UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL ATOMIC ENERGY AGENCY

INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS

Fourth College on Microprocessor-based Real-time Systems in Physics

Trieste, 7 October - 1 November 1996

LECTURE NOTES

Volume I

MIRAMARE - TRIESTE

October 1996

The United Nations University

Editors:
Abhaya S. Induruwa
Catharinus Verkerk

Presented by: Prof Indumathi - Dec '96 -

Foreword

The present volumes contain a first version of notes of the lectures delivered at the Fourth College on Microprocessor-based Real-time Systems in Physics, held at the International Centre for Theoretical Physics, Trieste from October 7 till November 1, 1996. It is hoped that these notes provide a readable record of the College.

The "Realtime Colleges" are an outgrowth of the "Microprocessor Colleges" which were organized since 1981 under the impulse of Professors Abdus Salam and Luciano Bertocchi, and of which several were held in developing countries. All these Colleges were sponsored by UNESCO, IAEA and UNU.

From the beginning, laboratory exercises and projects formed an essential ingredient of the course. They made use of equipment developed in-house until 1994, when a shift was made to the use of the Linux operating system and PCs.

The present College differs in several respects from the two previous ones: 1994 in Trieste and 1995 in Cape Coast, Ghana. Particular attention has been given to the configuration of the Linux operating system to make it present a nice-looking and user-friendly interface. More emphasis has been put on the development of embedded systems. To this end new boards were designed and produced in Turkey and Malaysia and the necessary software for cross-development implemented. This is also the first time that instead of copies of transparencies, the participants receive the lecture notes in a more readable form.

The preparation of all this required a large effort from a number of people. We gratefully acknowledge the essential contributions of Chu Suan Ang, Paul Bartholdi, Manuel Gonçalves, Carlos Kavka, Ulrich Raich, Pablo Santamarina, Alexei Tikhomorov and Jim Wetherilt. Without their great efforts, it would not have been possible to undertake the new things. We also wish to mention that in the past and present some 70-80 people, lecturers and instructors, have contributed to shaping this College into its present form. We gratefully acknowledge their contributions, which in a number of cases were immense.

We hope that the participants will enjoy the College and that they will benefit from it for their future activities.

Abhaya S. Induruwa,
Catharinus Verkerk,
Directors of the College,
Trieste, October 1996.

Contents

[1] This section has been prepared from notes by Merja Tornikoski, Finland

Principles of Real-Time Operating Systems, or Toward Real-Time

Fourth College on Microprocessor-based Real-time Systems in Physics

Trieste, 7 Oct–1 Nov 1996

Catharinus Verkerk*
email: Rinus.Verkerk@cern.ch

Abstract

We examine the operating system support needed for a real-time application. We'll see to what extent Linux satisfies the requirements and what can be done to adapt it.

*At present visitor at the International Centre for Theoretical Physics, Trieste, Italy

1 Introduction and a few definitions

A *real-time system* is defined as a system that *responds to an external stimulus within a specified, short time.* This definition covers a very large range of systems. For instance, a data-base management system can justly claim to operate in real-time, if the operator receives replies to his queries within a few seconds. As soon as the operator would have to wait for a reply for more than, say, 5 seconds, she would get annoyed by the slow response and maybe she would object to the adjective "real-time" being used for the system. Apart from having unhappy users, such a slow data-base query system would still be considered a real-time system.

The real-time systems we want to deal with are much more strict in requiring short response times than a human operator is, generally speaking. Response times well below a second are usually asked for, and often a delay of a few milliseconds is already unacceptable. In very critical applications the response may even have to arrive in a few tens of microseconds.

In order to claim rightly that we are having a real-time system, we must specify the response time of the system. If this response time can be occasionally exceeded, without any real harm being done, we are dealing with a *soft real-time system.* On the contrary, if it is considered to be a *failure* when the system does not respond within the specified time, we are having a *hard real-time system.* In a hard real-time system, exceeding the specified response time will generally result in serious damage of one sort or another, or in extreme cases even in the loss of human life. A data-base query system will generally fall in the first category: it will make little difference if a human operator will have to wait occasionally 6 seconds, instead of the specified 5 seconds response time, and nobody will dare to speak of a failure, as long as the replies to the queries are correct. This does not mean that all data-base systems are soft real-time systems: a data-base may well be used inside a hard real-time system, and its response may become part of the overall reaction time of the system.

Data-base systems are not at the centre of our attention in this course; we rather are interested in systems which control tne behaviour of some apparatus, machinery, or even an entire factory. We call these *real-time control systems.* We are litterally surrounded by such real-time control systems: video recorders, video cameras, CD players, microwave ovens, and washing machines are a few domestic examples. In the more technical sphere we will find the control of machine tools, of various functions of a car, of a chemical plant, etc., but also automatic pilots, robots, driver-less metro-trains, control of traffic-lights, and many, many more. Several of those systems are hard real-time systems: the automatic pilot is a good example.

We implicitly assumed that the systems we are dealing with are computer controlled. We are in fact interested in investigating the role the computer plays, what constraints are imposed by the part of the system external to the computer, or the environment in general, and what these constraints imply for the program that steers the entire process. We will pay particular attention to the role the underlying operating system plays and to what extent it may help in the development of a real-time control system.

At this point we should define two classes of real-time systems. On the one hand we have *embedded systems*, where the controlling microprocessor is an integral part of the entire product, invisible to the user and where the complete behaviour of the system is factory defined. The user can only issue a very limited and predefined set of instructions, usually with the help of switches, push-buttons and dials. There is no keyboard available to give orders to the device, nor is there a general output device which can give information on the state of the system. On a washing machine we can select four or five different programs, which define if we will wash first with cold and then with warm water, or if we skip the first, or which define how often we will rinse, if we will use the centrifugal drying or not, etc. If we add the control the user has over the temperature of the water, we have practically exhausted the possibilities of user intervention. The microprocessor included in the system has been programmed in the factory and cannot be reprogrammed by the user. Cost has been the overriding design consideration, user convenience played a secondary or tertiary role. These embedded systems run a monolithic, factory defined program and there is no trace of an interface to an operating system which would allow a user to intervene. This does not mean that such an embedded system does not take account of a number of principles, which should not be neglected in a system that claims to operate in real-time. All real-time aspects are folded into the monolithic program, indistinguishable of the other functions of the program.

The other class of real-time control systems comprises those systems that make use of a normal computer, which has not been severed of its keyboard and of its display device and where a human being can follow in some detail how the controlled process is behaving and where he can intervene by setting or modifying parameters, or by requesting more detailed information, etc. The essential difference with an embedded system is that a system in this second class can be entirely reprogrammed, if desired. Also, in contrast to an embedded system, the computer is not necessarily dedicated to the controlled process, and its spare capacity may be used for other purposes. So, a secretary may type and print a letter, while the computer continues to control the assembly line.

It is obvious that this class of real-time control systems needs to run an

operating system on the control computer. This operating system must be aware that it is controlling external equipment and that several operations initiated by it may be time-critical. The operating system must therefore be a **real-time operating system**. We will see in these lectures what this implies for the design and the capabilities of the operating system. We should keep in mind that we speak of generic real-time systems and generic real-time operating systems. The real-time control system does not necessarily use all features of the operating system, but the unused ones remain present, ready to be used at a possible later upgrade of the control system. This again is in contrast with the embedded system, where the parts of the operating system needed are cast in concrete inside the controlling program and where all other parts of it have been discarded.

2 The ingredients of a real-time computer controlled system

In order to investigate to some extent what the ingredients of a real-time control system are and what the implications are for a supporting operating system, we will take a simple example, which does not require any a-priori knowledge: a railway signalling system.

Safety in a railway system, and in particular collision-avoidance is based on a very simple principle. A railway track, for instance connecting two cities, is divided into sections of a few kilometers length each (the exact length depends on the amount of traffic and the average speed of the trains). Access to a section — called a block in railway jargon — is protected by a signal or a semaphore: when the signal exhibits a red light, access to the block is prohibited and a train should stop. A green light indicates that the road is free and that a train may proceed. The colour of the light is pre-announced some distance ahead, so that a train may slow down and stop in time. Access to a block is allowed if and only if there is no train already present in the block and prohibited as long as the block is "occupied". Normally all signals exhibit a red light; a signal is put to green only a short time before the expected passage of a train and if the condition mentioned above is satisfied. Immediately after the passage of the train, the signal is put back to red. The previous block is considered to be free only when the entire train has left it.

We will try to outline briefly what would be required if we decided to make a centralized, computer controlled system for the signalling of the entire railway system in a small or medium-sized country, comprising a few thousand kilometers of track, with hundred or so trains running simultaneously. This

would be a large-scale system, but it would be conceptually rather simple. The basic rule is: if there is a train moving forward in block $i-1$, and block i is free, the signal protecting the entrance to block i shall be put to green and back to red again as soon as the first part of the train has entered block i. For the time being we consider only double track inter-city connections, where trains are always running in the same direction on a given track.

From the rule we see that we need to know at any instant in time which blocks are free and which are occupied. So we need a sort of a *data-base* to contain this information. This data base must be regularly updated, to reflect faithfully the real situation. In fact, whenever a train is leaving a block and entering another, the data-base must be updated.

How do we know that a train moves from one block to the next? Trains are supposed to run according to a time table and at predefined speeds, so a simple algorithm should be able to provide the positions of all trains in the system at any moment. Unfortunately this assumption is not valid under all circumstances and we need a reliable signalling system, exactly to be able to cope with more or less unexpected situations where trains run too late, or not at all, or where an extra train has been added, or another ran into trouble somewhere. We conclude that it is better to actually *measure* the event that a train crosses the boundary between two blocks. We could put a switch on the rails, which would be closed by the train when it is on top of the switch. We could scan all the switches in our system at regular intervals. How long — or rather how short — should this interval be? A TGV of 200 meter length and running at close to 300 km/h, would be on top of a switch for $2\frac{1}{2}$ seconds. A lonely locomotive, running at 100 km/h would remain on top of the contact for less than a second. So we must scan some thousand or more contacts in, say $\frac{1}{2}$ second. This can be done, but it would impose a heavy load on the system and we would find the vast majority of the switches open in any case. We could refine our method and scan only those contacts where we expect a train to arrive soon. This would reduce the load on the system, as only hundred or so contacts have to be scanned, but it still is not very satisfactory, as we will continue to find many open contacts. Note that instead of contacts, we could have used other detection methods: strain gauges on the rails, or photo-cells.

A better way of detecting the passage of a train, is by using hardware interrupts [1]. We could generate an interrupt when the contact closes and another when it opens again, indicating the entrance of a train into block

[1] For those who may have forgotten: a hardware interrupt is caused by an external electrical signal. The normal flow of the program is interrupted and a jump to a fixed address occurs, where some work is done to *handle* the interrupt. A "return from interrupt" instruction brings us back to the point where the program was interrupted.

i and the exit of the same train from block $i-1$, respectively. We don't lose time then anymore for looking at open contacts. We also simplify the procedure, for we do not have to look anymore at the data-base before the start of a scan, to find out which contacts are likely to be closed by a train soon.

We have discovered here a very important ingredient of any real-time control system: the *instrumentation with sensors and actuators.* In our case we must sense the presence of a train at given positions along the tracks, and we must actuate the signals, putting them to green and to red again. Generally speaking, the instrumentation of a real-time control system is a very important aspect, which must be carefully considered. Usually, apart from *sensors which provide single-bit information*, such as switches, push-buttons, photocells, *which can also be used to generate hardware interrupts*, we will need *measuring devices, giving an analog voltage output*, which then has to be *converted into a digital value* with an analog-to-digital converter. Conversely, *output devices may be single bit*, such as relays, lamps and the like, or *digital values, to be converted into analog voltages.* **Accuracy, reproducibility, voltage range, frequency response** etc. have to be considered carefully. The operation of a system may critically depend on how it has been instrumented. The *interface to the computer* is another aspect to take into account for its possible consequences. **Speed, reliability and cost** are some of the concurring aspects. We will not dwell any further on these topics in these lectures, as they are too closely related to the particular application, making a general treatment impossible.

For our railway signalling system we mentioned the timetable, claiming that we could not rely on it. We can however use it to check the true situation against it to detect any anomaly. These anomalies could then be reported immediately. For instance, we could tell the station master of the destination, that the train is likely to have a delay of x minutes. Another useful thing is to keep a *log* of the situation. This can be used for daily reports to the direction (where they would probably be filed away immediately), but they could prove valuable for extracting statistics and for global improvement of the system. *Operator intervention* is also needed. For instance, when a train, running from station A to station B, leaves station A, it does not yet exist in the data-base. Likewise, when it arrives at B, it has to be removed from the data-base. This could be done automatically, in principle, but what do we do if it has been decided to run two extra trains, because there is an important football match? We conclude that *data-logging, operator intervention* and some *calculations* (to check actual situation against predicted one) are also essential ingredients of a real-time control system, in addition to the *interrupt handling, interfacing to the sensors and actuators* and *updating of the data-*

base reflecting the state of the system.

This idyllic picture of our railway signalling system might stimulate us to start coding immediately. A program which uses the principles outlined above does not seem too difficult to produce. We simply let the program execute a large loop, where all different tasks are done one after the other. The interrupts have made it possible to get rid of a serious constraint, so all seems to be nice and straightforward. Once we would have a first version of the program ready, we would like to test it. Hopefully we will use some sort of a test rig at this stage, and abstain from experimenting with real trains. During the testing stage, we will then quickly wake up and find that we have to face reality.

In our model, we assumed double track connections between cities, where on a given track, trains always run in the same direction. But, even in the case that the entire railway network is double track between cities, we must nevertheless consider also single track operation, because a double track connection may have to be operated for a limited period of time and for a limited distance as a single track, repair or maintenance work making the other track unusable.

Assume that, on a single track, we have two trains, one in block $k+1$, the other in block $k-1$, running in opposite directions, both toward block k. If we would apply our simple rule, they would both be allowed to enter block k (supposing it was free) and a head-on collision would result. The problem can be solved by slightly modifying our rule: If a train is moving forward in block $i-1$ toward block i, then access to block i will be allowed if blocks i and $i+1$ are free. So both trains will be denied access to block k in our example. We have eliminated the possibility of a head-on collision, but we now have another problem. Assume that our two trains are in block $k-2$ and $k+1$ respectively and running toward each other. Applying our new rule, they would be allowed to enter block $k-1$ and k respectively and both trains would stop, nose to nose at the boundary between these two blocks. We have created a sort of a *deadlock situation.*

The true solution is of course not to allow a south-bound train into the entire section of single track, as long as there is still a north-bound train somewhere in the entire section, and vice-versa. South-bound and north-bound trains compete for the same "**resource**", the piece of single track railway. They are **mutually exclusive** and only one type of train should be allowed to use the resource. If the stretch of single track is long enough, and comprises severa' blocks, more than one north-bound train can be running on that stretch of track. Now assume that several north-bound trains are occupying the stretch of single track and that a south-bound train presents itself at the nothern end of the stretch. It obviously has to wait, but while it

is waiting, do we continue to allow more north-bound trains into the stretch? This is a matter of *priority*, which should be defined for each train. A *scheduler* should take the priorities into account and deny the entrance into the stretch for a north-bound train if the waiting south-bound one has higher priority. As soon as the stretch has then been emptied of all north-going trains, the south-bound one can proceed, possibly followed by others.

A similar situation, where two trains may be competing for the same resource, arises when two tracks, coming from cities A and B, merge into a single track entering city C. Obviously, if two trains approach the junction simultaneously, only one can be allowed to proceed, which should be the one with the highest priority. It should be noted that the priority assigned to a train is not necessarily static. It may change dynamically. For instance, a train running behind schedule, may have its priority increased at the approach of the junction and allowed to enter city C, before another train which normally would have had precedence. This latter example illustrates a *synchronization problem*: some trains may carry passengers which have to change trains in city C; the two trains should reach the station of C in the right order.

We have thus discovered some more ingredients (or concepts) for a real-time control system: **priorities, mutual exclusion, synchronization**.

We started off by considering our railway signalling problem being controlled by a single program, which guides all trains through all tracks, junctions and crossings. We have gradually come to have a different look at the problem: *a set of trains, using resources* (pieces of railway track), *and sometimes competing for the same resource.* We can consider our trains as *independent objects*, more or less unaware of the existence of similar objects and of the competition this may imply. In order to get a resource, every train must put forward a request to some sort of a master mind (the real-time operating system), who will honour the request, or put the train in a waiting state.

At this stage, we realize that we better abandon our first version of the program, because it would have to be rewritten from scratch in any case. We have become aware that our particular real-time control system may have many things in common with other real-time systems and that it would be advantageous to take profit from the facilities a real-time operating system offers to solve the problems of mutual exclusion, priorities, etc. Once we have mastered the use of these facilities, we can build on our experience for the implementation of another real-time control system. In case we would obstinately continue to adapt our original program, we would probably find, after months of effort, that we have rewritten large parts of a real-time operating system, but which have been so intimately interwoven with the application

program, that it will be difficult, if not impossible, to re-use it for the next application we may be called to tackle.

Other aspects we have not yet considered may also build very nicely on the foundations laid by a real-time operating system. For instance, we have the problem of dealing with *emergencies*. A train may have derailed and obstructed both tracks. Such an unusual and potentially dangerous situation must be immediately notified to the operating system which can then take the necessary measures. If they cannot be notified, a mechanism for detecting potentially dangerous situations must be devised: in our particular case, the system should be alerted if a train does not leave its block within a reasonable time. In other words, a *time-out* could be detected.

Now that we mentioned time, we are reminded of the fact that **time** may play an important role in any real-time system, either in the form of *elapsed time*, or of the *time of the day*. It is difficult to think of a system that could operate without the help of a clock. A **real-time clock** and the possibility to program it to generate a clock interrupt at certain points in time, or after a given time-interval has elapsed, are therefore indispensable ingredients of a real-time control system.

Reliability of the entire system is another item for serious consideration. You certainly do not want a parity error in a disk record to bring your system to a halt or to create a chain of very nasty incidents.

In many cases, we are not dealing with a closed system, so there must be a means of *communicating with other systems* (our national railway network is connected to other networks, and trains do regularly cross the border). *User-friendly interfaces to human operators*, which usually implies the use of graphics, are also very likely to be an essential ingredient of our real-time control system. A large synoptic panel, showing where all trains are in the network, would be the supervisor's dream, not to speak of makers of science fiction films.

In the following lectures, we will investigate in more depth the various features a real-time operating system should provide. Making use of these features will prevent us from re-inventing the wheel.

The question then arises: which real-time operating system should I use? There are several on the market: *OS-9* for *Motorola 68000 machines*, and *QNX* or *LynxOS* for *Intel machines*, *Solaris* for *Sparc processors*, to mention a few. These systems are sold together with the tools necessary to build a real-time application: **compiler, assembler, shell, editor, simulator, etc**. A minimum configuration would cost US$ 2000-2500, a full configuration may push the price up into the 10 K$ range. This would cause no problem whatsoever for a railway company, but what about you?

Another solution is to use a **real-time kernel**, useful for embedded sys-

tems, which you *compile and link into your application. VxWorks, MCX11* and *µCOS* are examples. They are much cheaper —or practically free: *MCX11* and *µCOS* [2]—, but you will need a complete development system in addition. This development system could of course be Linux.

The ideal would be to be able to **use Linux for development of a real-time control system, as well as for running the application**. We will see shortly to what extent this is possible at present. Before proceeding, however, we will make sure that we understand the fundamental concept of a **process**.

3 Processes

In our example we have seen that a real-time system has a number of tasks to accomplish: besides ensuring that trains could proceed from block to block without making collisions, we had to log data, keep the data-base up-to-date, communicate with the operator, cater for emergency situations, etc. Not all of these tasks have the same priority, of course.

When we analyze a real-time system, we will almost invariably be able to identify *different tasks*, which are more or less *independent of each other*. "Independent of each other" really means that each task can be programmed without thinking too much of the other tasks the system is to perform. At most there is some *intertask communication*, but every task does its job on its own, without requiring assistance from other tasks. If assistance is required, the operating system should provide it. The system designer should identify and define the different tasks in such a way that they really are as independent of each other as possible. Some *synchronization* may be needed: certain tasks can only run after another task has completed. For instance, if some calculations have to be done on collected data, the data collection tasks could be totally separated from the calculation task. In order to make sense, the latter should only be executed when the data collection task has obtained all data necessary for the calculation. This implies that some inter-task communication is needed here. The true difficulty of dividing the overall system requirement up into different tasks consists of choosing the tasks for maximum independence, or —in other words— for **minimum need of inter-task communication and synchronization**.

These various tasks can now be implemented as different programs and then run as different **processes**.

What exactly is a process and what is the *difference between a program and a process*? A program is an orderly sequence of machine instructions,

[2] *RTEMS* is a recent, very complete real-time executive, also available for free.

which could have been obtained by compilation of a sequence of high-level programming language statements. It is not much more than the listing of these statements, which can be stored on disk, or archived in a filing cabinet. It becomes useful only when it is run on a machine and executing its instructions in the desired sequence, thus obtaining some result. It is only useful when it has become a *running process.*

A process is therefore a running (or runnable) program, together with its data, stack, files, etc. It is only when the code of a program has been loaded into memory, and data and stack space allocated to it, that it becomes a runnable process. The operating system will then have set up an entry in the *process descriptor table*, which is also part of the process, in the sense that this information would disappear when the process itself ceases to exist. The operating system may decide at a certain moment to run this runnable process, on the basis of its priority and the priorities of other runnable processes. This would happen in general when the running process is unable to proceed —e.g. because it is waiting for input to become available—, or because the time allocated to it has run out.

We should emphasize that we are considering only the case of a single processor system, where only one process can run at a time. The other runnable processes will wait for the CPU to become free again. If the different processes are run in quick succession, a human observer would have the impression that these processes are executed simultaneously.

The consequence of this is that we can write a program to calculate Bessel functions, without having to think at all about the fact that when we will run our program, there may already be fifty or more other processes running, some of them even calculating Bessel functions. In as far as we have written our program to be autonomous, it will not be aware of the existence of other runnable processes in the computer system. Consequently, it cannot communicate with the other processes either: its fate is entirely in the hands of the operating system[3].

There may exist on the disk a general program to calculate Bessel functions and on a general purpose time-sharing computer system several users may be running this program. A reasonable operating system should then keep only one copy of the program code in memory, but each user process running this program should have its *own* process descriptor, its *own* data area in memory, its *own* stack and its *own* files. All users of the computer system will presumably run a **shell**. Command shells, such as *bash* or *tcsh* are very large programs and it would be an enormous waste if every single

[3] And luckily so: the operating system will also provide protection, avoiding that other processes interfere with ours.

user of a time-sharing system would have his own copy of the shell in memory.

In general, we will have a number of runnable processes in our uniprocessor machine, and one process running at a given instant of time. When will the waiting processes get a chance to run? There are two reasons for suspending the execution of the running process: either the *time-slice* allocated to it *has been exhausted*, or *it cannot proceed any further without some event happening*. For instance, the process must wait for input data to become available, or for a *signal* from another process or the operating system, or it has to complete an output operation first, etc. The programmer does not have to bother about this. At a given point in the program, where it needs to have more input data, the programmer simply writes a statement such as: *read(file,buffer,n);*. The compiler will translate this into a call to a *library function*, which in turn will make a **system call**, (or **service request**), which will transfer control to the kernel. Our process becomes *suspended* for the time the kernel needs to process this system call. In the case of a read operation on a file, the kernel will set this into motion, by emitting the necessary orders to the disk controller. As the disk controller will need time to execute this order, the kernel will decide to **block** the execution of the process which was running and which made the system call. This blocked process will be put in the *queue of waiting processes*, and it will become runnable again later, when the disk controller will have notified the kernel —by sending a hardware interrupt— that the I/O operation has been completed. The kernel makes use of the **scheduler** to find, from the queue of runnable processes the one that should now be run. The kernel will then make a **context switch** and this will start the new process running.

A context switch is a relatively heavy affair: first all hardware registers of the old (running) process must be saved in the process descriptor of the old process. Then the new process must be selected by the scheduler. If the code and data and stack of the new process are not yet available in memory, they must be loaded. In order to be loaded, it may be necessary first to make room in memory, by swapping out some memory pages which are no longer needed or which are rarely used. The page tables must be updated, and the process descriptors must be modified to reflect the new situation. Finally the hardware registers of our machine must be restored from the values saved at an earlier occasion for the process now ready to start running. The last register to be restored is the program counter. The next machine instruction executed will then be exactly the one where the new process left off when it was suspended the last time.

The execution of a program will thus proceed piecemeal, but without the programmer having to bother about it: the operating system takes care of everything. So the application programmer can continue to believe that

his program is the only one in the world. The price to be paid for this convenience is the overhead in time and memory resources introduced by the intervention of the operating system.

For our Bessel function program we are entirely justified in thinking that we are alone in the world. There are however situations where this is not the case and where different processes interfere with each other, either willingly or unwillingly. Here is my favourite example of such a case of interference[4].

Assume that we have three separate bytes in memory which contain the hour, minutes and seconds of the time of the day. There is a hardware device which produces an interrupt every second and wakes up a process that will update these three bytes. Any process which wants to know the time, can access these three memory bytes, one after the other (we assume that our machine can address only one byte at a time). Now suppose that it is 10.59.59 and that a process has just read the first byte "10", when a clock interrupt –which has a higher priority than the running process– occurs. Our process is suspended, the clock process updates the time, setting it to 11.00.00. Control now returns to the first process which continues reading the next two bytes. The result is: 10.00.00; which is one hour wrong. What happened here is that *two processes access the same resource* —the three memory bytes— and that one or both of them can alter the contents. No harm would be done if both processes had *read-only* access to the shared resource.

The reader should note that the concept of a process has allowed us to speak about them as if they were really running simultaneously. We do not have to include in our reasoning the fact that there is a context switch and that complicated things are going on behind the scenes. We only have to be aware that access to shared resources must be protected, in order to avoid that another process accesses the same resource "simultaneously". On a multi-processor system "simultaneous" can really mean "at the same instant in time", on a uni-processor machine it really means "concurrently". The processor concept is equally valid for a uni- and a multi-processor machine.

The places in the program where a shared resource is accessed are so-called **critical regions**. We must avoid that two processes access simultaneously the resource and this can be done by ensuring that a process cannot enter a critical region when another process is already in a critical region where it accesses the same shared resource. The entrance to a critical region must be protected with a sort of a lock.

Two operations are defined on such a lock: *lock* and *unlock*. The lock operation tests the state of the lock and if it is unlocked, locks it in the

[4]The reader should be aware that the example describes a primitive situation; no modern operating system would allow this situation to occur.

same **atomic** operation. If the lock is already locked, the lock operation will stop the process from entering the critical region. The unlock operation will simply clear a lock which was locked, and allow the other process access to the critical region again. That these operations must be *atomic* means that it must be impossible to interrupt them in the middle. Otherwise we would get into awkward situations again. If the lock operation would not be atomic, we could have a situation where process 1 inspects the lock and finds it open. If immediately after this, process 1 gets interrupted, before it had a chance to close the lock, process 2 could then also inspect the lock. It finds that it is open, sets it to closed, enters the critical region where it grabs the resource (a printer for instance) and starts using it. Some time later process 1 will run again, it will also close the lock and it will also grab the same printer and start using it. Remember that it had found the lock to be open and process 1 is unaware that process 2 has been running in the meantime!

The lock and unlock operations must therefore be completed before an interruption is allowed. This can be done —primitively— by disabling interrupts and then enabling them after the operation. No reasonable operating system would allow a normal user to tinker with the interrupts, so most machines have a *test-and-set* instruction. The *test-and-set* instruction tests a bit and sets it to "one" if it was "zero". If it was already "one" it is left unchanged. The result of the test (i.e. the state of the bit before the *test-and-set* instruction was executed is available in the processor status word and can be tested by a subsequent *branch* instruction. The *test-and-set* instruction is a single instruction; a hardware interrupt arriving during the execution of the instruction will be recognized only after the execution is complete. This guarantees the atomicity of a *test-and-set* operation.

What do we do after the *test-and-set* instruction? If the lock was open, you can safely enter the critical region. If, on the contrary, process A finds the lock closed, it should go to **sleep**. The operating system will then suspend the execution of process A and schedule another process to run, say C, or E. The process B, which had closed the lock in the first place, will also be running again at some instant and eventually will unlock the lock and **wakeup** the sleeping process A. The system will then make process A runnable again.

Now suppose that process A gets interrupted immediately after doing its — unsuccessful — lock operation and before it could execute the *sleep()* call. Process B will at some stage open the lock and wakeup A. As A is not sleeping, this wakeup is simply lost. When A will run again, it will truely go to sleep, this time forever.

The solution to the problem was given in 1965 by Dijkstra, when he defined the **semaphore**. A semaphore counts the number of wakeups which have been "saved". It can therefore have a positive value, or "0". Two

atomic operations are defined on a semaphore, which we will call **up** and **down**[5]. The *down* operation checks the value of the semaphore. If it is greater than zero, it decrements the value and just continues. If it finds that the semaphore value is zero, the process is put to sleep. Once an operation on a semaphore is started, no other process can access the same semaphore. Thus **atomicity** of a semaphore operation is guaranteed. The work done for a *down* (and similarly for an *up*) operation must therefore be part of the operating system and not of a user process. The *up* operation on a semaphore increments its value. If one or more processes were sleeping, one of them is selected by the operating system. It can then complete its *down*, which had failed earlier, and it will be allowed to run. Thus, if the semaphore was positive, it will simply be incremented, but if it was "0", meaning that there are processes sleeping on it, its value will remain "0", but there will be one process less sleeping.

We have described the general form of a semaphore: the **counting semaphore**, which is used to solve **synchronization** problems, ensuring that certain events happen in the correct order. A **binary semaphore** can only take the values "0" or "1" and is particularly suited for solving problems of *mutual exclusion*, which explains its other name: **mutex**.

To illustrate the use of mutexes and counting semaphores we show an example of the **Producer-Consumer** problem. Suppose we have two collaborating processes: a *producer* which produces items and puts them in a *buffer* of finite size, and a *consumer* which takes items out of the buffer and consumes them. A data acquisition system which writes the collected data to tape is a good example of a producer-consumer problem. It is clear that the producer should stop producing when the buffer is full; likewise, the consumer should go to sleep when the buffer is empty. The consumer should wake up when there are again items in the buffer and the producer can start working again when some room in the buffer has been freed by the consumer. In order to obtain this synchronization between the two processes, two *counting semaphores* are used: *full* which is initialized to "0" and counts the buffer slots which are filled, and *empty*, initialized to the size of the buffer and which counts the empty slots. Access to the buffer, which is shared between the two processes, is protected by a *mutex*, initially "1" and thus allowing access. The example is taken from Andrew Tanenbaum's excellent book[6]. The

[5] Various other names are also used: post and signal, P and V (the original names given by Dijkstra), and possibly others. For mutexes, lock and unlock are often used.

[6] Andrew S. Tanenbaum, Modern Operating Systems, Prentice Hall International Editions, 1992, ISBN 0-13-595752-4. The reader is encouraged to read the chapter on Interprocess Communication, which provides a much more detailed treatment of synchronization problems than is possible here.

reader should study carefully the listing of the *Producer-Consumer* problem at the end of this section. He should be aware that the example is simplified: instead of two *processes* and a buffer structure in *shared memory*, the listing shows two functions, using global variables. Also the semaphores are not exactly what the standards prescribe. Using semaphores and mutexes remains a difficult thing: changing the order of two *down* operations in the listing below may result in chaos again.

```
#define N 100                   /* number of slots in buffer */
typedef int semaphore;          /* this is NOT POSIX !!  */
semaphore mutex=1;              /* controls access to critical region */
semaphore empty=N;              /* counts empty buffer slots */
semaphore full=0;               /* counts full buffer slots*/

void producer(void)
{
  int item;
  while(TRUE) {                 /* do forever (TRUE=1) */
      produce_item(&item);      /* make something to put in buffer */
      down(&empty);             /* decrement empty count */
      down(&mutex);             /* enter critical region */
      enter_item(&item);        /* put new item in buffer */
      up(&mutex);               /* leave critical region */
      up(&full);                /* increment count of full slots */
  }
}

void consumer(void)
{
  int item;
  while(TRUE) {                 /* do forever */
      down(&full);              /* decrement full count */
      down(&mutex);             /* enter critical region */
      remove_item(&item);       /* take item from buffer */
      up(&mutex);               /* leave critical region */
      up(&empty);               /* increment count of empty slots */
      consume_item(&item)       /* use the item */
  }
}
```

4 What is wrong with Linux?

UNIX has the bad reputation of not being a real-time operating system. This needs some explanation. Time is an essential ingredient of a real-time system: the definition says that a real-time system must respond within a given time to an external stimulus. Theoretically, it is not possible to guarantee on a general UNIX time-sharing system that the response will occur within a specified time. Although in general the response will be available within a reasonable time, the load on the system cannot be predicted and unexpected delays may occur. It would be a bad idea to try and run a time-critical real-time application on an overloaded campus computer. Nevertheless, before discarding altogether the idea of using UNIX or Linux as the underlying operating system for a real-time application, we should have a critical look at what the requirements really are, to what extent they are satisfied by off-the-shelf Linux, and what can be done (or has been done already) to improve the situation.

The UNIX and Linux schedulers have been designed for **time-sharing** the CPU between a large number of users (or processes). It has been designed to give a *fair share of the resources*, in particular of CPU time, to all of these processes. The priorities of the various processes are therefore adjusted regularly in order to achieve this. For instance, the numerical analyst who runs CPU-intensive programs and does practically no I/O, will be penalized, to avoid that he absorbs all the CPU time.

Such a scheduling algorithm is not suitable for running a real-time application. If the operating system would decide that this particularly demanding application had consumed a sufficiently large portion of the available CPU time, it would lower its priority and the application might not be able anymore to meet its deadlines.

A real-time application must have **high priority** and —in order to be able to meet its deadlines— *must run whenever there is no runnable program with a higher priority.* In practice, the real-time process should have the highest priority, and it should keep this highest priority throughout its entire life[7]. Another scheduling algorithm is therefore required: a certain class of processes should be allocated permanently the highest priorities defined in the system. The normal scheduler of Linux does not have this feature, but *another scheduler*, designed for real-time use is available and *can be compiled into the kernel.*

Time being a precious resource for a real-time system, overheads imposed

[7]It would be wise to run a shell with an even higher priority, in order to be able to intervene when the real-time process runs out of hand. This shell would be sleeping, until it gets woken up by a keystroke.

by the operating system should be avoided as much as possible. Some of the overheads can be avoided by careful design of the real-time program. For instance, knowing that forking a new process is a time-consuming business, all processes which the real-time application may need to run, should be forked and exec'ed (the *fork* and *exec* system calls will be illustrated in section 5) *during the initialization phase* of the application. Other overheads cannot be avoided so simply and need some adaptation or modification of the operating system.

Context switches may be very expensive in time, in particular when the code of the new process to be run is not yet available in memory and/or when room must be made in memory. All code and data of a real-time application should be **locked into memory**, so that this part of a context switch would not cause a loss of time. Locking everything into memory will also prevent *page faults* to happen, avoiding this way other *memory swapping* operations. Off-the-shelf Linux does not have the possibility of locking processes into memory, but again, *there is a package available* which, when compiled into the kernel, will do it.

A further help in reducing the overheads due to context switches is to use so-called *light-weight processes* or **multi-threaded user processes**. Linux as such does not provide these, but *a package does exist* to implement the latter.

Other places where to watch for lurking losses of time are Input/Output operations. Normally, when a file is opened for writing, an initial block of disc sectors is allocated —usually 4096 bytes— and *inodes* and *directory entries* are updated. When the file grows beyond its allocated size, the relatively lengthy process of finding another free block of 4096 bytes and updating inodes and directory entries is repeated. A real-time system should be able to grab all the disc space it needs during initialization, so that these time losses may be avoided. Linux does not allow this at present.

All input and output in Linux is **synchronous**. This means that a process requesting an I/O operation will be *blocked until the operation is complete* (or an error is returned). Upon completion of the operation, the process becomes *runnable* again and it will effectively run when the scheduler decides so. However, "completion" of an output operation means only that the data have arrived in an output buffer, and there is no guarantee that the data have really been written out to tape or disc. When the process is only notified of completion of the I/O operation when the data are really in their final destination, we have **synchronized I/O**, which may be a necessity for certain real-time problems. Linux does not spontaneously do *synchronized I/O*, but it *can be easily imposed by using sync or fsync.*

Asynchronous I/O may be another real-time requirement. It means

that *the process* requesting the I/O operation *should not block* and wait for completion, *but continue processing* immediately after making the I/O system call. The standard device drivers of Linux do not work asynchronously, but the primitive system calls allow the option of continuing processing. A special purpose device driver could make use of this and thus do asynchronous I/O. The process will then be notified with an interrupt when the I/O operation has been completed.

The designer of a real-time system should of course also be aware that no standard device drivers exist for exotic devices. They have to be written by the application programmer. In a standard UNIX system, such a new device driver must be compiled and linked into the kernel. Linux has a very nice feature: it allows to *dynamically load and link to the kernel so-called modules*, which can be —and very often are— device drivers.

We have shown before that it would be wise to divide a real-time system up into a set of processes, which can each care for their own business, without excessively interfering with each other. Nevertheless, some communication between processes may be needed. Old UNIX systems had only two interprocess communication mechanisms: **pipes** and **signals**. Signals have a very low information content, and only two user definable signals exist. System V UNIX added other IPC mechanisms: sets of **counting semaphores, message queues**, and **shared memory**. Most Linux kernels have the System V IPC features compiled in.

Probably no real-time system could live without a **real-time clock** and **interval timers**. They do exist in off-the-shelf systems, but the *resolution*, usualy 1/50 th or 1/100 th of a second, may not be enough. The user-threads package can work with higher resolutions, if the hardware is adequate.

The IEEE has made a large effort to standardize the user interface to operating systems. The result of this effort has been the POSIX.1 standard, which defines a set of system calls, and POSIX.2, which defines a standard set of Shell commands. POSIX.1 and POSIX.2 have been approved by IEEE and by ISO and have thus gained international acceptance. Also **real-time extensions to operating systems** have been defined in the **POSIX.4** (later renamed as POSIX 1003.1c-1994) standard, which has also been accepted by ISO. All the points discussed above are part of the POSIX.4 standard, except for the **multiple threads and mutexes, which are defined in POSIX.4a**. To the best of my knowledge, POSIX.4a is now also an international standard. Linux is POSIX.1 and POSIX.2 compliant, and a truly *POSIX.1 certified version is available.*

In summary, Linux is weak on the following:

- **Mutexes**. A simple mutex does not exist. The System V IPC semaphores

can be used, although they are overkill, introducing a large overhead. Atomic bit operations are defined in *asm/bitops.h* and can be used more easily, but care should be exercised (danger of *priority inversion*). Mutexes are defined in the **pthreads** package, but work only between user threads in the same process.

- **Interprocess Communication.** System V IPC is usually part of the Linux kernel and adds counting semaphores, message queues and shared memory to the usual mechanisms of pipes and of signals.
- **Scheduling.** A POSIX.4 compliant scheduler for Linux exists. We do not have experience with it yet.
- **Memory Locking.** A POSIX.4 compliant package exists for Linux which implements memory locking. We have not yet tested this package.
- **Multiple User Threads.** A library implementation of pthreads (as defined in POSIX.4a) exists and can be used. You will soon get into close contact with it.
- **Synchronized I/O.** Can be obtained easily with *sync* and *fsync*.
- **Asynchronous I/O.** Not available in standard device drivers. Could be implemented for special purpose device drivers.
- **Pre-allocation of file space.** Not available.
- **Fine-grained real-time clocks and interval timers.** They are part of one of the available pthreads packages and could be used if the hardware is capable.

5 Creating Processes

Creating a new process from within another process is done with the *fork() system call. fork()* creates a new process, called the **child process**, *which is an exact copy of the original (parent) process*, including open files, file pointers etc. Before the *fork()* call there is only *one* process; when the *fork()* has finished its job, there are *two*. In order to deal with this situation, *fork()* **returns twice**. To the parent process it returns the **process identification (PID)** of the child process, which will allow the parent to communicate later with the child. To the child process it returns a 0. As the two processes are

exact copies of each other, an *if* statement can determine if we are executing the child or the parent process.

There is not much use of a child process which is an exact copy of its parent, so the first thing the child has to do is to load into memory the program code that it should execute and then start execution at *main()*. A child is obviously too inexperienced to do this on its own, so there is a system call that does it for him: *execl()*. The entire operation of creating a new process therefore goes as follows:

```
/* here we have been doing things */
child=fork();           /* PID of new process --> child */
if(child){              /* here for parent process */
                        /*continue parent's business*/
    }
else {                  /* here for child process */
    execl("/home/boss/rtapp/toggle_rail_signal",\
     "toggle_rail_signal", N_sigs, NULL);
    perror("execl");    /* here in case of error */
    exit(1);
    }
/* here continues what the parent was doing */
```

execl() will do what was described above, so in our example it will load the executable file */home/boss/rtapp/toggle_rail_signal* and then start execution of the new process at *main(argc,argv)*. The other arguments of *execl()* are passed on to *main()*. *execl* is one of six variants of the exec system call: *execl, execv, execle, execve, execlp, execvp*. They differ in the way the arguments are passed to *main()*: **l** means that a list of arguments is passed, **v** indicates that a pointer to a vector of arguments is passed. **e** tells that environment pointer of the parent is passed and the letter **p** means that the *environment variable* **PATH** should be used to find the executable file.

This completes the creation of a new process. On a single CPU machine, one of the two processes may continue execution, the other will wait till the scheduler decides to run it. There is no guarantee that the parent will run before the child or vice versa.

The new process can *exit()* normally when it has done its job, or when it hits an error condition. The parent can *wait* for the child to finish and then find out the reason of the child's death by executing one of the following system calls:

```
pid_t wait(int *status); /* wait for any child to die*/
or: pid_t waitpid(pid_t which, int *status, int options)
            /* wait for child "which" to die */
```

These wait calls can be useful for doing some cleaning-up and to avoid leaving *zombies* behind. When the parent process exits, the system will do all the necessary clean-up, childs included.

We can now understand what the shell does when we type a command, such as *cp file1 dir*. The shell will parse the command line, and assume that the first word is the name of an executable file. It will then do a *fork()*, creating a copy of the shell, followed by an *execl()* or *execv()* which will load the new program, in our example the copy utility *cp*. The rest of the command line is passed on to *cp* as a list or as a pointer to a vector. The shell then does a *wait()*. When an & had been appended to the command line, then the shell will *not* do a wait, but will continue execution after return from the *exec* call.

The following gives a more complete and rather realistic example of a *terminal server and a client*[8]. The reader is invited to study this example in detail.

The code for the **server** looks like:

```
#define     POSIX_C_SOURCE 199309

#include    <unistd.h>
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/wait.h>
#include    <signal.h>
#include    <errno.h>
#include    "app.h" /* local definitions */

main(int argc, char **argv)
{
    request_t r;
    pid_t terminated;
    int status;

    init_server();   /* set things up * /
```

[8]the example is taken from Bill O. Gallmeister, POSIX.4, Programming for the Real World, O'Reilly, 1995.

```
    do {
        check_for_exited_children();
        r = await_request();   /*get some input*/
        service_request(r);   /*do what wanted*/
        send_reply(r);        /*tell we did it*/
    } while (r != NULL);

    shutdown_server();    /*tear things down*/
    exit(0);
}

void
service_request(request_t r)
{
    pid_t child;
    switch (r->r_op) {
        case OP_NEW:
            /* Create a new client process */
            child = fork();
            if (child) {
                /* parent process */
                break;
            } else {
                /* child process */
                execlp("terminal","terminal \
                 application","/dev/com1",NULL);
                perror("execlp");
                exit(1);
            }
            break;
        default:
            printf("Bad op %d\n", r->r_op),
            break;
    }
    return;
}
```

The **terminal** end of the application looks like:

```
#define     POSIX_C_SOURCE 199309

#include    <unistd.h>
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/wait.h>
#include    <signal.h>
#include    "app.h" /* local definitions */

char *myname;

main(int argc, char **argv)
{
    myname = argv[0];
    printf("Terminal \"%s\" here!", myname);
    while (1) {
        /* deal with the screen */
        /* await user input */
    }
    exit(0);
}
```

Presumably *request_t* is defined in *app.h* as a pointer to a structure. *await_request()* is a function which sleeps until a service request arrives from a terminal. The operations performed by the other functions: *init_server, service_request(), check_for_exited_children(), send_reply()* and *shutdown_server()* are implied by their names.

6 Interprocess Communication

In the case where we have a real-time application with a number of processes running concurrently, it would be a normal situation when some of these processes need to communicate between them. We said already that the classical UNIX system only knows pipes and signals as communication mechanisms. Interprocess communication, suitable for real-time applications is an essential part of the POSIX.4 standard, which adds a number of mechanisms to the minimal UNIX set. In the following we will briefly describe the various IPC mechanisms and how they can be invoked. We will follow as much as possible the POSIX.4 standard, except where the facilities are not

implemented in Linux. In that case we will describe the mechanism Linux makes available.

6.1 UNIX and POSIX.1 Signals

The old signal facility of UNIX is rather limited, but it is available on every implementation of UNIX or one of its clones. Originally, signals were used to *kill* another process. Therefore, for historical reasons, the system call by which a process can send a signal to another process is called *kill()*. There is a set of signals, each identified by a number (they are defined in <*signal.h*>), and the complete system call for sending a signal to a process is:
`kill(pid_t pid, int signal);`
The integer *signal* is usually specified symbolically: *SIGINT, SIGALRM* or *SIGKILL*, etc., as defined in <*signal.h*>. *pid* is the process identification of the process to which the signal shall be sent. If this receiving process has not been set up to **intercept signals**, its *execution will simply be terminated by any signal sent to it.* The receiving process can however be set up to *intercept certain signals* and to perform certain actions upon reception of such an intercepted signal. Certain signals cannot be intercepted, they are just killers: SIGINT, SIGKILL are examples. In order *to intercept a signal*, the receiving process must have *set up a signal handler* and notified this to the operating system with the *sigaction() system call.* The following is an example of how this can be done:

A *structure sigaction* (not to be confounded with the system call of the same name!) is defined as follows:

```
struct sigaction {
    void (*sa_handler)();
    sigset_t sa_mask;
    int sa_flags;
    void(*sa_sigaction)(int,siginfo_t *,void *); };
```

This structure encapsulates the action to be taken on receipt of a signal.

The following is a program that shall exit gracefully when it receives the signal *SIGUSR1*. The function *terminate_normally()* is the **signal handler**. The administrative things are accomplished by defining the elements of the structure and then calling *sigaction()* to get the signal handler registered by the operating system.

```
void
```

```
terminate_normally(int signo)
{
    /* Exit gracefully */
    exit(0);
}

main(int argc, char **argv)
{
    struct sigaction sa;
    sa.sa_handler = terminate_normally;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGUSR1, &sa, NULL)) {
        perror("sigaction");
        exit(1);
    }
 ...
}
```

The operating system itself may generate signals, for instance as the result of machine exceptions: *floating point exception, page fault*, etc. Signals may also be generated by something which happens *asynchronously* with the process itself. The signals then *aim at interrupting the process*: I/O completion, timer expiration, receipt of a message on an empty message queue, or typing CTRL-C or CTRL-Z on the keyboard. Signals can also be sent from one user process to another.

The structure *sigaction* does not only contain the information needed to register the signal handler with the operating system (in the process descriptor), but it also contains information on what the receiving process should do when it receives the registered signal. It can do one of three things with the signal:
- it can block the signal for some time and later unblock it.
- it can ignore the signal, pretending that nothing has happened.
- it can handle the signal, by executing the signal handler.

The POSIX.1 signals, described so far, have some serious limitations:
- there is a lack of signals for use by a user application (there are only two: SIGUSR1 and SIGUSR2).
- signals are not queued. If a second signal is sent to a process before the first one could be handled, the first one is simply and irrevocably lost.
- signals do not carry any information, except for the number of the signal.

– and, last but not least, signals are sent and received asynchronously. This means in fact that a process may receive a signal at any time, for instance also when it is updating some sensitive data-structures. If the signal handler will also do something with these same data-structures, you may be in deep trouble. In other words, when you write your program, you must always keep in mind that you may receive a signal exactly at the point where your pencil is.

Linux is compliant with this POSIX.1 definition of signals.

6.2 POSIX.4 signals

From the description above, we have seen that the POSIX.1 signals are a rather complicated business (in UNIX jargon this is called flexibility). The POSIX.4 extensions to the signal mechanism introduces even more flexibility. POSIX.4 really defines an entirely *new set of signals*, which can peacefully co-exist with the old signals of POSIX.1. The historical name *kill()* is replaced by the more expressive *sigqueue()*.

The main improvements are:
– a far larger number of user-definable signals.
– signals can be queued; old untreated signals are therefore not lost.
– signals are delivered in a fixed order.
– the signal carries an additional integer, which can be used to transmit more information than just the signal number.

POSIX.4 signals can be sent *automatically* as a result of *timer expiration, arrival of a message on an empty queue*, or *by the completion of an asynchronous I/O operation.* Unfortunately, the POSIX.4 signals are not yet part of Linux, so we will not dwell any further on them.

6.3 pipes and FIFOs

Probably one of the oldest interprocess communication mechanisms is the **pipe**. Through a pipe, the *standard output* of a program is pumped into the *standard input* of another program. A pipe is usually set up by a shell, when the pipe symbol (|) is typed between the names of two commands. The data flowing through the pipe is lost when the two processes cease to exist. For a **named pipe**, or **FIFO** *(First In, First Out)*, the data remains stored in a file. The *named pipe* has a name in the filesystem and its data can therefore be accessed by any other process in the system, provided it has the necessary permissions.

A running process can set up a pipe to communicate with another process. The communication is uni-directional. If duplex communication is needed, two pipes must be set up: one for each direction of communication. The two "ends of a pipe" are nothing else than file descriptors: one process writes into one of these files, the other reads from the other.

Setting up a pipe between two processes is not a terribly straightforward operation. It starts off by making the *pipe()* system call. This creates *two file descriptors*, if the calling process still has file descriptors available. One of these descriptors (in fact the second one) concerns the end of the pipe where we will write, the other descriptor (the first one) is attached to the opposite end, where we will read from the pipe. If we now create another process, this newly created process will inherit these two file descriptors. We now must make sure that both parent and child processes can find the file descriptors for the pipe ends. The *dup2* system call will in fact do this, by duplicating the "abstract" file descriptors *pipe_ends[0]* and *pipe_ends[1]* into well-known ones. *dup2* copies a file descriptor into the first available one, so we should close first the files where we want the pipe to connect (usually *standard out* for the process connected to the writing end and *standard in* for the process which will read from the pipe). Here is a *skeleton* program for doing this in the case of a *terminal server*, which *forks off a terminal process to display messages from the server*:

```
/* First create a new client */
if (pipe(pipe_ends) < 0) {
    perror("pipe");
    exit(1);
}

global_child=child=fork();
if (child) {
    /*here for parent process*/
    do_something();
}
else {
    /*here for the child*/
    /* pipe ends will be 0 and 1 (stdin and stdout) */
    (void)close(0);
    (void)close(1);
    if (dup2(pipe_ends[0], 0) < 0)
        perror("dup2");
    if (dup2(pipe_ends[1], 1) < 0)
```

```
        perror("dup2");
    (void)close(pipe_ends[0]);
    (void)close(pipe_ends[1]);
    execlp(CHILD_PROCESS, CHILD_PROCESS, "/dev/com1", NULL);
    perror("execlp");
    exit(1);
}
```

The terminal process, created as the child could look:

```
#include <fcntl.h>

char buf[MAXBYTES]

/* pipe should not block, to avoid waiting for input */
if(fcntl(channel_from_server, F_SETFL, O_NONBLOCK) < 0){
    perror("fcntl");
    exit(2);
}
while (1) {
    /* Put messages on the screen */
    /* check for input from the server */
    nbytes = read(channel_from_server, buf, MAXBYTES);
    if (nbytes < 0) && (errno != EAGAIN))
        perror("read");
    else if (nbytes > 0) {
        printf("Message from the Server:  \"%s\"\n", buf);
}
...
```

In this example[9], the server process simply writes to the write end of the pipe (which has become stdout) and the child reads from the other end, which has been transformed by *dup2* into stdin. To set up a communication channel in the other direction as well, the whole process must be repeated, inverting the roles of the server and the terminal client (the first becomes the reader, and the second the writer) and using two other file descriptors (for instance 3 and 4 if they are still free). Note that the *dup* calls must be made

[9]Which was also taken from Gallmeister's book.

before the child does its *exec* call, otherwise, the file descriptors for the two pipe ends would be lost.

The use of named pipes is simpler: the *FIFO* exists in the file system and any process wanting to access the file can just open it. One process should open the *FIFO* for reading, the other for writing. A FIFO is created with the POSIX.1 *mkfifo()* system call.

6.4 Message Queues

When we have compiled the *System V IPC facilities* into the Linux kernel, we have **message queues** available, which however do not conform to the POSIX.4 standard. We will nevertheless describe them briefly, as they are the only ones we have at present.

In system V the **message resource** is described by a *struct msqid_ds*, which is allocated and initialized when the resource is created. It contains the permissions, a pointer to the last and the first message in the queue, the number of messages in the queue, who last sent and who last received a message, etc. The messages itself are contained in:

```
struct  msgbuf {
        long mtype;
        char mtext[1]; }
```

To set up a message queue, the creator process executes a *msgget* system call:

```
msqid = msgget(key_t key, int msgflg);
```

The *key* is a unique identification of the particular message queue which ensures that messages are delivered to the correct destination. A key can be fabricated with the *ftok(filename,character)* library call; *IPC_PRIVATE* is frequently used as the key. The use of *IPC_PRIVATE* will create a new resource if it does not already exist. The processes wanting to receive messages on this queue must also perform a *msgget* call, in order to obtain the *msqid*. A message is sent by executing:

```
int msgsnd(int msqid,struct msgbuf *msgp,int msgsz, \
int msgflg);
```

and similarly a message is received by:

```
int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, \
long msgtyp, int msgflg);
```

msgtyp is used as follows:

if msgtyp $= 0$: get first message on the queue,
> 0 : get first message of matching type,
< 0 : get message with smallest type which is $\leq$abs(msgtyp).

Finally, the *msgctl* calls allow you to get the status of a queue, modify its size, or destroy the queue entirely.

The message queue can be empty. If a message is sent to an empty queue, the process reading messages from the queue is woken up. Similarly, when the queue is full, a writer trying to send a message will be blocked. As soon as a message is read from the queue, creating space, the writer process is woken up.

6.5 Counting Semaphores

System V **semaphore arrays** are an **oddity**. The *semget* call allocates an **array of counting semaphores**. Presumably, and hopefully, the array may be of length 1. You also specify operations to be performed on a series of members of the array. The operations are only performed if they will **all succeed!**

Counting semaphores can be useful in **producer-consumer problems**, where the producer puts items in a buffer and the consumer takes items away. Two counting semaphores keep track of the number of items in the buffer and allow to "gracefully" handle the *buffer empty* and *buffer full* situations.

Producer-consumer situations can easily arise in a real-time application: the *producer collects data from measuring devices*, the *consumer writes the data to a storage device* (disk or tape).

Another example is a large paying car park: There is one *counting semaphore* which is initialized to the total number of places in the car park. A *separate process is associated* with *each entrance or exit gate.* The process at an entrance gate will **do a wait on the semaphore, e.g. decrement it.** If the result is greater than zero, the process will continue, issue a ticket with the time of entrance, and open the gate. It closes the gate as soon as it has detected the passage of the car. If the *value of the counting semaphore* is zero when the decrement operation is tried, the process is **blocked** and added to the pile of blocked processes. This is just what is needed: the car park is full and the car will have to wait, so no ticket is issued, etc.

The processes at the exit gates do the contrary: after having checked the ticket, they open the gate and then do a *post or increment* operation on the semaphore, effectively indicating that one more place has become free. This operation will always succeed.

The *System V counting semaphore mechanism* is rather similar to the message queue business: You create a semaphore (array) as follows:

```
int semid = semget(key_t key, int nsems, int semflg);
```

The key IPC_PRIVATE behaves as before. All processes wanting to use the semaphore must execute this *semget* call. You can then operate on the

semaphore:
`int semop(int semid, struct sembuf *sops, unsigned nsops);`
(here is the oddity, you do nsops operations on nsops members of the array; the operations are specified in an array of *struct sembuf*). This structure is defined as:

```
struct sembuf
       ushort sem_num;  /*index in array*/
       short sem_op;    /*operation*/
       short sem_flg    /*operation flags*/
```

Two kinds of operations can result in the process getting blocked:
i) If *sem_op* is 0 and *semval* is non-zero, the process *sleeps* on a queue, waiting for *semval* to become zero, or returns with error *EAGAIN* if either of *(IPC_NOWAIT | sem_flg)* are true.
ii) If *(sem_op < 0)* and *(semval + sem_op < 0)*, the process either sleeps on a queue waiting for *semval* to increase, or returns with error *EAGAIN* if *(sem_flg & IPC_NOWAIT)* is true.

Atomicity of the semaphore operations is guaranteed, because the mechanism is embedded in the kernel. The kernel will not allow two processes to simultaneously use the kernel services. In other words, a system call will be entirely finished before a context switch takes place.

Note: If you want to use a semaphore which takes only the values 0 or 1 (for instance for mutual exclusion), you are better off by using the atomic bit operations, defined in <asm-i386/bitops.h>: *test_bit, set_bit and clear_bit.*

6.6 Shared Memory

Shared Memory is exactly what its name says: two or more processes **access the same area of physical memory**. This segment of physical memory is **mapped into** two or more **virtual memory spaces**.

Shared Memory is considered a low-level facility, because the shared segment **does not benefit from the protection** the operating system normally provides. To compensate for this disadvantage, **shared memory is the fastest IPC mechanism**. The processes can read and write shared memory, without *any system call being necessary.* The *user himself must provide the necessary protection*, to avoid that two processes "simultaneously" access the shared memory. This can be obtained with a **binary semaphore** or **mutex**.

A *mutex* can be simulated by performing the *set_bit(int nr, void * addr)* call, which sets the desired bit *nr* and returns the old value of the bit. The

short integer on which this operation is performed must also reside in shared memory, in order to be accessible by both processes.

At present the only shared memory facility available in Linux comes from System V, and is therefore **not** conforming to POSIX.4. The related system calls are similar to the System V calls we have already seen:

There is, of course, a *shared memory descriptor*, `struct shmid_ds`.
Shared memory is allocated with the system call:

```
int shmget(key_t key, int size, int shmflg);
```

The *size* is in bytes and should preferably correspond to a multiple of the page size (4096 bytes). All processes wanting to make use of the shared memory segment must make a *shmget* call, with the same key.
Once the memory has been allocated, you map it into the virtual memory space of your process with:

```
char *virt_addr;
virt_addr = shmat(int shmid, char *shmaddr, int shmflg);
```

shmaddr is the requested attach address:
if it is 0, the system finds an unmapped region;
if it is non-zero, then the value must be *page-aligned.*
By setting *shmflg* = *SHM_RDONLY* you can request to attach the segment *read-only.*

You can get rid of a shared memory segment by:

```
int shmdt(char *virt_addr);
```

Finally, there is again the *shmctl* call, which you may use to get the status, or also to destroy the segment (a shared segment will only be destroyed after all users have *detached* themselves).

If you are using shared memory, and you need *malloc* as well, you should `malloc` a large chunk of memory first, before you attach the shared memory segment. Otherwise *malloc* may interfere with the shared memory.

A word about the Linux implementation of the System V IPC mechanisms is in order. All System V system calls described above make use of a single Linux system call: ipc(). A library of the system V IPC calls is available, which maps each call and its parameters into the Linux ipc() call. An example is:

```
int semget (key_t key, int nsems, int semflg)
{
    return ipc (SEMGET, key, nsems, semflg, NULL);
}
```

The constants are defined in <linux/ipc.h>

7 Scheduling

The scheduling algorithm of Linux aims at giving a **fair share of the resources** to each user. It is therefore a typical **time-sharing scheduler**. A time-sharing scheduler is based on priorities, like any other type of scheduler, but the system keeps changing the priorities to attain its aim of being fair to everyone.

For *time-critical real-time applications* you want another sort of scheduler. You need a **high priority for the most critical real-time processes**, and a *scheduler* which will run such a high priority process whenever **no process with higher priority is runnable**[10].

Less critical processes of the real-time application can run at *lower priorities* and other user jobs could also be fitted in at priorities below.

SVR6 (System V, Release 6) has a scheduler that does both time-sharing and real-time scheduling, depending on the priority assigned to a process. Critical processes run at priorities between, say, 0 and 50, and benefit from the priority scheduling. Other jobs run at lower priorities and have to accept the time-sharing scheduler. This aspect of System V has not (yet) been ported to Linux.

A POSIX.4 compliant scheduler **has** been ported to Linux. In order to make use of it, you must make *patches to the kernel code* and recompile the kernel together with this POSIX.4 scheduler. At the time these notes were prepared, we had not yet had a chance to try it.

The advantage of a POSIX.4 scheduler is, of course, that your application program will be portable between different platforms.

What does a POSIX.4 scheduler do? Here is what it provides[11]:

```
#include          <unistd.h>
#ifdef _POSIX_PRIORITY_SCHEDULING
#include          <sched.h>
int i, policy;
struct sched_param *scheduling_parameters;
pid_t pid;

sched_setscheduler(pid_t pid, int policy, \
    struct sched_param *scheduling_parameters);
int sched_getscheduler(pid_t pid);
int sched_getparam(pid_t pid, \
    struct sched_param *scheduling_parameters);
```

[10] Remember that you need a sleeping shell at a still higher priority.

[11] Again from Gallmeister's book.

```
int sched_setparam(pid_t pid, \
    struct sched_param *scheduling_parameters);
int sched_yield(void);
int sched_get_priority_min(int);
int sched_get_priority_max(int);
#endif _POSIX_PRIORITY_SCHEDULING
```

You see that you define a *scheduling "policy"*. You have a choice:

SCHED_FIFO: pre-emptive, priority-based scheduling,
SCHED_RR: pre-emptive, priority-based with *time quanta*,
SCHED_OTHER: implementation dependent scheduler.

With the first choice, the process *will run* until it gets blocked for one reason or another, or *until a higher priority process becomes runnable.* The second policy adds a **time quantum**: a process running under this scheduling policy will only run for a certain duration of time. Then it goes back to the end of the queue for its priority (each priority level has its own queue). Thus, at a given priority level, all processes in that level are scheduled **round-robin**. In future, **deadline scheduling** will probably have to be added as another choice.

There is a range of priorities for the *FIFO* scheduler and another range for the *RR* scheduler.

After a *fork()*, the child process *inherits the scheduling policy and the priority* of the parent process. If the priority of the child then gets increased above the priority of the running process, the latter is *immediately pre-empted*, even before the return from the *sched_setparam* call! So be careful, you may seriously harm yourself.

On the other hand, you may *"yield"* the processor to another process. You cannot really be sure which process this is going to be. As a matter of fact, the only thing *yield* does, is to put your process at the end of the queue at your particular priority level.

All this is nice, but we are still *stuck with the fact that the kernel itself cannot be pre-empted.* This is usually not too much of a problem. Most of the system calls will take only a short time to execute.

Usually, the system calls that may take a considerable time (such as certain I/O related calls), should be relegated — as far as possible — to those tasks that run at a lower priority level. Also some common sense will help: it is much faster to write once 512 bytes to disk than to write 512 times a single byte!

Other system calls do take a long time. *fork* and *exec* for example. You should therefore **create** all necessary processes during the **initialization**

phase of your application. Let the processes that you only need sporadically just *sleep* for most of the day.

8 Timers

You may want to arrange for certain things to happen at **certain times**, or a given **time interval after** something else happened. So you will nearly always have the need for a **timer** and/or an **interval timer.**

Standard UNIX (and Linux) has a **real-time clock**. It counts the number of seconds since 00:00 a.m. January 1, 1970. (called the *Epoch*). You get its value with the *time()* function:

```
#include <time.h>
time_t time(time_t *the_time_now);
```

You can also call *time* with a NULL pointer.

Linux also has the *gettimeofday* call, which stores the time in a structure:

```
struct timeval {
    time_t tv_sec   /* seconds */
    time_t tv_usec }    /* microseconds */
```

`gettimeofday` returns a 0 or -1 (success, failure respectively).

You can make things happen **after a certain time interval** with *sleep*:

```
unsigned int sleep(unsigned int n_seconds);
```

The process which executes this call will be stopped and resumed *after n_seconds* have passed. The resolution is very crude! As a matter of fact, many real-time systems would need a resolution of milliseconds and, in extreme cases, even microseconds.

To overcome this drawback, Linux has also **interval timers.** each process has three of them:

```
#include      <sys/time.h>

int setitimer(int which_timer, \
    const struct itimerval *new_itimer_value, \
    struct itimerval *old_itimer_value);
int getitimer(int which_timer, \
    struct itimerval *current_itimer_value);
```

The first argument, *which_timer*, has one of three values: *ITIMER_REAL*, *ITIMER_VIRTUAL* and *ITIMER_PROF*. *setitimer()* sets a new value of the

interval timer and returns the old value in *old_timer_value*. When a timer **expires, it delivers a signal**: *SIGALRM, SIGVTALRM* and *SIGPROF* respectively. The calls make use of a structure:

```
struct itimerval {
    struct timeval it_val     /* initial value */
    struct timeval it_interval }     /* interval */
```

The *ITIMER_REAL* measures the time on the "wall clock" and therefore includes the time used by other processes. *ITIMER_VIRTUAL* measures the time spent in the user process which set up the timer, whereas *ITIMER_PROF* counts the time spent in the user process **and** in the kernel on behalf of the user process. It is thus very useful for *profiling*.

The resolution of these interval timers is given by the constant *HZ*, defined in *<sys/param.h>*. On Linux machines, *HZ=100*, so the resolution of the interval timers is 10000 microseconds.

POSIX.4 extends the timer facilities to a number of implementation defined clocks, which may have different characteristics. Timers and intervals can be specified in nanoseconds.

9 Memory Locking

As we already pointed out before, the real-time processes – at least the critical ones – should be **locked into memory**. Otherwise you could have the very unfortunate situation that your essential task has been swapped out, just before it becomes runnable again. **Faulting** a number of pages of code back into memory may add an intolerable overhead.

Remember also that *infrequently used pages may be swapped out* by the system, without any warning. Faulting them back in again may make you miss a **deadline**. Thus, not only the *program code*, but also the *data and stack pages* should be locked into memory.

A POSIX.4 conformant memory locking mechanism is available for Linux. Unfortunately, we have not yet been able to test it. It does the following:

```
#include        <unistd.h>
#ifdef _POSIX_MEMLOCK
#include        <sys/mman.h>
int mlockall(int flags);
int munlockall(void);
#endif /* _POSIX_MEMLOCK */
```

mlockall will lock **all** your memory, e.g. *program, data, heap, stack and also shared libraries.* You may choose, by specifying the flags, to lock the space you occupy at present, but also what you will occupy in future.

Instead of locking everything, you may also lock parts:

```
#include        <unistd.h>
#ifdef _POSIX_MEMLOCK_RANGE
#include        <sys/mman.h>

int mlock(void *address, size_t length);
int munlock(void *address, size_t length);
#endif /* _POSIX_MEMLOCK_RANGE */
```

Finally, you may want to lock just a few essential functions: a signal handler or an interrupt handler, for instance. You should not do this from within the interrupt handler, but from a separate function:

```
void intr_handler()
{
    ...  /* do your work here */
}
void right_after_intr_handler()
{
        /* this function serves to get an address */
        /* associated with the end of intr_handler() */
}

void intr_handler_init()
{
    ...
    i = mlock(ROUND_DOWNTO_PAGE(intr_handler),\
      ROUND_UPTO_PAGE(right_after_intr_handler - \
      intr_handler));
}
```

The function *right_after_intr_handler()* does nothing. It serves only to get an address associated with the *end* of the interrupt handler. This is needed to calculate *length* argument for the *mlock()* call.

10 Multiple User Threads

All we have seen so far happened at the *process* level and *kernel intervention* was needed for every coordinating action between processes. The overall picture has become quite complicated and a programmer must master many details or else he runs into trouble.

Is there not another solution, where the user has more direct control over what is going on? Fortunately, there is: **multiple user threads**. POSIX.4a (or POSIX.1c if you prefer) standardizes the **API (Application Programmer's Interface)** for multiple threads.

Threads are independent flows of control *inside a single process*. Each *thread* has its *own thread structure* –comparable to a *process descriptor*–, its *own stack* and its *own program counter*. All the rest, i.e. *program code*, *heap storage* and *global data*, is **shared** between the threads. Two or more threads may well execute the same function simultaneously. The services needed to *create threads, schedule their execution, communicate and synchronize between threads* are provided by the **threads library** and *run in user space*. For the kernel exists only the *process*; what happens inside this process is invisible to the kernel.

Lightweight Processes, as in Solaris or SunOS 4.x, are somewhere midway: a small part of the process structure has been split off and can be replicated for several LWPs, all continuing to be part of the same process, using the same memory map, file descriptors, etc. The split-off part is still a kernel structure, but the kernel can now make rapid context switches between LWPs, because only a small part of the complete process structure is affected. Inside a LWP, multiple threads may be present.

Multiple threads offer a solution to programming which has a number of advantages. The model is particularly well suited to *Shared-memory Multiple Processors*, where the code, common to all threads, is executed on different processors, one or more threads per processor. Also for real-time applications on uniprocessors, threads have advantages. In the first place, the *fastest, easiest intertask communication mechanism, – shared memory –* is there for *free*!

There are other advantages as well. The **responsiveness** of the process may increase, because when one thread is *blocked*, waiting for an event, the other can continue execution. The fact that threads offer a sort of "do-it-yourself" solution makes the user have a better grasp of what he is doing and thus he can produce better structured programs. *Communication* and *synchronization* between threads is easier, more transparent and faster than between processes. Each thread conserves its ability to communicate with another process, but it is wise to concentrate all *inter-process communication*

within a single thread.

Multiple threads will in general lead to performance improvements on shared memory multiprocessors, but on a uniprocessor one should not expect miracles. Nevertheless, the fact that there is **less overhead** and that some threads may **block while others continue**, will be felt in the **performance**.

It sounds as if we just discovered a gold mine. Well ..., there are a few things which obscure the picture somewhat. For threads to be usable with no danger, the **library functions** our program uses must be **threads-safe**. That is, they must be *re-entrant*. Unfortunately, most libraries contain functions which modify global variables and therefore are **not** re-entrant. For the same reason, your threaded program must be re-entrant, so it has to be compiled with *_REENTRANT* defined. In addition, for a real-time application, you still need at least a few facilities from the operating system: *memory locking and real-time priority scheduling*[12].

Threads can be implemented as a *library of user functions*. One standard set is defined in POSIX.4a, but other implementations also exist. The package we are using implements the **POSIX.4a pthreads**. There are some 50 service requests defined. They are briefly described in Annex III and in more detail in the man pages. We will illustrate only a few of them, the most important ones.

pthreads defines functions for *Thread Management, Mutexes, Condition Variables, Signal Management, Thread Specific Data* and *Thread Cancellation*. Threads, mutexes and condition variables have *attributes*, which can be modified and which will change their behaviour. Not all options defined by the various attributes need to be implemented. <*pthread.h*> defines eight types:

Type	Description
`pthread_attr_t`	`Thread attribute`
`pthread_mutexattr_t`	`Mutex attribute`
`pthread_condattr_t`	`Condition variable attribute`
`pthread_mutex_t`	`Mutual exclusion lock (mutex)`
`pthread_cond_t`	`Condition variable`
`pthread_t`	`Thread ID`
`pthread_once_t`	`Once-only execution`
`pthread_key_t`	`Thread specific data key`

[12] Alternatively, you run on a dedicated machine, where you have killed all daemons, so that your application is the only active process in the system.

Attributes can be *set or retrieved* with calls of the following type:

```
int pthread_attr_setschedpolicy( pthread_attr_t *attr, \
int newvalue);
```

or:

```
int pthread_mutexattr_getprotocol( pthread_mutexattr_t *attr, \
        *protocol);
```

See Annex III for the complete list. The *scheduling policy* can be one of: `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`, as for the POSIX.4 standard. The scheduling parameters can also be set and retrieved.

When the process is forked, *main(argc, argv)* is entered. In the main program you may then create threads. Each thread is a function, or a sequence of functions. At thread creation, the *entry point* must be specified:

```
int pthread_create( pthread_t *thread, \
          const pthread_attr_t *attr, void *(*entry)(void *), \
          void *arg );
```

```
void pthread_exit( void *status );
```

does what is expected from it. It should be noted that *NULL* may often be used to substitute an argument in the function call. This is notably the case for `pthread_attr_t *attr` and `void *status` above.

An important function is:

```
int pthread_join( pthread_t thread, void **status );
```

When this primitive is called by the running thread, its execution will be suspended until the target `thread` terminates. If it has already terminated, execution of the calling thread continues. *pthread_join()* is therefore an important mechanism for synchronizing between threads. So-called *detached threads* cannot be joined. You specify at creation time or at run time if the thread has to be detached or not.

Mutexes can have as the *pshared* attribute `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`, meaning that the mutex can be accessed also by other processes or that it is private to our process. Private mutexes are defined in all implementations, shared mutexes are an option. The two usual operations on a mutex are:

```
int pthread_mutex_lock( pthread_mutex_t *mutex );
```

and

```
int pthread_mutex_unlock( pthread_mutex_t *mutex );
```

but you can also try if a mutex is locked and continue execution, whatever the result:

```
int pthread_mutex_trylock( pthread_mutex_t *mutex );
```

All memory occupied by the process is shared among the various threads, which we said was an important advantage of threads. Nevertheless, sometimes a thread needs to protect its data against attacks from other threads. For this reason a few primitives which allow to create and manipulate *thread specific data* are defined. For details see the man pages.

We have not yet met **condition variables**, which are another feature of pthreads. Condition variables are always associated with a *mutex* and they are useful if you want to make things depend on complex situations. Maybe you want to execute a piece of code in a thread only if the temperature is below some limit, it is raining outside and the number of your ticket of the lottery is odd. If you can express your condition as a piece of program, you can use it as a condition variable (CV). It works as follows:

Thread 1	**Thread 2**
lock the mutex	
test the condition	
FALSE! *unlock mutex*	
sleep on CV	
	lock the mutex
	change the condition
	signal thread 1
	unlock mutex
lock mutex	
test condition again	
TRUE! *do the job*	
unlock mutex	

Translated into code, this becomes:

Thread 1	**Thread 2**
`pthread_mutex_lock(&m);`	
`while (!my_condition)`	
`while (pthread_cond_wait(&c, &m) != 0) ;`	
	`pthread_mutex_lock(&m);`
	`my_condition = TRUE ;`
	`pthread_cond_signal(&c);`

```
                                        pthread_mutex_unlock(&m);
do_thing();
pthread_mutex_unlock(&m);
```

Note that *pthread_cond_wait()* will free the mutex *for you* and your thread will go to sleep on the condition variable.

pthreads is really a subject in itself and our quick review has been very superficial. Threads are well suited for implementing **Server-Client problems**. Due to the shared memory, the communication between the *server* and the –possibly many– *clients* is easy. We will see an example of a *Server-Client* problem in one of the coming afternoons.

A brief resume of the POSIX.4a definitions is given in Annex III. For more details, the reader is referred to the "man pages".

We close this section with a complete code example. In the example a *reader thread* reads characters from standard input and puts them into one of two buffers, a *writer thread* reads from a buffer and sends the characters to standard out. The reader or the writer want to use a buffer which must be in a given state (DIRTY or CLEAN). If the buffer of interest is not in the desired state, the thread will block. The reader runs first. This example illustrates several aspects of multiple threads, including the use of mutexes and condition variables. The reader is again invited to study this example carefully.

```
/*
*  DBCP - Double-Buffer Copy
*
*  Copyright (C) 1995 by Sun Microsystems, Inc.
*  All rights reserved.
*
*  This file is a product of SunSoft, Inc. and is provided for
*  unrestricted use provided that this legend is included on all
*  media and as a part of the software program in whole or part.
*  Users may copy, modify or distribute this file at will.
*
*  THIS FILE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND
*  INCLUDING THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND
*  FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE
*  OF DEALING, USAGE OR TRADE PRACTICE.
*
*  This file is provided with no support and without any
*  obligation on the part of SunSoft, Inc. to assist in its use,
```

```
*  correction, modification or enhancement.
*
*  SUNSOFT AND SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY
*  WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS
*  OR ANY PATENTS BY THIS FILE OR ANY PART THEREOF.
*
*  IN NO EVENT WILL SUNSOFT OR SUN MICROSYSTEMS, INC. BE LIABLE
*  FOR ANY LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT
*  AND CONSEQUENTIAL DAMAGES, EVEN IF THEY HAVE BEEN ADVISED OF
*  THE POSSIBILITY OF SUCH DAMAGES.
*
*  SunSoft, Inc.
*  2550 Garcia Avenue
*  Mountain View, California  94043
*/

/* Include Files        */

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

/* Constants and Macros        */

#define    NBUF        2

#define    ST_DIRTY    0x01
#define    ST_CLEAN    0x02
#define    ST_INUSE    0x04

/* Data Declarations        */

typedef struct {
    char        buffer[BUFSIZ];
    int         state;
    int         nbytes;
    pthread_mutex_t    lk;
    pthread_cond_t     cv;
} buffer_t;
```

```
/* External Declarations    */

buffer_t    buf[NBUF];

/* External References        */

extern    void        *reader( void * );
extern    void        *writer( void * );
extern    buffer_t    *alloc_buffer( int, int );
extern    void        enable_buffer( buffer_t *, int );
extern    int         read_buffer( int, buffer_t * );
extern    int         write_buffer( int, buffer_t * );

/* Main                */

main( int argc, char *argv[] ) {
    int               i;
    int               n;
    pthread_attr_t    attr;
    pthread_t         tid;

    for ( i=0; i < NBUF; ++i ) {
        buf[i].state    = ST_CLEAN;
        pthread_mutex_init( &buf[i].lk, NULL );
        pthread_cond_init( &buf[i].cv, NULL );
        buf[i].nbytes    = 0;
    }

    pthread_attr_init( &attr );
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);

    if(n=pthread_create(&tid,&attr,reader,(void *)STDIN_FILENO)){
      fprintf(stderr,"pthread_create:reader: %s\n",strerror(n));
      exit( 1 );
    }

    if(n=pthread_create(&tid,&attr,writer,(void *)STDOUT_FILENO)){
      fprintf(stderr,"pthread_create:writer: %s\n",strerror(n));
      exit( 1 );
    }
```

```
    pthread_attr_destroy( &attr );
    pthread_exit( 0 );
/* NOTREACHED */
    return( 0 );
}

/* Reader - reader thread        */

    void *
reader( void *arg ) {
    int    fd      = (int) arg;
    int    eof     = 0;
    int    iter    = 0;

    while ( !eof ) {
        buffer_t    *bp    = alloc_buffer( iter++, ST_CLEAN );

        eof    = read_buffer( fd, bp );
        enable_buffer( bp, ST_DIRTY );
    }

    return( 0 );
}

/* Writer - writer thread        */

    void *
writer( void *arg ) {
    int    fd      = (int) arg;
    int    eof;
    int    iter    = 0;

    do {
        buffer_t    *bp    = alloc_buffer( iter++, ST_DIRTY );

        eof    = write_buffer( fd, bp );
        enable_buffer( bp, ST_CLEAN );
    } while ( !eof );

    return( 0 );
```

```
}

/* alloc_buffer - allocate the buffer for iteration 'iter'    */
/* when in 'state'.    */

    buffer_t *
alloc_buffer( int iter, int state ) {
    buffer_t    *bp    = &buf[iter % NBUF];

    pthread_mutex_lock( &bp->lk );

    while ( bp->state != state )
        pthread_cond_wait( &bp->cv, &bp->lk );

    bp->state    = ST_INUSE;

    pthread_mutex_unlock( &bp->lk );

    return( bp );
}

/* enable_buffer - change 'state' of buffer 'bp'.        */

    void
enable_buffer( buffer_t *bp, int state ) {
    pthread_mutex_lock( &bp->lk );
    bp->state    = state;
    pthread_cond_signal( &bp->cv );
    pthread_mutex_unlock( &bp->lk );

    return;
}

/* read_buffer - read into buffer 'bp'.                    */

    int
read_buffer( int fd, buffer_t *bp ) {
    if((bp->nbytes=read(fd,bp->buffer,BUFSIZ))==-1) {
        fprintf(stderr,"read_buffer: %s\n",strerror(errno));
        exit( 1 );
    }
```

```
    return( bp->nbytes == 0 );
}

    int
write_buffer( int fd, buffer_t *bp ) {
    if ( bp->nbytes )
        if(write(fd,bp->buffer,bp->nbytes )!=bp->nbytes) {
          fprintf(stderr,"write_buffer: %s\n",strerror(errno));
          exit( 1 );
        }

    return( bp->nbytes == 0 );
}
```

11 Conclusion

We have tried in this course to give a brief overview of the requirements of a real-time application and we have investigated to what extent Linux can do the job. We have also mentioned the improvements to Linux which have already been made. We are confident that more will come. To our knowledge there is however no concerted effort to develop a real-time version of Linux, so –at least for the time being– , all improvements have to be added to the kernel individually.

The **pthreads** package does contain the major part of the improvements a user would like to see. When a real-time application has been written using *pthreads*, the only essential features the operating system has still to provide are *memory locking* and *real-time scheduling*.

The important thing to remember is that you should analyze your problem very carefully, before deciding that you can (or cannot) use such or such an operating system. It is not very likely that you will need a large array of semaphores, but if you do, this course has shown you where to find them.

All depends therefore on your application. If you expect **high data rates** or **high interrupt rates** or if you are otherwise pressed for time constraints, or if you must meet stringent deadlines, then you will need many of the mechanisms described and you may have to accept acquiring a true real-time operating system.

This can be the case in physics experiments, in particular in Particle Physics and in Nuclear Physics.

There will however be situations where you don't need the heavy guns and where the standard Linux system will do the job. To give you an idea: Ulrich Raich runs a real-time application on a 66 MHz 486 machine, concurrently with X11. The machine sustains a rate of 200 external interrupts per second, in addition to the 100 Hz clock interrupts. It obviously all depends on what has to be done as the result of an interrupt.

With prices of PCs and PC-boards going down, there is now a tendency to use a PC-board also for an **embedded system**, where before you would have used a small, dedicated microprocessor. Using a PC-board has the obvious advantage of **portability**: you can develop your application on a large configuration, and then **download** it to the embedded system.

The reader may be interested to learn that there is a Linux Development Project to port a cut-down version of Linux to the Intel 8086 processor[13].

Many people may be just interested in hooking up existing instruments, for instance those which are equipped with an interface to the GPIB bus. This situation arises routinely in chemistry labs, or medical analysis labs, etc.

There is good news for those people as well: There is a **project to develop for Linux a complete package for controlling instruments with GPIB and Camac**. The first parts of it have been released already a year ago. It has graphical interfaces, uses X11 and is extensible[14]. And **it is free!** This project will certainly deliver the ideal solution for laboratories using standard equipment. No need to spend a lot of money on LabView or other products.

To end, I wish you a happy time programming your real-time applications. Enjoy!

[13]The project is called ELKS (Embeddable Linux Kernel Subset) and is led by Alan Cox. The project can use more volunteers. For more information, consult the Linux Documentation Project homepage (http://sunsite.unc.edu/mdw/linux.html), where you will find ELKS under the link "projects".)

[14]You can ftp this package from chemie.fu-berlin.de.

12 Annex I – Annotated bibliography

The last year has seen a real explosion of the number of books on **Linux**. A number of them are nothing but collections of **HOW-TOs** from the **Linux Documentation Project**. Others are specific for certain **Linux Distributions**, e.g. *Slackware, RedHat, Caldera Desktop, Yggdrasil Plug and Play Linux*. These books contain one or more *CD-ROMs*, or the CD-ROM set is sold separately (the *Linux Developers Resource* from *InfoMagic* is an example).

Below is an annotated bibliography of the books I found most useful and which is limited to those publications which are **not** specific to a distribution, or just collections of HOW-TOs.

1. Matt Welsh and Lar Kaufman, *Running Linux*, Sebastopol, CA95472, 1995, O'Reilly & Associates, Inc; ISBN 1-56592-100-3
   An excellent book, very complete and very readable. Contains extensive indications on how to obtain and install Linux, followed by chapters on UNIX commands, System Administration, Power Tools (including X11, emacs and LaTeX), Programming, Networking. The annexes contain a wealth of information on documentation, ftp-sites, etc. One of the most readable books on Linux.

2. Marc Ewing, *Running Linux Installation Guide and Companion CD-ROM*, O'Reilly & Associates, Inc.; no apparent ISBN.

3. Matt Welsh, *Linux Installation Guide*, 1995, Pacific Hi-Tech, 3855 South 500 West Suite M, Salt Lake City, Utah 84115,
   email: orders@pht.com; No ISBN found.
   The book is thin (221 pages) and cheap ($ 12.95). It contains a few extra chapters on XFree86, TCP/IP, UUCP, e-mail and usenet.

4. Olaf Kirch, *Linux Network Administrator's Guide*, Sebastopol CA95472, 1995, O'Reilly & Associates, Inc; ISBN 1-56592-087-2
   Another excellent book on Networking for Linux. Covers not only local networks and TCP/IP, but also the use of a serial line to connect to Internet, and other chapters on NFS, Network Information System, UUCP, e-mail and News Readers. Essential reading if you want to use your Linux box on the network.

5. Stefan Strobel and Thomas Uhl, *Linux, unleashing the workstation in your PC*, Berlin, 1994, Springer Verlag; ISBN 3-540-58077-8
   This book is good to whet the appetite of someone who has no idea of

what Linux is or what it can do. It has many illustrations, in particular of graphics applications and it mentions many software packages which are not part of the usual Linux distributions, together with indications on how to obtain and install the package. It mentions on the cover: ***Friends don't let friends use DOS***.

6. *Linux Bible*, 1994, San Jose, Yggdrasil Computing. No apparent ISBN. I know about this book only from the advertisements.

7. Kamram Hussain, Timothy Parker et al., *Linux Unleashed*, 1996, SAMS Publishing, ISBN 0-672-30908-4
   Approx 1100 pages of text, covering Linux and many tools and applications: Editing and typesetting (groff and Tex), Graphical User Interfaces, Linux for programmers (C, C++, Perl, Tcl/Tk, Other languages, Motif, XView, Smalltalk, Mathematics, Database products), System Administration, Setting up an Internet site and Advanced Programming topics. The book contains a CD-ROM with the Slackware distribution.

8. Randolph Bentson, *Inside Linux, a look at Operating System Development*, 1996, Seattle, Specialized system Consultants, Inc; ISBN 0-916151-89-1.
   This book provides some more insight into the internal workings of operating systems, with the emphasis being placed on Linux. It is written in general terms and does not contain code examples.

9. John Purcell (ed.), *Linux MAN, the essential manpages for Linux*, 1995, Chesterfield MI 48047, Linux Systems Lab, ISBN 1-885329-07-5.
   Indispensable for those who cannot stare at a screen for more than 8 hours a day, or who like to sit down in a corner to write their programs with pencil and paper, but want to be sure they use system calls correctly. As the title says, 1200 pages of "man pages" for Linux, from *abort* to *zmore*, and including system calls, library functions, special files, file formats, games, system administration and a kernel reference guide.

10. M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner, *Linux Kernel Internals*, 1996, Addison Wesley, ISBN 0-201-87741-4.
    For the real sports! A translation of a german book, revealing all the internals of the Linux kernel, including code examples, definitions of structures, tables, etc. The book contains a CD-ROM with Slackware and kernel sources. Indispensable if you want to make modifications to the kernel yourself.

The following books concern **real-time** and **POSIX.4 and 4a**:

i) Bill O. Gallmeister, *POSIX.4: Programming for the Real World*, 1995, O'Reilly & Associates, Inc.; ISBN 1-56592-074-0.
This book gives an in-depth treatment of programming real-time applications, based on the POSIX.4 standard. Several of the examples in the present course were taken from this book. In addition to approximately 250 pages of text, the book contains 200 pages of "man pages" and solutions to exercises.

ii) Bil Lewis, Daniel J. Berg, *Threads Primer, A Guide to Multithreaded Programming*, 1996, Sunsoft Press (Prentice Hall);
ISBN 0-13-443698-9.
An introduction to threads programming, mainly based on the Solaris implementation of threads, but containing comparisons to POSIX threads and a full definition of the *Applications Programmer's Interface* to **POSIX.4a pthreads**.

iii) S. Kleiman, Devang Shah, B. Smaalders, *Programming with Threads*, 1996, Sunsoft Press (Prentice Hall;
ISBN 0-13-172389-8.
At the time of writing these notes, this book had just been published and I had not yet seen it. It should contain a more in-depth treatment of threads programming than the previous title. It should also be more pthreads-oriented.

iv) Andrew S. Tanenbaum, *Modern Operating Systems*, 1992, Prentice Hall; ISBN 0-13-595752-4.
This excellent book is not specifically tuned to real-time, but it provides a comprehensive introduction to the features of modern operating systems and their implementation. An older edition of the book contained a complete listing of the **minix** operating system. The reader may appreciate that Linux was born when Linus Torvalds set out to improve minix ...

Note that there are many more books available, in particular from O'Reilly, which may be of relevance to topics treated in the present course.

13 Annex II – CD-ROM sets

The following CD-ROMs or CD-ROM sets are available at affordable prices. The list is certainly incomplete; new titles have appeared during the last year. *Caldera desktop* and *RedHat distribution* are obvious examples of CD-ROMs which do exist, but for which I have no reference. I find the last item in the list (*InfoMagic's Linux Developers Resource*) particularly useful.

1. "Linux Developers Resource 6 CD set", approx.$ 50.00 (I paid 57.00 CHF). Contains Slackware, SLS, RedHat, Debian, Bogus, JE and JF distributions, from: *InfoMagic, P.O. Box 30370, Flagstaff, AZ 86003, fax: +1-602-526-9573, e-mail: info@infomagic.com.*
   This is probably the most useful CD-ROM set.

2. "Slackware Linux", a 2 disc set with Slackware and archives, from: *Walnut Creek CDROM, 4041 Pike Lane, Suite D-461, Concord, CA 94520, e-mail: info@cdrom.com, WWW: http://WWW.cdrom.com, fax: +1-510-674-0821.* $39.95.

3. "Linux Developers Kit", 2 disc set,
   "Linux Runtime System", 1 CD with a runnable system (no file compression), both from: *Pacific Hi-Tech, $19.95. Address: see above.*

4. "WGS Linux Pro CD", $ 19.95, or "WGS Linux Pro 4 CD set", $ 29.95, or the latter + "WGS Linux Compendium" (1200 pages), $ 69. All from: *Work Group Solutions, Inc., P.O. Box 460190, Aurora, CO 80046-0190, e-mail: info@wgs.com, URL: ftp://ftp.wgs.com/pub2/wgs/Filelist.*

5. "SoftCraft Linux", 1 CD-ROM + 30 support, $ 69.95 from: *Solutions R Us, 4320 Stevens Creek Blvd, Suite 170, San Jose, CA 95129, e-mail: info@sru.com, fax: (408) 985-1880.*

6. "LinuxWare", 1 CD-ROM $29.95, Supplement of 3 CD-ROMs with Slackware $ 9.95, both together $34.95, from: *Trans-Ameritech Systems, 2342A Walsh Ave., Santa Clara, CA 95051, e-mail: order@trans-am.com, fax: (408) 727-3882.*

7. "S.u.S.E. Linux", 3 disc set with Slackware etc., DM 89,-/US$ 49.95, from: *S.u.S.E., Gebhardtstrasse 2, D-90762 Fuerth, e-mail: suse@suse.de, WWW: http://www.suse.de.*

Annex III — Resume of POSIX.4a definitions

POSIX.1c/D10 Summary

Disclaimer

SunSoft, Inc.
2550 Garcia Avenue
Mountain View, California 94043

Introduction

All source that uses POSIX.1c threads must include the header file.

```
#include <pthread.h>
```

In addition, Solaris requires the pre-processor symbol **_REENTRANT** to be defined in the source code before any C source (including header files).

```
#define_REENTRANT
```

The POSIX.1c thread library should be the last library specified on the `cc(1)` command line.

```
voyager% cc -D_REENTRANT ... -lpthread
```

Name Space

Each POSIX.1c type is of the form:

```
pthread[_object]_t
```

Each POSIX.1c function has the form

```
pthread[_object]_operation[_np|_NP]
```

where *object* is a type (not required if object is a thread), `operation` is a type-specific operation and `np` (or `NP`) is used to identify non-portable, implementation specific functions

All POSIX.1c functions (except for `pthread_exit`, `pthread_getspecific` and `pthread_self`) return zero (0) for success or an `errno` value if the operation fails.

There are eight(8) POSIX.1c types:

Table 0-1 POSIX.1c types

Type	Description
pthread_attr_t	Thread attribute
pthread_mutexattr_t	Mutual Exclusion Lock attribute
pthread_condattr_t	Condition variable attribute
pthread_mutex_t	Mutual Exclusion Lock (mutex)
pthread_cond_t	Condition variable (cv)
pthread_t	Thread ID
pthread_once_t	Once-only execution
pthread_key_t	Thread Specific Data (TSD) key

Feature Test Macros

POSIX.1c consists of a base (or common) component and a number of implementation optional components The base is the set of required operations to be supplied by every implementation. The pre-processor symbol (**_POSIX_THREADS**) can be used to test for the presence of the POSIX.1c base. Additionally, the standards document describes a set of six (6) optional components. A pre-processor symbol can be used to test for the presence of each All of the symbols appear in the following table

Table 0-2 POSIX.1c Feature Test Macros

Feature Test Macro	Description
_POSIX_THREADS	base threads
_POSIX_THREAD_ATTR_STACKADDR	stack address attribute
_POSIX_THREAD_ATTR_STACKSIZE	stack size attribute
_POSIX_THREAD_PRIORITY_SCHEDULING	thread priority scheduling
_POSIX_THREAD_PRIO_INHERIT	mutex priority inheritance
_POSIX_THREAD_PRIO_PROTECT	mutex priority ceiling
_POSIX_THREAD_PROCESS_SHARED	inter-process synchronization

Macro Dependency

If **_POSIX_THREAD_PRIO_INHERIT** is defined then **_POSIX_THREAD_PRIORITY_SCHEDULING** is defined.

If **_POSIX_THREAD_PRIO_PROTECT** is defined then **_POSIX_THREAD_PRIORITY_SCHEDULING** is defined.

If **_POSIX_THREAD_PRIORITY_SCHEDULING** is defined then **_POSIX_THREADS** is defined.

If **_POSIX_THREADS** is defined then **_POSIX_THREAD_SAFE_FUNCTIONS** is defined.

POSIX.1c API

In the following sections, function arguments that are of the form:

```
type name = NULL
```

indicate that a value of **NULL** may safely be used for name.

```
int   pthread_atfork( void (*prepare)(void) = NULL,
                      void (*parent)(void) = NULL,
                      void (*child)(void) = NULL );
```

Register functions to be called during fork execution.
errors **ENOMEM**
notes `prepare` functions are called in reverse order of registration.
`parent` and `child` functions are called in order of registration.

Thread Attributes

All thread attributes are set in an attribute object by a function of the form:

```
int   pthread_attr_setname( pthread_attr_t *attr, Type t );
```

All thread attributes are retrieved from an attribute object by a function of the form:

```
int   pthread_attr_getname( const pthread_attr_t *attr, Type *t );
```

Where *name* and *Type* are from the table below.

Table 0-3 Thread Attributes

Name and Type	Feature Test Macro	Value(s)
`int inheritsched`	**_POSIX_THREAD_PRIORITY_SCHEDULING**	**PTHREAD_INHERIT_SCHED,**
		PTHREAD_EXPLICIT_SCHED
`int schedpolicy`	**_POSIX_THREAD_PRIORITY_SCHEDULING**	**SCHED_FIFO,**
		SCHED_RR,
		SCHED_OTHER
`struct sched_param schedparam`	**_POSIX_THREADS**	POSIX.1b, Section 13
`int contentionscope`	**_POSIX_THREAD_PRIORITY_SCHEDULING**	**PTHREAD_SCOPE_SYSTEM,**
		PTHREAD_SCOPE_PROCESS
`size_t stacksize`	**_POSIX_THREAD_ATTR_STACKSIZE**	**>= PTHREAD_STACK_MIN**

Table 0-3 Thread Attributes

Name and Type	Feature Test Macro	Value(s)
`void *stackaddr`	**_POSIX_THREAD_ATTR_STACKADDR**	`void *stack`
`int detachstate`	**_POSIX_THREADS**	**PTHREAD_CREATE_DETACHED,**
		PTHREAD_CREATE_JOINABLE

```
int   pthread_attr_init( pthread_attr_t *attr );
```

Initialize a thread attribute object.
errors **ENOMEM**

```
int   pthread_attr_destroy( pthread_attr_t *attr );
```

Destroy a thread attribute object.
errors none

Thread Management

```
int   pthread_create( pthread_t *thread,
                      const pthread_attr_t *attr = NULL,
                      void *(*entry)(void *), void *arg );
```

Create a new thread of execution.
errors **EAGAIN, EINVAL**
note Maximum number of **PTHREAD_THREADS_MAX** threads per process.

```
int   pthread_detach( pthread_t thread );
```

Set the `detachstate` of the specified thread to **PTHREAD_CREATE_DETACHED**.
errors **EINVAL, ESRCH**

```
pthread_t   pthread_self( void );
```

Return the thread ID of the calling thread.
errors none

```
int   pthread_equal( pthread_t t1, pthread_t t2 );
```

Compare two thread IDs for equality.
errors none

```
void   pthread_exit( void *status = NULL );
```

Terminate the calling thread.
errors none

```
int   pthread_join( pthread_t thread, void **status = NULL );
```

Synchronize with the termination of a thread.
errors **EINVAL, ESRCH, EDEADLK**
note This function is a cancellation point.

```
#include <sched.h>
int   pthread_getschedparam( pthread_t thread, int *policy, struct sched_param *param );
```

Get the scheduling policy and parameters of the specified thread.
control **_POSIX_THREAD_PRIORITY_SCHEDULING**
errors **ENOSYS, ESRCH**

```
#include <sched.h>
int   pthread_setschedparam( pthread_t thread, int policy,
                             const struct sched_param *param );
```

Set the scheduling policy and parameters of the specified thread.
control _POSIX_THREAD_PRIORITY_SCHEDULING
errors ENOSYS, EINVAL, ENOTSUP, EPERM, ESRCH
policy [SCHED_RR, SCHED_FIFO, SCHED_OTHER]

Mutex Attributes

All mutex attributes are set in a mutex attribute object by a function of the form:

```
int    pthread_mutexattr_setname( pthread_attr_t *attr, Type t );
```

All mutex attributes are retrieved from a mutex attribute object by a function of the form:

```
int    pthread_mutexattr_getname( const pthread_attr_t *attr, Type *t );
```

Where *name* and *Type* are from the table below

Table 0-4 Mutex Attributes

Name and Type	Feature Test Macro	Value(s)
`int protocol`	_POSIX_THREAD_PRIO_INHERIT, _POSIX_THREAD_PRIO_PROTECT	PTHREAD_PRIO_NONE, PTHREAD_PRIO_PROTECT, PTHREAD_PRIO_INHERIT
`int pshared`	_POSIX_THREAD_PROCESS_SHARED	PTHREAD_PROCESS_SHARED, PTHREAD_PROCESS_PRIVATE
`int prioceiling`	_POSIX_THREAD_PRIO_PROTECT	POSIX.1b, Section 13

```
int    pthread_mutexattr_init( pthread_mutexattr_t *attr );
```
Initialize a mutex attribute object.
errors ENOMEM

```
int    pthread_mutexattr_destroy( pthread_mutexattr_t *attr );
```
Destroy a mutex attribute object.
errors EINVAL

Mutex Usage

```
int    pthread_mutex_init( pthread_mutex_t *mutex, const pthread_mutexattr_t *attr = NULL );
pthread_mutex_t    mutex        = PTHREAD_MUTEX_INITIALIZER;
```
Initialize a mutex.
errors EAGAIN, ENOMEM, EPERM, EBUSY, EINVAL

```
int    pthread_mutex_destroy( pthread_mutex_t *mutex );
```
Destroy a mutex.
errors EBUSY, EINVAL

```
int    pthread_mutex_getprioceiling( const pthread_mutex_t *mutex, int *prioceiling );
```
Get the `prioceiling` value of the specified mutex.
control _POSIX_THREAD_PRIO_PROTECT
errors ENOSYS, EINVAL, EPERM

```
int    pthread_mutex_setprioceiling( pthread_mutex_t *mutex, int prioceiling,
                                     int *old_ceiling );
```
Set the `prioceiling` value and return the old `prioceiling` value in the specified mutex.
control _POSIX_THREAD_PRIO_PROTECT
errors ENOSYS, EINVAL, EPERM

```
int    pthread_mutex_lock( pthread_mutex_t *mutex );
```
Acquire the indicated mutex.
errors EINVAL, EDEADLK

```
int    pthread_mutex_trylock( pthread_mutex_t *mutex );
```
Attempt to acquire the indicated mutex.
errors EINVAL, EBUSY, EINVAL

```
int    pthread_mutex_unlock( pthread_mutex_t *mutex );
```
Release the (previously acquired) mutex.
errors EINVAL, EPERM

Once-only Execution

```
pthread_once_t    once    = PTHREAD_ONCE_INIT;
```
Initialize a once control variable.

```
int    pthread_once( pthread_once_t *once_control, void (*init_routine)(void) );
```
Execute `init_routine` once.
errors none specified

Condition Variable Attributes

All condition variable attributes are set in a condition variable attribute object by a function of the form:

```
int    pthread_condattr_setname( pthread_condattr_t *attr, Type t );
```

All condition variable attributes are retreived from a condition variable attribute object by a function of the form:

```
int    pthread_condattr_getname( const pthread_condattr_t *attr, Type *t );
```

Where *name* and *Type* are from the table below

Table 0-5 Condition Variable Attributes

Name and Type	Feature Test Macro	Value(s)
`int pshared`	_POSIX_THREAD_PROCESS_SHARED	PTHREAD_PROCESS_SHARED, PTHREAD_PROCESS_PRIVATE

```
int    pthread_condattr_init( pthread_condattr_t *attr );
```
Initialize a condition variable attribute object.
errors ENOMEM

```
int    pthread_condattr_destroy( pthread_condattr_t *attr );
```
Destroy a condition variable attribute object.
errors EINVAL

Condition Variable Usage

```
int    pthread_cond_init( pthread_cond_t *cond,
```

```
                        const pthread_condattr_t *attr = NULL );
pthread_cond_t      cond        = PTHREAD_COND_INITIALIZER;
```

Initialize a condition variable.
errors EAGAIN, ENOMEM, EBUSY, EINVAL

```
int   pthread_cond_destroy( pthread_cond_t *cond );
```

Destroy a condition variable.
errors EBUSY, EINVAL

```
int   pthread_cond_signal( pthread_cond_t *cond );
```

Unblock at least one thread currently blocked in the specified condition variable.
errors EINVAL

```
int   pthread_cond_broadcast( pthread_cond_t *cond );
```

Unblock all threads currently blocked on the specified condition variable.
errors EINVAL

```
int   pthread_cond_wait( pthread_cond_t *cond, pthread_mutex_t *mutex );
```

Block on the specified condition variable.
errors EINVAL
note This function is a cancellation point.

```
int   pthread_cond_timedwait( pthread_cond_t *cond, pthread_mutex_t *mutex,
                        const struct timespec *abstime );
```

Block on the specified condition variable not longer than the specified absolute time.
errors ETIMEDOUT, EINVAL
note This function is a cancellation point.

Thread Specific Data

```
int   pthread_key_create( pthread_key_t *key, void (*destructor)(void *) = NULL );
```

Create a thread-specific data key.
errors EAGAIN, ENOMEM
note system limit of **PTHREAD_KEYS_MAX** per process.
system limit of **PTHREAD_DESTRUCTOR_ITERATIONS** calls to `destructor` per thread exit.

```
int   pthread_key_delete( pthread_key_t key );
```

Destroy a thread-specific data key.
errors EINVAL

```
void  *pthread_getspecific( pthread_key_t key );
```

Return the value bound to the given key for the calling thread.
errors none

```
int   pthread_setspecific( pthread_key_t key, const void *value );
```

Set the value for the given key in the calling thread.
errors ENOMEM, EINVAL

Signal Management

```
#include <signal.h>
int   pthread_sigmask( int how, const sigset_t *newmask = NULL, sigset_t *oldmask = NULL );
```

Examine or change calling threads signal mask.
errors EINVAL
how { SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK }

```
#include <signal.h>
int   pthread_kill( pthread_t thread, int signo );
```

Deliver signal to indicated thread.
errors ESRCH, EINVAL

```
#include <signal.h>
int   sigwait( const sigset_t *set, int *sig );
```

Synchronously accept a signal.
errors EINVAL, EINTR
note This function is a cancellation point.

Cancellation

```
int   pthread_setcancelstate( int state, int *oldstate );
```

Set the cancellation state for the calling thread.
errors EINVAL
`state` { PTHREAD_CANCEL_ENABLE, PTHREAD_CANCEL_DISABLE }

```
int   pthread_setcanceltype( int type, int *oldtype );
```

Set the cancellation type for the calling thread.
errors EINVAL
`type` { PTHREAD_CANCEL_DEFERRED, PTHREAD_CANCEL_ASYNCHRONOUS }

```
int   pthread_cancel( pthread_t thread );
```

Cancel the specified thread.
errors ESRCH
note threads that have been cancelled terminate with a status of **PTHREAD_CANCELED**.

```
void  pthread_testcancel( void );
```

Introduce a cancellation point.
errors none
note This function is a cancellation point.

```
void  pthread_cleanup_pop( int execute );
```

Pop the top item from the cancellation stack and optionally execute it.
errors none specified
note push and pop operations must appear at the same lexical level.
`execute` { 1, 0 }

```
void  pthread_cleanup_push( void (*routine)(void *), void *arg );
```

Push an item onto the cancellation stack.
errors none specified

C Refresh

Fourth College on Microprocessor-based Real-time Systems in Physics

Trieste, 7 Oct–1 Nov 1996

Carlos Kavka*
Departamento de Informática
Universidad Nacional de San Luis
San Luis
Argentina.

email: ckavka@unsl.edu.ar

Abstract

This chapter is intended to refresh your C programming language knowledge. It is not a complete guide or reference on the language. The topics are introduced mainly through simple examples.

*at present visitor at International Centre for Theoretical Physics, Trieste, Italy

1 Introduction

The C programming language was developed by Dennis Ritchie in the Bell Laboratories and was designed to be run on a PDP-11 computer with a Unix operating system. It is a small, flexible and concise language, with a mix of low-level assembler-style commands and high-level commands. It is an excellent selection in those areas where you may want to use assembly language but would keep it a 'simple to write' and 'easy to maintain' program.

The first standard was the Kernighan and Ritchie's book: "The C programming language" (1988). The ANSI C standard was defined when it was evident that the C programming language was becoming a very popular language. The ANSI C standard defines not only the syntax and semantics of the programming language but also a standard library. We will follow this standard in all the examples.

There is also another standard known as POSIX.1, which defines the interface of the system calls and some library functions, used to obtain services from the operating system. We will encounter this standard in the examples.

2 Getting started

The first example (see figure 1) is a program that prints the mean of two integer values. Not so much, but enough to begin. Note that the line numbers in the left column do not belong to the program; they are intended only for reference.

All C programs need a `main` function, and this is the place where the execution begins. In this example, three local variables in `main` are created. The first two, named `a` and `b` are of type integer, and the other one, named `answer` is of type float.

The sentences in line *11* assign values to the two integer variables. Then the function `mean` is called to calculate the mean of the two integer arguments given to it. The types of the formal parameters of the function (in this case `x` and `y`) should be compatible with the actual parameters in the call. The initial values of `x` and `y` are copied from the variables mentioned in the call (`a` and `b`).

The function `mean` returns the mean of the two integer arguments (a float, hence the float before the function name). It also declares a local variable `f` of type float to be used to store the mean value, which is computed in line *5*. This value is returned to the `main` program through the return statement.

We have used 2.0 (a float constant) instead of 2 (an integer constant) in line *5*, because we want a float as the result of the division operation. If we

```
1.  #include <stdio.h>
2.  #include <stdlib.h>

3.  float mean(int x,int y) {
4.    float f;

5.    f = (x + y) / 2.0;
6.    return f;
7.  }

8.  int main() {
9.    int a,b;
10.   float answer;

11.   a = 3; b = 2;
12.   answer = mean(a,b);
13.   printf("the mean of %d and %d is %f\n",a,b,answer);
14.   exit(0);
15.  }
```

Figure 1: a program to print the mean of two integers

divide an integer by another integer, we obtain an integer:

```
(3 + 2) / 2   = 2
(3 + 2) / 2.0 = 2.5
```

The values are printed in the `main` function by using the standard library function `printf`. The on-line manual page describes this function fully. For now, just note that the first argument is a string in which some format specifiers are embedded: %d for integers and %f for floats. The string is printed with these format specifiers replaced with the values of the variables that follow the string as arguments.

The `exit` function terminates a program normally. It expects a single integer as argument, which is called the *exit status*, and can be examined by the process which puts this process to run (possibly the shell). If simply `main` 'falls off the end' (implicit return), the exit status of the process is undefined. By convention, an argument of 0 means OK, and an argument between 1 and 255 means an error has occurred.

The two `#include` directives found in lines *1* and *2*, instruct the preprocessor to include the definitions and declarations from the include files `stdio.h` and `stdlib.h`. For standard include files we use the form <*filename*> to indicate that the standard include directory must be searched. Our own directory is searched first if we use the form "*filename*" instead. The on-line manual page for each function shows which files must be included.

3 Control structures

3.1 Repetition statements

The C programming language provides three structures for looping: the `while` loop, the `do while` loop and the `for` loop.

The `while` loop continues to loop while some condition is true. When the condition is false, the looping is discontinued. Let's see an example (figure 2).

```
1.  #include <stdio.h>

2.  int main() {
3.    int i = 1,sum = 0;

4.    while(i < 5) {
5.      sum += i;
6.      i++;
7.    }
8.    printf("summation is %d\n",sum);
9.    exit(0);
10. }
```

Figure 2: `while` statement

This small program just prints the summation of the integer numbers from 1 to 4. As long as the expression of the while statement in parenthesis is true, all statements within the braces are repeatedly executed.

If the variable `i` were initialized to any number greater than or equal to 5, the statements inside the braces of the while loop would not be executed at all. If the variable were not incremented in the loop, the loop would never terminate. If there were just one statement to be executed within the loop, no braces would be needed.

Note the short expressions used in lines *5* and *6*. The operator ++ is called the increment operator. The meaning of these expressions is the following:

```
sum += i      sum = sum + i
i++           i = i + 1
```

The `for` loop is nothing new; just a new way to describe the `while` loop. The same example from figure 2 is re-written using the `for` statement in figure 3.

```
1.  #include <stdio.h>

2.  int main() {
3.    int i,sum;

4.    for(sum = 0,i = 1;i < 5;i++)
5.      sum += i;

6.    printf("summation is %d\n",sum);
7.    exit(0);
8.  }
```

Figure 3: `for` statement

The `for` statement has three expressions separated by semi-colons (;). The first one contains sentences that are executed prior to the first pass through the loop. In this case, two assignments. The comma (,) operator allows to put more than one expression, where only one is allowed. The second field is the test which is evaluated at the beginning of each pass through the loop. The third field is executed in every pass, but after all the statements in the body of the loop.

The `for` loop is convenient because all the control information of the loop is in one place. We will see later more examples on the use of the `for` statement.

The other construction, the `do while` loop is a variation of the `while` loop. The main difference is that the condition is evaluated at the end of the loop. This means that the body is executed at least once. The same example is re-written in figure 4. Note that the meaning of the program is the same, because in both cases, the body of the loop is executed at least once.

```
1.  #include <stdio.h>

2.  int main() {
3.    int i = 1,sum = 0;

4.    do {
5.      sum += i;
6.      i++;
7.    } while (i < 5);
8.    printf("summation is %d\n",sum);
9.    exit(0);
10. }
```

Figure 4: `do ... while` statement

3.2 break and continue

The `break` statement is used to jump out from a loop.

The `continue` statement does not cause a termination of the loop but causes a jump out of the present iteration. It always jumps to the end of the loop just prior the terminating brace. The loop is terminated or not based on the loop test. In the `for` statement, the last expression is evaluated as usual. A complete example is provided in figure 5.

3.3 if and switch

In the simplest form, the `if` statement has a condition and a statement. If the condition is true, the statement is executed, and if it is false, the statement is skipped. Note that the single statement can be replaced by a compound statement composed of several statements between braces.

The second form is similar, but with the addition of the word `else` and another statement. If the condition is false, this statement is executed.

The `switch` statement is like a multi-branch if. The key word `switch` is followed by a value between parenthesis, and a set of cases between braces, identified by the word `case` followed by a constant. The control is transferred to the first statement of the case whose constant is the same as the value between parenthesis. If no constant is found, the control is then transferred to the first sentence after the key word `default`, if there is one. If no case

```
1.  #include <stdio.h>
2.  #define N     50

3.  int main() {
4.    int i,c;
5.    int n_spaces = 0,n_symbols = 0,n_chars = 0;

6.    for(i = 0;i < N;i++) {
7.      c = getchar();

8.      if (c == EOF) break;
9.      if (c == '\n') continue;

10.     switch(c) {
11.       case ' ': n_spaces++;
12.                 break;
13.       case ',':
14.       case '.':
15.       case ';': n_symbols++;
16.       default:  n_chars++;
17.     }
18.   }
19.   printf("chars: %d spaces: %d symbols %d\n",
              n_chars,n_spaces,n_symbols);
20.   exit(0);
21. }
```

Figure 5: control statements

matches, and there is no default, no action is performed. Once an entry point is found, statements will be executed until a **break** is found, or until the control runs out of the switch braces.

An example that shows the use of most of the control structures discussed so far is presented in figure 5.

The objective of this program is to read at most 50 characters from the standard input, and print the number of spaces, the number of punctuation symbols (only '.', ';' and ','), and the number of characters read without considering spaces. Newlines ('\n') must be ignored, but considered in the

50 characters limit. EOF should be considered as the end of the input.

The line *2* contains a definition of a constant by using the preprocessor directive `#define`. Each occurrence of the identifier `N` in the program is replaced with the string 50. It can also be used to define macros.

The function `getchar()` reads a character from the standard input.

The `break` in line *8* will cause a jump out of the loop effectively terminating the loop, if the character read is an EOF. The `continue` statement on line *9* will cause a jump to the end of the loop if the character read is a newline. The third expression of the `for` statement (`i++`) will be executed.

If the character is a space, the corresponding counter (`n_spaces`) will be incremented, and the `break` in line *12* will cause a jump out of the `switch` statement.

If the character is a symbol, the corresponding counter (`n_symbols`) will be incremented, and the execution will continue also with the default sentences, allowing the counter of characters (`n_chars`) to be incremented.

If the character does not belong to this set, the default sentences are executed, incrementing the counter of characters (`n_chars`).

4 Expressions

- Most operations in C that are designed to operate with integers will work equally well with characters, because they are a form of integer values. The following code will convert upper case characters to lower case.

```
int c;

c = getchar();
if (c >= 'A' && c <= 'Z')
  c = c - 'A' + 'a';
```

  The `&&` operator is the *logical and*. The *logical or* operator is `||` and the *negation* operator is `!`.

- The operators `++` and `--` are known as the increment and decrement operators respectively. `i++` is equivalent to `i = i + 1`, and `i--` is equivalent to `i = i - 1`. The operation can be done after the variable is used, or before, by using `i++` or `++i`, so

```
...
```

```
i = 10;
printf("i = %d\n",i++);
```

and

```
...
i = 10;
printf("i = %d\n",++i);
```

will both leave i as 11, but in the first example 10 will be printed, and in the second 11 will.

- There is an abbreviated form that can be used with binary operators. For example, the following expressions are equivalent:

```
i = i + 6       i += 6
i = i * 12      i *= 12
```

- The assignment expressions produce a value: the value that is effectively assigned, so

```
a = (b = 1 + 2) + 4;
```

will assign 3 to b and 7 to a.

- Comparisons will return 1 if the comparison is true, and 0 if it is false, so

```
i = (3 <= 8) + 2;
```

will assign 3 to i.

- There is no boolean type in C; the integer 0 stands for false, and any number different from 0 is considered as true. So

```
while(i != 0)
```

where != stands for *different*, is equivalent to `while(i)`.

- Some conditional expressions can be abbreviated by using the conditional operator (`?:`). For example,

```
if (x < 2)
  a = 5;
else
  a = 12;
```

can be re-written as

```
a = (x < 2) ? 5 : 12;
```

- There are also operations for bit manipulation, that can be applied to operands of types `int`, `short`, `long`, `unsigned` and `char`. They are the *bitwise and* `&`, the *bitwise or* `|`, the *bitwise exclusive or* `^`, the *left shift* `<<`, the *right shift* `>>` and the *one's complement* `~`.

  The example of the figure 6 shows a function used to count the number of bits in 1 in an `unsigned long`.

```
1.  int n_bits(unsigned long x) {
2.    int n = 0;

3.    while (x) {
4.      if (x & 0x01) n++;
5.      x >>= 1;
6     }
7.  }
```

Figure 6: bit manipulation

As we previously said, `while(x)` is equivalent to `while(x != 0)`. This is a safe stop point, because we are shifting `x`, and as it is unsigned, it is filled with 0's from the left.

The test on line *4* checks if the least significant bit of `x` is 1. Note that the constant 0x01 is hexadecimal. If a constant begins with 0 (zero) it is an octal one (like 077).

The expression in line *5* is an abbreviated form of `x = x >> 1`.

- The *cast* operator can be used to prescribe a conversion to a target data type, independent of the context. For example,

```
int x = 5,y = 2;
float f,g;

f = x / y;
g = x / (float)y;
```

will assign 2 to `f`, and 2.5 to `g`. The cast consists of the name of a type between parenthesis.

5 Arrays, Structures and Unions

An array is a set of contiguous variables of the same type, that can be accessed through an integer index. For example, the declaration:

```
int a[100];
```

reserves memory for 100 integer variables. They can be accessed by using subscripts from 0 to 99. For example, the program in figure 7 initializes all the components in an array and then print the summation of them.

A structure is a collection of variables grouped as a single object, where each one could be from a different type. For example, the following structure could be used to define a point giving its x and y coordinates:

```
structure point {
  float x;
  float y;
};
```

We can declare variables of this type:

```
struct point a,b;
```

and fill data by using the dot (.) operator:

```
a.x = 2.5;
a.y = 5.6;
```

We could have created an initialized point by using:

```
1.  #include <stdio.h>
2.  #define N      50

3.  int main() {
4.    int a[N],i,sum = 0;

5.    for(i = 0;i < N;i++)
6.      a[i] = i * 2;

7.    for(i = 0;i < N;i++)
8.      sum += a[i];

9.    printf("summation is %d\n",sum);
10.   exit(0);
11. }
```

Figure 7: arrays

```
struct point b = { 5.0 , 1.25 };
```

Structures can be assigned, passed to functions and returned, but they cannot be compared, so:

```
c = a;
```

is possible (all the fields from a are copied into c), but you cannot do:

```
if (a == b) ...                    /* not possible */
```

The figure 8 shows a program that assigns into a point structure c the structure a if a is equal to b. If this is not the case, the greater coordinates between a and b are assigned to c.

A union is like a structure, but the fields occupy the same memory locations, with enough memory allocated to hold the largest one. For example, the following union has two fields that overlap.

```
union option {
  int number;
  float price;
};
```

```
1.  struct point {
2.    float x;
3.    float y;
4.  };

5.  int main() {
6.    struct point a = { 2.3 , 3.1 },b,c;

7.    b.x = 1.5;
8.    b.y = 8.9;

9.    if (a.x == b.x && a.y == b.y)
10.     c = a;
11.   else {
12.     c.x = (a.x > b.x) ? a.x : b.x;
13.     c.y = (a.y > b.y) ? a.y : b.y;
14.   }
15    exit(0);
16. }
```

Figure 8: structures

An assignment to one of its fields overlap what it has in the other, so

```
union option x;

x.number = 13;
x.price = 12.5;
```

the value 13 will be over-written with the value 12.5. The programmer has to remember what the union is used for.

Structures and arrays can be combined, for example,

```
struct point arr[10];
```

is an array of then structures point, and their components can be accessed for example as follows:

```
arr[4].x = 3;
```

6 Type declarations

The `typedef` declaration allows us to give an identifier to a type, so it can be used in the same way as the predefined ones. For example,

```
typedef int integer;
```

will define the type `integer` as the standard type `int`, so now we can declare an `int` variable `x` by doing:

```
integer x;
```

A more useful example is the following:

```
typedef int array[100];
typedef struct point Point;
```

Now, we can declare:

```
array a;
Point x;
```

and `a` is an array of 100 integers, and `x` a structure. An array of 10 structures point can be defined as:

```
Point b[10];
```

7 Pointers

All variables are stored in some position in the memory, for example, as a result of

```
int i = 10;
```

the situation in the memory (simplified) could be as is shown in figure 9, assuming the base address of the variable is 3000.

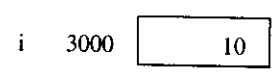


Figure 9: memory situation 1

A pointer to an integer can be defined as follows:

```
int *p;
```

and it can point to `i` by assigning to it the address of the variable. This value can be obtained by using the `&` operator:

```
p = &i;
```

and the situation in memory will be as is shown in figure 10.

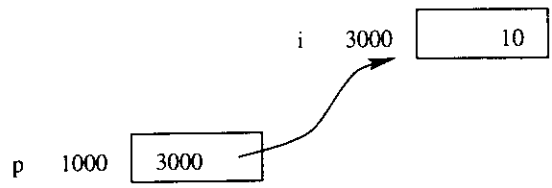


Figure 10: memory situation 2

The value pointed by a pointer can be accessed by using the operator `*`. We can print the value pointed by a pointer, and modify it by executing:

```
printf("value pointed by p = %d\n",*p);  /* prints 10 */
*p = 5;
printf("value of i = %d\n",i);           /* prints 5 */
```

8 Pointers as parameters

We have seen that C copies the values of the actual arguments into the formal parameters of the function when it is called. It is not possible for a function to modify the arguments, so a function that swaps the values of the arguments cannot be defined.

To be able to remove this restriction, the addresses of the arguments can be passed as parameters. In figure 11 the code for a function that swaps the values of the arguments is shown.

The memory situation when the function is called is depicted in figure 12.

Note that the function defines the parameters as pointers to integers. The addresses of the variables are passed by *value*, so they can not be modified. But this is not important. We want to use them to be able to interchange the values of the original variables.

The standard library function `scanf` can be used to read from the standard input. The first argument is a format string which gives information on the external representation of the data (similar to the one used in `printf`).

```
1.  void swap(int *x,int *y) {
2.    int temp = *x;
3.    *x = *y;
4.    *y = temp;
5.  }

6.  int main() {
7.    int a = 2,b = 5;
8.    swap(&a,&b);
9.  }
```

Figure 11: pointers as arguments

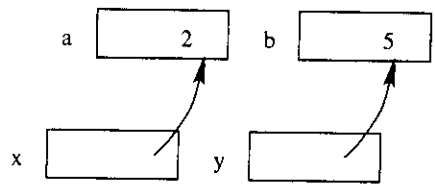


Figure 12: parameters when swap is called

The next arguments are the addresses of the variables where the input values must be stored. For example, to read two integers and one float value from standard input, we can do:

```
int i,j;
float f;

scanf("%d %d %f",&i,&j,&f);
```

We must pass the address of the variables. This is the only way in which the `scanf` function will be able to store the values.

9 Pointers to structures

It is also possible to assign the address of structures to pointers. For example, if we declare a structure of type Point (declared in section 6):

```
Point s = { 2.0 , 3.0 };
```

and a pointer to Point structures:

```
Point *p;
```

We can make the pointer p to point to s by executing:

```
p = &s;
```

so the situation in memory may now be as is shown in figure 13.

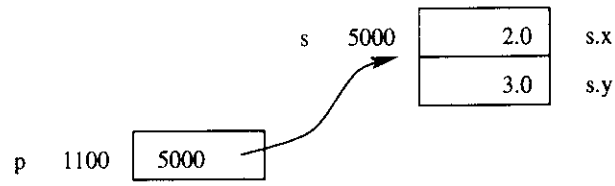


Figure 13: pointer to a structure

To modify a field of the structure by using the pointer, we can write:

```
(*p).y = 8.0;
```

The parenthesis can not be omitted, since the dot operator has a higher precedence than the asterisk operator. The same behavior of these two operators can be obtained with the operator ->. So, we can write:

```
p->y = 8.0;
```

10 Program structure

We have seen that a program consists of a set of functions. The variables we have used so far were all local variables to these functions. When the program is not executing statements in a function, these local variables do not even exist. Space is created for them when the function is called. This space is deallocated when the function is abandoned. These variables have an *automatic* storage class.

Local variables can be defined in such a way that the values they can have will still remain between calls, even if they can not be accessed when the statements of the function are not under execution. These variables have an *static* storage class. See the example in figure 14.

This program has a function f that receives no arguments and has no return value (this is the meaning of the void key word). It defines in lines *3*

```
1.  #include <stdio.h>
2.  void f() {
3.    int a = 0;
4.    static int b = 0;

5.    printf("a = %d b = %d\n",a++,b++);
6.  }

7.  int main() {
8.    f();
9.    f();
10.   exit(0);
11.  }
```

Figure 14: storage classes

and *4* two local variables named a and b initialized to 0. a has an automatic storage class, and b has a static storage class. This means that a is initialized every time the function is called. b is initialized just the first time the function is called, and the value is maintained through successive function calls. So, the values printed by the program are:

```
0    0
0    1
```

We have defined only variables that are local to functions. It is possible to define variables that can be accessed in more than one function, and also local to some compound statement.

Large C programs usually consist of several source files. They are compiled separately, and the object files are combined into one executable program. C provides the possibility that variables and functions defined in one module can be used in another one. They are called *external.*

This is illustrated in figure 15. The example is not meaningful, but it shows the different possibilities.

In module one.c, two variables are declared outside the scope of the functions. The variable b is static. This means it can be accessed in all functions, but in the same file in which it is defined (it is called *file scope*). The variable a is an extern variable, and can be accessed in the file in which it is defined, and also in all the files in which it is declared (*program scope*).

```
1.  /* module one.c */
2.  int a;
3.  static float b;

4.  int main() {
5.    int f(int,float);
6.    extern float g(int);
7.    b = 3.9 + g(2);
8.    a = f(2,b);
9.    exit(0);
10. }

11. int f(int x,float y) {
12.   return a + b + x;
13. }

14. /* module two.c */
15. extern int a;
16. extern int f(int,float);

17. float g(int x) {
18.   return (x + a + f(x,3.1)) / 2.0;
19. }
```

Figure 15: scope

The definition is in line *2* and a declaration is in line *15*. A declaration just specifies the attributes, and a definition does the same thing, but it also allocates memory space. An external variable has only one definition, but it can have several declarations.

The declaration in line *15* allows the variable a from module `one.c` to be accessed in module `two.c`.

In order to access the function g in module `one.c`, a declaration is provided in line *6*. As it is a local declaration, the function g can be called just from the `main` function.

The line *5* contains a declaration of the function `f`, which is defined later in the same file. This declaration is called a prototype and is required every time we want to call a function that is defined later in the file.

The declaration of the function `f` in line *16* allows this function to be called from functions in the file `two.c`.

11 Bibliography

Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice Hall, 1988.

Leendert Ammeraal. *C for programmers.* John Wiley and Sons Ltd., 1986.

W. Richard Stevens. *Advanced programming in the Unix Environment.* Addison Wesley Professional Computing Series, 1992.

Tim Love. *ANSI C for Programmers on Unix Systems.* Cambridge University Engineering Department.

Available on ftp::/svr-ftp.eng.cam.ac.uk:misc/love_C.ps.Z.

Advanced C

Fourth College on Microprocessor-based Real-time Systems in Physics

Trieste, 7 Oct–1 Nov 1996

Carlos Kavka *
Departamento de Informática
Universidad Nacional de San Luis
San Luis
Argentina.

email: ckavka@unsl.edu.ar

Abstract

In this chapter, we will cover some more advanced characteristics of the C programming language.

*at present visitor at International Centre for Theoretical Physics, Trieste, Italy

1 Pointers and Arrays

The relation between pointers and arrays in C is quite strong. All operations that could be defined with arrays can also be implemented by using pointers.

Let us define an array of integers, and a pointer to integer:

```
int a[5] = { 7 , 4 , 9 , 11 , 8 };
int *p;
```

After the assignment

```
p = &a[0];
```

the pointer `p` points to the beginning of the array `a`. This could have been done also by:

```
p = a;
```

because the name of the array represents also a pointer to the first element `a[0]`.

The situation in memory may now be as follows:

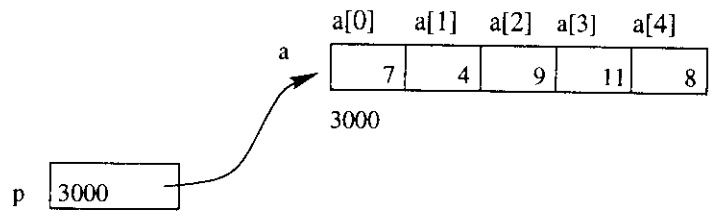


Figure 1: a pointer to an array

The value of the first component of the array can be assigned into an integer variable `x` by using an index `i`

```
x = a[0]
```
(where i = 0 in this case)

or through the pointer

```
x = *p;
```

It is allowed to add an integer constant to a pointer. By definition, if a pointer `p` points to a component of an array `a`, `p+i` points to `i` components after `p`. See figure 2.

The value of the fourth component of the array `a` can be assigned into the integer variable `x` by using the index

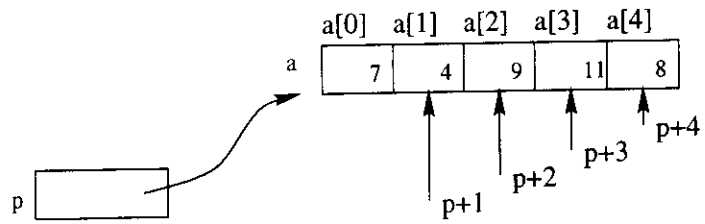


Figure 2: adding constants to p

```
x = a[3];
```

or by using the pointer

```
x = *(p+3);
```

If a pointer `p` points to the beginning of the array `a`, then `*(p+i)` is equivalent to `a[i]`. Note that this is also valid for the name of the array, so `*(a+i)` is equivalent to `a[i]`.

2 Pointer arithmetic

C allows several forms of arithmetic operations with pointers, and this is one of the distinctive features of the language.

We have seen in the previous section that it is possible to add a constant to a pointer that points to an array. Likewise, subtraction is permissible. When a constant `i` is added (subtracted) to a pointer `p`, `p` is moved ahead (back) in the array `i` positions, without considering the size of the components.

Let us, for example, assume that we have declared an array of structures:

```
typedef struct Point {
  int x;
  int y;
};
Point a[4];
```

and two pointers to this kind of structure:

```
Point *p,*q;
```

These pointers can point to some components in the array:

```
p = &a[1];
q = &a[3];
```

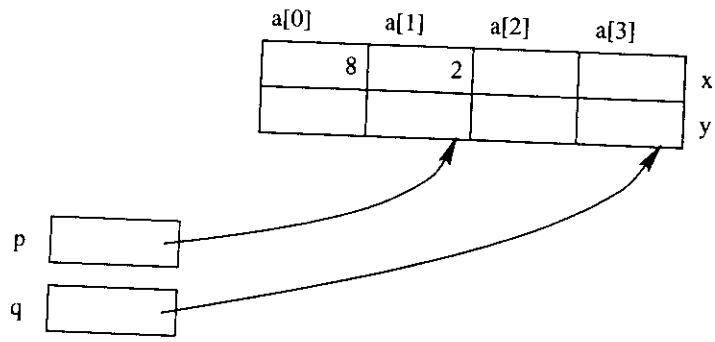


Figure 3: pointers to array of structures

as can be seen in figure 3.

It is possible to access the components of the structure pointed by a pointer using the pointer operator, for example,

```
p->x = 2;
```

which in this case, is equivalent to `a[1].x = 2`.

By subtracting the constant 3 from the pointer `q`, we can access the 0th component of the array:

```
*(q - 3).y = 8;
```

Pointers to components of an array can also be compared. In the following example, the first two conditions evaluate to true (1) and the last one to false (0):

```
p < q
p != q
p >= q
```

Pointers can also be modified. As an example, if we execute:

```
p = p - 1;  (or p--)
```

`p` will now point to the previous component in the array, as is shown in figure 4.

The subtraction of pointers is also valid, and it produces an integer that represents the number of components between the two pointers. As an example,

```
q - p    returns 3
```

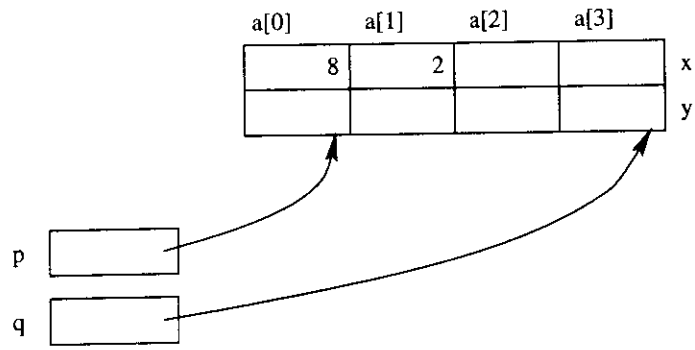


Figure 4: after p--

`p - q` returns -3

To initialize all the fields of the structures of the array `a` to 0, we can execute:

```
for(p = &a;p <= &a[3];p++) {
  p->x = 0;
  p->y = 0;
}
```

3 Pointers to void

A pointer to `void` is called a *generic pointer*, and it can point to objects of any type. We have defined in the previous sections just pointers that point to an object of a specified type. Let us see an example:

```
int i;
float f[5];

void *p,*q;
p = (void*)&i;
q = (void*)&f[3];
```

In this example, two generic pointers `p` and `q` are defined. The pointer `p` is pointed to the integer variable `i`, and `q` is pointed to a component of the array `f` of type float. Note that we must use the cast operator to explicitly convert the types.

These pointers can point to objects of any type. However, there are some operations that cannot be done with these pointers. For example, it is

not possible to add a constant to a generic pointer. The reason is that the compiler does not know the size of the object pointed by the pointer. So, for example,

`q++` cannot be done

although, with the appropriate cast, the following operation can be done:

`(float*)q++` is legal.

As another example, to print the integer value pointed to by p, we can do:

```
printf("%d\n",*((int*)p));
```

4 Strings

A string is represented in C as an array of characters. The end of the string is denoted by a *null* character, which is written as `'\0'`. So, one extra byte is needed to represent the string.

As an example,

```
char str1[] = "C is nice";
```

will define an array of 10 elements, as is shown in figure 5. Note that the size of the array is obtained from the length of the string plus one byte for the null character. If we had defined a longer array, the extra space will remain uninitialized.

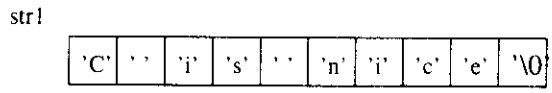


Figure 5: our first string

The name of the array can be considered as a pointer. However, there is a significant difference if we define a string like this

```
char *str2 = "C is nice";
```

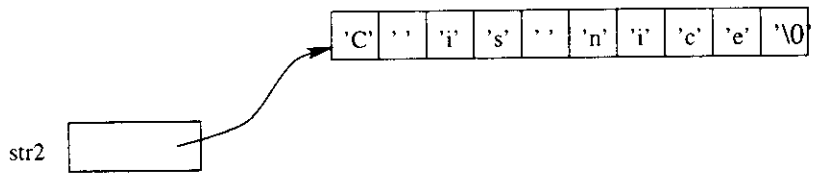


Figure 6: our second string

In this case, we obtain a real pointer and an array, as is shown in figure 6.

In both cases, references to individual characters can be done, by using both the notation of pointers or with indexes:

`str1[2]`	is equivalent to	`*(str1+2)`
`str2[2]`	is equivalent to	`*(str2+2)`

However, an important difference is that the name of the array is a constant pointer, so it cannot be modified:

`str1++`	is not allowed, and
`str2++`	advances the pointer by one position.

C does not provide operators that work with whole strings. As an example, if we have two strings `s1` and `s2`, we would like to execute `s1 = s2` to assign strings. This is not possible, because they are pointers, and we would have just copied the addresses. We must copy the characters one by one, by using a loop. The next function `strcpy` allows us to do this.

```
void strcpy(char *s1,char *s2) {
  while (*s1++ = *s2++);
}
```

The following is a situation in which the function `strcpy` can be used:

```
char a[12];
char b[] = "C is nice";

strcpy(a,b);
```

Note that as we are passing the name of the arrays as arguments, we are really passing the addresses of these arrays as arguments. The figure 7 shows the situation when the function `strcpy` is just called.

The code of the `strcpy` function is extremely compact and efficient, and it could be intimidating. The `while` loop has no body. This means that the condition will be evaluated, until it becomes false. Note that the condition is an assignment expression,

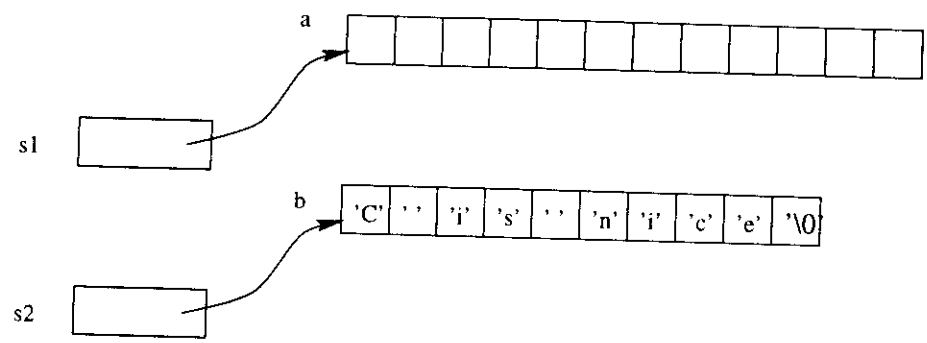


Figure 7: just to execute `strcpy`

```
*s1++ = *s2++
```

so the assigned value will be used to determine if the condition is true or not: if this value is 0, the condition will be considered false, and true if it is different from 0.

In the assignment expression, the right hand side is considered first:

```
*s2++
```

The `++` is executed first, so the pointer `s2` is advanced to the next position. However, it is a *post*-increment operation, this means, that it returns the pointer as it was before the operation. This value is de-referenced with the `*` operation. In the example, the first time this expression is evaluated, `s2` will point to the position b[1], and the character obtained will be `'C'`.

On the left side:

```
*s1++
```

the process is the same. The character is assigned to the position pointed to by `s1`, and the pointer is advanced to the next position. The process is repeated until the character `'\0'` is copied. In this last case, the value returned by the assignment will be 0, and the condition will be evaluated to false. This situation can be seen in figure 8.

Note that the target string must have enough space to contain the characters to be copied from the source string.

5 Library functions for strings

There exist many functions that work with strings in the standard library. We will go through some of them.

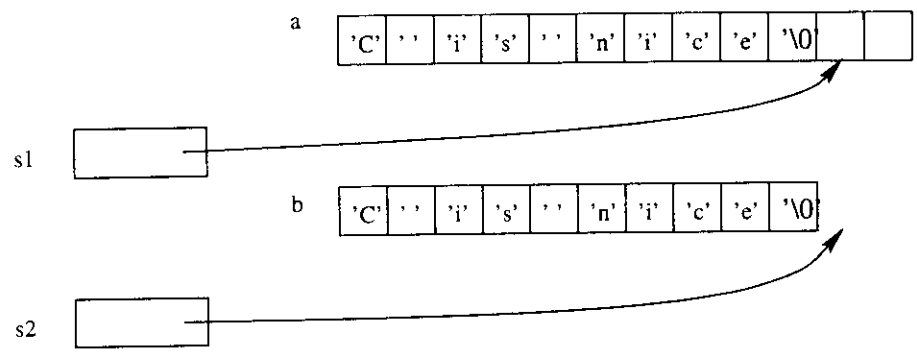


Figure 8: just to return from `strcpy`

- `void strcpy(char *s1,char *s2)`

  We have already seen the operation of the function `strcpy` in the previous example. It copies all the characters pointed to by `s2` to the area pointed to by `s1`, until the null character is copied. There must be enough space for them on the area pointed to by `s1`. As an example:

```
char a[12];
char b[] = "C is nice";

strcpy(a,b);
printf("a: %s\n",a);
```

  will print:

```
a: C is nice
```

- `void strcat(char *s1,char *s2)`

  This function concatenates the characters pointed to by `s2` to the string pointed to by `s1`. There must be enough space in the area pointed by `s1` to store the characters from both strings. As an example:

```
char a[12];

strcpy(a,"C is ");
strcat(a,"nice");
printf("a: %s\n",a);
```

  will print

```
a: C is nice
```

- `int strcmp(char *s1,char *s2)`

  This function allows to compare lexicographically strings `s1` and `s2`. It returns 0 if both strings are equal, a negative value if `s1` is before `s2` and a positive value if `s1` is after `s2`.

- `int strlen(char *s1)`

  This function will return the number of characters in the string `s1` without considering the null character. As an example:

```
char a[] = "C is nice";

printf("length of a: %d\n",strlen(a));
```

  will print

```
length of a: 9
```

- `int sprintf(char *s,char *format,...)`

  This function works like `printf`, but the actual output goes to the string `s` instead of the standard output. The notation ... indicates that the number of arguments is variable, and in this case, it depends on the number of format specifiers in the format string. Let us see an example:

```
char a[20];
int i = 5;
float f = 3.5;

sprintf(a,"%d -- %f",i,f);
printf("a: %s\n",a);
```

  will print

```
a: 5 -- 3.5
```

6 Using strings

It is possible to define an array of strings, and initialize it at the same time. For example:

```
char *a[3] = { "C" , "is" , "nice" };
```

In this example, a is an array of three pointers to characters, or, an array of three strings. The memory may be as is shown in figure 9.

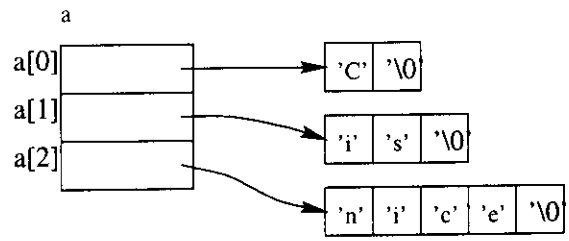


Figure 9: an array of strings

the expression `a[i]` can be used to access the i-th string. As an example:

`strlen(a[2])` returns 2.

Strings can also be used as fields in structures. For example, a structure that represents a person with his or her name and age, could be defined as follows:

```
typedef struct person {
  char name[10];
  int age;
};
```

and created and initialized by:

```
person x;

strcpy(x.name,"John");
x.age = 30;
```

Note that we must define the string as an array of characters, and not as a pointer. If we would have defined it as a pointer to characters, there would not have been space for the characters to be copied by `strcpy`.

If we would like to reserve just the exact amount of characters needed by the name of the person, we can define the field as a pointer, and ask for memory in a dynamic way. This point will be introduced in the next section.

7 Dynamic memory administration

Dynamic memory administration is the process by which memory can be allocated and freed at any point during the execution of the program.

C provides some functions in its standard library related to dynamic memory administration. The two most important ones are:

```
void *malloc(size_t n);
void free(void *p);
```

The `malloc` function asks for a memory block of size `n` (in bytes). If `n` consecutive bytes are available, it returns a pointer to the first byte. Otherwise, it returns the constant `NULL`.

As an example, if we want to copy the string `b` into `a`, we can reserve space for the exact amount of characters, and then copy the string:

```
char *a;
char *b = "a string";

if ((a = (char*)malloc(strlen(b)+1)) == NULL) {
  printf("not enough memory\n");
  exit(1);
}
```

Note that we must consider also the null character when we ask for memory space.

Let us suppose we need to obtain space for n integers during the execution of the program. The `malloc` function requires the size expressed in bytes. To know how many bytes an integer uses, we can use the operator `sizeof`, which takes as argument the name of a type or an object, and returns its size in bytes. The following piece of code allocates dinamically space for an array of `n` integers, and initializes all its components to zero.

```
int *arr,n,i;

scanf("%d",&n);

if ((arr = (int*)malloc(n * sizeof(int))) == NULL) {
  printf("not enough memory\n");
  exit(1);
}
for(i = 0;i < n;i++) arr[i] = 0;
```

As another example, to reserve memory for n structures `person` (as defined in the previous section), we can execute:

```
person *p;

scanf("%d",&n);

if ((p = (person*)malloc(n * sizeof(person))) == NULL) {
  printf("not enough memory\n");
  exit(1);
}
for(i = 0;i < n;i++) {
  strcpy(p[i].name,"");
  p[i].age = 0;
}
```

The standard library function `free`, is used to return back the memory that was obtained by calling `malloc`. For example, to return back all the memory that was dynamically assigned in the examples in this section, we can execute:

```
free((void*)a);
free((void*)arr);
free((void*)p);
```

8 A bigger example

In section 6 we have seen that an array of strings could be defined and initialized in a very simple way. For example:

```
char *a[3] = { "C" , "is" , "nice" };
```

However, if we want to build a structure like this in a completely dynamic way, it is not so easy. Remember that in this example, a is a pointer to pointers of characters, because a is the name of an array, and the name can be considered as a pointer to the first element.

We must begin with an empty structure

```
char **b;
```

First, we need to create the array of pointers.

```
if ((b = (char**)malloc(3 * sizeof(char*))) == NULL) {
  printf("not enough memory\n");
  exit(1);
}
```

then we can ask memory for the individual strings and we are ready to copy them

```
for(i = 0;i < 3;i++) {
  if ((b[i] = (char*)malloc(strlen(a[i])+1)) == NULL) {
    printf("not enough memory\n");
    exit(1);
  }
  strcpy(b[i],a[i]);
}
```

The three steps we have followed are shown in figure 10.

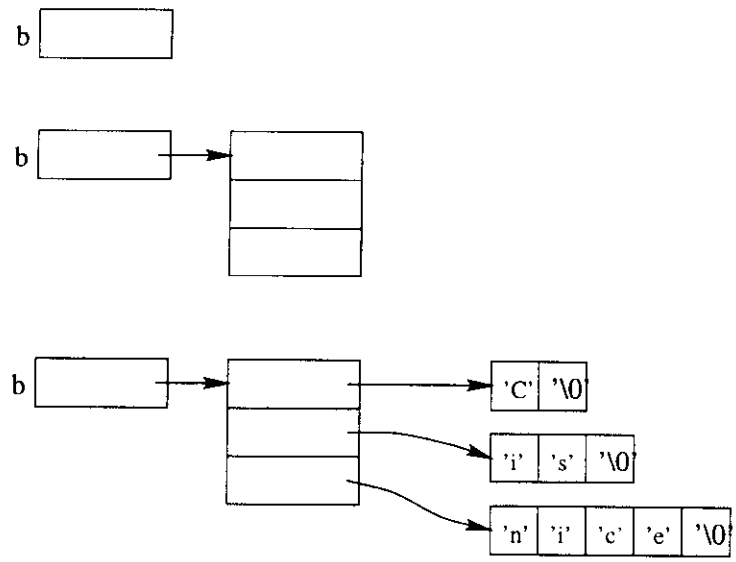


Figure 10: steps in dynamic memory allocation

To release the area that was dynamically allocated, we must follow the opposite procedure:

```
for(i = 0;i < 3;i++)
  free((void*)b[i]);
free((void*)b);
```

9 Program arguments

It is possible from the C program, to access the arguments that are passed in the command line. For example, if our program is called **program**, it can be executed from the shell prompt with a series of arguments, like for example:

```
$ program file.tex -b 123
```

The program arguments can be accessed through two parameters of the `main` function, named by convention **argc**, the argument count, and **argv**, the argument vector. **argc** is an integer, and **argv** is an array of strings, like the one we have been discussing in the previous sections. In this example, the values of the parameters are shown in figure 11.

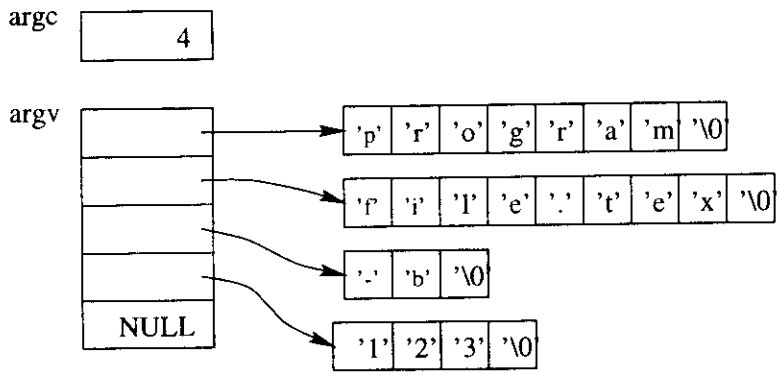


Figure 11: program arguments

The following program prints all the strings that are passed as arguments to the program.

```
#include <stdio.h>
int main(int argc,char *argv[]) {
  int i;

  for(i = 0;i < argc;i++)
    printf("%s\n",argv[i]);
  exit(0);
}
```

If we compile this program under the executable name **program**, and we execute it as it was shown before, we will obtain:

```
program
file.tex
-b
123
```

Note that the name of the program is the first string in the argv argument. Also the ANSI standard guarantees that argv[argc] is a NULL pointer. The notation char *argv[] is equivalent to char **argv, and can be used just when defining the arguments of a function.

We will see now an example, which will show how a program can deal with parameters as the standard Unix commands do. The program expects a file name as argument, and it has two options: -a and -b. The usual notation for this is the following:

```
program [[-a][-b]] <filename>
```

So, the program can be called, for example, as follows:

```
$ program a.tex
$ program -a a.tex
$ program -b a.tex
$ program -a -b a.tex
$ program -b -a a.tex
```

The code for the program is the following:

```
#include <stdio.h>
int main(int argc,char *argv[]) {
  int a_option = 0,b_option = 0;
  char **p_to_arg = &argv[1];

  while (--argc && (*p_to_arg)[0] == '-') {
    if ((*p_to_arg)[1] == '\0') {
      printf("invalid option\n"); exit(1);
    }
    switch((*p_to_arg)[1]) {
      case 'a': a_option = 1; break;
      case 'b': b_option = 1; break;
      default:  printf("invalid option\n"); exit(1);
    }
  p_to_arg++;
  }
```

```
  if (argc != 1) {
    printf("invalid arguments\n"); exit(1);
  }
  printf("a option: %s\n", (a_option) ? "yes" : "no");
  printf("b option: %s\n", (b_option) ? "yes" : "no");
  printf("file: %s\n",*p_to_arg);
  exit(0);
}
```

In the program, `p_to_arg` is a pointer initialized to point to the second entry in the `argv` argument, the one that corresponds to the first program argument.

The `while` loop processes the optional arguments. In every iteration the pointer `p_to_arg` is advanced to the next argument, and the argument count is decremented. This last operation is valid because `argc` is a local variable. The condition stands for: continue iterating while there are still arguments and the first character of the current argument is a `-`. At the end of the loop, `p_to_arg` will point to the filename, if there is one in the input line.

10 Pointers to functions

The functions are also stored in memory, and in C, pointers are allowed to point to them. As an example, let us define a pointer to functions, that take two integers as arguments and return an integer value:

```
int (*p)(int,int);
```

and define two functions with these characteristics:

```
int add(int x,int y) {
  return x + y;
}

int sub(int x,int y) {
  return x - y;
}
```

The pointer p can point to each of them. With the expression:

```
p = add;
```

the pointer p will point to add. This function add can be called through the pointer, by de-referencing it:

```
printf("%d\n",(*p)(2,3));
```

The important point is that the function pointed to by p is evaluated. So, if we make p a pointer to the function sub

```
p = sub;
```

and we return back to execute the printf function, sub will be called.

A pointer to a function can be passed to another function as a parameter and can be used within the function to call the function which it is pointing to. It is not permitted to increment or add a constant to a function pointer.

As an example, the following function do_op receives an integer n as argument, two arrays of integers with n elements in each and a pointer to a function:

```
int do_op(int n,int x[],int y[],int (*f)(int,int)) {
  int i,sum = 0;

  for(i = 0;i < n;i++)
    sum += (*f)(x[i],y[i]);
  return sum;
}
```

This function can be invoked, for example, as follows:

```
int a[3] = { 2 , 1 , 5 };
int b[3] = { 1 , 3 , 4 };

printf("%d\n",do_op(3,a,b,add));
printf("%d\n",do_op(3,a,b,sub));
```

The first printf will print 16 (2+1 + 1+3 + 5+4), and the second 0 (2-1 + 1-3 + 5-4).

11 Linked lists

We will see now an example in which we will combine dynamic memory allocation with structures.

A linked list has a pointer to access the first node, this node contains a pointer that points to the second one, and so on. The last node contains a null pointer. Each node keeps some information. In our example, we will assume it contains an integer data.

A node is an ideal candidate to be implemented with a structure. It must contain a field to store data, and the pointer to the next node. Note that this structure is recursively defined:

```
typedef struct node {
  int data;
  node *next;
};
```

The list will be a pointer to the first node:

```
typedef node *list;
```

A list `l` could be defined as follows.

```
list l;
```

An example of a list built with the previous structures is shown in figure 12.

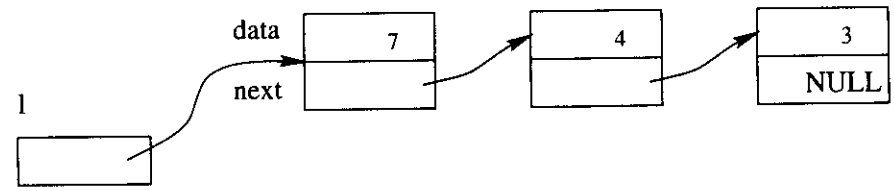


Figure 12: a linked list

A list like this could be traversed printing the integer information by using the following function:

```
void print(list l) {
  node *p;

  for(p = l;p != NULL;p = p->next)
    printf("%d\n",p->data);
}
```

The pointer p begins by pointing to the first node, and if its value is not NULL, the data field of the node pointed to by p is printed. Then p is advanced to the next node, by using the address stored in the **next** field. This process continues until p is NULL.

To create the list, we can define a function to create and insert a node into the list l in the position pos with a data value. Note that this function must allocate dynamically space for the node, and modify the involved pointers. If the node must be created in the first position, the pointer l that points to the first element of the list must be modified. So, its address is passed as argument (not the value).

```
void insert(list *first,int value,int pos) {
  node *p = *first,*prev = NULL;
  node *new_node;

  /* a new node must be created */
  if ((new_node = (node*)malloc(sizeof(node))) == NULL) {
    printf("not enough memory\n"); exit(1);
  }

  /* advance the pointers to reach insertion position */
  while (--pos) {
    prev = p;
    p = p->next;
  }

  if (prev == NULL) {  /* first position */
    *first = new_node;
    new_node->next = p;
  } else {             /* other position */
    prev->next = new_node,
    new_node->next = p;
  }
}
```

The pointer p is used to point to the node which is currently at the pos position. The pointer prev (previous) is used to point to the previous position. If we want to insert a node in the first position, the pointer p will point to this position, and the pointer prev will be NULL (no position to point). This situation is shown in figure 13. In this case, the pointer to the first element must be modified to point to the new first one. The new pointers are drawn with a dotted line.

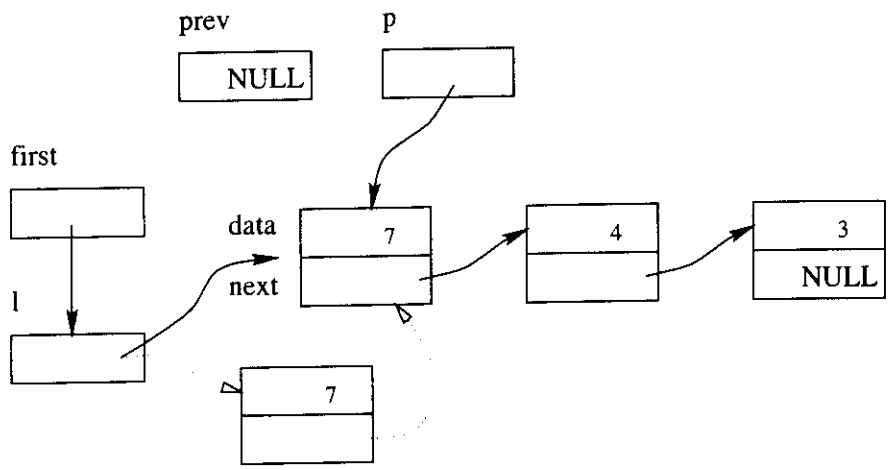


Figure 13: insertion in the first position

A situation in which the node to be inserted is not the first is shown in figure 14.

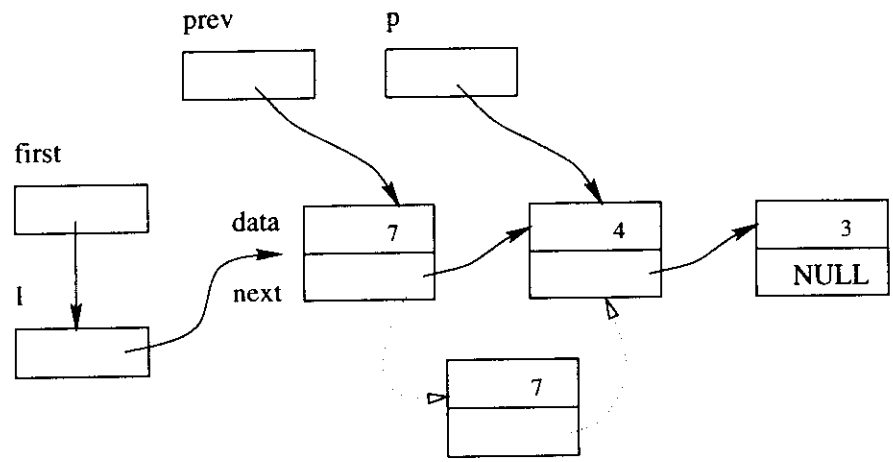


Figure 14: insertion in another position

12 Files

There are two possibilities to work with files in C. The first one is called *unbuffered I/O*. It is not part of the ANSI C standard, but it is part of POSIX.1. The term *unbuffered* refers to the fact that each *read* or *write* invokes a system call in the kernel. The other one, usually called, the *standard*

I/O routines belongs to the ANSI C standard, and provides higher level services.

12.1 Unbuffered I/O

All open files are referred to by file descriptors. A file descriptor is a non-negative integer. When we open a file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with this integer value.

By convention the Unix shell associates the file descriptor 0 to standard input, file descriptor 1 to standard output and file descriptor 2 to standard error. In POSIX.1 these numbers are replaced by the constants `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`.

The functions available for file I/O are five: `open`, `read`, `write`, `lseek` and `close`. We will look at them right now:

- `int open(char *pathname,int oflag)`

  The `pathname` is the name of the file to open or create. The value to be passed to the argument `oflag` is obtained from one of the following constants:

`O_RDONLY`	Open for reading only.
`O_WRONLY`	Open for writing only
`O_RDWR`	Open for reading and writing

  optionally OR'ed with constants from the following set (Not all the possibilities are shown):

`O_APPEND`	Append to the end of file on each write.
`O_CREAT`	Create the file if it does not exist. This option requires a third argument specifying the access permission bits of the new file.
`O_TRUNC`	If the file exists, and its open mode allows write operations, truncate its length to 0.
`O_NONBLOCK`	Sets the non blocking mode.

- `int close(int filedes)`

  Close the file with file descriptor `filedes`.

- `off_t lseek(int filedes,off_t offset,int whence)`

  Every open file has an associated 'current file offset'. It is a non negative integer that measures the number of bytes from the beginning of the file. The interpretation of the `offset` argument depends on the value of the `whence` argument.

  - If `whence` is `SEEK_SET`, the offset of the file is set to `offset` bytes from the beginning of the file.
  - If `whence` is `SEEK_CUR`, the offset of the file is set to its current value plus the offset. The offset can be positive or negative.
  - If `whence` is `SEEK_END`, the offset of the file is set to the size of the file plus the `offset`. The offset can be positive or negative.

  The offset of the file can be greater than the current size, in which case, the next `write` to the file will extend it.

- `ssize_t read(int filedes,void *buff,size_t nbytes)`

  This function read `nbytes` from the file and store them in memory beginning at the address pointed to by `buff`. It returns the number of bytes successfully read.

- `ssize_t write(int filedes,void *buff,size_t nbytes)`

  This function writes `nbytes` from the address pointed to by `buff` to the file. It returns the number of bytes successfully written.

12.2 Some examples

Let us see some examples. The following program, opens a file named `file.data`, writes a string and closes it.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main() {
  int fd;
  char *str = "some data";
  int n = strlen(str)+1;

  if ((fd = open("file.data",O_WRONLY | O_CREAT | O_TRUNC,
```

```
                                    S_IRUSR | S_IWUSR)) < 0) {
    perror("can not open");
    exit(1);
  }

  if (write(fd,str,n) != n) {
    perror("can not write");
    exit(1);
  }

  exit(0);
}
```

The file is opened only for writing; it is truncated to zero length if it existed, and if it is created, permissions to read and write are granted to the user. The function `perror` prints the string it receives as argument and then print the system error message. Note that a `close` is not necessary at the end, because all files are closed automatically when the program exits.

The following program can read from the file just created. Ten characters are read from the file to the area pointed to by `str`. Note that we must have enough space for the characters read.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main() {
  int fd;
  char str[10];

  if ((fd = open("file.data",O_RDONLY)) < 0) {
    perror("can not open");
    exit(1);
  }

  if (read(fd,str,10) != 10) {
    perror("can not read");
    exit(1);
  }

  exit(0);
}
```

The following program can create an empty file with 1KB size:

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main() {
  int fd;
  char c = '\0';

  if ((fd = open("file.data",O_WRONLY | O_CREAT | O_TRUNC,
                             S_IRUSR | S_IWUSR)) < 0) {
    perror("can not open");
    exit(1);
  }

  if(lseek(fd,1024,SEEK_END) != 1024) {
    perror("can not seek");
    exit(1);
  }
  if (write(fd,&c,1) != 1) {
    perror("can not write");
    exit(1);
  }

  exit(0);
}
```

12.3 Standard I/O library

The ANSI C standard I/O library handles details such as buffering allocation and performing I/O in optimal-sized chunks.

When we open a file, the standard I/O function returns a pointer to a FILE object. This object contains all the information required by the other library functions. We never use this structure directly. We pass a pointer to this structure to the other standard functions, in the same way as we were using the file descriptors before.

Some of the functions available are described now. Note that just a short description is presented. For more information, please refer to man pages.

- `FILE *fopen(char *pathname,char *type)`

This function opens the file `pathname`. `type` could be one of the following values:

`"r"` open for reading.

`"w"` open for writing.

`"a"` append; open for writing at the end of the file, or create for writing.

`"r+"` open for reading and writing.

`"w+"` truncate to 0 length or create for reading and writing.

`"a+"` open or create for reading and writing at the end of file.

If the file can be successfully opened, a pointer to a `FILE` structure is returned. If there is a problem, a `NULL` pointer is returned.

- `int fclose(FILE *fp)`

  Closes the file specified by `fp`.

- `int fgetc(FILE *fp)`

  Reads one character from the file specified by `fp`. `getc` is equivalent to `fgetc`, but it is implemented as a macro. It returns `EOF` to indicate an error condition.

- `int ungetc(int c,FILE *fp)`

  Push back the character `c` into the stream specified by `fp`. This means that it will be available on a subsequent reading.

- `int fputc(int c,FILE *fp)`

  Write the character `c` in the file specified by `fp`. `putc` is equivalent to `fputc`, but it is defined as a macro. It returns `EOF` to indicate an error condition.

- `char *fgets(char *buff,int n,FILE *fp)`

  This function reads at most `n-1` characters into the area pointed to by `buff` from the file specified by `fp`. The reading is stopped after an EOF or a newline. It returns `NULL` to indicate an error condition.

- `int fputs(char *str,FILE *fp)`

  This function writes the string pointed to by `str` to the file specified by `fp`. It returns `EOF` to indicate an error condition.

- `int fflush(FILE *fp)`

  This function causes any unwritten data to be passed to the kernel. If `fp` is NULL, all output streams are flushed.

  The output cannot be directly followed by input without an intervening `fflush` or `fseek`. The same is true in the other way.

12.4 Some examples

Let us see some examples. The following program opens a file and writes three strings.

```
#include <stdio.h>
int main() {
  FILE *fp;
  char *str[] = { "one" , "two" , "three" };
  int i;

  if ((fp = fopen("file.data","w")) == NULL) {
    perror("can not open");
    exit(1);
  }

  for(i = 0;i < 3;i++)
    if (fputs(str[i],fp) == EOF) {
      printf("can not write");
      exit(1);
    }
  exit(0);
}
```

There exists three predefined file pointers: `stdin`, `stdout` and `stderr`. These pointers can be used with all the functions shown here. As an example, the following program copies its standard input into its standard output:

```
#include <stdio.h>
int main() {
  int c;

  while ((c = fgetc(stdin)) != EOF)
    if (fputc(c,stdout) == EOF) {
      perror("can not write");
```

```
        exit(1);
      }
    exit(0);
  }
```

12.5 Binary I/O

Most of the functions shown operate with one character at the time or one line at a time. If we are doing binary I/O we would like to read or write an entire structure at a time. Two functions are provided for this purpose:

- `size_t fread(void *ptr,size_t size,size_t nobj,FILE *fp)`

  Reads `nobj` objects of size `size` from the file specified by `fp` and stores them in memory starting at the address pointed to by `ptr`. It returns the number of objects successfully read.

- `size_t fwrite(void *ptr,size_t size,size_t nobj,FILE *fp)`

  Writes `nobj` objects of size `size` into the file specified by `fp` copied from the memory address pointed to by `ptr`. It returns the number of objects successfully written.

We can write the elements 3 through 6 of a floating point array into a file by executing:

```
float data[10];

if (fwrite(&data[3],sizeof(float),4,fp) != 4)
  perror("can not write");
```

or write a complete structure as follows:

```
struct person {
  char name[10];
  int age;
};
struct person x;

if (fwrite(&x,sizeof(x),1,fp) != 1)
  perror("can not write");
```

There are two functions related to the file position:

- `long ftell(FILE *fp)`

  Returns the current file position.

- `int fseek(FILE *fp,long offset,int whence)`

  with the same semantic of the `lseek` function.

12.6 Formatted I/O

The formatted I/O functions allow to read or write from a file in a similar way as `scanf` and `printf` work with standard input and output. The information is always written in *ASCII* code into the file. The great advantage is that they are simple to use, and the files can be read directly with a text editor. The disadvantage is that files are usually bigger.

- `int fprintf(FILE *fp,char *format,...)`

  It works like `printf`, but the output goes to the file specified by `fp`.

- `int fscanf(FILE *fp,char *format,...)`

  It works like `scanf`, but the input data comes from the file specified by `fp`.

As an example, the following program writes into a file called `numbers` the integers from 0 to 5.

```
#include <stdio.h>

int main() {
  FILE *fp;
  int i;

  if ((fp = fopen("numbers","w")) == NULL) {
    perror("can not open");
    exit(1);
  }

  for(i = 0;i < 6;i++)
    if (fprintf(fp,"%d\n",i) != 1) {
      perror("can not write");
      exit(1);
    }
  exit(0);
}
```

If we list the contents of this file, we will see the numbers.

```
$ cat numbers
  0
  1
  2
  3
  4
  5
```

13 Bibliography

Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice Hall, 1988.

Leendert Ammeraal. *C for programmers.* John Wiley and Sons Ltd., 1986.

W. Richard Stevens. *Advanced programming in the Unix Environment.* Addison Wesley Professional Computing Series, 1992.

Tim Love. *ANSI C for Programmers on Unix Systems.* Cambridge University Engineering Department.

Available on ftp::/svr-ftp.eng.cam.ac.uk:misc/love_C.ps.Z.

Real Time Data Communication in Computer Networks

Fourth College on Microprocessor-based Real-time Systems in Physics

Trieste, 7 Oct–1 Nov 1996

Abhaya S Induruwa*
Dept of Computer Science & Engineering
University of Moratuwa
Moratuwa
Sri Lanka.

email: abhaya@cse.mrt.ac.lk

Abstract

These lectures are intended to help understand the computer network architecture comprising network protocols, standards, hardware and supporting technologies needed to perform real time data transfer. The IP architecture and its components used in real time data communication are discussed. IP/TV is illustrated as a real life example.

*At present at the Computing Laboratory, University of Kent at Canterbury, England.

1 Introduction

Computer networks are increasingly becoming an integral and indispensable part of scientific as well as public life. Over the last couple of decades data networks have changed their character from a slow speed point to point connection to a high speed data communication backbone supporting full multimedia information transfer.

Today the technology offers the possibility of merging Real Time applications such as voice and data acquisition services which are time sensitive, with time insensitive non Real Time services on a single network infrastructure. The largest Real Time Network in the world (also the oldest) is the telephone network which provides only 4 kHz bandwidth per voice channel. The newer architectures such as ISDN (Integrated Services Digital Network) and Broadband ISDN offer channel bandwidths of 64 Kbps and above. These higher bandwidths are suitable to carry either a number of basic voice channels or a single application requiring a larger bandwidth. Asynchronous Transfer Mode (ATM), a cell based transport technique has been developed to support the B-ISDN services.

RTP (Real time Transport Protocol), together with RTCP (Real time Transport Control Protocol), have been devised to facilitate the communication of Real Time data over computer networks, which have been designed and built to guarantee the delivery of time insensitive bursty data.

2 Network Classification

A computer network is a collection of computers interconnected by one or more transmission paths for the purpose of transfer and exchange of data between the computers. Today these networks span the entire globe and belong to many different nations and network operators. Such networks can be classified in many ways depending on the switching mechanism, transmission speed, etc [Black 93], [Stallings 94a], [Stallings 94b].

For the purpose of this discussion it is appropriate to classify them based on their:

a. geographical coverage

b. network topology.

2.1 Geographical Coverage

Networks can be classified into 4 categories depending on their geographical coverage (from the smallest to the largest) as follows:

1. Desktop Area Networks (DANs) [1]
2. Local Area Networks (LANs)
3. Metropolitan Area Networks (MANs)
4. Wide Area Networks (WANs).

There is a significant level of deployment of all or some of the above even in developing countries (most notably LANs and WANs with DANs just appearing) and hence should be of interest to research scientists.

2.2 Network Topology

Networks can be classified according to their topology in the following manner [Stallings 93], [Stallings 94b].

1. Bus Topology (eg. CSMA/CD; Ethernet)
2. Ring Topology (eg. Token Ring, FDDI)
3. Star Topology (eg. ArcNet, Switched networks)
4. Mesh Topology (eg. Telephone network).

The bus topology has been used initially in the Ethernet (broadcast) and later in Token Bus. Today it is important in the DQDB (Distributed Queue Dual Bus).

The token passing mechanism shown in Figure 1 has been originally used in the Token Ring and later in FDDI.

If however networks are categorised according to their transfer mechanism, then the following classification results.

1. Broadcast Networks

   Figure 2 shows a broadcast network which is used to broadcast from 1 to many. CSMA/CD (Carrier Sense Multiple Access/Collision Detect) is a medium access protocol which is used to broadcast on a bus topology.

[1] a recent classification arising out of delivering ATM to the desktop at 25 Mbps. DANs are used to interconnect devices such as camera, telephone and workstations.

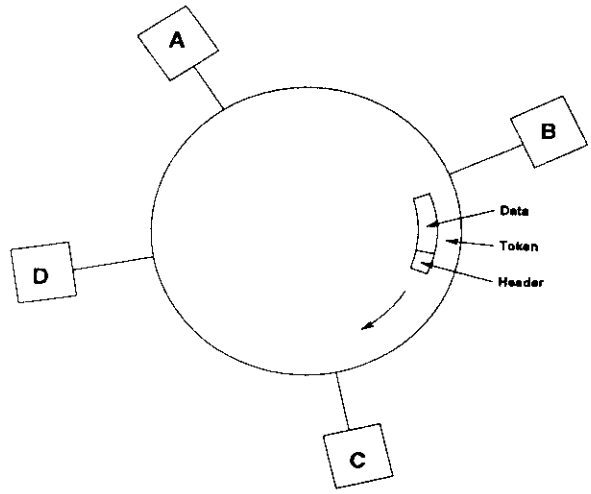


Figure 1: Token Passing Ring

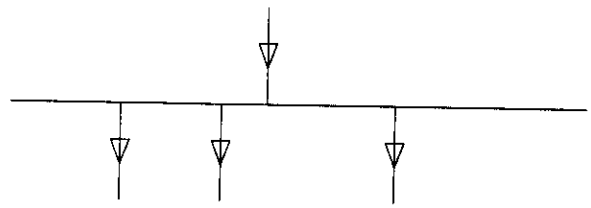


Figure 2: Broadcast Network

2. Switched Networks

   In Figure 3 is shown a switched network which is used to switch data from 1 to 1, 1 to many, or many to many. The telephone network is an example of a switched network. It can be used to support applications such as tele conferencing which involves the switching of many to many.

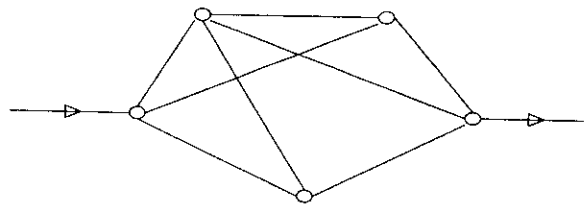


Figure 3: Switched Network

Although initially switched networks were mainly used in telecommunication networks, today because of its superior performance, switched technologies are used in LANs (for example switched Ethernet) and ATM networks.

3. Hybrid Networks

   Figure 4 shows a hybrid network which consists of a switched part and a broadcast part.

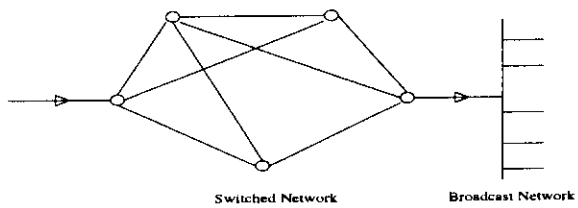


Figure 4: Hybrid Network

3 Network Architecture

The topology, transmission mechanism, and a protocol which manages the transmission mechanism together define a network architecture.

3.1 What is a Network Protocol?

A network protocol is used to facilitate the transmission of data between a sender (transmitter) and a recipient (receiver) across a data communication network in an agreed manner. Over the years several different network protocol architectures have evolved, the most notable being:

i. ISO - OSI (Open Systems Interconnect) Reference Model

ii. IP (Internet Protocol)

iii. ISO 8802.X (for LANs and MANs - same as IEEE 802.X)

In addition to the above there are hundreds of vendor specific protocols such as:

a. IBM's SNA (Systems Network Architecture)

b. DEC's DNA (DEC Network Architecture),

which are proprietary and hence are not truly interoperable.

Because todays networks span the entire globe, it is important to utilise standard protocols to facilitate seamless data communication over the networks belonging to different network operators to ensure interoperability.

This has been the major objective of the bodies involved in preparing standards, such as the International Standards Organisation (ISO), Internet Engineering Task Force (IETF), International Telecommunication Union (ITU), Institute of Electrical and Electronic Engineers (IEEE), American National Standards Institute (ANSI) and the ATM Forum.

Figure 5 shows the layered architectures of the protocols developed by the above standards bodies.

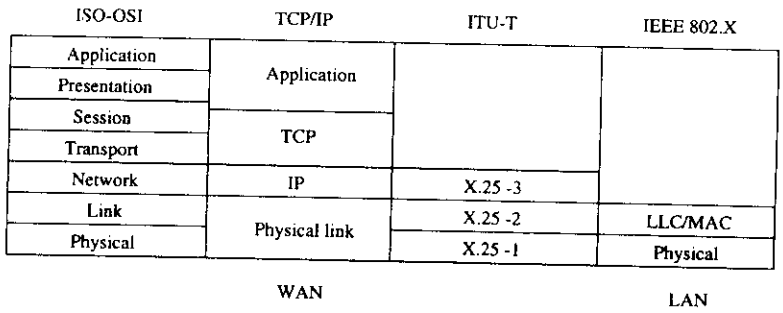


Figure 5: Layered Architectures of different Protocols

In Figure 6 is shown the architecture of the IEEE 802 family of standards for LANs and MANs. FDDI is a standard developed by ANSI-ASC X3T9 (Accredited Standards Committee) and provides services specified by the ISO Data link and Physical layers (ISO 9314).

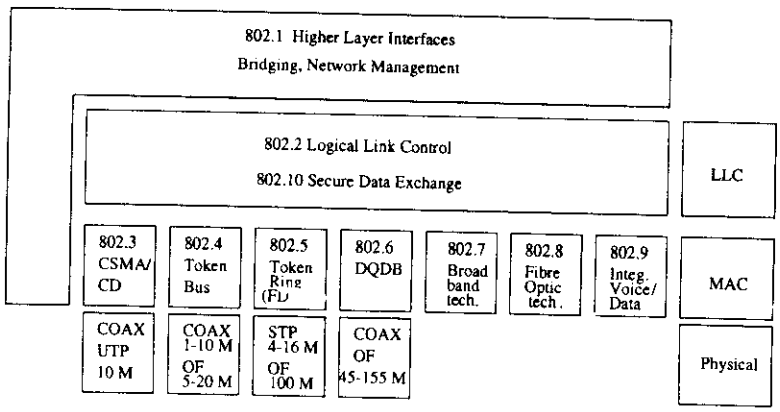


Figure 6: The Architecture of IEEE 802 LAN/MAN Standards

The key features of FDDI are 100 Mbps data rate, use of optical fibre (*multi mode fibre, single mode fibre, low cost fibre*), the token ring style protocol and the reconfiguration concept (automatic healing property in case of faults). The FDDI standard however allows the FDDI to be carried on other physical media such as twisted pair copper (CDDI).

IEEE 802.6 MAN standard specifies a DQDB (Distributed Queue Dual Bus) protocol which can support data, voice and video traffic. It can also serve as a LAN. DQDB MAN operates on a shared medium with two unidirectional buses that flow in opposite directions. Two methods of gaining access to the medium depending on the type of traffic have been specified. In the first method a node on the DQDB subnetwork can queue to gain access to the medium by using a distributed queue or by requesting a fixed bandwidth through a prearbitrated access method. Data is transmitted in fixed size units called slots of length 53 bytes (52 bytes data + 1 byte access control field).

DQDB private networks are connected to the public network by point to point links. A DQDB MAN can typically range upto more than 50 km in diameter and can operate at a variety of speeds ranging from 34 Mbps to 150 Mbps.

DQDB has been conceived to integrate data and voice over a common set of equipment, thereby reducing maintenance and administrative costs. B-ISDN is an attempt to provide universal and seamless connectivity for multimedia services. IEEE 802.6 MAN has been designed to provide an interim solution and to act as a migration path to B-ISDN.

Nodes in a DQDB subnetwork are connected to a pair of buses flowing in opposite directions and can operate in one of two topologies, namely; *open bus* (Figure 7) or *looped bus* (Figure 8) (open bus topology is similal to Ethernet and looped bus topology is simialr to token ring).

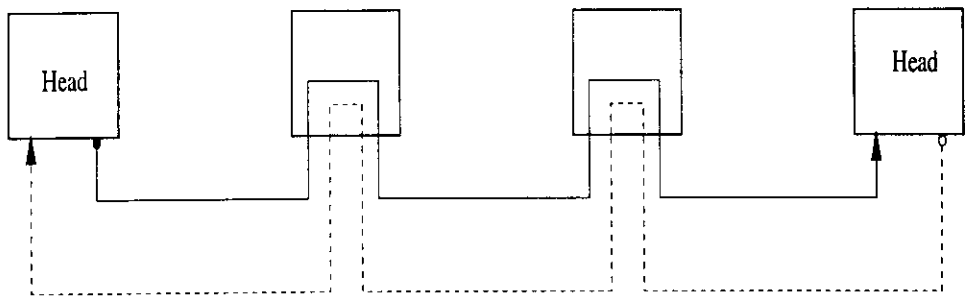


Figure 7: Open Bus DQDB Network

Although the DQDB access layer is independent of the physical medium, the speeds at which DQDB MANs operate demands the use of fibre or coaxial cables. For example, ANSI-DS3 operates at 44.736 Mbps over 75 Ω coax or fibre and ANSI SONET STS3 operates at 155.52 Mbps over single mode fibre. The ITU-T G.703 operates at 34.368 Mbps and 139.264 Mbps over a metallic medium.

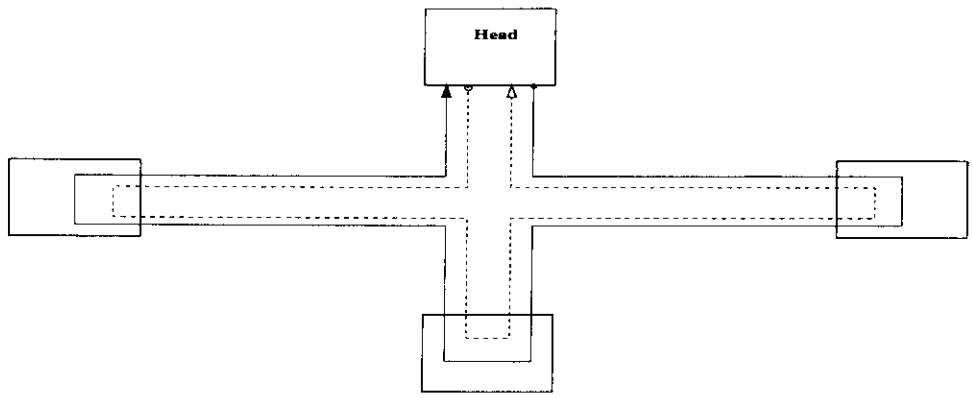


Figure 8: Looped Bus DQDB Network

A complete treatment of both the FDDI (Fibre Distributed Data Interface) and the DQDB (Distributed Queue Dual Bus) along with LANs and MANs is found in [Stallings 93], [Stallings 94b].

3.2 Transmission Mechanism

From the beginning, the voice data in a telephone network had been transmitted in real time using circuit switching techniques. Since circuit switching is not an efficient transmission mechanism for data communication, packet switching techniques have been devised.

3.2.1 Packet Switching

Transmission mechanisms based on packet switching allows the multiplexing of data packets from different sources on the same transmission path thereby making use of the channel bandwidth more efficiently. It also allows the transmission of packets from one source along different paths thus taking care of line congestion and availability problems. When the packets reach the intended destination in whatever path they may have taken, they are reassembled and presented to the user application.

The most widely used protocols which manage the packet transfer across a data network belong to the family of standards conceived by the ITU-T (X.25) and Internet Architecture Board (IP). However the variable packet lengths and the multiplexing technique introduce jitter making their Quality of Service (QOS) unacceptable for real time applications. The inherent limitation of X.25 in high speed data transfer has been removed to some extent in the Frame Relay [Smith 93] transfer mechanism.

3.2.2 Frame Relay

Frame Relay offers a high speed version of packet switching and has the potential of operating effectively at much higher speeds compared to X.25, reaching speeds of 45 Mbps. Frame relay is well suited to high speed data applications, but not suited for delay sensitive applications such as voice and video because of the variable length of frames [Smith 93], [Stallings 95].

3.2.3 Cell Switching

Cell Relay is a transmission mechanism that combines the benefits of time division multiplexing with packet switching. It operates on the packet switching principle of statistically interleaving cells on a link on an 'as required basis', rather than on a permanently allocated time slot basis. The fixed cell size used enables a reasonably deterministic delay to be achieved across a network. This deterministic nature of cell relay makes it suitable for all traffic types within a single network.

ATM (Asynchronous Transfer Mode) is fast becoming the dominant form of cell relay. It uses a cell of 53 bytes long (5 bytes header and 48 bytes data) and typically operates at speeds of 155 and 622 Mbps. ATM is delivered to the desktop at 25 Mbps and Gbps platforms are being tested [Partridge 94]. ITU-T has selected ATM as the transport technique for B-ISDN (Broadband Integrated Services Digital Network) [De Prycker 95], [Handel et al 94], [Stallings 95].

3.3 Physical Media

The most popularly used physical media are:

- Optical fibre cables
- Coaxial cables
- Unshielded Twisted Pair (UTP) cables
- Shielded Twisted Pair (STP) cables

Today all of the above media are used to carry data in excess of 100 Mbps speeds. Only the distances they cover are different (for example, optical fiber can operate at gigabit speeds for a few km whereas UTP can operate at 100 Mbps for a 100 m without repeaters).

4 Internetworking

By internetworking is meant the process of interconnection of computers and their networks to form a single internet. The Internet (note the uppercase 'I') is such an internet and uses TCP/IP (Transmission Control Protocol /Internet Protocol) protocol suite.

TCP is connection oriented and provides a reliable stream transport. Although TCP is commonly associated with IP (as its underlying protocol), it is an independent, general purpose protocol that can be adapted to use with other delivery systems. The popularity of TCP has resulted in ISO – TP4, which has been derived from TCP. TCP, together with IP, provides a reliable stream delivery for data traffic.

TCP/IP protocol suite has become the defacto standard for open system interconnection in the computer industry. It is used world wide in academic, government, private and public institutions. Some of the reasons for its wide acceptance can be attributed to the following:

- It provides the highest degree of interoperability.
- It encompasses the widest set of vendors' systems.
- It runs over more network technologies that any other protocol suite.

Over the years Unix (and hence linux) and TCP/IP have become almost synonymous. Today it has become part of the operating system kernel. The OS runs a separate process for IP, TCP input/output and UDP input/output.

In building internets, following hardware devices used to interconnect networks are of interest [Black 93], [Comer 88].

1. Repeaters
2. Bridges
3. Routers
4. Gateways.

Figure 9 shows the use of these components in relation to the ISO-OSI Reference Model.

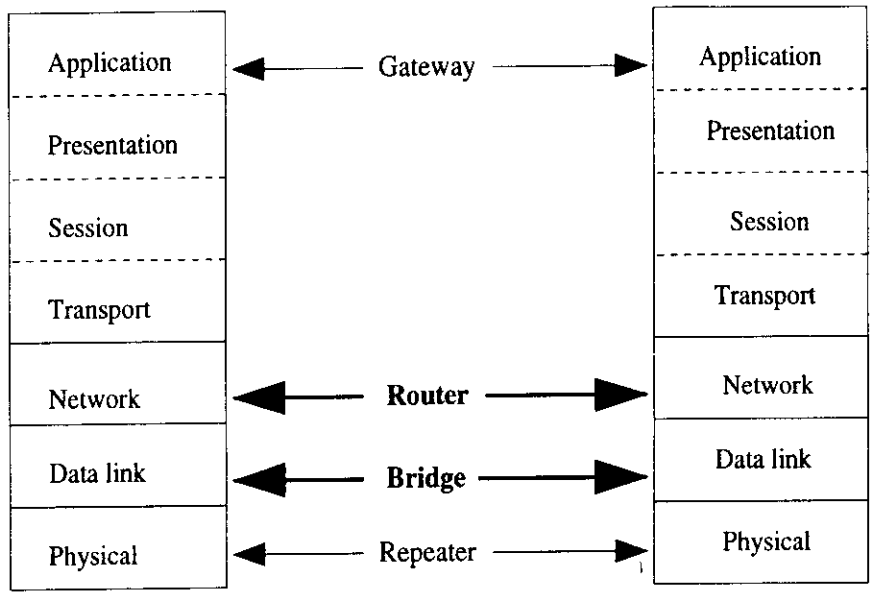


Figure 9: Interconnection Components

4.1 Repeaters

When two networks are to be connected at the lowest level, ie. the physical level, an interconnecting device known as a repeater is used. A repeater simply takes bits arriving from one network and repeats them on to the other. In some cases a repeater might have to translate between two different physical layer formats, for example from optical fiber to UTP (Unshielded Twisted Pair) cable. This may involve some processing of the received signal such as signal regeneration for noise elimination. Repeaters pass on the data received without paying attention to the address information.

4.2 Bridges

Bridges are used to interconnect networks at the medium access control (MAC) layer. Typically this requires the interconnected networks to have identical MAC layers although networks with different but related MAC layers can be interconnected. Since Bridges operate at the MAC layer, they can be used to effectively segment the traffic by filtering the traffic entering to one segment from another thereby reducing the unwanted traffic flow on network segments.

Unlike a repeater which replicates electrical signals, bridges replicate packets. They are superior in their function, because they do not replicate noise and errors or malformed frames. Moreover, bridges implement

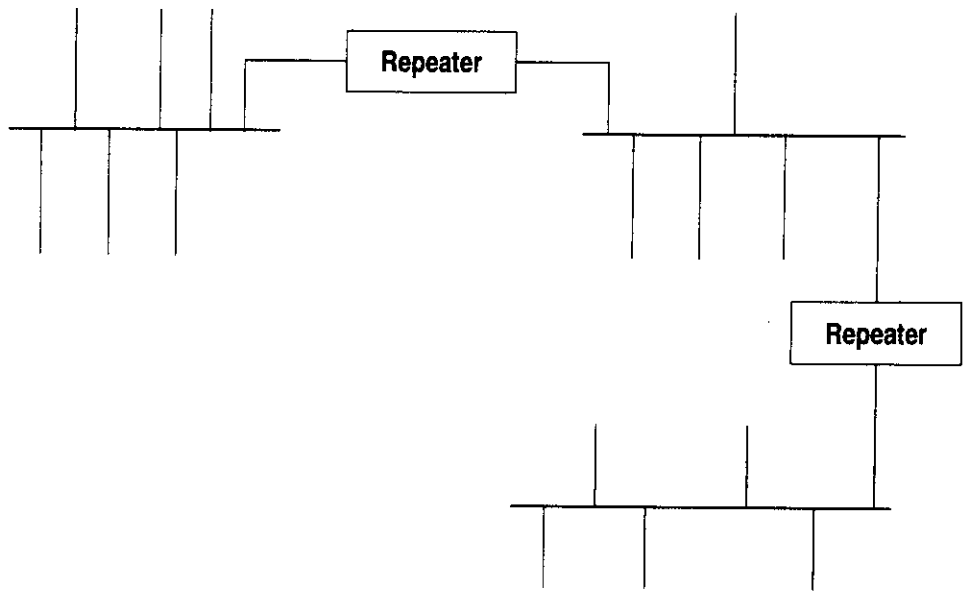


Figure 10: Use of Repeaters

CSMA/CD rules and hence collisions and propagation delays remain isolated without affecting the other segments. As a result an almost arbitrary number of Ethernet segments can be connected together with bridges whereas with repeaters the maximum number of segments is five giving a total length of 2.5 km. Since bridges hide details of interconnection, a set of bridged segments acts like a single Ethernet.

A bridge can be used to make a decision on which frames to forward from one segment to another. Such bridges are called *adaptive* or *learning* bridges. They learn over time, which hosts are connected to which segment. Thus an adaptive bridge builds up the address table automatically without human intervention.

Bridges are often used to improve the performance of an overloaded network by effectively partitioning the network into segments.

The type of bridges used in CSMA/CD LANs are known as *transparent bridges* (IEEE 802.1D) since their presence is not visible to the stations. The type of bridges used to interconnect token rings are called *source routing bridges* (IEEE 802.5) and the routing information is provided by the source station.

4.3 Routers

Routers interconnect networks at the network layer (level 3 in the OSI-RM and the IP layer of the TCP/IP suite) and perform routing functions. This is the main building block in internetworking using IP. Most routers now support at least one of the multicast routing protocols, which is an essential functionality to support the delivery of Real Time data over IP.

4.4 Gateways

Gateways are used when networks have to be interconnected at layers higher than the network layer and when protocol translation is necessary. A typical example is interconnecting two networks based on TCP/IP suite and OSI suite of protocols. An application level gateway is required to support FTP on TCP/IP and FTAM on OSI. From this it should be clear that a different application level gateway is required for every application supported across the interconnected networks.

4.5 Multiport-Multiprotocol Devices

The multiplicity of transmission media and protocols used in today's networks require the use of multiport repeaters and bridges, as well as multiprotocol routers which support more than one protocol stack.

5 A word about the Internet

The one and only global network of interest to the whole scientific community of the world today is the Internet, which is based on the Internet Protocol (IP).

Internet Protocol is a truly scalable protocol used to connect computers to small LANs and to WANs forming a global internetwork. IP is available for almost all computing platforms ranging from the smallest laptop to the largest ultrasuper computer.

The Internet is an ever expanding network of networks. As of this writing (September '96), it interconnects 13 million computers and 135,000 computer networks in 154 countries. It experiences a staggering growth rate of 100% per year. The Internet connectivity in the world is shown in Figure 11.

Due to this unprecedented popularity and the growth of usage of IP, today IP suite is included as a standard component of all UNIX and UNIX like (and hence linux) distributions, making the networking of any computer

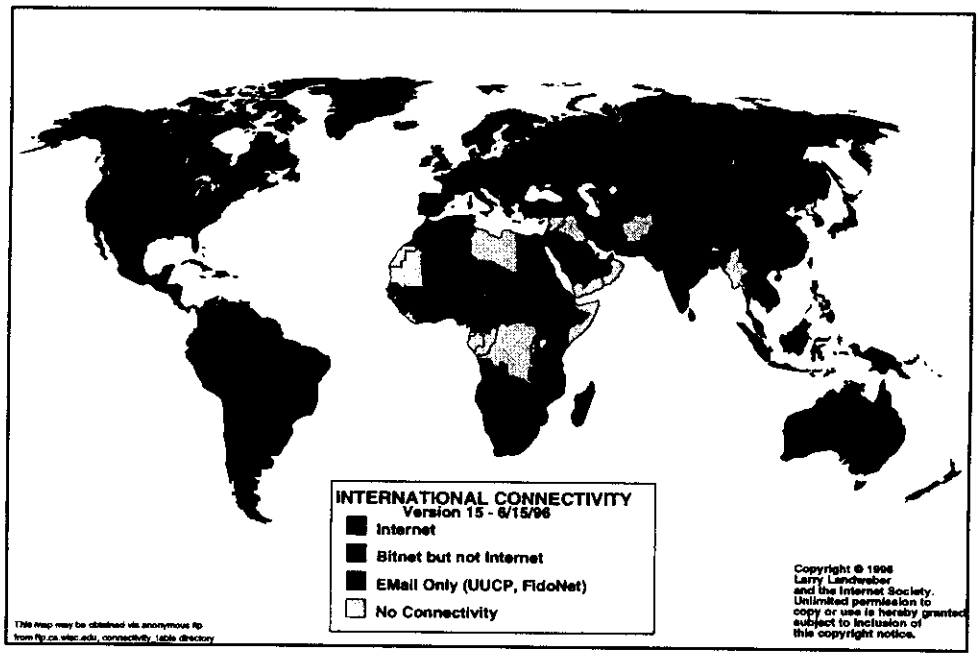


Figure 11: Worldwide Internet Connectivity

running linux much easier. IP is also incorporated into the OSI-RM thus ensuring compliance with ISO standards.

6 Internet Protocol Architecture

By far the most successful and the most widely used protocol architecture for LANs and WANs alike is the Internet Protocol (IP) [Black 93], [Comer 88], [Stallings 89].

IP is a layered architecture (see Figure 4) consisting of only 4 layers (ISO-OSI RM has 7 !). TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are the two transport protocols supported over IP. Both the TCP and UDP make the assumption that the link is of acceptable reliability (measured in terms of its BER) and is capable of delivering a packet to the intended destination without the intervention of the upper layers. Some of the services supported in TCP/IP, including RTP are shown in Figure 12.

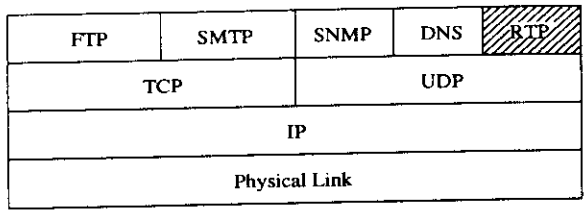


Figure 12: Services Supported Over IP

6.1 IP Addressing

IPv4 uses 5 classes of addresses as shown in Figure 13.

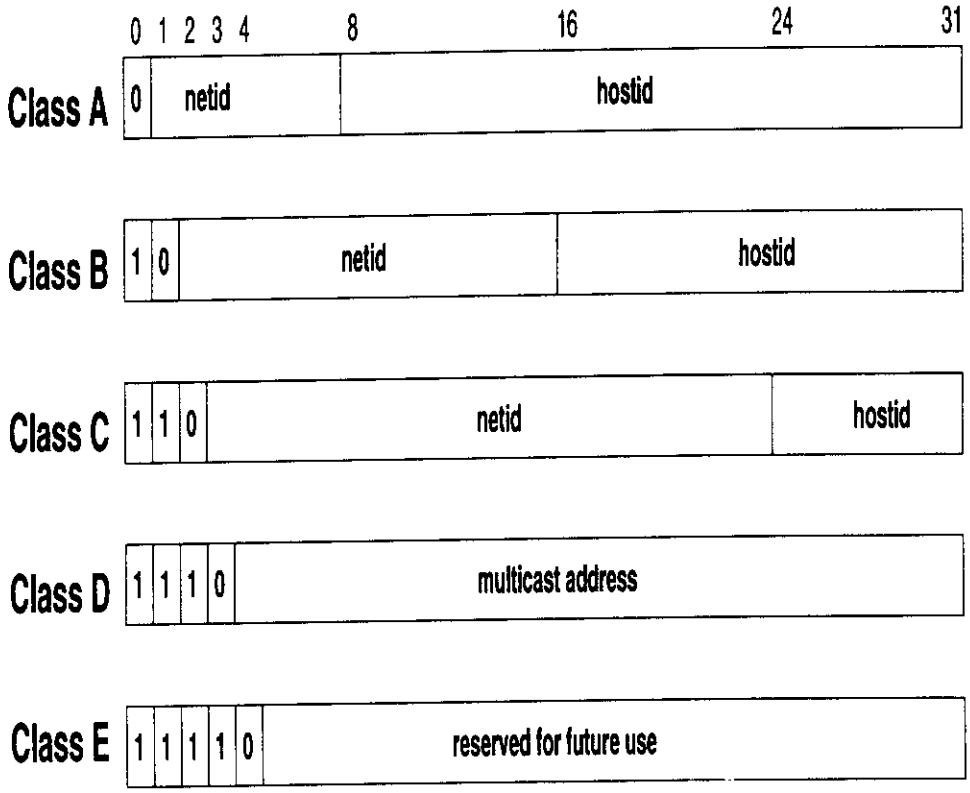
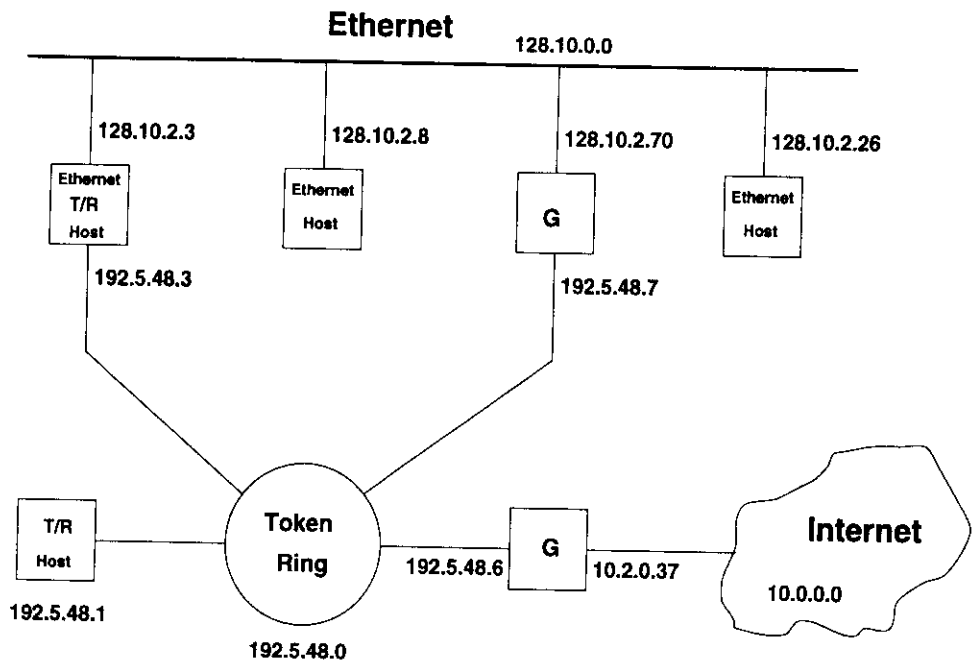


Figure 13: IP Addresses

An example of IP address allocation in a typical network is shown in figure 14.

Many hosts have a host name associated with the IP address which can be used to address them. However it must be understood that an IP address does not identify a host. It identifies a network connection to a host because an IP address encodes both a network and a host on that network. Hence if the host is moved to another network, its IP address has to be changed.



One address for each network / segment

Internet	**10.0.0.0**	**(Class A)**
Ethernet	**128.10.0.0**	**(Class B)**
Token Ring	**192.5.48.0**	**(Class C)**
Network Hosts		
	128.10.2.8	
	192.5.48.1	
	128.10.2.3	
	192.5.48.3	

Two addresses for each gateway.

Figure 14: IP Addresses on an internet

No two devices on the global Internet should be allocated the same IP address, although on a private network with no connection to the outside world, arbitrary addresses can be allocated. The address allocation has been initially handled by the Network Information Centre (NIC). Now it is handled by the NIC as well as RARE in Europe and APNIC in the Asia- Pacific region. The addresses must be officially obtained from one of the above before using them on your network.

Nowadays there are Internet Service Providers (ISPs) in many countries (most of the time more than one !) who are allocated blocks of IP addresses by the relevant NIC. These ISPs in turn allocate the IP addresses to their customers.

The explosive growth of the Internet has resulted in the exhaustion of the address space provided in IPv4 which uses a 32 bit (4 byte) address. IPng (IP new generation), also known as IPv6, has been designed to provide among other things an enhanced address space using 16 bytes (128 bits) [RFC 1883].

6.2 The Internet Protocol

The Internet Protocol is a connectionless network layer protocol. TCP is a higher layer protocol which sits on top of IP. IP provides a connectionless (or datagram type) service to its user. In other words data given to IP (by the higher layer) is not guaranteed to be delivered.

The IP datagrams are the encapsulation of data packets passed from the higher layer with an IP header as shown in Figure 15.

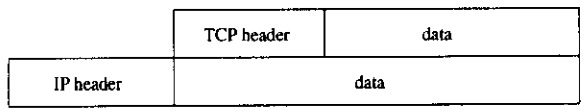


Figure 15: IP Datagram

6.3 Internetworking with IP

Figure 16 shows how IP used to internetwork two networks running different medium access control mechanisms namely, CSMA/CD and Token Ring.

6.4 IP Datagram Format

The IP datagram format which has a preamble of ten 16 bit words and an option field of variable length is shown in Figure 17.

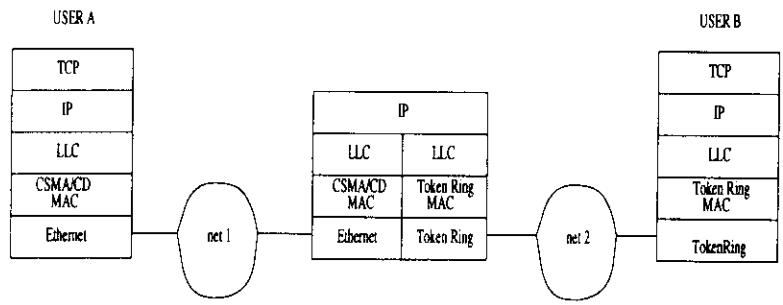


Figure 16: Internetworking Using IP

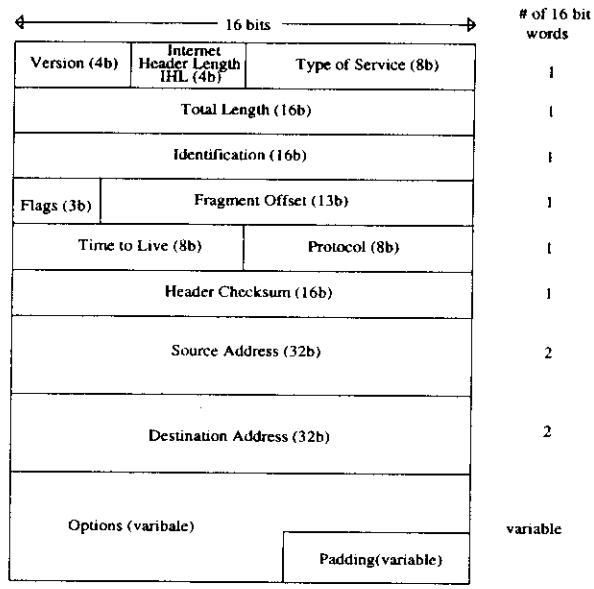


Figure 17: IP Datagram Format

6.5 Brief Description of TCP and UDP

TCP and UDP are the two transport protocols used in the IP architecture [Comer 88], [Stallings 89].

TCP is a connection oriented transport protocol designed to work in conjunction with IP. TCP provides its user (application layer) with the ability to transmit reliably a byte stream to a destination and allows for multiplexing multiple TCP connections within a transmitting or receiving host computer.

Being connection oriented, TCP requires a connection establishment phase (like dialing a number to make a phone call) which is followed by the data transmission phase. A connection is terminated when it is no longer in use. TCP/IP is ideal for the transmission of bursty data. It works on the principle of retransmission of dropped packets which is one of the major contributors

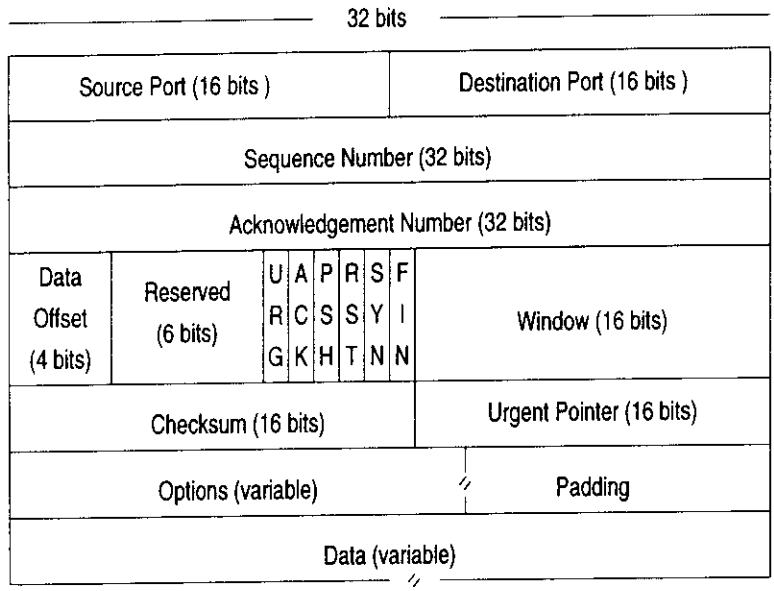


Figure 18: TCP packet format

to delays in transmission. However, since voice and video data are time sensitive, packet technologies such as TCP/IP cannot guarantee the proper delivery of such data. Figure 18 shows the TCP header format.

UDP, on the other hand, is a connectionless transport protocol designed to operate over IP. Its primary functions are error detection and multiplexing. UDP does not guarantee the delivery of packets (compare with the ordinary postal service) but guarantees that if a packet is ever delivered in error, such error will be detected (use of checksum). It also allows for communicating with multiple processes residing on the same host computer.

UDP packet format is simple (see Figure 19). It is also fast compared to the use of TCP, since there is no connection establishment phase. Moreover, UDP is important since RTP (Real time Transport Protocol) is supported over UDP.

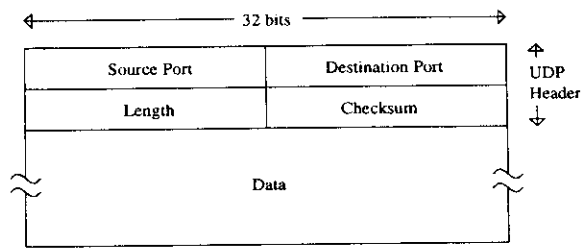


Figure 19: UDP Packet Format

6.6 IP Multicasting

Multicasting is important in allowing a stream of data to be sent efficiently to many receivers. In multicasting, rather than sending a separate stream of data packets to each intended user (unicasting) or transmitting all packets to everyone (broadcasting), a stream is transmitted simultaneously to a designated subset of network users. The concept of multicasting is shown in Figure 20.

IP Multicasting is defined in [RFC 1112] of 1989. A key to IP multicasting is the Internet Group Management Protocol (IGMP). IGMP enables users to sign up for multicast sessions and allows these multicast groups to be managed dynamically, in a distributed fashion. Enhancements have been done to existing protocols to direct the traffic to the members of the group. These include Distance Vector Multicasting Routing Protocol (DVMRP) and Multicast Open Shortest Path First (MOSPF) protocol. An entirely new protocol developed specifically for multicasting is the Protocol Independent Multicast (PIM).

At the start of a multicast session group addresses are allocated which are relinquished at the end of that session and reused later.

Internet MBONE is Internet's multicast backbone which is a collection of multicast routers that can distribute multicast traffic. MBONE participants use class D Internet addresses which identify a group of hosts rather than individual hosts.

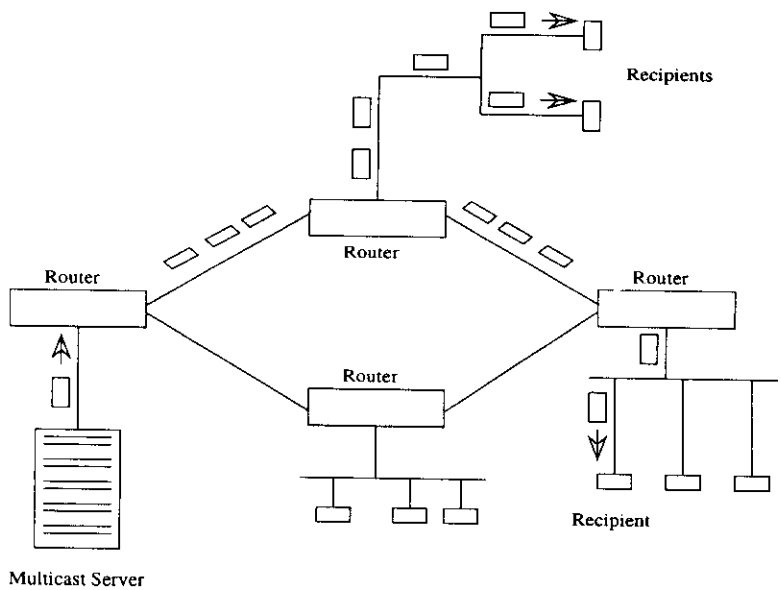


Figure 20: Multicasting

6.7 Resource Reservation Protocol (RSVP)

A key factor in achieving real time quality of service is a reservation set up protocol, a mechanism for creating and maintaining flow specific state information in the end point hosts and in routers along the data flow path. The IETF has developed its Resource Reservation Protocol (RSVP) specifically for the packet switched multicast environment.

RSVP has been designed to meet a number of requirements:

1. support for heterogeneous service needs;
2. flexible control over the way reservations are shared along branches of multicast delivery trees;
3. scalability to large multicast groups;
4. and the ability to preempt resources to accommodate advance reservations.

The RSVP protocol basically acts according to its name. An RSVP request specifies the level of resources to be reserved for some or all of the packets in a particular session. An application requests resources by specifying a flow specification, which describes the type of traffic anticipated (for example, average and peak bandwidths and level of burstiness), and a resource class specifying the type of service required (such as guaranteed delay). A filter specification is also specified, which determines the sources to which a given reservation applies.

RSVP mandates that a resource reservation be initiated by the receiver rather than the sender. While the sender knows the properties of the traffic stream it is transmitting, it has been found that the sender initiated reservation scales poorly for large, dynamic multicast delivery trees. Receiver initiated reservation deals with this by having each receiver request a reservation appropriate to itself; differences among heterogeneous receivers are resolved within the network by RSVP. After learning sender's flow specification via a higher level "out of band mechanism", the receiver generates its own desired flow specification and propagates it to senders, making reservations in each router along the way.

RSVP itself uses a connectionless approach to multicast distribution. The reservation state is cached in the router and periodically refreshed by the end station. If the route changes, these refresh messages automatically install the necessary state along the new route.

RTP and RTCP information is simply data from the point of view of routers that move the packets to their destination. RSVP prioritises multimedia traffic and provides a guaranteed quality of service. Routers that have

been upgraded to support RSVP can reserve carrying capacity for video and audio streams and prevent unpredictable delays that would interfere with their transmission.

7 Data Communication in Real Time

Packet switching techniques based on ITU-T X.25 and IP have been traditionally used for non Real Time data transfer. Since they do not guarantee packet sequence integrity and consistent latency times in delivery, they are inherently unsuitable for Real Time applications.

The following are required to carryout Real Time data transfer on existing networks.

1. Enough bandwidth for extremely dense audio and video traffic.
2. A transport protocol appropriate for the streaming requirements of real time data (RTP).
3. A protocol to reserve network bandwidth and assigning priorities for various traffic (RSVP).

7.1 RTP Data Transfer Protocol

A Real time Transport Protocol is therefore needed to provide end to end network transport functions suitable for applications communicating in real time. Such applications include transmission of interactive audio and video data or real time simulation data over multicast or unicast network services.

The largest (and the oldest) network which supports real time data communication is the telephone network which falls in to the category of a circuit switched network. However in terms of a network protocol there is not much. Once the network connection is established the communication process is largely in the hands of the two persons communicating with each other.

In view of this a Real time Transport Protocol (RTP) along with a profile for carrying audio and video over RTP were defined by the IETF in January 1996 [RFC 1889], [RFC 1890].

7.1.1 Characteristics of RTP

The Realtime Transport Protocol has the following characteristics.

i. Payload type identification

ii. Sequence numbering

iii. Time stamping

iv. Delivery monitoring.

Real Time applications typically run RTP on top of UDP to make use of its multiplexing and checksum services (see Figure 21). Tailoring RTP to the application is accomplished through auxiliary profile and payload format specifications. A payload format defines the manner in which a particular payload, such as an audio or video encoding, is to be carried in RTP. A profile assigns payload type numbers for the set of payload formats that may be used in the application.

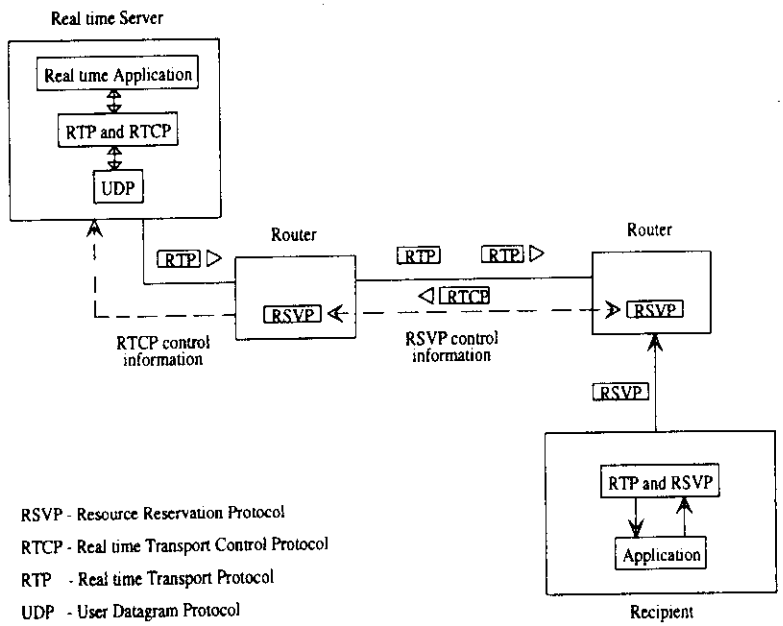


Figure 21: Real Time Application running RTP on top of UDP

However RTP is not limited to be used with UDP/IP. It can be used equally with other underlying network or transport protocols such as ATM or IPX. Moreover, RTP supports data transfer to multiple destinations using multicast distribution if provided by the underlying network, a feature which makes RTP ideal for multi party multimedia conferencing.

RTP is designed to work in conjunction with RTCP (Real time Transport Control Protocol) to monitor the quality of service. RTP delivers real time traffic with timing information for reconstruction as well as feedback on reception quality. The Resource Reservation Protocol (RSVP) is used to reserve network bandwidth and assign priority for various traffic types.

7.1.2 Definitions in RTP

The following definitions are extracts from [RFC 1889, RFC 1890].

- RTP Payload

  The data transported by RTP in a packet, for example audio samples or compressed video data.

- RTP Packet

  A data packet consisting of the fixed RTP header, a possibly empty list of contributing sources, and the payload data. Typically one packet of the underlying protocol contains a single RTP packet, but several RTP packets may be contained if permitted by the encapsulation method.

- RTCP Packet

  A control packet consisting of a fixed header part similar to that of RTP data packets, followed by structured elements that vary depending upon the RTCP packet type. Typically multiple RTCP packets are sent together as a compound RTCP packet in a single packet of the underlying protocol. This is enabled by the length field of the fixed header of each RTPC packet.

- Port

  The "Abstraction" that transport protocols use to distinguish among multiple destinations within a given host computer (TCP/IP protocols identify ports using small positive integers and the transport selectors (TSEL) used by the OSI Transport layer are equivalent to ports). RTP depends on the lower layer protocol to provide some mechanism such as ports to multiplex the RTP and RTCP packets of a session.

- Transport Address

  The combination of a network address and port that identifies a transport level end point, for example an IP address and a UDP port. Packets are transported from a source transport address to a destination transport address.

- RTP Session

  The association among a set of participants communicating with RTP. For each participant, the session is defined by a particular pair of destination transport addresses consisting of one network address and a port pair for RTP and RTCP. The destination transport address pair

may be common for all participants, as in the case of IP multicast, or may be different for each, as in the case of individual unicast network addresses plus a common port pair. In a multimedia session, each medium is carried in a separate RTP session with its own RTCP packets. The multiple RTP sessions are distinguished by different port number pairs and/or different multicast addresses.

- Synchronisation Source (SSRC)

  The source of a stream of RTP packets, identified by a 32 bit numeric SSRC identifier carried in the RTP header so as not to be dependent upon the network address. Examples of synchronisation sources include the sender of a stream of packets derived from a signal source such as a microphone, a camera or an RTP mixer. If a participant generates multiple streams in one RTP session, for example from separate video cameras, each must be identified as a different SSRC.

- Contributing Source (CSRC)

  A source of a stream of RTP packets that has contributed to the combined stream produced by an RTP mixer. The mixer inserts a list of the SSRC identifiers of the sources that contribute to the generation of a particular packet into the RTP header of that packet. This list is called the CSRC list. An example application is audio conferencing where a mixer indicates all the talkers whose speech was combined to produce the outgoing packet, allowing the receiver to indicate the current talker, even though all the audio packets contain the same SSRC identifier (that of the mixer).

- End System

  An application that generates the content to be sent in RTP packets and/or consumes the content of received RTP packets.

- Mixer

  An intermediate system that receives RTP packets from one or more sources, possibly changes the data format, combines the packets in some manner and then forwards a new RTP packet. Since the timing among multiple input sources will not generally be synchronised, the mixer will make timing adjustments among the streams and generate its own timing for the combined stream. Thus all data packets originating from a mixer will be identified as having the mixer as their synchronisation source.

- Monitor

  An application that receives RTCP packets sent by participants in an RTP session, in particular the reception reports, and estimates the current quality of service, fault diagnosis and long term statistics.

7.1.3 RTP Fixed Header Fields

The RTP header format is shown in Figure 22. The first twelve octets are present in every RTP packet, while the list of CSRC identifiers is present only when inserted by a mixer.

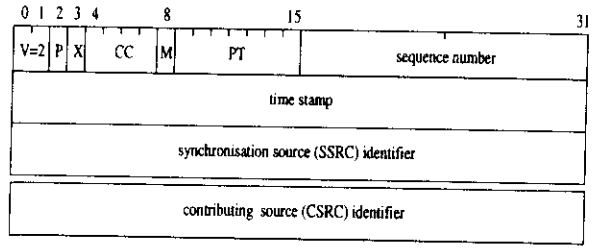


Figure 22: RTP Header Format

The RTP header provides the timing information necessary to synchronise and display audio and video data and to determine whether packets have been lost or arrive out of order. In addition, the header specifies the payload type, thus allowing multiple data and compression types. This is a key advantage over most proprietary solutions, which specify a particular type of compression and thus limit users' choice of compression schemes.

7.1.4 Multiplexing RTP Sessions

In RTP, multiplexing is provided by the destination transport address (network address and a port number) which define an RTP session.

7.1.5 Real time Transport Control Protocol (RTCP)

The RTCP is based on the periodic transmission of control packets to all participants in the session, using the same distribution mechanism as the data packets. The underlying protocol must provide multiplexing of the data and control packets, for example using separate port numbers with UDP.

The primary function of RTCP is to furnish information on the quality of data distribution. This feedback is a critical part of RTP's use as a transport protocol, since applications can use it to control how they behave. The feedback is also important for diagnosing distribution faults. For instance, by

monitoring reports from all data recipients, network managers can determine the spread of a problem. When used in conjunction with IP multicast, RTCP enables the remote monitoring and diagnosis.

In addition RTCP controls the rate at which participants in an RTP session transmit RTCP packets. In a session with a few participants, RTCP packets are sent at the maximum rate of one every five seconds whereas for a larger group, RTCP packets may be sent only once every 30 seconds. In other words, the more participants there are in a conference, the less frequently each participant sends RTCP packets. This makes RTCP scalable to accommodate tens of thousands of users.

7.2 Real Time Data Transfer using ATM

Audio and video applications generate lots of bits, and the traffic has to be streamed or transmitted continuously rather than in bursts. This is in contrast to conventional data types such as text, files and graphics, which are able to withstand short and inconsistent periods of delay between packet transmissions. What is needed then is a network capable of transporting both streaming and bursty data. ATM (Asynchronous Transfer Mode) is a technique which just does this [De Prycker 95], [Stallings 95].

ATM is a connection oriented protocol designed to support high bandwidth, low delay (even services with predictable delay), packet like switching and multiplexing. The design of ATM ensures the capability to carry both stream traffic (such as voice and video) and bursty traffic (such as interactive data). It uses a fixed cell size for all types of traffic. In the case of stream traffic ATM guarantees the integrity of cell sequence which is essential for the successful delivery of such traffic.

ATM has grown out of the need for a worldwide standard to allow interchange of information regardless of the "end system" or type of information. Historically there have been separate methods used for the transmission of information among users on LANs and the users on WANs. This situation has been made more complex by the user's need for connectivity expanding from the LAN to MAN to WAN. ATM is a method to unify the communication of information on LANs and WANs (Figure 23).

ATM is the only technology based on standards, and has been designed from the beginning to accommodate the simultaneous transmission of data, voice and video. It is an easily scalable backbone which can be upgraded merely by adding more switches or links. ATM is switched instead of routed and therefore it is faster since not every IP packet at every node is examined to determine its destination. ATM LAN Emulation (LANE) allows transparent interconnection of "legacy" LANs based on Ethernet or FDDI

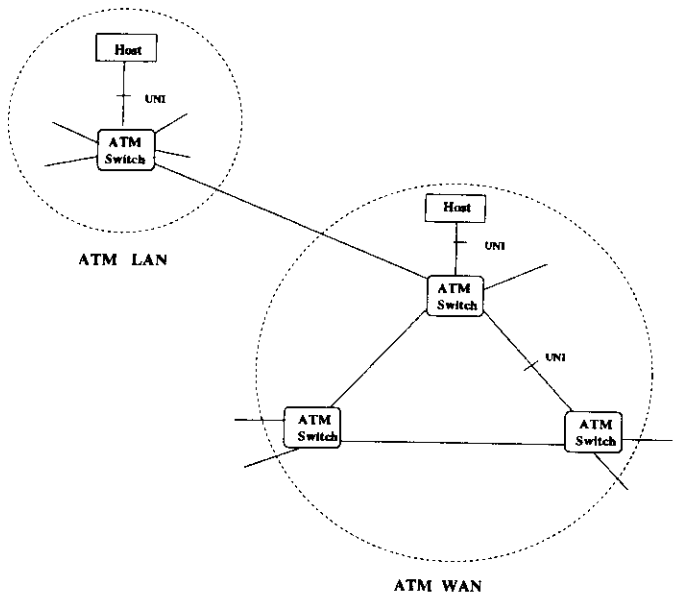


Figure 23: ATM LAN-WAN Connectivity

technology, making the ATM backbone look like a fast Ethernet or FDDI to workstation applications. As more and more ATM nodes are deployed, the differences between local and wide area networks will disappear to form a seamless network based on one standard.

To use the limited bandwidth more efficiently, ATM uses circuit switching principles to give the users a full channel to themselves. Since these users do not use the full channel all the time, ATM uses statistical analysis to time division multiplex several users onto the same line. This allows each user to have all of the channel's bandwidth for the period of time in which it is needed.

To achieve this ATM uses two connection concepts; the *Virtual Channel (VC)* and the *Virtual Path (VP)*.

A virtual channel (also known as a virtual circuit) provides a logical connection between end users and is identified by a VCI (Virtual Channel Identifier) in the ATM header (see Figure 24). A virtual path defines a collection of virtual circuits traversing the same path in the network and is identified by a VPI (Virtual Path Identifier). The VPI emulates the functions of the trunk concept in circuit switching. Thus virtual paths define the cross connection functions across the network, whereas virtual channels are concerned with switching and connection establishment functions. Virtual paths are statistically multiplexed on the physical link on a cell multiplexing basis.

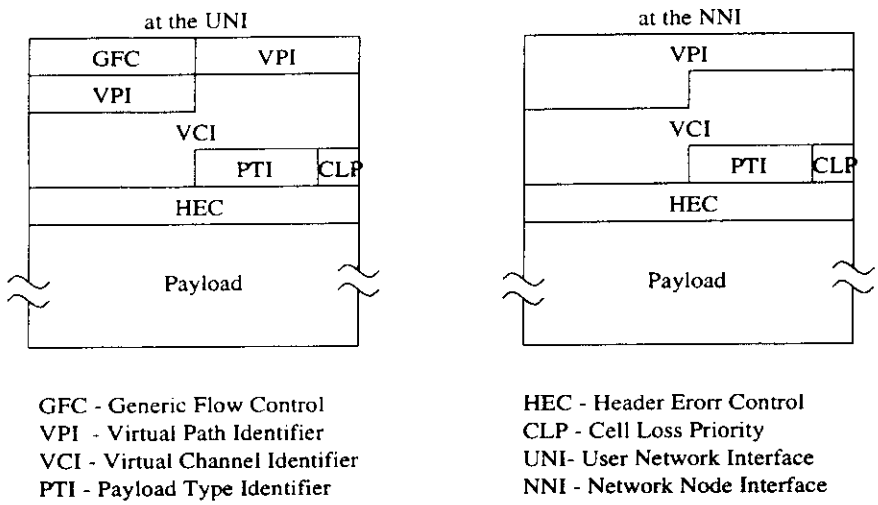


Figure 24: ATM Cell Formats

GFC (Generic Flow Control) is used to control the amount of traffic entering the network.

VPI and VCI are used for routing. VPI will change from one node to the next when it travels through the ATM layer. VCI is predefined and usually remains the same throughout the duration of the transmission. PTI (Payload Type Identifier) is used to distinguish between cells that are carrying user data and those carrying control information. CLP is a single control bit which provides selective discard during network congestion and HEC is used to check header errors.

The ATM cell formats used at the UNI (User-Network Interface) and NNI (Network-Node Interface) are shown in Figure 24.

7.2.1 ATM Protocol Structure

Figure 25 shows the ATM layered architecture as described in ITU-T recommendation I.321 (1992). This is the basis on which the B-ISDN Protocol Reference Model has been defined.

- ATM Physical Layer

  The physical layer accepts or delivers payload cells at its point of access to the ATM layer. It provides for cell delineation which enables the receiver to recover cell boundaries. It generates and verifies the HEC field. If the HEC cannot be verified or corrected, then the physical layer will discard the errored cell. Idle cells are inserted in the transmit direction and removed in the receiving direction.

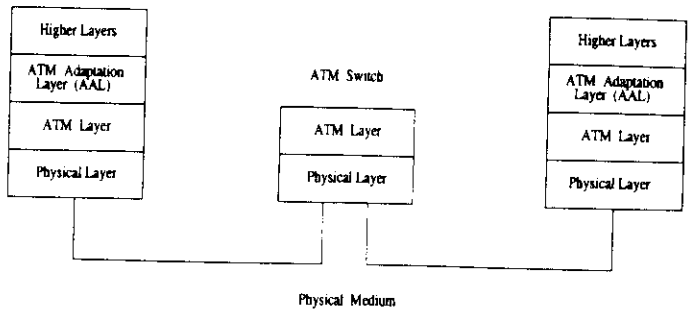


Figure 25: ATM Protocol Architecture

For the physical transmission of bits, 5 types of transmission frame adaptations are specified (by the ITU and the ATM Forum). Each one of them has its own lower bound or upper bound for the amount of bits it can carry (from 12.5 Mbps to 10 Gbps so far).

1. Synchronous Digital Hierarchy (SDH) $\geq$ 155 Mbps;
2. Plesiochronous Digital Hierarchy (PDH) $\leq$ 34 Mbps;
3. Cell Based $\geq$ 155 Mbps;
4. Fibre Distributed Data Interface (FDDI) = 100 Mbps;
5. Synchronous Optical Network (SONET) $\geq$ 51 Mbps.

The actual physical link could be either optical or coaxial with the possibility of Unshielded Twisted Pair (UTP Category 3/5) and Shielded Twisted Pair (STP Category 5) in the mid range (12.5 to 51 Mbps).

- ATM Layer

  ATM layer mainly performs switching, routing and multiplexing. The characteristic features of the ATM layer are independent of the physical medium. Four functions of this layer have been identified.

  1. cell multiplexing (in the transmit direction)
  2. cell demultiplexing (at the receiving end)
  3. VPI/VCI translation
  4. cell header generation/extraction.

  This layer accepts or delivers cell payloads. It adds appropriate ATM cell headers when transmitting and removes cell headers in the receiving direction so that only the cell information field is delivered to the ATM Adaptation Layer.

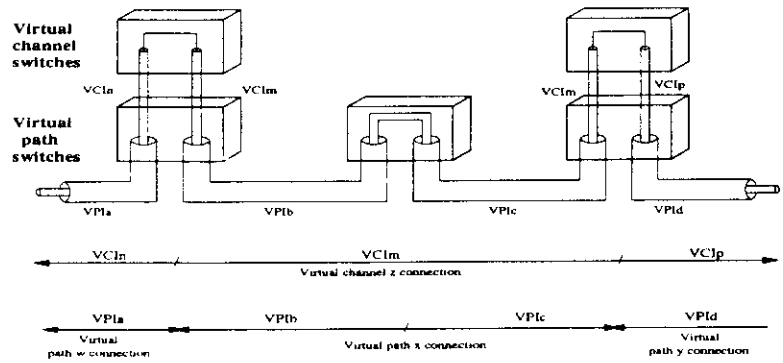


Figure 26: VC/VP Switching in ATM

At the ATM switching/cross connect nodes VPI and VCI translation occurs. At a VC switch new values of VPI and VCI are obtained whereas at a VP switch only new values for the VPI field are obtained (see Figure 26). Depending on the direction, either the individual VP's and VC's are multiplexed into a single cell or the single cell is demultiplexed to get the individual VP's and VC's.

- ATM Adaptation Layer (AAL)

  The ATM Adaptation Layer (AAL) is between ATM layer and the higher layers. Its basic function is the enhanced adaptation of services provided by the ATM layer to the requirements of the higher layers.

  This layer accepts and delivers data streams that are structured for use with user's own communication protocol. It changes these protocol data structures into ATM cell payloads when receiving and does the reverse when transmitting. It inserts timing information required by users into cell payloads or extracts from them. This is done in accordance with five AAL service classes defined as follows.

  1. AAL1 - Adaptation for Constant Bit Rate (CBR) services (connection oriented, 47 byte payload);
  2. AAL2 - Adaptation for Variable Bit Rate (VBR) services (connection oriented, 45 byte payload);
  3. AAL3 - Adaptation for Variable Bit Rate data services (connection oriented, 44 byte payload);
  4. AAL4 - Adaptation for Variable Bit Rate data services (connection less, 44 byte payload);
  5. AAL5 - Adaptation for signalling and data services (48 byte payload).

In the case of transfer of information in real time, AAL1 and AAL2 which support connection oriented services are important. AAL4 which supports a connection less service was originally meant for data which is sensitive to loss but not to delay. However, the introduction of AAL5 which uses a 48 byte payload with no overheads, has made AAL3/4 redundant. Frame Relay and MPEG -2 (Moving Pictures Expert Group) video are two services which will specifically use AAL5.

7.2.2 ATM Services

- CBR Service

  This supports the transfer of information between the source and destination at a constant bit rate. CBR service uses AAL1. A typical example is the transfer of voice at 64 Kbps over ATM. Another usage is for the transport of fixed rate video.

  This type of service over an ATM network is sometimes called circuit emulation (similar to a voice circuit on a telephone network).

- VBR Service

  This service is useful for sources with variable bit rates. Typical examples are variable bit rate audio and video.

- ABR and UBR Services

  The definition of CBR and VBR has resulted in two other service types called Available Bit Rate (ABR) services and Unspecified Bit Rate (UBR) services.

  ABR services use the instantaneous bandwidth available after allocating bandwidths for CBR and VBR services. This makes the bandwidth of the ABR service to be variable. Although there is no guaranteed time of delivery for the data transported using ABR services, the integrity of data is guaranteed. This is ideal to carry time insensitive (but loss sensitive) data such as in LAN-LAN interconnect and IP over ATM.

  UBR service, as the name implies, has an unspecified bit rate which the network can use to transport information relating to network management, monitoring, etc.

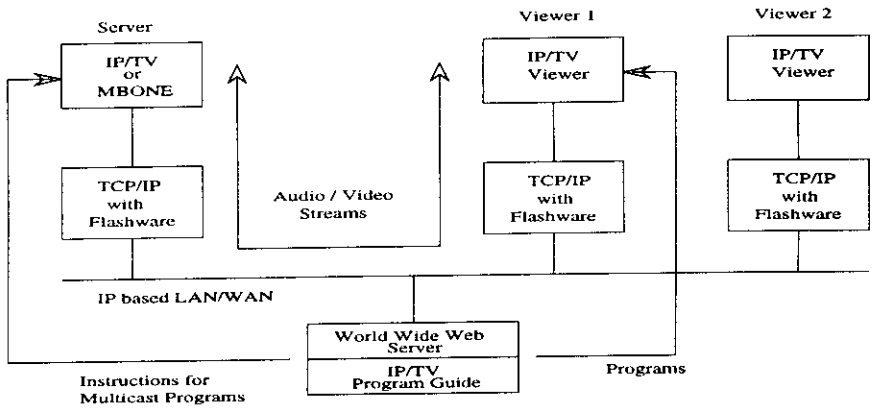


Figure 27: Real Time Audio/Video Over IP based LAN/WANs

7.3 IP/TV - A Real life Example

IP/TV^{TM} is a client server application that multicasts live or prerecorded digital video and audio streams in real time to an unlimited number of users over any IP based local or wide area network including the global Internet, using fully compliant TCP/IP protocol stacks supporting real time protocol components. It uses state-of-the-art Internet standards such as IP multicasting, RTP, RTCP and RSVP in its $Flashware^{TM}$ software suite to provide high quality, synchronised audio/video information over existing packet switched networks simultaneous with current network data traffic.

IP/TV [2] consists of a Viewer, a Program Guide and a Server (Figure 27). The program guide shows a schedule of multicasts and can be accessed via Web browser with HTTP (Hyper Text Transport Protocol). MBONE session information can be accessed with the Program Guide which controls the number of streams allowed on the network and the format of those streams, ie. audio only, audio and video or some other combination. The server delivers prerecorded or live multimedia streams based on the Program Guide schedule and parameters such as start time and file name.

The IP/TV viewer, a tool for signing up for scheduled multicasts, is designed to provide VCR like controls as well as "channel changing" controls. With a software based codec (compliant with ITU video conferencing standard H.320/H.261) colour video running at a rate of 30 frames per second uses about 500 Kbps bandwidth. The use of IP multicasting helps to conserve network bandwidth by transmitting over the network a single data stream that can be picked up by any interested user. The use of RSVP provides the ability to reserve bandwidth on RSVP compliant routers, thereby giving pri-

[2] available with Flashware from Precept Software Inc.

ority to time dependent audio/video streams over less critical network traffic thus ensuring the desired Quality of Service (QOS).

8 Summary

Real time Transport Protocol (RTP), together with a host of other protocols facilitate the transfer of real time data streams over existing LANs and WANs based on the Internet Protocol (IP) technology. The efforts in the commercial sector had been focussed mostly towards the support of multimedia audio and video streams on PCs running Windows environments (such as Windows 3.11, Windows 95 and Windows NT). IP/TV is a strong case in point which demonstrates how fast commercial products adhering fully to international standards appear (RTP/RTCP on which IP/TV is based were proposed only in January 1996!!).

However, the technology and the tools developed are available for other real time data transfer applications, such as data acquisition, which are of interest to research scientists.

9 Bibliography

[**Black 93**] Black U, *Computer Networks, Protocols, Standards and Interface (2nd Edition)*, Prentice Hall, 1993.

[**Black 95**] Black U, *TCP/IP and Related Protocols (2nd Edition)*, McGraw Hill, 1995.

[**Comer 88**] Comer D, *Internetworking with TCP/IP: Principles, Protocols and Architecture*, Prentice Hall, 1988.

[**Comer 91**] Comer D, *Internetworking with TCP/IP - Vol I: Principles, Protocols and Architecture (2nd Edition)*, Prentice Hall, 1991.

[**Comer et al 91**] Comer D and Stevens D L, *Internetworking with TCP/IP - Vol II: Design, Implementation, and Internals*, Prentice Hall, 1991.

[**De Prycker 95**] De Prycker M, *ATM Solutions for Broadband ISDN (3rd Edition)*, Prentice Hall, 1995.

[**Handel et al 94**] Handel R, Huber M N, and Schroder S, *ATM Networks - Concepts, Protocols, Applications*, Addison-Wesley, 1994.

[**Partridge 94**] Partridge C, *Gigabit Networking*, Addison Wesley, 1994.

[**RFC 1112**] *Host Extensions for IP Multicasting*, August 1989.

[**RFC 1883**] *Internet Protocol, Version 6 (IPv6) Specification*, April 1996.

[**RFC 1889**] *RTP: A Transport Protocol for Real Time Applications*, January 1996.

[**RFC 1890**] *RTP Profile for Audio and Video Conferences with Minimal Control*, January 1996.

[**Smith 93**] Smith P, *Frame Relay - Principles and Applications*, Addison Wesley, 1993.

[**Stallings 89**] Stallings W, *Handbook of Computer-Communications Standards Vol 3: The TCP/IP Protocol Suite (2nd Edition)*, Howard W. Sams, 1989.

[**Stallings 93**] Stallings W, *Local and Metropolitan Area Networks (4th Edition)*, Macmillan, 1993.

[**Stallings 94a**] Stallings W, *Data and Computer Communications (4th Edition)*, Macmillan, 1994.

[**Stallings 94b**] Stallings W, *Advances in Local and Metropolitan Area Networks*, IEEE Computer Society Press, 1994.

[**Stallings 95**] Stallings W, *ISDN and Broadband ISDN with Frame Relay and ATM (3rd Edition)*, Prentice Hall, 1995.

Software Design

Fourth College on Microprocessor-based Real-time Systems in Physics

Trieste, 7 Oct–1 Nov 1996

Paul Bartholdi and Denis Mégevand
Geneva Observatory
51, ch. des Maillettes
CH-1290 Sauverny
Switzerland

e-mail: Paul.Bartholdi@obs.unige.ch
URL: http://obswww.unige.ch/

Abstract

In this chapter, we will look at various topics concerning Software Design, from program documentation to very specific aspects of real-time. It contains also an introduction to shell programming and the use of various unix tools.

1 Documentation

Some program are used once and never used again.

However most programs

- will be used many times;
- will be changed, upgraded;
- will go to other users;
- will contain undetected errors.

Maintaining, upgrading, using again, debugging, cost more time and money **after** a program is "finished" than **before**.

Good programming + Good documentation = lower future cost

1.1 Various Types of Documentation

Documentation will serve many goals, and be read by many different users.

It should be

- Useful, that is concise and readable;
- Consistent, any change should be time stamped;
- Maintainable, indexes and cross-references should be produced automatically;
- Up-to-date, in parallel with the codes.

Here is a *short* list of various situations:

1. Source Code Comments
2. Maintenance Manual
3. User's Guide (Tutorial)
4. Reference Manual

5. Reference Card

6. Administrator's Guide

7. Teaching Notes

8. General Index

Depending on the importance of the system, some of these points may be ignored, or be part of others. For large project, they should be independent documents.

1.2 Internal Documentation to the Code

Goal: Document each module at the local level for the programmer. It should be short and informative (not paraphrase), easily readable on a screen.

Header

- name + descriptive title
- programmer's name and affiliation
- date and version of revisions with changes
- short description of what it does and how
- input expected, limits
- output produced
- error conditions, special cases
- other modules called

In-line comments

- should help to follow execution
- break into sub-sections
- indent if useful
- use meaningful names
- do not duplicate code

1.3 Maintenance Manual – Programme Logic

Goal: Present a global view of the product to a programmer, at the functional and structural level.

- table of contents
- program purpose, what it does and how
- names and purpose of principal modules
- cross-reference between modules
- name and purpose of main variables
- flow chart of main activities, dynamical behaviour
- debugging aids, how to use them
- interface for new modules
- index

It should complement the internal documentation (not duplicate it)

Look at your program from above, think about it as an outsider.

1.4 User's Guide

Goal: Should help the **user**, present him a global overview of the product and how to use it!

- Table of contents
- how to use the documentation
- how to contact author/maintainer (E-Mail) addresses, phones etc
- acknowledgements
- program name(s)
- what it does (briefly)
- explanation of the main notions and concepts used

- references (how it does it)
- how to start and stop the programs
- input expected, controls available
- unusual conditions, errors, limitations
- sample run with input, output and comments
- index

1.5 Reference Manual

Goal: Present an exhaustive and formal description for the various elements of the product.

- table of contents
- table of function, with a short description
- reference pages: list of all functions in a standard form, with a complete description similar to the module headers
- table of global variables with complete description and cross-indexing
- glossary for all specific words
- table of errors
- table of drivers
- annexes
- index

1.6 Reference Card

Goal: Single sheet with formal references for rapid consultation.

List of all commands, with their syntax, ordered by subject. Should be produced automatically from the Reference Manual and User's Guide.

1.7 Administrator's Guide

Goal: Easy installation and maintenance of the product in various environments.

- Table of contents
- minimum configuration and necessary associated products
- installation
- documentation production
- updates
- des-installation procedure
- list of supported machines and configurations
- list of attached files
- table of variables
- index

1.8 Teaching Manual, Primer

Goal: Easier understanding and learning of the product.

Step by step introduction of the various concepts and commands of the system, with examples, exercises, answers etc

It will depend considerably on the product. It could be part of the User's Guide.

As a rule, make suggestions for serial execution, avoid to force the reader on a given path, let him try whatever he wants, put data files at his disposition. In my opinion, many *Introduction to ...* are far too restrictive in this sense.

1.9 General Index

Goal: Find information anywhere in the documentation.

Should be prepared at the same time as the various documents.

1.10 Reference Page Contents

Here is a quite exhaustive list of fields for a reference page:

name		
list of commands	linkages to other products	
short description	long description	remarks
synopsis	(BNF) syntax	return value(s)
options	global variables	context
input parameters	output parameters	optional parameters
author	version	date
examples	keywords	optional keywords
known bugs	limitations	cross-references
errors	level of errors	bibliography
algorithms	precision	complexity
input files	library files	external references
temporary files	used files	modified files

1.11 Literate Programming

Knuth, while writing his set of books on TEX in parallel with the design of the product, has build a new concept for the documentation of codes, where the text around the code is the main object of attention.

`cweb` is particularly well adapted to `C` programming.

Here is a small extract from a *cweb file*:

```
@ Most \.{CWEB} programs share a common structure.  It's probably a
good idea to state the overall structure explicitly at the outset,
even though the various parts could all be introduced in unnamed
sections of the code if we wanted to add them piecemeal.

Here, then, is an overview of the file \.{wc.c} that is defined
by this \.{CWEB} program \.{wc.w}:

@c
@<Header files to include@>@/
@<Global variables@>@/
@<Functions@>@/
@<The main program@>
```

```
@ We must include the standard I/O definitions, since we want
to send formatted output to |stdout| and |stderr|.

@<Header files...@>=
#include <stdio.h>

@ The |status| variable will tell the operating system if the
run was successful or not, and |prog_name| is used in case
there's an error message to be printed.

@d OK 0 /* |status| code for successful run */
@d usage_error 1 /* |status| code for improper syntax */
@d cannot_open_file 2 /* |status| code for file access error */

@<Global variables@>=
int status=OK; /* exit status of command, initially |OK| */
char *prog_name; /* who we are */
```

From this code, two files can be extracted, a `.tex` for the printed document, and a `.c` file for the compiler.

Here is the corresponding extract in printed form:

2 AN EXAMPLE OF CWEB WC §1

2. Most CWEB programs share a common structure. It's probably a good idea to state the overall structure explicitly at the outset, even though the various parts could all be introduced in unnamed sections of the code if we wanted to add them piecemeal.

Here, then, is an overview of the file `wc.c` that is defined by this CWEB program `wc.w`:

⟨ Header files to include 3 ⟩
⟨ Global variables 4 ⟩
⟨ Functions 20 ⟩
⟨ The main program 5 ⟩

3. We must include the standard I/O definitions, since we want to send formatted output to *stdout* and *stderr*.

⟨ Header files to include 3 ⟩ ≡
#include <stdio.h>
This code is used in section 2.

4. The *status* variable will tell the operating system if the run was successful or not, and *prog_name* is used in case there's an error message to be printed.

#define OK 0 /* *status* code for successful run */
#define *usage_error* 1 /* *status* code for improper syntax */
#define *cannot_open_file* 2 /* *status* code for file access error */

⟨ Global variables 4 ⟩ ≡
int *status* = OK; /* exit status of command, initially OK */
char **prog_name*; /* who we are */
See also section 14.
This code is used in section 2.

and the C code:

```
#define OK 0
#define usage_error 1
#define cannot_open_file 2 \

#define READ_ONLY 0 \

#define buf_size BUFSIZ \

#define print_count(n)printf("%8ld",n) \

/*2:*/
#line 30 "wc.w"

/*3:*/
#line 39 "wc.w"

#include <stdio.h>

/*:3*/
#line 31 "wc.w"

/*4:*/
#line 50 "wc.w"

int status= OK;
char*prog_name;

/*:4*//*14:*/
#line 150 "wc.w"

long tot_word_count,tot_line_count,tot_char_count;
```

2 Quality Assurance[1]

The goal of Quality Assurance is to systematise the process of verification and validation:

[1]This section has been prepared from notes by Merja Tornikoski, Finland

- Verification: *Are we building the the product right?*
- Validation: *Are we building the right product?*

2.1 Standards, Practices and Conventions

Will depend on the environment (ex. programming language). It should be

- generally agreed on,
- then followed by evry one.

In general:

- The code should reflect the problem, not the solution;
- the methods used has to be predictable;
- the style has to be consistent throughout the program;
- special features of the programming language or hardware environment should be used very carefully, or avoided altogether;
- the program should be written for a reader as much as for a computer.

2.2 Software Quality Factors

Correctness does it satisfy its specifications and fulfill the objectives?
Does it do what I want?

Reliability does it perform its intended functions?
Does it do it accurately all the time?

Efficiency Amount of resources required
Will it run on a given hardware as well it can?

Security controlled access to the code and data
Is it secure?

Usability Effort required to learn, operate, upgrade the code
Can I run it in the long term?

Maintainability Effort required to locate and fix errors in the code
Can I fix it?

Flexibility Effort required to modify an operational program
Can I upgrade it?

Testability Effort required to test fully a program
Can I test/trust it?

Portability Effort required to transfer the program to another system
Will I be able to change my OS or hardware?

Re-usability Reuse of parts of a program in another application
Can I reuse some of my work?

Interoperability Effort required to couple one system to another
Can I interface my program to another system?

2.3 Review and Audits

An innocent view on your work can be very useful to

- uncover errors in function, logic or implementation;
- verify that it meets the requirements;
- agree with accepted standards;
- achieve consistency with other works;
- ease management.

A technical review should take place each time a module of a reasonable size has been completed, or results from some extensive test exist.

The review team should be small: 2–3 persons. E-Mail has the advantage that everything will be documented.

Imaginary checklist for a review:

1. System engineering: definitions, interfaces, performances, limitations, consistency, alternative solutions
2. Project planning: budgets, deadlines, schedules
3. Software requirements

4. Software design: modularity, functional dependencies, interfaces, data structures, algorithms, exception handling, dependencies, documentation, maintainability

5. Testing: identification of test phases, resources, tools, record keeping, error handling, performance, tolerance

6. Maintenance: side effects, documentation, change evaluation and approval ...

2.4 Testing

1. Executing a program with the intent of finding an error

2. Successful test: one that uncovers an as-yet undiscovered error, with minimum amount of time and effort.

3. Testing cannot prove the absence of defects

2.4.1 Black Box Testing

Using only the specified functions and input/output description, demonstrate that each function is fully operational in all circumstances, and has no defective side effects.

Some questions:

- Which functions are tested?
- Which classes of inputs are used?
- Is the system sensitive to input values? to user errors?
- What data rates and volumes can be accepted?
- How does it affect system operations?

2.4.2 White Box Testing

Using not only the external specifications, but also the internal working of the modules, demonstrate that it does work in the expected way, exercising all internal components.

All procedural details should be closely examined.

Exhaustive testing is generally impossible for large modules.

Some questions:

- Do the data structures maintain their integrity during the execution?
- Which paths are exercised, which are not?
- How are "special paths" executed?
- How is error handling executed?
- How does the system react to stress, deliberate attacks?

2.5 Defensive Programming in the Lab

The previous section is mainly valid for large projects, in particular when a team of many people is involved with external requirements.

Here are a few hints that can be applied during the exercises in the lab:

- Try to explain clearly what you are doing to your colleague. It is not far from a psychiatric experience. You will find your own errors that way.
- Do not trust anything!
  - Print the status for all file operations
  - When you open a file, verify that it exists
  - When you read a record, check that you are not at `eof`
    check that the data are valid
  - When you write a record, check that you have write permissions
  - When you do some complex calculation, check that the results are in the right order
  - If some input data must be on a given range, check its bounds
  - If anything may last more than a few seconds, print some flags or indications
  - When your program has terminated, check the size and contents of every file involved (it may not be a bad idea to print inside the program a summary of all written files with their length)

- Keep a backup of all important (a constantly changing concept) files
- Use the facilities of UNIX, like `make`, `grep`, `tee` ...

2.6 Debugging

Almost all programmes contain errors (= bugs in relay). You can help the detection of them:

- add guards while coding
- prepare simulated input, first simple (easy to trace by hand), then more complex (difficult)
- Debug each module alone, then in small integration
- chose critical points where you know what you should get if previous step are correct.
- advance by small steps
  - from input forward
  - from output backward
- analyse wrong results to see what/where this value comes from
- try all (very) improbable cases

Rules :

- if some thing can go wrong, it will !
- if an error can be damaging, it will !
- if it is very improbable, it will still exist !

2.7 Murphy's Laws

Murphy was an american engineer whose pessimism paid — his famous law, "If anything can go wrong, it will," should remain a model of conservative system design. Many scientists were inspired by him (as seen from the following):

- Any given program, when running, is obsolete.
- Any given program costs more and takes longer to develop.
- If any program is useful, it will have to be changed.
- If a program is useless, it will have to be documented.
- Any given program will expand to fill all available memory.
- The value of a program is proportional to the weight of its output.
- Program complexity grows until it exceeds the capability of the programmer who must maintain it.
- If the input editor has been designed to reject all bad input, an ingenious idiot will discover a method to get bad data past it.
- Make it possible for programmers to write in English and you will find the programmers cannot write in English.
- *Bolub's Fourth Law of Computerdom*: Project teams detest weekly progress reporting because it so vividly manifests their lack of progress.
- *The Briggs/Chase Law of Program Development*: To determine how long it will take to write and debug a program, take your best estimate, multiply that by two, add one, and convert to the next higher units.
- Computers are unreliable, but humans are even more unreliable.
- Any system which depends on human reliability is unreliable.
- A carelessly planned project takes three times longer to complete than expected; A carefully planned project takes only twice as long.
- *Grosch's Law*: Computing power increases as the square of the cost.
- *Putt's-Brook's Law*: Adding manpower to a late software project only makes it later.

- *Shaw's Principle*: Build a system that even a fool can use, and only a fool will want to use it.
- *Weinberg's First Law*: If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilisation.
- *Weinberg's Second Law*: A computer can make more mistakes in 2 seconds than 50 mathematicians in 200 years.
- Efforts in improving a program's "user friendliness" invariably lead to work in improving user's "computer literacy".
- "But I only changed one line and it won't affect anything!"

3 UNIX Tools

The goal of this section is not to introduce UNIX *per se*, but to show how some UNIX tools can help in the production of good software.

3.1 Pipes and Redirections

Pipes permit to write small modules dedicated to simple tasks, and to interconnect them through standard input/output. Such modules are much simpler to develop and test individually, while the pipe checks for the interfaces. When fully tested, these modules can be put together in larger ones.

Redirection is a good way to have all input data (including test ones) in files that can be text-edited. Output redirection, in particular using `tee`, builds sets of files against which future version's output can be compared (use `diff` for that).

3.2 UNIX as a Programming Language

Forty years ago, much programming was done in assembler, if not with wires. Then higher level languages like Fortran, C, Cobol etc. permitted the development of codes more or less independent of the hardware and operating system, that is much easier to read, that can be developed in reusable modules. Yet, the basic building blocks are still relatively low level instructions

that are combined into higher and higher modules to form a single large program, where the modules are 'hard' interconnected.

The pipes and redirections, the very large number of simple standard tools available in UNIX and the facilities to build newer tools in the same spirit, and then interconnect them into streams and shells, make UNIX an ideal interactive programming environment.

3.3 Aliases

Every complex command that may be used regularly could be aliased into a simple mnemonic name :

```
alias mnemonic 'equivalent command string'
```

The exact form of *alias* depends on the shell used. Here I have adopted the *cshell* form. Another method, probably safer and shell independent, is to have a reserved `~/bin` directory, and a corresponding scripts for each alias:

Put in your `.login` file a command:

```
PATH=~/bin:$PATH
```

and then:

```
echo "/bin/rm -i" > ~/bin/rm
```

instead of the `alias` command.

Many examples of aliases are given below.

Aliasing into usual UNIX command should be carefully avoided if the use of the original version can be dangerous when the aliased one is expected.

```
alias rm '/bin/rm -i'
```

is a typical example. In another environment, `rm` will not ask you for confirmation when you expected it.

Inversely, tools that require a mode, should specify so: use "`~/bin/rm -f`" and not "`rm`".

But:

```
alias ls '/bin/ls -CF'
```

is a perfectly acceptable one.

3.4 Searching Tools

grep is a very powerful tool to do all sorts of searches and filters, in particular as part of a pipe stream. It looks for all occurrences of a pattern inside a set of files, and print the corresponding lines.

For example, finding all files that use stdio.h:

```
grep stdio.h *.c *.h  or          *.[ch]
```

Printing error messages only, with full output into a file:

```
test < test.data | tee test.res | egrep -i error
```

grep can also be used very effectively to "search" through a "data base". Suppose that you have a file with names, phone numbers and remarks, more or less in free form, another with hints on different subjects concerning your programs etc.

Then you can define the following aliases:

```
alias help     "egrep -ih \!* \~{ }./help ./.help"
alias tel      "egrep -ih \!* \~{ }/.phones /share/phones"
```

help xxx will print all lines from ~/.help and ./.help that contains the string 'xxx'.

and tel nnn will do the same for the phone files. With tel or help you can look for anything, not necessarily name or first name, but also for partial phone numbers etc. tel 0039 will list all entries in Italy.

Here is another application, to list only the files that have been modified this day in the current directory:

```
alias today 'set TODAY=`date +"%h %d"` ; ls -al | egrep "$TODAY"'
```

A similar command to see all files modified this day, in alphabetical order:

```
alias Today 'find .  -ctime 0 -print | sort'
```

head and tail can be used to select only a few useful lines:

To see only the first line of a set of subroutines:

```
head -1 *.c
```

To see only the largest (or the most recently modified) files:

```
ls -l | sort +4 -5 | tail -16
ls -rtl | tail
```

Long output could also be piped into `more` (or `less`, `most`).

`uniq` can be combined efficiently with `sort` to find 'words' that are rarely used, and so possibly wrong (`sort -u` would do the same).

3.5 Stream Editor: `sed` and `gawk`

`sed` is a very simple yet powerful editor that can be inserted in the middle of a stream. `gawk` can be used in the same way for very complex text manipulation. The simplest use of `sed` looks like:

```
... | sed -e 's/abc/efgh/g' | ...
```

It will simply replace everywhere the pattern 'abc' with 'efgh'. The first character after `s` will be used as separator, it is not necessarily a `/`.

3.6 Executing just What is Necessary, using `make`

When a project gets larger, it becomes more and more difficult to track which compilations, link and execution are necessary.

`make` permits to do such operations automatically, based on declared dependencies and last modification time. The set of commands executed in each case is completely open and not restricted in any way to compilation or link. Further, the dependencies can be given explicitly, supplied by compilers like `gcc -M`, or even assumed implicitly by `make` itself in many cases from the file suffixes.

The use of implicit assumptions make it faster to write but more difficult to read the dependency file.

The general form of a dependency file (usually named `Makefile`) is the following:

```
target(s): dependencies
<TAB>      commands to produce the target(s)
```

`make` without a parameter will check the first target for dependencies, and then recursively through the file. If a target is older than a dependency, then the corresponding commands are executed.

If `make` is used with a parameter (a target in the `Makefile`), then the search starts from this target.

Here is a small example of a `Makefile`

```
all:     prog test

prog:    main.o  sub.o
         $(LINK.c) -o $@ main.o sub.o

main.o:  incl.h main.c
         gcc -c main.c

sub.o:   incl.h sub.c
         gcc -c sub.c

test:    prog test.data
         prog < test.data > test.results
```

`touch` can be used to change the date of last modification.

`make` can also be used as a simple user interface for commands, when there are dependencies among them. Suppose that you have a dBase on which you can edit, make extraction, preformat, visualize or print. The user could then say: `make visualise` or `make edit`, and all necessary operations will be done automaticaly. Here is the corresponding makefile:

```
all : catalogue stickers

catalogue : Catalogue.dvi
        dvips -PhpOd Catalogue

stickers : Stickers.dvi
        dvips -PhpO Stickers

catalogue.win : Catalogue.dvi
        xdvi Catalogue &

stickers.win : Stickers.dvi
        xdvi Stickers &

Catalogue.ps : Catalogue.dvi
        dvips Catalogue -o

Stickers.ps : Stickers.dvi
        dvips Stickers -o
```

```
Catalogue.dvi : Catalogue.tex catalogue.tex
        latex Catalogue

Stickers.dvi : Stickers.tex stickers.tex
        latex Stickers

catalogue.tex : m.rdb
        report catalogue.report < m.rdb > catalogue.tex

stickers.tex : m.rdb
        report stickers.report < m.rdb > stickers.tex

m.rdb : mediatheque.rdb
        cp mediatheque.rdb m.rdb

mediatheque.rdb : mediatheque.db
        m.awk mediatheque

clear :
        rm catalogue.tex stickers.tex Catalogue.dvi Stickers.dvi \
        Catalogue.ps Stickers.ps Catalogue.log Stickers.log      \
        Catalogue.aux Stickers.aux
```

If the files reside on more than one machine (using NFS for example), they should all be synchronised with `ntp` or similar time protocols.

For very large projects, when many persons are involved in the development, `make` is not sufficient. `make` ignores the notion of version or file locking that are necessary in these circumstances.

Other tools exists for them, in particular `sccs`, `RCS` or `CV`. `diff` and `patch` can be used to keep track of incremental updates and versions (including the recovery of previous code).

3.7 RCS and SCCS: Automatic Revision Control

RCS and SCCS designate sets of tools that help maintaining revisions of a product. Only RCS will be discussed; SCCS offers approximately the same capabilities while having an older, clumsier syntax.

If a program of a certain importance is being developed, it is essential to keep

all versions of the source code — not just the last, or the ten last. All versions should be numbered; a log file should account for all the modifications made between two numbers; version numbers should be allowed to ramify in a tree-like manner; the binary code produced should be stamped with the version number; and if many people work on the same project, there should be some coordinating means between them.

RCS is a set of tools for UNIX that manages automatically these tasks. Text files are normally hidden by RCS. A developer may *check a file out*, that is make it visible in his directory for modification, while locking other developer's access to it; edit it, write appropriate logging information; and *check it in*. Initially, a file `f.c` is placed under RCS' supervision with

```
ci f.c
```

with initial version 1.1. The file is moved to a special directory, usually ~/RCS. An edit cycle would now be:

```
co f.c
edit f.c
ci f.c
```

If you have EMACS, you may use its built-in capabilities to simplify this process: edit the file using its true path (~/RCS/f.c), and type Ctrl-X and Ctrl-Q to check the file in and out respectively.

It is not necessary to modify your `Makefiles`, as `make` automatically checks out and deletes files it doesn't find. If you really wanted to, you would just put:

```
...
f.c: /home/mickeymouse/RCS/f.c
<TAB>   co $<
...
```

RCS can stamp source and object code with special identification strings. To obtain them, place the marker "`$Id$`" somewhere inside your source file. `co` will automatically replace it with `$Id:` *filename revision_number date time author state locker*`$` and the marker "`$Log$`" is replaced by the log messages that are requested during a check-in.

RCS keeps all your previous versions through *reverse deltas*, i.e. keeps the last version in full, and reverse diff's to obtain previous revisions. These are accessed through

```
co -r<revision #>
```

and a sub-branch, new level major release etc. may be defined with

```
ci -r<new revision #>
```

Besides `ci` and `co`, RCS provides a few commands:

`ident`	extracts identification markers
`rlog`	extracts log information about an RCS file
`rcs`	changes an RCS file's attribute
`rcsdiff`	compares revisions

Refer to the manual pages for more detail.

3.7.1 Remarks concerning RCS

1. The directory `~/RCS` is **not** made automatically (use `mkdir RCS`)
2. `ci` will nor move `...c,v` files automatically to `RCS` (use `mv`)
3. `co` and `ci` will look automatically in `~/RCS/` if the file is not found in the current directory, and `~/RCS` exists.
4. `co` and `ci` will **not** lock automatically the files, use `co -l` instead.
5. `co` and `ci` work also on wild card. For example, `co -l *.c` will extract all `.c` files at once.
6. `rcs -l` *file* will lock the file. This is necessary if you modified a non locked file.
7. `rcs -U/rcs -L` *file* will enable/disable the file, doing strict locking.

3.8 Shell programming

When a set of commands is repeated more than 2 or 3 times, then it is usually worth puting them into a file and executing the file, passing possibly parameters. Such files are called script files in UNIX.

All UNIX shells offer lots of usual programming constructions, as variables, conditionals and loops, input and output, even some rudimentary arithmetic. Shell programming cannot replace C programming, in particular it is much

slower, but it can be very effective to organize together the repetitive and possibly conditional execution of programs.

Writing script files can have two other advantages:

– They can be edited until it works, even once ...

– They keep track of what was done, either as a log, or as an example for a similar problem in the future.

To be executable, a file just needs the `x` bit set. This is done with the `chmod +x script` command.

As many different shells can be used in UNIX, it is preferable to add as a first line a comment telling the system which one is used. So the first line of a script file should look like `#!\bin\sh` or whatever other shell is used (remember they have different syntax, and should not be confused).

3.8.1 Comments

Any character between the # and the end-of-line is treated as a comment. The example just above is realy a comment, and is interpreted by the shell as a possible indication about which shell should be used. In such a case, the # is called the *magic number*.

3.8.2 Quotes

Two quotes symbols can be used: ' and ".

Inside ' ', no special character is interpreted.

Inside " ", then $, `, !, and \ are the only ones interpreted.

Any special character can be transformed into a normal one with a \ in front.

Try:

```
Test="NoGood"
echo 1. Test          # just ascii string
echo 2. $Test         # $    in front
echo 3. \$Test        # \$   in front
echo 4. \\$Test       # \\$  in front
```

3.8.3 Parameter passing

A command can be followed by parameters as "words" separated with spaces or tabs. The end-of-line, a ;, redirections or pipes end the command.

Inside a script, `$n`, where n is a digit, will be replaced by the corresponding parameter. Notice that `$0` corresponds to the name of the command itself.

As a very simple example, here is a script that will compile a C program, and execute it immediately. The name of the program is passed as a parameter.

```
...) cat ccc
#!/bin/sh -x
gcc -O3 -o $1 $1.c
$1
```

To compile and execute threads.c, one would type `ccc threads` .

3.8.4 Variables

Variables can be defined inside a shell. Except if exported, they are not seen outside the shell. Variable names are made of letters, digits and underscores only, starting with a letter or an underscore.

They can be defined with =, or read from the terminal or a file.

```
Test="Order==$1"
read answer
```

and used, as for parameters, with a $ in front for them to be replaced with their content.

```
if [ "x$answer" = "xY" ]; then
   SetPower $level
fi
select "$Test"
```

3.8.5 Environment variables `PATH` , `MANPATH` and `LD_LIBRARY_PATH`

When the name of a program (a file name effectively) is given for execution, the system will look in successive directories, and execute the first one found.

In the same way, `man` looks in successive directories and prints the first corresponding pages found, and the loader looks in the list of directories for dynamic libraries.

These lists of directories are given in the variables `LD_LIBRARY_PATH`, `MANPATH` and `PATH`.

The directory names are separated with colon (":") characters.

To add a new directory, use command:

`setenv PATH ${PATH}:<my_dir>`

or

`setenv PATH <my_dir>:${PATH}`

The first version puts the new directory at the end, the second in front of the list. Both versions have some advantages.

`tcsh` keeps a hash table of all executables found in the `PATH`. This table is setup at login, but it is not automaticaly updated when `PATH` changes. The command `rehash` can be used to update manually the hash table.

- a "generous" `PATH` is predefined in most *Linux* systems
- the current directory "." is usually part of the `PATH` . It is better to put it at the end of the list to avoid replacing a system program.
- you can put all your executables in a directory called `~/bin` and add `~/bin` to your `PATH` . (in the file `~/.login`).
- you can do the same for your personal `man` pages.
- to see the full `PATH` as defined now, use the command:

  `echo $PATH`
- to see all environment variables:

  `env`
- to find where an executable is:

  `which my_program`
- to find where are all copies of a program (in the list defined by `PATH`):

  `whereis your_program`

  You may have to redefine `whereis` in an alias to search the full `PATH` :

  `alias whereis "whereis -B $PATH -f"`

- If you add directories in an uncontrolled way, the same directory may appear in different places ... To avoid this, you can use the PD program `envv` :

  ```
  eval 'envv add PATH my_dir 1'
  ```

  The last number, if present, indicates the position of the new directory in the list. Without a number, the new directory is put at the right end of the list.

3.8.6 Reading data

Variables can be read from the keyboard with the `read` command as seen above. Any file can be redirected to the standard input with the command `exec 0<file`. Then the `read` command gets lines form the file into the variables. The arguments can be individualy recovered with the `set` command:

```
exec 0< Classes
read head
set $head
echo The heads are: $1 $2 $3
```

3.8.7 Finding something in a large directory tree – `find`

`find` allows to search through any directory tree, looking for matching file names or files modified before or after a given date for example, and then execute any sort of command, like printing file name with full path, deleting, executing a grep on them etc.

`find` has many options, but we will see only four. Refer to the man pages for all other ones.

`find . -name` <*file_name, possibly with wild card*> `-print`

`find . -ctime` <*n*> `-exec` <*command*>

In the command, use {} to replace the file name, ending the command with \;

The first parameter (".") is the starting point, root of the directory we are searching.

The second is the selection criteria, according to file names or times.

Then comes the execution for all files that match the selection criteria.

Examples:

1. Remove all core files, printing their full path:
   ```
   find .  -name core -exec rm -f {} \;
   ```
2. List all files created today in any subdirectory:
   ```
   find .  -ctime 0 -print
   ```
3. Search for use of `stdio.h` in all c files:
   ```
   find .  -name \*\.c -exec grep stdio {} \;
   ```

3.8.8 Loop – `foreach` command

In `csh`, the command `foreach` permits to loop over many commands with a variable taking successive values from a list.

The syntax is:

`foreach` <*variable name*> (<*list of values*>)
<*commands*>
<*commands*>
...
`end`

The variable names can be modified with the following modifiers:

<*variable name*>`:r` suppresses all the possible suffixes.

<*variable name*>`:s/`<*old*>`/`<*new*>`/` substitutes <*new*> for <*old*>.

Example:

1. Save all executables and recompile:

```
foreach file ( *.c )
   echo $file
   cp $file:r $file:r_org
   gcc -g -o $file:r $file
end
```

2. Repeat 10 times a benchmark:

```
foreach bench ( 1 2 3 4 5 6 7 8 9 10 )
   echo Benchmark Nb: $bench
   benchmark | tee bench.log_$bench
end
```

3. Doing `ftp` to a set of machines. We assume that the commands for `ftp` have been prepared in a file `ftp.cmds`:

```
foreach station ( 1 2 3 7 13 19 27 )
   echo "Connecting to station infolab-$station"
   ftp infolab-$station < ftp.cmds
end
```

Such commands enable us to update a lot of stations in a relatively easy way.

3.9 Use of the `history`

`tcsh` keeps a log of the last *n* commands. *n* is defined with the command `set history=`*n* in the file `.cshrc` .

This log can be used in the following ways:

`history` prints (on screen) the list of the last *n* commands executed,

`^`***old*`^`*new* repeats the last command, replacing the first occurrence of *old* by *new*

`!!` repeats the last command,

`!`***n*** repeats a given command,

`!`***abc*** repeats the last command starting with the same letters,

`!!:s/`***old*`/`*new*`/g` repeats the last command with editing (substitution [+global]),

`!$` reuses the last parameter of the last command.

3.10 Command/file name completion

After you have typed a few letters of a command or file name:

<*TAB*> will complete it if possible and unique,

<*ctrl*>d will list all possible completions.

3.11 Very High Level Programming

Many tools exist now where the basic data unit is not numbers or words, but vectors, matrices, records or files, whose internal structure and detailed manipulation can be ignored by the user.

`matlab`, `SciLab`, `Yorick` or `SuperMongo` are good examples of very high level programming environments for graphic, vector and matrix manipulation.

`/rdb` is a similar environment to manipulate relational tables.

For example, here is a small program in `SM`, that reads a file, does some computation, and draws a graph with points of various sizes:

```
data cluster.dat
read{size 1  viscosity 2  temperature 5}
set LogT = lg(temperature)
set size = 0.1 + 2 * viscosity
expand viscosity
Diag size LogT
```

and another that selects some columns and rows from a table, using their names and a selection criteria, then prepares a file for later processing with LaTeX.

```
column name first_name institute < ictp.rdb | \
row ' country == "India" || country == "China" ' | \
jointable -j1 institute - addresses.rdb | \
tabletotex > addresses.tex
```

3.12 Notes about Relational Data Bases

Data Base systems are not part of this course, but it is difficult to build real time systems without producing data that must be stored for later analysis. Environmental parameters, usually noted in log books, should also be put in files.

Many models have been invented to organize (some very large) sets of data, the final goal being to be able to extract rapidly part of these data according to given criteria (see the example in page 174).

The relational model is probably the simplest to understand and use, the only one where mathematical proofs can be used and for which a standard interrogation language (SQL) has been defined.

3.12.1 The relational model

The relational model was introduced by E. F. Codd of IBM in 1970. Its main characteristics are:

- it is mathematically defined
- it is always coherent
- it is fully predictable
- it contains no redundancy

Many commercial or not relational data Bases are now available, for example DB2, Informix, Ingres, Oracle, Sybase, /rdb ...

In a relational RdB the data are organized in sets of rectangular tables:

PIN	name	surname	birth	...
9318	Weber	Luc	610711	

PIN	Insurance
9318	Medica

Test	Blood	Sugar
316	...	...
...		
495	...	

PIN	Diag	Interv	**Test**
9318	...	...	316
9318	...	...	495

Some columns (in bold) are key columns. Usually, each row has a different value in them. They do not depend on another one. Non key columns depend on a key one.

The rule behind the choice of columns and the structure of tables, is that no information should appear twice or more anywhere.

3.12.2 RdB basic commands

The basic commands are: insert, delete, sort, search, edit, append and join.

The **join** commands combine two or more tables whose records match on a given column.

Example: `Join Personal Medical on PIN`
`Join Medical Lab on Test`

SQL, the Standard Query Language, is a standard way to do interrogation on a RdB. SQL commands can be embedded into C or Fortran programs, but this is not standardized.

3.12.3 Real Time RdB

Concept: Associate with critical columns a trigger function(s) that is executed whenever an entry is added or changed in it.

The trigger has access to any other data, and can start any operation, including modification in the dB that may start another trigger.

Example of applications:

- stock exchange
- patient monitoring
- central control for complex instruments
- storage monitoring ($\Delta t > 1d$)

Real time dBs are good examples of the concept of "Objects = Data + Functions".

4 Use of network

The network concepts are part of another chapter. Here are just a few notes on how to use the network for file transfer and remote connection.

4.1 File transfer

File transfer between two computers can be done with the program `ftp` (*file transfer protocol*)

`ftp` <*remote host name*>

On some computers (including infolab-*n*), `ncftp` is available with some extra facilities. It will record all recent hosts you have been connected to and in which directory you worked. It will reuse this information the next time you connect to the same host. Hosts can have short nick names.

4.1.1 Host names

The computer you want to connect to can be local, part of your local network, or nonlocal, part of the rest of the world.

For a local host, the host name is sufficient.

For a non local host, the full name of the host.domain.country is necessary.

For example: `infolab-27` is locally acceptable, but `obsmp2.unige.ch` must be given in full.

Every computer on the Internet has an IP number, made of 4 groups of digits (1-255). For example, infolab-20 has the number `140.105.28.186` .

Both full name and IP number are unique in the world, and must stay so! They can usually be used interchangeably.

4.1.2 User names

If you have an account on the remote computer, then use your own *username* and your own *password* on that machine to transfer files back and forth between your local and your remote computer.

If the remote machine is an `anonymous` server, from which you intend to fetch or send files, then you must use `anonymous` as user name, and your email address, in the form `user@host.domain.country` as password. Some servers will accept anything as password, some others will check that it is a valid address. In any case, politeness dictate that you use your true email address, or at least your name and host.

4.1.3 Going to the right directory

When you are connected to the remote computer, you can use the usual `cd` and `ls` or `dir` command to locate your files.

Note that on anonymous servers, directories ready to accept files from anonymous users are usually not readable! ...but you can still fetch a file from them if you know its name.

4.1.4 Setting the mode of transfer

The files can be transmitted either in `ascii`, possibly with code conversion if necessary, or in `binary` mode. The `tenex` mode is for binary files with very long records.

4.1.5 Getting files

`get` <*remote file*> <*local file name*> will fetch the file.

`mget` <*first file*> <*second file*> ... file fetch a set of files.

`reget` <*remote file*> <*local file name*> will restart the transfer of the file **after** the last previously transferred block (after a problem on the line ...).

4.1.6 Putting files

`put` <*local name*> <*remote file name*> will transfer the file to the remote host.

`mput` <*first file*> <*second file*> ... will transfer a set of files to the remote host.

4.1.7 Compression and tar files

Some servers are set to compress files before transferring them. They can also tar a complete directory and even compress it before sending.

To use these facilities, one must add `.gz` , `.tar` or `.tar.gz` after the file or directory names.

4.1.8 Decompressing a file or directory

`gzip -d` <*compressed file*> will decompress that file.

`tar xzvf` <*compressed tar file*> will decompress and detar the full tar file.

`gzip -dc` <*compressed tar file*> | `tar vxf -` will do it if the decompression is not available within `tar`.

4.2 Working on another computer

To do so, you MUST have an account on the remote machine. No anonymous user is possible (On infolab-*nn* machines, the username `public`, possibly with password `public` can be used in a way similar to anonymous!).

`telnet` < *remote host name* > will establish the connection to the remote host.

`rlogin -l` < *username* > < *remote host name* > will establish a new session for you on the remote host.

4.2.1 Password transfer

If you have in your home directory a file called `.rhosts` with entry lines in the form:

```
host1 username
host2 username
```

with your current host name on the left part of this file, then the remote system will not ask you for your password if you use the `rlogin` connection.

4.3 Executing a command on a remote host

It is possible to execute a line of commands on a remote station with:

`rsh` < *remote host* > "< *command line* >"

Your local host should be present in the `.rhosts` file in your remote home directory.

If more than one command is present on the line, they should be separated with ";" characters.

For example, to list your files in the directory `tbl` on the remote host `infolab-21`, use the command:

```
rsh infolab-21 "cd tbl; ls -l"
```

4.4 Remote copying a file

`rcp` < *local file* > < *remote host* >:< *remote file* > will copy the local file onto the remote system. Your local host should be present in the `.rhosts` file in your remote home directory.

4.5 Displaying on another station

To have a process running on a station with a X11 display on another, you must:

On the display station: give the permission to write on its screen with the command:

`xhost` <*process station name or IP address*>

(`xhost +` will give permission to any computer in the world. This can be dangerous ...)

On your process station, you may have to redefine the global variable `DISPLAY` with the command:

`setenv DISPLAY` <*display address*>`:0.0`

Then on, all your X11 output will go to the screen of the display station.

5 Structured Design

5.1 Introduction

The continued improvement of computer performances have permitted to develop more and more complex programs, leading to a posteriori misunderstanding of the code, and to difficulties in the support and modification of the program.

This has lead Dijkstra in 1965 to the concept of *structured programming*, which can be understood as creating programs recursively consisting of modules of lower level complexity. The modules should describe a whole, a logical entity, at all levels. The operations involved in each module should be described in the most general terms available at this level, and hide the unnecessary details.

5.2 Program Development Phases

Any software project goes through a series of phases, possibly with many loopbacks to previous steps. This is true for both simple modules and large projects as a whole.

The main steps are:

user's requirements , idenfication of what are the data, what has to be done, which results are expected, what is the time-scale for the project, what money, what hardware is available?

system definitions , formalisation of the previous informations

system analysis , looking for solutions to the requirements

program design , software architecture for the adopted solution

program coding , implementation of the architecture

testing , verification against definitions and requirements

improving , smoothing the bottlenecks, getting better user's interface

upgrading , to new requirements, new hardware available etc

The analysis phase is very important as it should lead to a good design for simple programming, maintenance and should enable anyone to further enhance the program without having been involved in the original programming work.

The question is how to decompose the problem in modules ?

There are two main ways in the decomposition process:

5.3 Ascending Design and Programming

Ascending approach is the construction of a complex system by combining modules from the lowest level operations to the complete system, in increasing order of complexity. This is also called Bootom-Up design.

Pros: The modules can be tested in their real functioning at the time they are built.

Cons: We can't know at the module's programming time if it will best fit the next level module.

We don't have a general sight of the problem to be presently solved.

The interfaces are difficult to fix from below.

5.4 Descending Design and Programming

Descending approach is the construction of a complex system by expressing it in terms of simpler layers, with stepwise refinement. This is also called Top-Down design. The descending design presents exactly the reverse situation for the programmer, that is:

Pros: General problem is more correctly decomposed in sub-problems.

Good sight of the problem or sub-problem to be solved at any time of the design.

Design error can be detected and corrected at programming time quite easily.

The interfaces between modules are defined from above.

Cons: Testing the already built modules (which are the higher level modules) need to write drivers submodules simulating the input-output behaviour of the real submodules.

In practice, a mixture of both approaches is often used, by combining a descending analysis and design with a ascending programming phase. This mixture can be a good practical way *as long as the analysis and design phase are kept detailed and precise enough to avoid design errors.* If one has to correct the design at the programming time, this one should also be descending.

5.5 Structured Design Principles

Structured programming enables to:

- give a program a better clarity, so that future enhancements may be easily done.
- augment the reliability of the code, because modules can be tested as soon as they are built.
- hide unnecessary details.

The principle of structured programming is to give the programmer tools enabling him to express his problem in structured blocks.

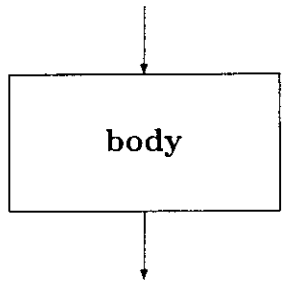


Figure 1: Structured block construction

A structured block is a module (at design level), or a piece of code (at programming level) which stands on its own and has only 1 input and 1 output.

The content of this block may be very simple **or** very complex, in which case it should be decomposed itself into other structured blocks.

This can be done with the flow control instructions. With these instructions, the `GOTO` instruction is not needed anymore, so that one can avoid the unverifiable and multiple paths in a program.

5.6 Flow Controlling

Each language defines it's own set of flow control instructions, and renames them differently. In this section, we will describe the main flow control instructions, which can be separated into three groups:

Conditional instructions

5.6.1 `IF...THEN...ELSE...`

Only one of two possible blocks is executed (figure 2):

if *condition* **then** *body_true*
else *body_false*
end

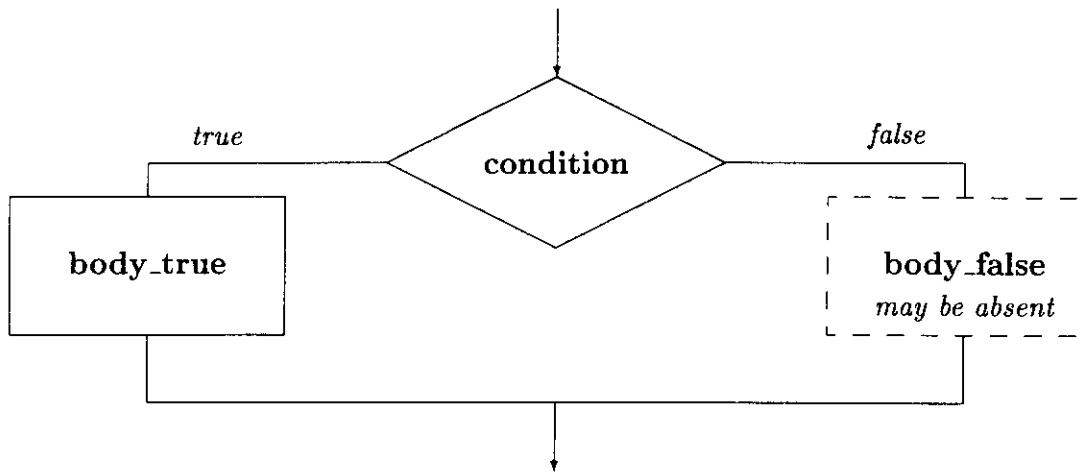


Figure 2: `if...then...else` construction

5.6.2 `CASE...OF...`

Only one of many possible blocks is executed (figure 3):

case *expression* **of**
    value_1:= *body_1*
    value_2:= *body_2*
    ⋮
end

Please take notice that *expression* and *values* are sometimes replaced by *conditions.*

Counting loops

5.6.3 `FOR...DO...`

A given block is executed an exact number of times (figure 4):

for *variable*:= *first_value* **to** *last_value* **do**
    loop_body
end

Please take notice that *loop_body* is usually executed at least once.

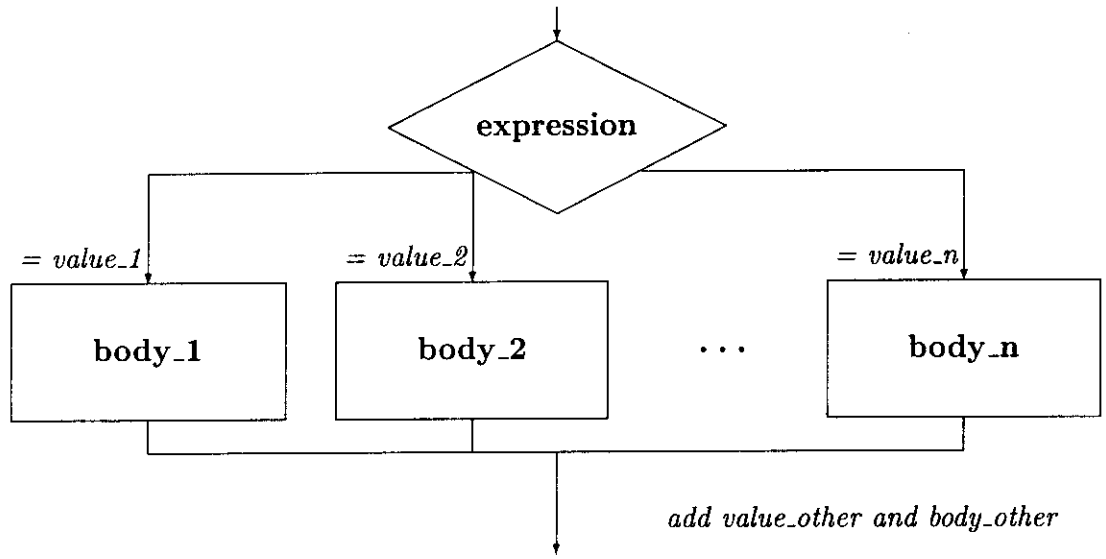


Figure 3: `case...of` construction

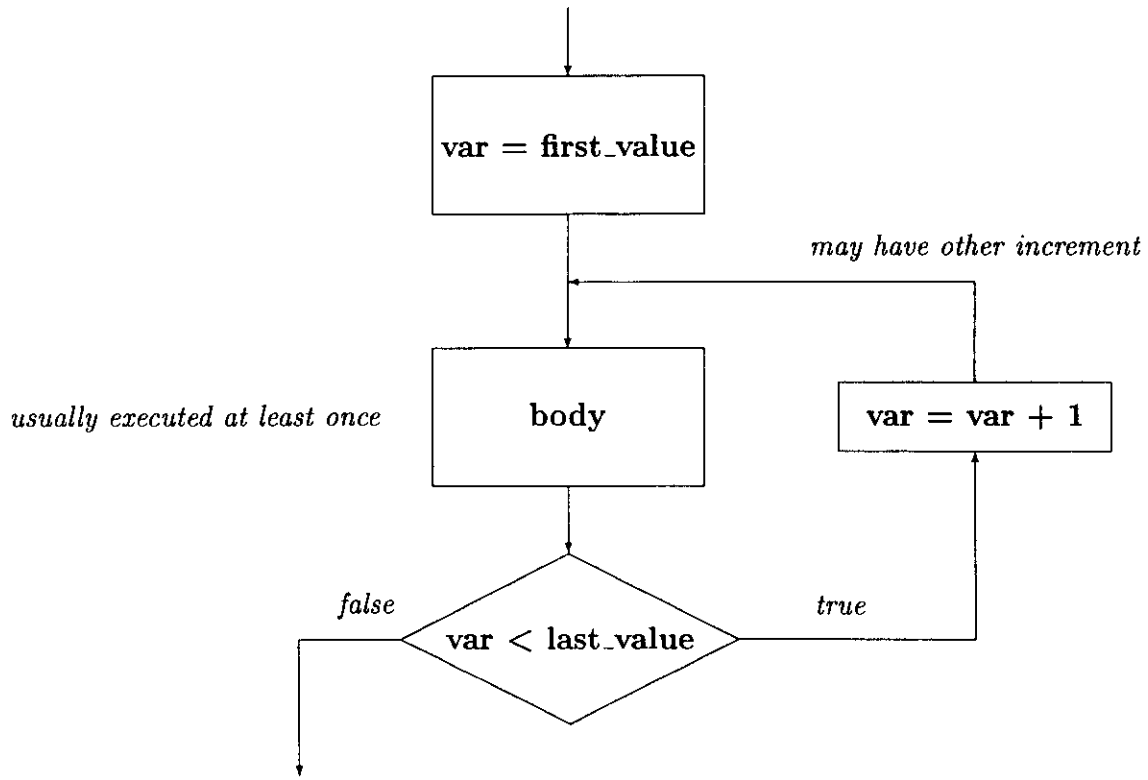


Figure 4: `for...do` construction

Conditional loops A given block may or may not be executed many times depending on a condition. The condition may be set inside the block.

5.6.4 `WHILE...DO...`

while *condition* **do**
    conditional_body
end

Please take notice that *condition* is tested before the first execution of the *conditional_body* (figure 5).

If the condition is the constant 1, then the loop will go for ever. You will have to use `break` to get out of it.

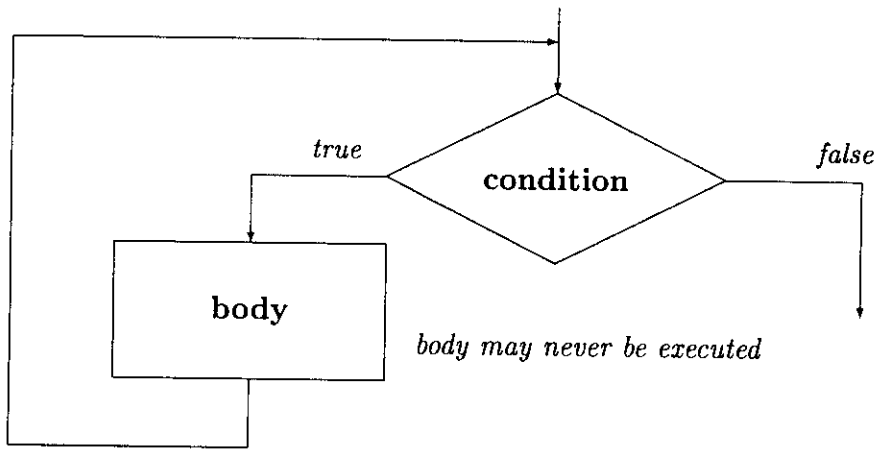


Figure 5: `while...do` construction

5.6.5 `REPEAT...UNTIL...`

repeat
    conditional_body
until *condition*

Please take notice that *condition* is tested after the first execution of the *conditional_body* (figure 6).

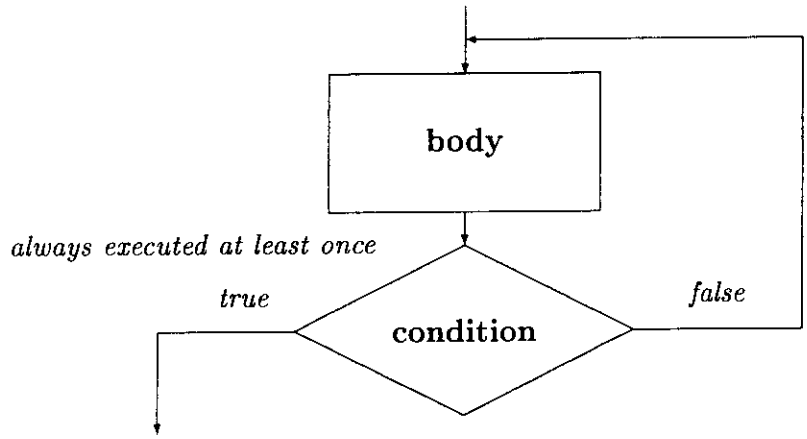


Figure 6: `repeat...until` construction

5.6.6 `REPEAT...WHILE...DO...`

repeat
body_1
while *condition* **do**
body_2
end

Please take notice that *condition* is tested after the first execution of the *body_1*, but before the first execution of the *body_2* (figure 7.

Consider the following real situation (in pseudo code):

```
read;
if not EOF  do computations
read again
```

We can solve it in three ways:

1. as in Pascal:
   `s=read( );`
   `while(s!=EOF) { calculations; s=read( )};`
   `read` is used twice, and appear illogicaly after the calculations ...
2. `while((s=read( ))!=EOF) { calculations } ;`
   Now we have side effects in the condition, doing two things in one statement;

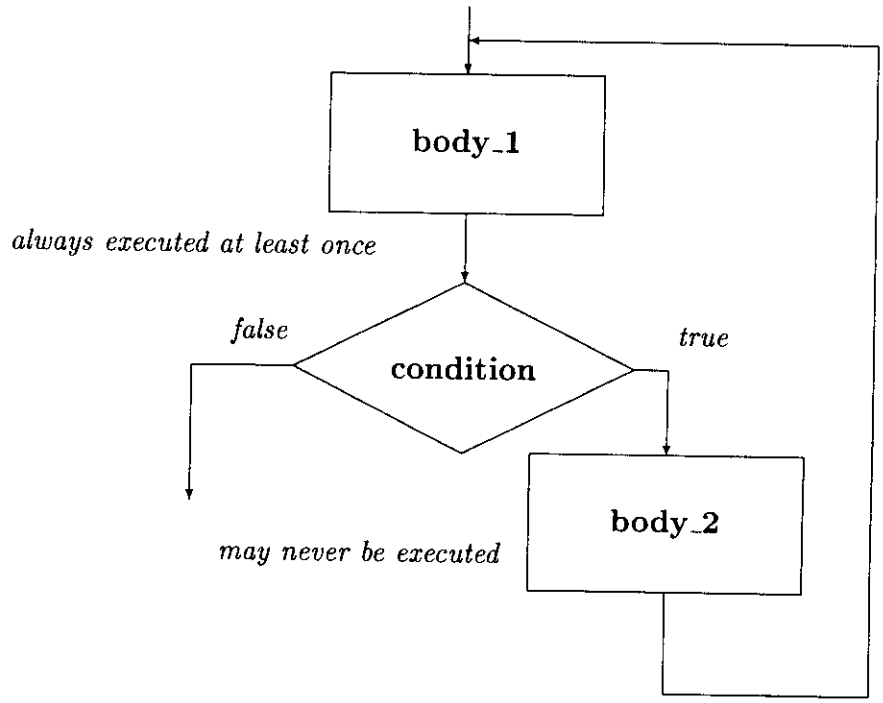


Figure 7: `repeat...while...do` construction

3.
```
while(1) { s=read( );
if(s==EOF) break;
calculations
```
This matches the logic much better, though the code is longer.

5.6.7 Side effects

In C and many other languages, the tested condition can be any expression, possibly with strong side effects, that is variables get changed by the condition evaluation. For example, consider the expression, given in the C notes:

```
while ( *S1++ = *S2++ ) ;
```

Such expressions are very compact, but rather difficult to read, and quite prone to errors.

5.7 Implementation Addresses

Some languages are more appropriate than others to structured programming, and amongst the procedural languages, let's cite as examples the Pascal, C, Modula and Ada languages.

These languages offer all the preceding possibilities by specialised instructions, except the last one which should be programmed with a loop and an internal if instructions. Moreover, Ada language doesn't support the `repeat...until` structure.

Other branching instructions complete the set, enabling the program to interrupt or skip an occurrence of a loop.

Most of the procedural languages offer a `goto` instruction, just in case..., but to avoid using it will lead to better design, and maintenance.

5.8 Weaknesses of the Structured Approach

The modules are based on their functionality, and define procedures and functions, while variables are often passed as parameters, or are globally (on the outside) defined.

This leads to

- logically incomplete modules.
- difficulty to reuse a module in a slightly different way
- variables can be modified from the outside of the module.

5.9 Practical remarks concerning the exercises

1. All system calls and standard library routines return a value indicating the success or failure of the operation. The error code is also returned in the variable `errno`.

   This value should always be checked, with an erorr message and appropriate action (continue with default, do it again, exit ...) in case of failure (See the examples below).

2. `stdio.h` and other header files (including your own) contain list of declarations like `#DEFINE OEF (-1)` or even `#DEFINE NULL (0)` and also `typedef ... { ... } FILE;`

Use them! They help you hide something and make the code easier to read, check and understand.

Examples:

```
FILE *Pn
      Pn = fopen("/ds", "w" );  /* not "2" but "w" */

if( (fn=open("/ds", "w")) == NULL )
   { printf("cannot open file /ds \n") ;
     exit (11);
}

if( (fn=open("specific", "r")) == NULL )
if( (fn=open)"default",  "r")) == NULL )
   { printf("neither specific nor default available \n");
     exit (13);
   }
```

Notice in the last example, that the second `if` is skiped if the first succeeds.

6 Data structures

Data structures can be classified into two main categories: linear and non-linear. Linear structures are composed of a sequence of elements and include *arrays*, *linked lists*, *stacks* and *queues*. Non linear structures include *trees* and *graphs*. We will limit our scope to a general introduction to the linear structures, as they are the basis of the structures used in real-time systems.

The operations that can be performed on a linear structure are:

- Traverse the structure and process each element.
- Search a particular element of the structure.
- Add a new element to the structure.
- Remove an element from the structure.
- Rearrange the elements in some order.

The internal representation of a linear structure may take two shapes:

- Array representation, where logically consecutive elements of the structure are represented by *sequential memory locations.*

- Linked list representation, where the relation between the elements are represented by means of *pointers.*

The type of representation one chooses for a particular structure depends on how it will be accessed, and on how many times the different operations will be performed.

6.1 Arrays

Arrays can be linear or multidimensional homogeneous structures. We will limit our scope to linear arrays; the extrapolation of the algorithms to the other cases is relatively easy.

The linear array is a finite list of data elements. The elements are referenced by an *index*, which is the ordering number of the element. The elements are stored in consecutive memory locations. That implies that the index set is composed of consecutive numbers.

The smallest index is called the *lower bound* (LB), and the largest is the *upper bound* (UB). The length of the array is given by the formula

$$L = UB - LB + 1$$

Usually, $LB = 0$ and $L = UB + 1$, or $LB = 1$ and $L = UB$.

The logical representation of an array consist of a series of compartments pictured either vertically or horizontally, depending on the number of elements and on the available space, as shown on the figure 8.

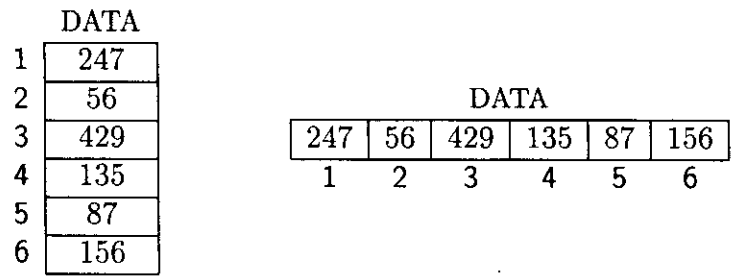


Figure 8: Logical pictures of array DATA.

The computer keeps only track of the *base address* (BA) of the array A, and calculates the position of the kth element by the formula:

$$LOC(A[k]) = BA(A) + w \cdot (k - LB)$$

where w is the number of memory words (bytes for an 8-bit architecture) per element for the array A. The figure 9 shows the internal representation of an array AUTO, with $BA = 200$, $LB = 1932$, and $w = 4$.

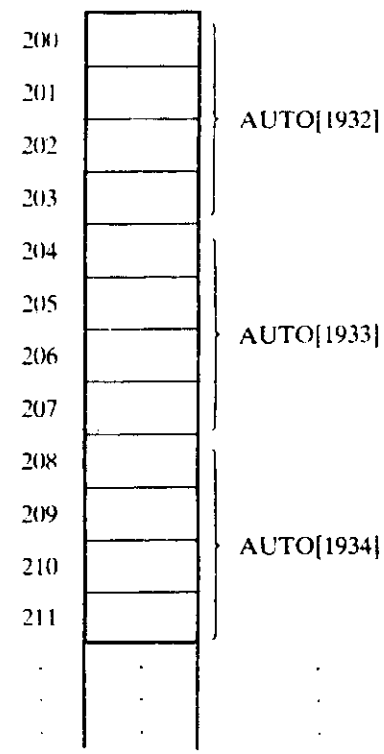


Figure 9: Memory representation of array AUTO.

6.1.1 Operations on linear arrays

Operations on arrays are simple, due to the linear structure of the arrays.

Traversing an array is done by a counting loop (5.6.3), the index of the array being used as the control variable of the loop. The body of the loop defines the operations to do on each element.

Inserting an element at the end of an array is quite simple. Inserting an element in the middle of the array implies moving all the elements located after the insertion point up back a position. This again may be done by using a counting loop initialized at the upper bound, and running down to the insertion point. One has to do it this way, as the higher indexed memory locations may be overwritten without problem.

The figure 10 illustrates this by inserting the value "Ford" in a string array at position 3.

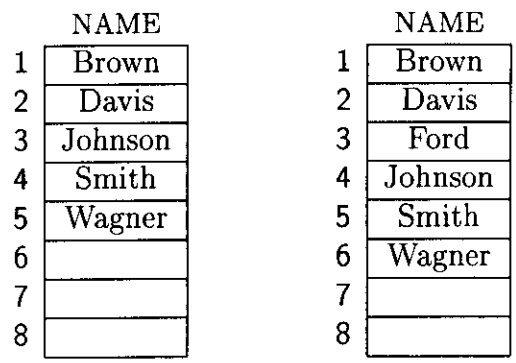


Figure 10: Insertion of an element in an array.

Notice that decreasing index counting loops are not supported by all languages. If not supported, this operation can be simulated by a conditional loop (5.6.4).

Deleting an element of the array is very similar to inserting, at the algorithmic level. A counting loop running upward from the deletion point should be used to move down the succeeding elements.

Searching an element in the array can be done through two algorithms: linear and binary search.

Linear search implies a conditional loop executed at least once. The loop body should check if the element fits the desired item and if the bound of the array is reached. This implies two comparisons at each occurrence of the loop, leading to a possible $2N$ comparisons.

The estimation of the number of basic operations an algorithm needs to be completed is called the complexity of the algorithm. It gives the notion of computation time for the implementation of the algorithm. It is sometimes expressed with the O notation:

$$f(n) \quad \text{is} \quad O(g(n))$$

Where $f(n)$ is the complexity, $g(n)$ is a simple function.

An enhanced algorithm will first write the searched item at the end of the array, in position $N+1$. Then a single comparison is done in the loop, checking for the item, and when successful, a last comparison determines if the item was found in the array or in position $N+1$. The maximum comparisons number is thus $N+1$.

The average number of comparisons, in case of equally probable position of the item, with an absence probability of ε is given by

$$\begin{aligned} 1 \cdot \frac{1}{N} + 2 \cdot \frac{1}{N} + \cdots + N \cdot \frac{1}{N} + (N+1)\varepsilon &= \frac{N(N+1)}{2} \cdot \frac{1}{N} + (N+1)\varepsilon \\ &= (N+1)(\frac{1}{2} + \varepsilon) \end{aligned}$$

If the absence probability is very small, the average number of comparisons will be about half the length of the array.

Binary search is used for maximum efficiency. The array *needs to be somehow sorted.* The comparisons will not be done sequentially, but accessing recursively the middle of the part of the array containing the item to find. At the beginning, the containing part is the whole array.

After M comparisons, the segment containing the item is $\frac{N}{2^M}$ long. Locating the item implies thus a maximum of $M = \log_2(N) + 1$ comparisons. This means that a 65000 element array could be searched successfully in 16 comparisons.

So why not use always a so economical algorithm ? Binary search is only possible if the array is sorted, and maintaining a sorted array can be very resource-consuming, for big arrays with a lot of modifications.

Sorting an array is a bit more complicated. There are several algorithms suitable for different data structure. The most simple is called *bubble-sort.*

Let's have a N-element array. The algorithm consists of traversing the array, comparing each element with the element immediately following it and swapping the two elements if necessary. This traverse operation,

called a *pass*, enables to put the smallest or the largest element (according to the test) at the upper bound, in element N. This step is repeated $N-1$ times with the subarrays upper-bounded by the element indexed $N-1$, $N-2$, etc.

The complete sort is a $N-1$ passes process. The passes involve $N-1$, $N-2$, etc. comparisons, so the entire sort process need, to be complete, a total of

$$(N-1)+(N-2)+(N-3)+\cdots+2+1=\frac{N(N-1)}{2}$$

which is proportional to N^2.

Another well-known sorting algorithm is the *quicksort* algorithm.

In this algorithm, each *step* (fig. 11) is used to find the proper place for one element of the array. Let's take the first number of the array. We compare it with the others, starting backwards from the last. When a smaller number is found, we exchange the two numbers, and start again traversing from left to right the array until we find a larger number. This step stops when the comparison with the element itself. This element is at its correct place in the array.

We then have two subarrays which are themselves to be quicksorted.

Comparison 1	[44]	33	11	90	40	22	88	66
Comparison 2	[44]	33	11	90	40	22	88	66
Comparison 3	[44]	33	11	90	40	22	88	66
Swap 1	22	33	11	90	40	[44]	88	66
Comparison 4	22	33	11	90	40	[44]	88	66
Comparison 5	22	33	11	90	40	[44]	88	66
Comparison 6	22	33	11	90	40	[44]	88	66
Swap 2	22	33	11	[44]	40	90	88	66
Comparison 7	22	33	11	[44]	40	90	88	66
Swap 3	22	33	11	40	[44]	90	88	66
	subarray 1					subarray 2		

Figure 11: One step of the quicksort algorithm.

The quicksort algorithm is in the worst case when the array is already sorted. Each step needs N comparisons and produces only one subar-

ray, of length $N-1$, leading to a total of

$$N + (N-1) + (N-2) + (N-3) + \cdots + 2 + 1 = \frac{N^2}{2}$$

comparisons, which is proportional to N^2. The advantage over the bubble-sort appears for the average case. Bubble-sort has a constant number of comparisons. Quicksort, on the other hand, produces 2 subarrays in each step, so the successive levels place $1, 2, 4, \ldots, 2^{k-1}$ elements. About $\log_2(N)$ levels will be necessary to sort the array, with a maximum of N comparisons at each level. The average number of comparison for the quicksort is thus proportional to $N \log(N)$.

6.2 Linked lists

As the insertion or deletion of an element in an array is a quite expensive operation, and as arrays are static structures that cannot easily be expanded, it is sometimes necessary to use another type of structure, whose elements contain, in addition to the data, a link to the next element. This way, successive elements need not occupy consecutive memory locations.

This type of structure is called a *linked list*, and is widely used in computer science, due to it's dynamic behavior. A linked list is composed of *nodes*. Each node is divided into two parts: the *information part* and the *link field* or *next pointer field*, which contains the address of the next node in the list.

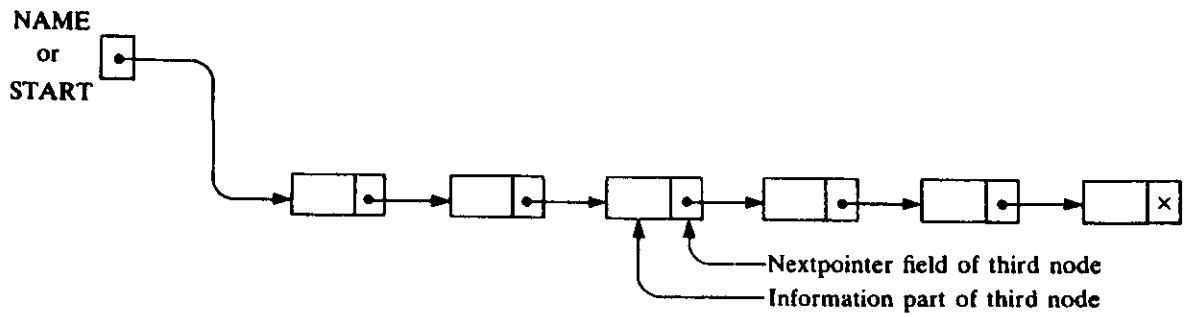


Figure 12: Horizontal representation of a linked list.

A linked list is represented by a series of double boxes linked by vectors, either horizontally or vertically, as shown in figures 12 and 13. The information part may be further subdivided, as seen in figure 13. A separate variable indicates the first element of the list. It is the list pointer variable ($START$). The last element of the list contains a null pointer to indicate the end of the list.

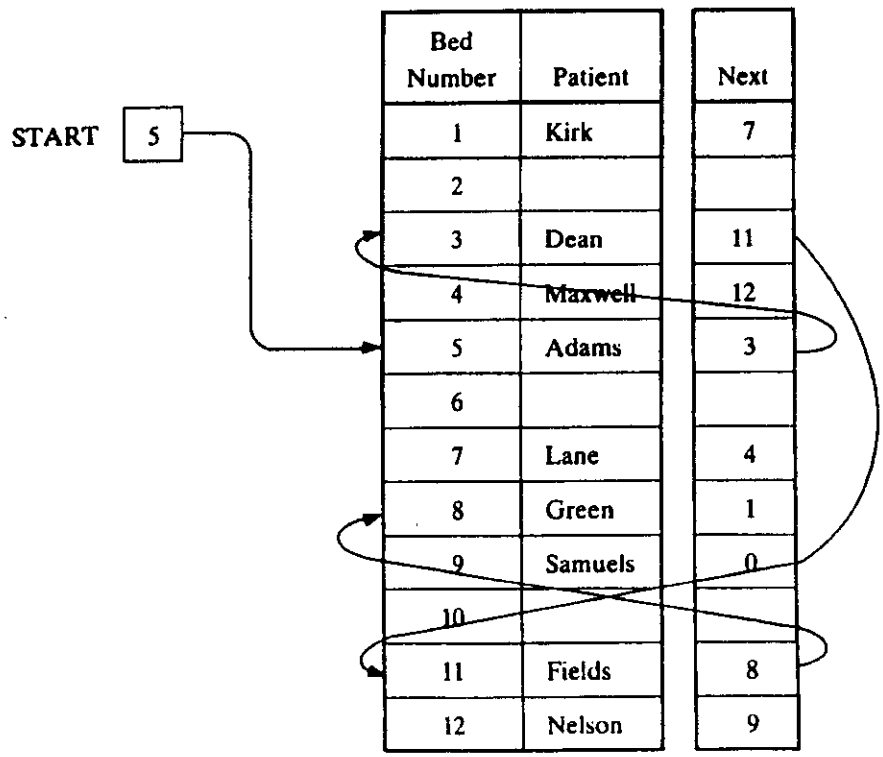


Figure 13: Vertical representation of a linked list.

6.2.1 Operations on linked lists

A linked list may be maintained in memory by means of two arrays, one containing the data and the other the links, or by using an array of records containing both the data and the links. Let the informative part of element K be $INFO[K]$ and the link field of the same element be $LINK[K]$. Let also $START$ contain the first node address and $NULL$ be the content of the last link.

Traversing a linked list is done by using a variable PTR containing initially the address of the first node ($PTR := START$). After having processed the first node's data, the pointer is updated to point to the next node ($PTR := LINK[PTR]$) and the loop is repeated until $PTR = NULL$.

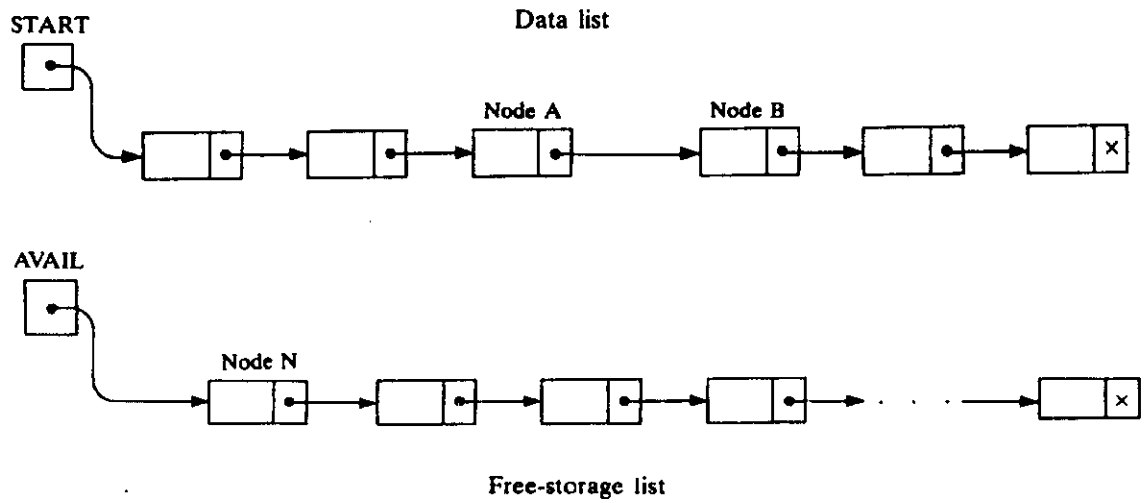


Figure 14: Linked list before an insertion.

Insertion To insert a new node in a list, we need to have some available memory locations, and to be able to allocate them to the list. This is done by maintaining a parallel list called the *list of available space*, the *free-storage list* or the *free pool*. Let this list be called $AVAIL$.

The insertion of a node between nodes A and B of a list (fig. 14) is done by removing the first node of $AVAIL$ and storing its address in an auxiliary variable NEW ($NEW := AVAIL$). The $AVAIL$ is updated ($AVAIL := LINK[AVAIL]$); we will then copy the new data in the new node ($INFO[NEW] := ITEM$), and at last we have to insert

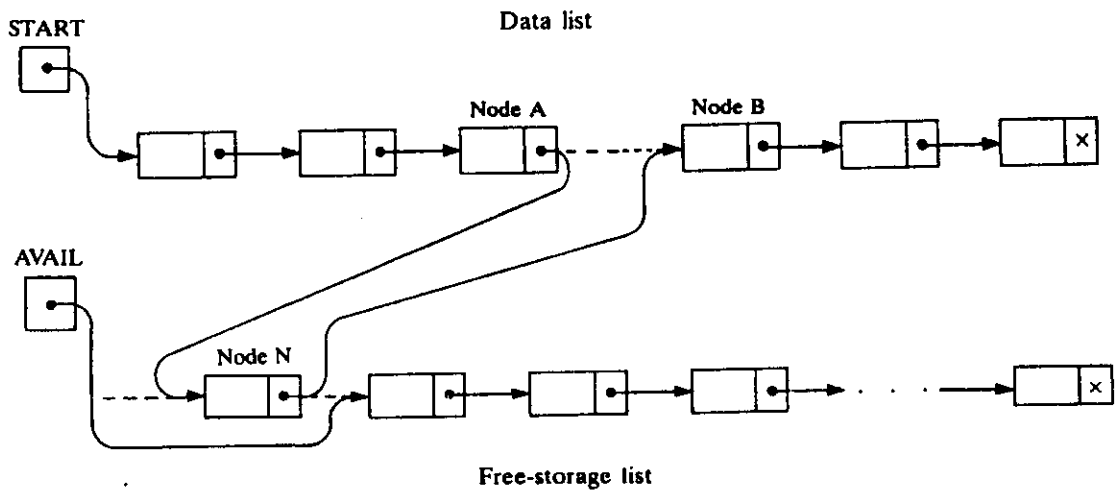


Figure 15: Linked list after an insertion.

the new nodes in the list ($LINK[NEW] := LINK[A]$; $LINK[A] := NEW$). The resulting lists are presented on figure 15. Note that were the insertion point be the first node, the two last assignments would have been $LINK[NEW] := START$; $START := NEW$.

Deleting a node of a list seems very simple, as we have only to reassign the pointer of the preceding node to point to the next node. In reality, we can't know the address of the preceding node without traversing the list to compare each node with the deletion point, while remembering the preceding node until the actual node is processed. Another problem is to deallocate the memory we don't use anymore. This task is called *garbage collection* and is done by returning the node to the $AVAIL$ list (fig. 16). Thus, deleting an element of a list is done by traversing the list once, and then returning the node to the free pool, which implies about the same operations as inserting a node. While doing the traversing, we are able to do another task, as searching, for example, a node with specific data, which we want to delete.

Searching a specific item throughout a list implies a loop with an internal concordance test. If the list is sorted, the test may be smarter to check if the item position is already overpassed, which would lead us to stop the loop.

Binary search is not possible with linked lists, since there is no way to point to the middle of a list.

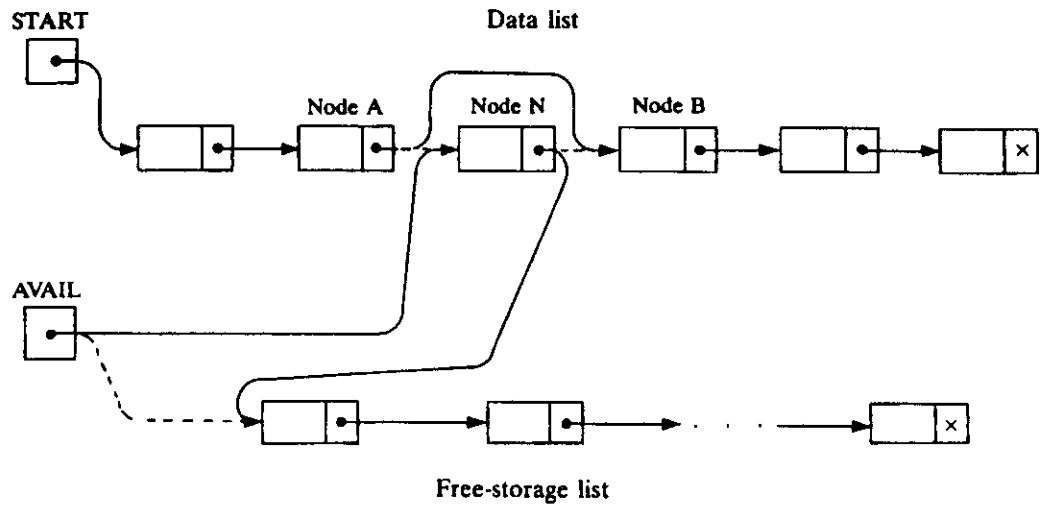


Figure 16: Deletion in a linked list.

Sorting a list may be done by different algorithms. The bubble-sort algorithm (6.1.1) will be suitable for a linked list, but the quicksort algorithm (6.1.1) will need the particular properties of a two-ways list (6.2.2).

Another good way to have a sorted list is to keep it sorted, i.e. insertion is done at the right place (searching).

6.2.2 Particular lists

There are several particular forms of lists that can be used in different situations.

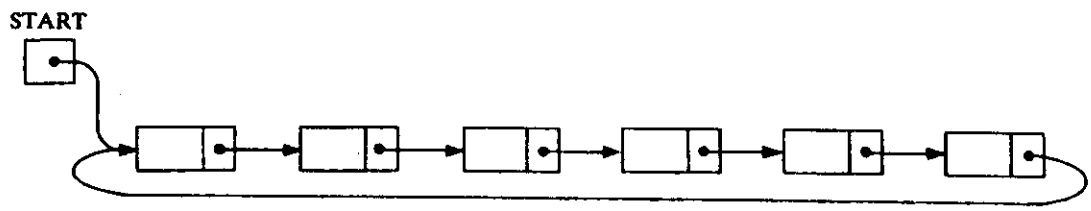


Figure 17: Circular linked list.

A *circular list* (fig. 17) is a linked-list whose last node's link points to the first node. This kind of list is widely used in computer science, because all the

pointers contain valid addresses, and no special treatment is thus required neither for the first node, nor for the last.

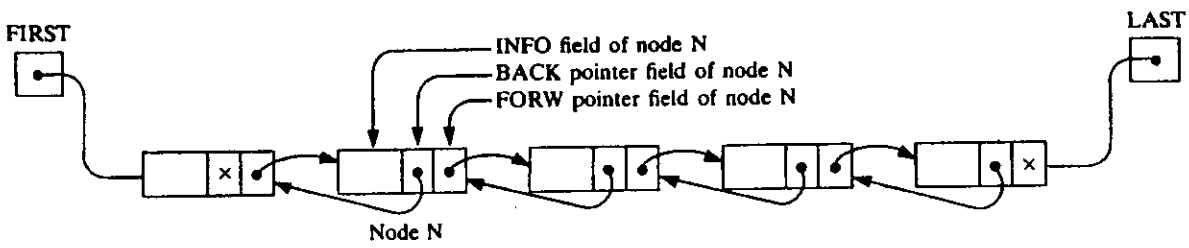


Figure 18: Two-ways linked list.

A *two-ways list* (fig. 18) contains three parts nodes. In addition to the data part and the link field $LINK[K]$ now called $FORW[K]$, there is a second link $BACK[K]$ pointing to the preceding node. The $START$ variable is replaced by two entry point variables $FIRST$ and $LAST$. A two-ways list has the fgllowing properties:

- $FORW[A] = B \iff BACK[B] = A$
- Operations can be done in either direction.
- For deletion, the localization of the preceding node is trivial.
- Insertion is a bit more complicated by the presence of the second pointer, i.e. needs two more assignments than insertion in a one-way list.

A *two-ways circular list* mixes the properties of the two previous lists.

6.3 Stacks

A *stack* is a linear structure accessible only by one extremity. This notion is very familiar to us, as we use a lot of stacks in everyday's life, as illustrated in figure 19.

All the operations will be done on a particular point called the *top of the stack*. Adding an element is done by *pushing* it on the stack. Removing an element from the stack is called *popping* (fig. 20). As the top is the only access to the stack, the last element pushed in will be the first popped out from the stack. This *last-in, first-out* property has given to the stack its second name: *LIFO*.

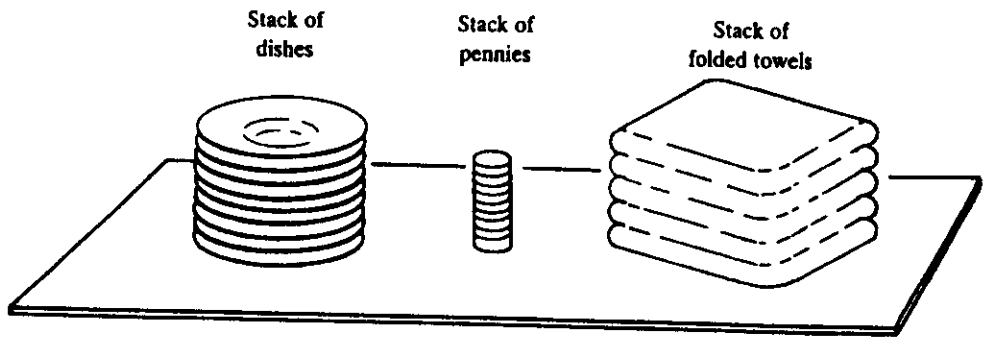


Figure 19: Everyday's life stacks.

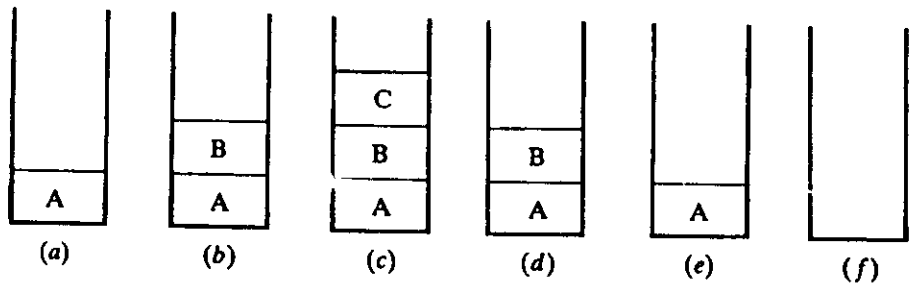


Figure 20: Stack push and pop operations.

Stacks are widely used in computer science. They are the basic structures on which the notion of recursion is implemented, and many well-known algorithms or problems have been implemented and solved through its usage.

Remember the quicksort algorithm (6.1.1). A practical way to keep track of all the subarray bounds while processing one of them is to put them on stacks. The *Towers of Hanoi* problem is implemented recursively (recursion uses stacks), or may be implemented with stacks in an iterative way. *Reverse Polish Notation* (RPN) which writes operations as operands followed by the operator uses stacks: The operands are put on the stack, where each operator pops the number of operands it needs.

6.4 Queues

A *queue* is another familiar concept (fig. 21). In computing, queues are also widely used for bufferizing data arriving from or leaving to a peripheral, or to schedule tasks to a processor. They have a *first-in, first-out* structure, and thus are also called *FIFO*.

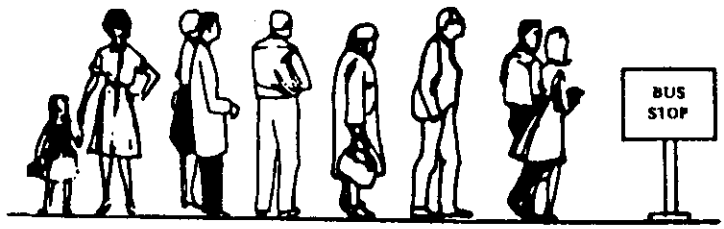


Figure 21: Familiar queue.

Data may be added in a queue only at the end called the *front*, and removed only at the other end, called the *rear* .

Special implementations of queues allow other types of access:

Deques are double ended queues, that can be accessed by either ends, but not in the middle.

Priority queues are queues where the highest priority element is to be processed first. The implementation will determine the ease of inserting or deleting the element in a priority queue. A way to implement a priority queue is to use a linked list with its usual properties for insertion,

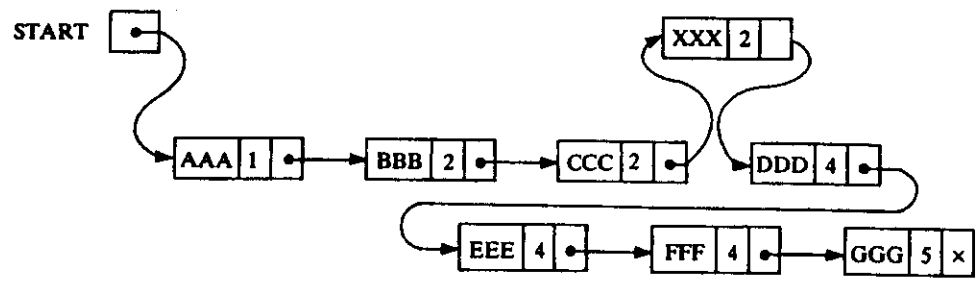


Figure 22: Representation of a priority queue implemented as a list.

but where processing and deletion is limited to the first element. In the figure 22, successive deletions will remove AAA, BBB, etc., while insertion of an element XXX is done at a place determined by the algorithm according to its priority (2) .

7 Object Oriented Computing

It is highly preferable to group in one unit a logically linked data set. On the other hand, it is not necessary that higher level modules know the internal functioning of the routines or the structure of a complex data set. An external module should perceive them as a functional black box. This vision is close to the block diagrams used in electronics or automatics.

The interface of the module is its visible and accessible part. It represents the specifications of the module and can be separated from the implementation part, which describes the functionalities of the module.

The Modula-2 language was one of the first languages to comply with the separate compilation of the modules. It addresses the notion of visibility, with library modules consisting of a definition and an implementation parts.

7.1 Objects

The basic concept of object-oriented description is to consider a program as a model for a real world situation. Now, the real world consists of related objects. Objects are thus more stable than relations in the system evolution.

It seems thus natural to decompose this real world situation's model in objects models rather than in models of the relations existing between these objects.

From now, we will call objects the models of real world objects. *An object is the whole set of characteristic properties satisfactory to describe the object with regard to the studied model.*

In classical programming, we consider an algorithmic description of the system, in which we introduce data. In object-oriented programming (OOP), we consider objects, whose behaviour is described by algorithms.

Object = Data structure + Related operations

7.2 Object Oriented Design

Different advantages of the object-oriented approach are examined in the next sections:

7.2.1 Easy Design

Our brain is used to apprehend real objects. The definition of a program's main concepts as objects enables us to better conceive, thus better express the application's goal.

7.2.2 Better Support and Debugging

With the gathering of data structures and related procedures in a single locus (the object), the localisation is better, leading to more direct access and easier debugging.

7.2.3 Data Security

An object is a black box. The OOP insists on the separation between the object's properties, described by related operations, and the internal representation of this object. An object provides the handling interface, while hiding the implementation details.

Seen from the outside, an object will be manipulated only on its properties knowledge, without considerations to its realisation.

7.2.4 Flexibility

Internal representation of the objects can be modified, adapted to the hardware and so allow performance optimisation, without meddling with the application software.

7.2.5 Recycling

An application can be developed from existing objects. This can speed up software production and decrease the development costs.

7.3 Competence Sharing

The overall process of software development involves three aspects:

7.3.1 The Role of the Application's Conceptor

He has to define the objects in three phases:

1. What are the intervening objects of the application ?
2. What are they doing ?
3. How do they interact with each other ?

7.3.2 The Role of the Objects' Programmer

He will create the objects defined by the first above-mentioned point. The answers of the second and third questions will give him the data structure and the associated operations, as illustrated in figure 23.

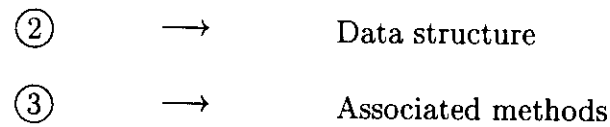


Figure 23: Concepts – objects relation.

The objects' programmer should be aware of the hardware, to be able to optimise the objects' code, if necessary. No hardware dependent programming should be done at another level, and even at any level, if possible.

7.3.3 The Role of the Application's Programmer

He uses the objects according to the functionalities defined by the conceptor. He is responsible for the application's functioning optimisation.

7.4 Object Oriented Programming

Procedural (algorithms based) languages such as Fortran or Pascal associate data to procedures, but OOP associate procedures to data structures to create objects. New languages with some new characteristic are to be used for objects creation and manipulation. As we'll see in the next sections, the object approach is implemented in these languages, as well as some other ideas allowing an easy and complete implementation of the objects.

First, the notions of abstract data types, which defines meta-objects, and of encapsulation is the implementation of the objects themselves. The concepts of inheritance enables the creation of hierarchy of related data types. The polymorphism allows an object to take several shapes, and the dynamic binding dispatches general calls to specific methods adapted to the object type.

In the object-oriented concept, the communication between the objects is done via *messages*, which are used to schedule the methods.

7.4.1 Data Abstraction and Encapsulation

Every data structure should give rise to a control of its manipulation, in order to guarantee the data consistency. One should think of this structure in terms of the actions to be carried out on it, rather than in terms of its representation.

The definition of a type as the whole set of operations linked to a data structure meet this view, provided that one can only manipulate this structure by these operations.

Such a type, whose name is associated with the data structure and whose internal functioning and representation details are hidden by providing the

appropriate operations for the variables of this type is called an *abstract data type.*

The gathering of hidden data structures and appropriate operations is called *encapsulation.* The data structures embedded in an abstract type are called *members data*, and the operations making up its interface are called *methods* and form the *specification* of the abstract type.

The specification should be complete in the sense that no access to a variable of the type should neither be necessary nor even possible, without going through the specified operations.

This will increase the data security.

7.4.2 Inheritance

Inheritance or *class derivation* is a mechanism by which OOP languages allow relations between types and sub-types to be defined.

New abstract types can be defined, sharing the properties (including methods) of an already defined abstract type, without having to re-implement these characteristics. The new type inherits all the members data and methods from a defined type, and may modify some of the already defined methods, as well as it may define some new members data and methods. The figure 24 illustrates this concept.

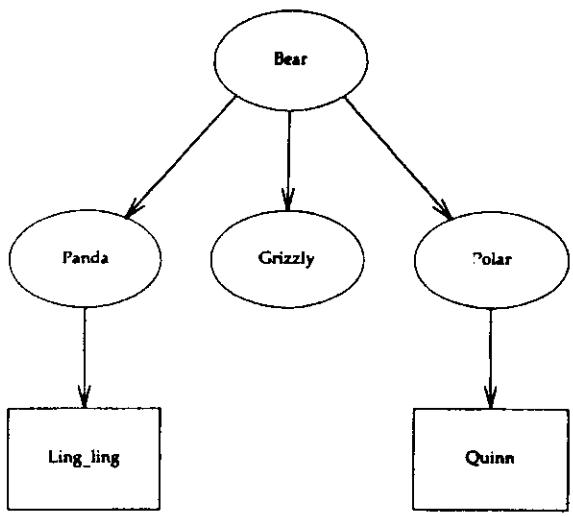


Figure 24: Single inheritance: The bear family class

Some object-oriented languages implement the multiple inheritance concept, allowing a class to be derived from more than one base class, with the aim of inheriting members from different and independent classes. This concept is illustrated in figure 7.4.2.

The new type is said to be a *subclass* or *derived class* of the original type, which is a *superclass* or *base class*. We will adopt this terminology from now.

A derived class has obviously to be declared as inheriting, by specifying it's base class.

A derived class can itself be an object of derivation, as seen on the figure 26.

In the base class, the hidden objects have to be declared as accessible to the derived classes: C++ defines three access levels for the members data or functions of a class: public, private and protected.

The *public* declaration in a class enables visibility and access from outside, and usually includes the manipulating functions of the embedded objects.

The *private* declaration is provided for the hidden data structures and functions, that are not accessible, even by a derived class.

The *protected* declared members and methods are only accessible by the derived classes.

The zoo animals fit nicely in an inheritance hierarchy, as already seen in figures 24 and 7.4.2. The figure 26 show a three level inheritance hierarchy with multiple base classes and multi-level derivation.

7.4.3 Polymorphism

Derived class variables (objects) can be assigned to its base class variables. Only the inherited methods and data structures will be copied, specific members added after the derivation will be ignored. This rule is very important to guarantee the compatibility between related classes, and implies that an object declared of the base class can take the shape of any object of the derived classes. This peculiarity is known as the *polymorphism* concept.

The special relationship existing between derived classes promotes a *generic* style of programming. The polymorphism mechanism implies the *dynamic binding* concept, which authorises the run-time address resolution of the method to use for a specific object.

C++ provides *virtual member functions*, which can be implemented differently for each derived class, despite it is referred to by the same name for all these classes, and the adequate function can be binded at run-time.

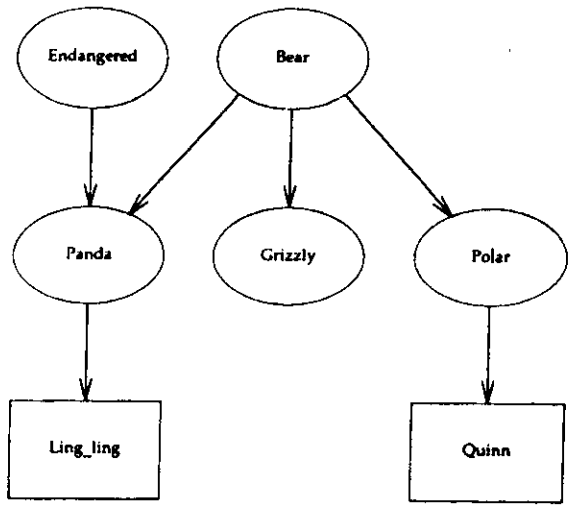


Figure 25: Multiple inheritance: The bear family class, with endangered species indication. Notice that multiple inheritance transform the tree structure of the class hierarchy into a directed graph structure.

A function should only be declared virtual if the class is supposed to be a base class, the implementation of the function is type-dependent, and it will be called through the base class. In the other cases, the code will be more efficient if the function is declared as a usual member function. The multiple definition process necessary for implementing the class-dependent versions of a function is called *overloading.*

In C++, even basic operators can be overloaded. One may define, for example, a + (plus) operator for adding strings, graphs, or stacks. The multiple declaration of the + operator stands out the necessity to choose from the different functions at some point. If this choice is done at the compilation time, this is called *early-binding* or *static binding.* The *late-binding* approach, where the choice is carried out at run-time, is used with virtual functions (dynamic binding).

7.5 OOP Languages

Languages supporting the different concepts of object-orientation to a certain extent include amongst other Ada, C++, Eiffel, Oberon, Simula, and

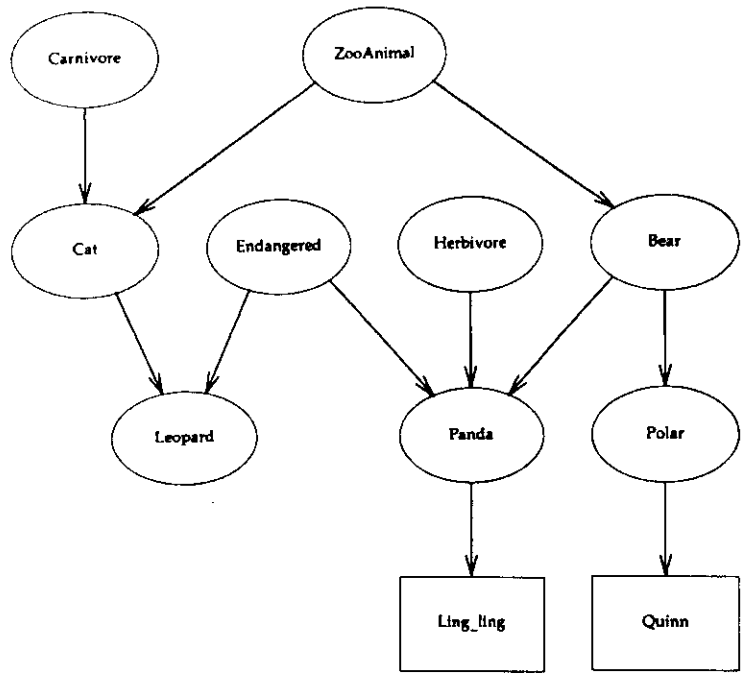


Figure 26: Complete inheritance: A zoo animal inheritance graph.

Smalltalk. We will restrict our view to C++, which is not the most secure and consistent object-oriented language, but which is compatible with its predecessor, the C language, with all the advantages and drawbacks this compatibility involves.

The Eiffel, Simula and Smalltalk are real object-oriented languages, while Ada and Oberon are conventional languages with minimal object-oriented programming support: the concept of object is defined, but neither classes, nor inheritance, even though these concepts may be simulated with some programming effort.

7.5.1 C++

C++ is an extension to C language, for supporting object orientation. The most important extensions from this point of view are

- data abstraction
- operator overloading
- classes with multiple inheritance
- objects with dynamic binding

Abstract data types: Classes

Classes consist of data and functions members, and is divided into a public and a private part. The public part describe the interface, while the private part is inaccessible for heirs and clients. Members may be declared as protected in order to be used by subclasses.

The declarative part of the class is stored in a separate header file, inserted in each file using the class, as well as in the implementation file for the class itself.

Inheritance: Derived classes

Publicly inherited members are public both in the superclass and in the subclass. Privately inherited members cannot be accessed from outside the subclass, and thus inhibit polymorphism.

Multiple inheritance is supported. Virtual derivation allows multiple inheritance while avoiding multiple copy of inherited parts.

Polymorphism:

The assignation of subclass objects to variables of the superclass is allowed, with the previously mentioned exception.

Dynamic binding:

Virtual functions allow to override inherited methods. Functions declared as such in the superclass are dynamically binded.

Objects:

Objects are created either by declaration or by the *new* operator.

An initialisation procedure called *constructor* is automatically called by the compiler each time an object is created. This procedure has the same name as the type. Another function of each class is the *destructor* which is used to delete objects, as well as the memory allocated by those objects.

8 Real-Time Systems

Real-time applications are characterised by the strict requirements they impose on the timing behaviour of their system. Systems ensuring that those *timing requirements* are met are called *real-time systems.* We will exclude from the beginning the *transactions processing systems* (seat reservations, banking), where the transactions are done in real-time, but without any constraint.

8.1 Concurrent and Real-Time Concepts

A *concurrent program* is a non-sequential program, in the sense that some operations are performed simultaneously. This technique, obviously useful in the case of a multiprocessor system, can even be attractive in a monoprocessor environment, to take full advantage of the independence of the processor and the peripherals.

Consider for example that we want to write characters on a terminal. The figure 27 illustrates the activities of both the processor and the terminal interface.

- The processor has to wait until the terminal is ready to accept a character, it then sends the character to the interface and loops back to its waiting state.

- The interface waits for a character, accepts it, write it to the screen and loops back to its waiting state.

That description shows that both processes are waiting for an information given by the other party, before doing any useful task. This is solved by task or process *synchronisation*. In this example, the synchronisation is done for one way by an interrupt, and for the other direction by means unspecified at this point. There are several mechanisms able to signal that the character is ready to be processed by the interface process.

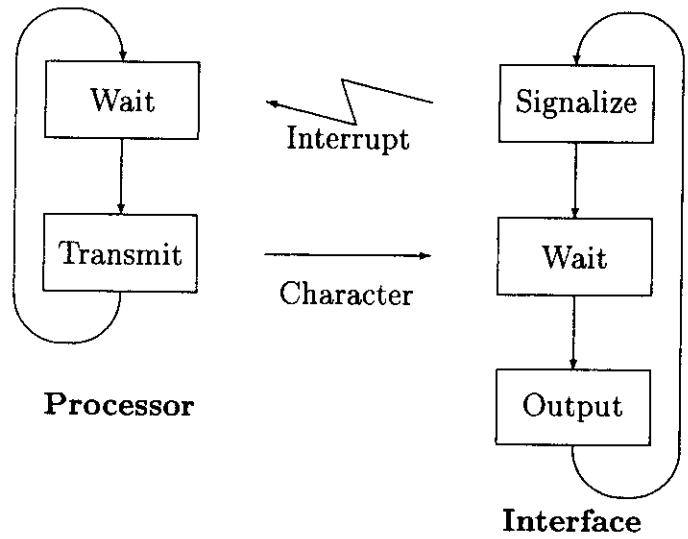


Figure 27: Respective activities of processor and terminal interface for writing a character.

During this time, a concurrent program can perform another task!

Of course, even with the synchronisation, one of the two processes will be faster than the other. In our example, the processor will be mostly waiting for the interface to be ready.

Concurrent tasks should avoid accessing shared data simultaneously. This could lead to incoherent informations if two processes write at the same time in a data structure. Concurrent programs always present these two problems:

- Mutual exclusion (Critical resource access).

- Synchronisation between processes.

These problems are solved by tools (mechanisms) specific to concurrent programming, called *locks*, *events*, *semaphores*, *monitors*, *mailboxes*, *rendez-vous* or *interrupts*.

A *real-time program* is very much like a concurrent program. It has to manage peripherals, and the mechanisms mentioned above still apply. A real-time program includes a supplementary issue: *timing constraints* imposed by the fact that a real-time program controls an external system.

With the improvement of the performance of the microcomputers, and as their price, size, weight, and power requirements decrease, real-time systems are more and more widespread.

Current fields of applications include scientific instrumentation, medicine, industry, cars and military. For example, a real-time system may drive and monitor an astronomical telescope or an X-ray medical scanner, control an industrial production line or a car motor and navigation system, as well as drive a weapon delivery system or control a entire nuclear powerplant.

You have noticed that the word control or a synonym come often in those examples:

Timing and *control* are the master-words in the real-time systems world.

In general, we'll call real-time system any system meeting external timing constraints and able to solve these constraints during its execution; without any specification on the architecture of the system.

A Real-time system can be divided into two groups: The *hard real-time systems*, for which a failure to meeting the timing constraints is considered as a major failure (crash) of the system, and the *soft real-time systems* that will give an error or a warning on such failures, without stopping execution.

8.2 Embedded and Distributed Real-Time Systems

Many complex systems require nowadays an elaborate control system to support their internal functioning. Such systems often use a dedicated computer as controller. Such a computer is called an *embedded computer*.

An embedded computer system has to control the rest of the system. It gets information like data and status from sensors, then issues control commands to actuators.

One feature that distinguishes embedded systems from other real-time sys-

tems is that they are only executing a task relative to a fixed and well-defined workload. They don't provide any development environment.

Study of embedded systems must consider the controlled system as a whole: In particular, mechanical, electro-mechanical parts and electronics should be considered at the specification level of such a real-time embedded system.

The most general way of defining a real-time system is to consider a *multi-machine, distributed computing environment.* The term multi-machine implies that, in addition to the internal timing constraints due to its peripherals, each machine (node) has to deal with timing constraint requests of the other nodes of the system.

8.3 Implementation Issues

Most of the real-time applications cannot be programmed with traditional languages under a traditional operating system, or at least at their standard level, as those languages don't know how to handle the timing constraints imposed by the system. Additional features known as *real-time extensions* are defined for some languages, enabling such systems to be programmed and checked. These extensions often enable the programmed real-time system to override the operating system mechanisms to control directly the hardware.

On the other hand, real-time systems can be programmed with classical languages such as C, if there is a library of functions implementing the real-time mechanisms. In this case, the real-time aspects of the application is shared between the language and the real-time operating system (LynxOS, OS/9).

Another aspect of the implementation of complex, multi-machines real-time applications is the operating system. The traditional approach to multi-tasking operating systems design is to split the time in slices and to attribute those slices to the different computing-resources demanding applications. This kind of management is called *time-sharing.* Time-sharing doesn't address correctly the problems arising in real-time systems.

So, the execution of real-time applications has to be supported by a correct environment, which is obtained through a *real-time operating system.*

These real-time operating systems have to manage timing and interactions problems. Different mechanisms allow them to handle timing constraints correctly, including *interrupts* and *signals.* They also contain mechanisms to solve the processes scheduling problem, that can be quite difficult, with

preemptive tasks and *dynamic priority* setting. Another aspect treats the communications between tasks, with *semaphores* and *shared data* zones.

8.4 Time Handling

Time handling is the most important issue in real-time systems. Time handling includes:

- Knowledge of time
- Time representation concepts
- Time constraints representation

8.4.1 Knowledge of Time

Time is given by *clocks*. In a multi-machine environment, multiple clocks may exist and should be *synchronised*, in order to get a coherence between the different timing constraints and interactions specifying the real-time system.

A clock is characterised by its *correctness*, which defines the quality of the knowledge of time, and by its *accuracy*, which defines the way the clock *drifts*. The accuracy is given by the derivative of the clock signal, as shown by the following definitions:

A *standard or reference clock* is one for which the relation

$$C(t) = t \ , \forall t$$

is confirmed. A clock is *correct* at time t_0, if

$$C(t_0) = t_0$$

A clock is *accurate* at time t_0, if

$$\left.\frac{dC(t)}{dt}\right|_{t_0} = 1$$

8.4.2 Clock Systems

There are different clock systems.

The simplest one consists of one central *clock server*, that should be very accurate and reliable, even though a redundant system can be used. Therefore, this kind of clock system is quite expensive.

Another type of clock system defines a *master clock* polling multiple slave clocks, measuring their differences and sending to them the corrections to do. All the clocks can be of the same accuracy, and if the master fails, another one amongst the other is elected to become the new master. This type of clock system is called *centrally controlled.*

A *distributed clock system* consists of an interlinked network of clocks, which all run the same algorithm, polling the other clocks to get their time, and then estimate their correctness. This type of system can be simple or enhanced, depending on the complexity of the algorithms used at the nodes, and implies a *relatively heavy traffic load* on the communication network.

The graph linking the nodes can be *closely connected*, with any of the clock polling all the others, or *loosely connected* with only a subset of the connections used for time synchronisation.

A protocol named *xntp* working through network with the *UDP protocol* is publicly available, and works as a distributed clock system with a hierarchy defining more or less reliable clocks. This hierarchy is organised in levels (strati), a lower level number meaning a more preemptive clock. Each node can be configured to communicate with a certain number of other clocks, either for synchronising itself (same or lower levels), or to only read the time on higher level clocks.

The Global Positioning System (GPS) is a satellite based navigation system providing precise position, velocity and time information. The heart of the GPS consists of 21 satellites and three spares, that revolve round the earth twice a day, at an altitude of 20000 kms. They allow a 24 hours per day worldwide coverage by more than 3 satellites. This system can be used by special hardware to get a good timing information to synchronise clocks. The receivers are cheap (about $ 600-1000).

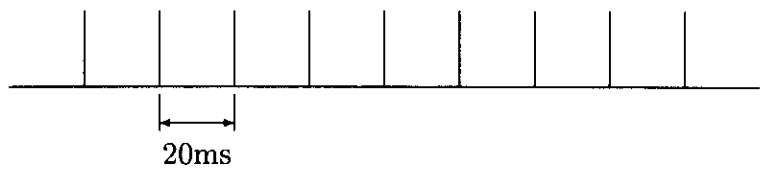
Other special hardware may take advantage of the time signals broadcasted by radio waves from different standard clock systems in the world, as DCF in Germany, WWV in Boulder, Colorado, WWVH in Hawai or JJY in the Pacific North.

8.4.3 Time Representation

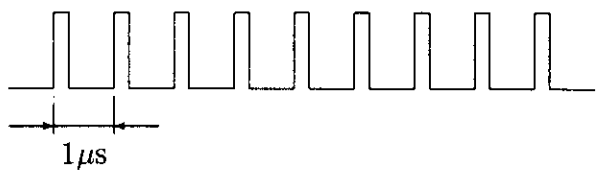
Time representation in real-time systems should be sufficiently well-designed to take into account the properties of the system, and to allow a precise definition of the characteristics of the time constraints.

As a preliminary definition, we should state that the *time granularity* of a system is the clock resolution. This notion is more complex than it seems. Each operating system uses a system clock (fig. 28a) to manage the timing synchronisation between processes. This clock gives interrupts to the system at a certain rate, which can usually be modified, but which should neither be too high, for fear of excessive system overhead, nor too low, because it would penalise the interactive processes by a long response time. This time is usually about some tens of milliseconds. This gives the granularity for scheduling processes, or time-slicing in a classical operating system.

There is another clock used for time measurement (fig. 28b), which can also be used to drive a programmable timer for scheduling events at certain time. This is called the real-time clock, and has a granularity of about microseconds. A real-time operating system will usually use this clock to synchronise the processes or manage timing constraints.



a) System Clock



b) Real-Time Clock

Figure 28: Different clocks are defined in a system.

Point-based representation defines events of zero-length duration, occur-

ring at some time instants in a system, which are responsible for a change in the state of the system.

Interval-based representation defines activities of finite duration, having a start and a stop time. These activities can exist simultaneously.

Both approaches have their drawbacks:

Point-based-representation	Interval-based-representation
Events cannot be decomposed while maintaining an order, as they have no duration. Partially overlapping activities cannot be described by this model.	It is difficult to take into account the time granularity of the system.

The best solution is highly dependent of the system, but will often be based on a compromise between both approaches, leading to an *interval based representation, with system's granularity support.*

8.4.4 Timing Constraints Representation

A real-time system has to deal with the arrival of time-constrained requests, i.e. the invocation of processes to be executed in due time.

The system has to allocate the resources to meet the specifications, in order that the process can begin at a specified time, and be completed at another specified time.

The minimal definition of a timing constraint is the triple

$$(Id, T_{begin}(condition1), T_{end}(condition2))$$

where Id is the name or ID-number of the process.
$T_{begin}(condition1)$ is the starting time of the process.
$T_{end}(condition2)$ is the completion time of the process.

Depending on the system and the temporal uncertainties on the allocation time of certain resources, we may need some additional time parameters in the constraint representation.

In particular, the completion time may not be a very severe constraint, and in case of earlier process completion, the resources should be freed for other processes.

On the other hand, a very long process should not monopolise the resources of the system, and the global efficiency of the system would be improved, if time-slices were attributed to this process.

This leads to the more mature definition of a timing constraint as the quintuple

$$(Id, T_{begin}(condition1), c_{Id}, f_{Id}, T_{end}(condition2))$$

where Id is the name or ID-number of the process.
$T_{begin}(condition1)$ is the starting time of the process.
c_{Id} is the computation time of the process, or the time-slice.
f_{Id} is the frequency with which the time-slices have to be attributed.
$T_{end}(condition2)$ is the completion time of the process.

8.4.5 Interrupts driven Systems

Interrupts are often used as a *synchronisation mechanism* in real-time systems, particularly in control applications.

An *interrupt* is a signal occurring asynchronously and triggering a *service routine*. This routine is called by the *interrupt handler*, which identifies the interrupt, locates in a table the appropriate address, and passes it to the program counter (instruction pointer). The handler or the service routine itself has to save the current environment before beginning processing the request, as it could modify this environment.

A signal enabling the interrupt system (IE) is disabled by the acceptance of an interrupt by the handler. It is usually the service routine's responsibility to re-enable it, at some time. In the figure 29, we have a first interrupt arriving (IR5). The interrupt handler accepts it, as there are no other interrupts being processed, and passes control to the IR5 service routine. A second non-preemptive interrupt arrives before the routine has released the IE signal. This interrupt is blocked for a while, until the interrupt handler being re-enabled. Then it is normally processed. This illustrates the fact that response time to interrupt may vary.

The routine has to be carefully designed to meet the time constraints on it's duration, deadline and frequency. Sometimes, the task has also a starting

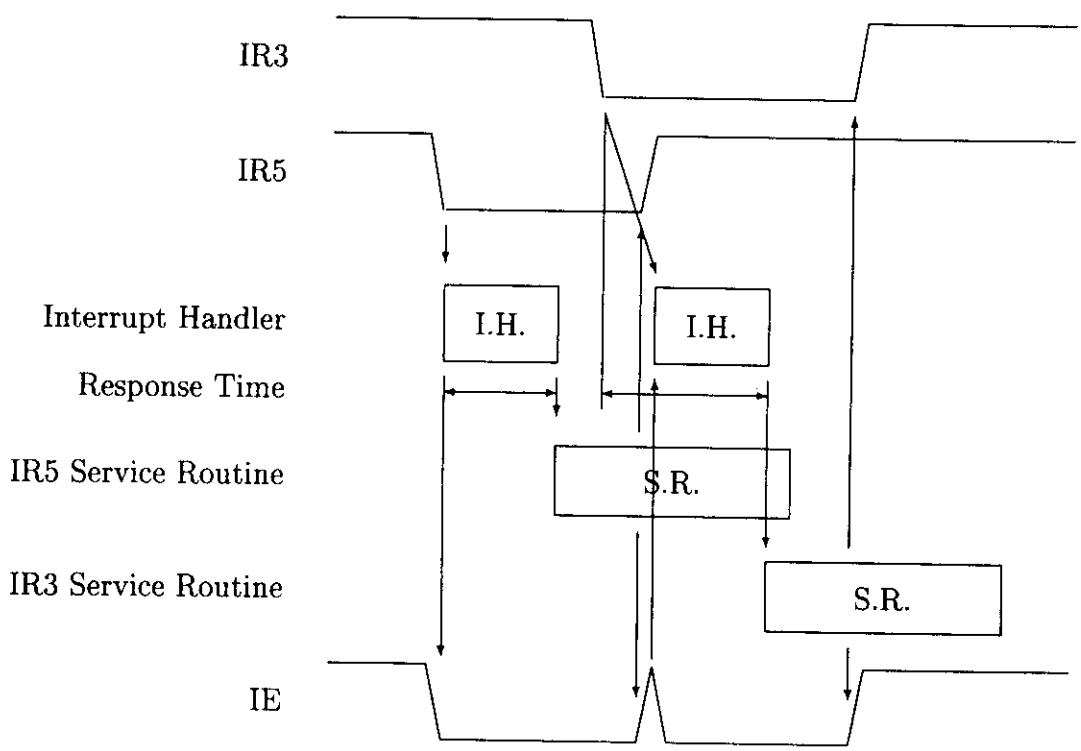


Figure 29: Interrupt Service Scheme

time condition, in which case it can be executed only if both the interrupt has occurred, and the starting condition is met.

8.4.6 Signal Synchronisation

Another way to synchronise processes is to signal certain states of the system. Typically, one process needs the system to be in a certain state which it cannot control for continuing it's execution. Arrived at that point, it checks a signal specifying the desired state, and if unsatisfied, waits until the signal arrives, indicating the change in the system state.

On the other hand, another process is responsible of modifying the state of the system, and has to signal it after completion. This method leading to *mailbox* or *rendez-vous synchronisation* does not fit well to real-time systems, because it cannot ensure that deadlines are respected, and is mainly used for concurrent processing.

8.5 Real-Time Systems Design

The design of any system should begin by a *requirement specification* phase, followed by the design phase itself. These phases will be followed by the implementation, tests, etc. The design phase can also be decomposed into a preliminary and a detail phase. The different phases and sub-phases may sometimes overlap each other in time.

Take care that a too rigid approach in the design, obtained for example by avoiding any time-overlap between phases, may lead to a very formal and well-documented design, but that will possibly be neither creative nor the best one.

Another aspect is that a project is in itself very much like a "real" real-time system, with timing constraints and deadlines. To achieve a project in the specified delays, one will tend to minimise the specification and design phases to begin as quickly as possible the implementation. This attitude may lead to a badly-designed and possibly fragile system. A better way is to begin the implementation of well-designed parts while refining the design of the rest, ensuring both a good overall design and a quick development of the system.

Let's examine the two phases of the design.

8.5.1 Requirements Specifications

The requirement specification phase is important in real-time systems, because the descriptive aspect of the document enables to easily include the timing constraints.

The requirement specification document should:

- state external behaviour of the system.
- avoid specifying any implementation details, but only constraints on the implementation, as the details of the hardware interface.
- state the responses to the exceptions.
- be easily modified.
- be well documented to serve as a reference during all phases of the project.
- specify the timing constraints and deadlines of the project itself.

Some systems may be described in a verbose documentation style only, while others may need some more sophisticated tools as, for example, *state-charts.*

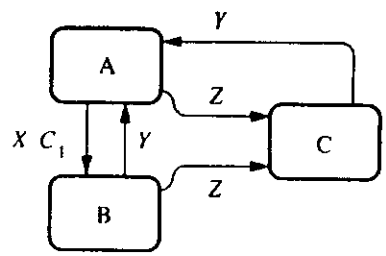


Figure 30: State-chart example.

8.5.2 State-Charts

State-charts describe the system as *states* and *transitions* between them, triggered by *events* and *conditions.* States are represented by boxes, transitions by arrows, events and conditions are labels for the arrows (figure 30).

States can be decomposed to lower level states or combined into a higher level state (figure 31). These operations are called *refinement* and *clustering.*

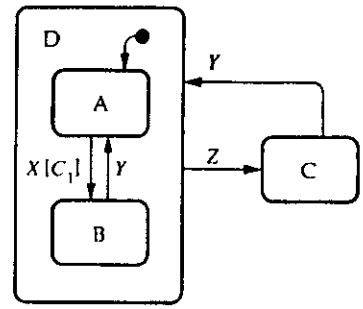


Figure 31: Clustering states in a state-chart.

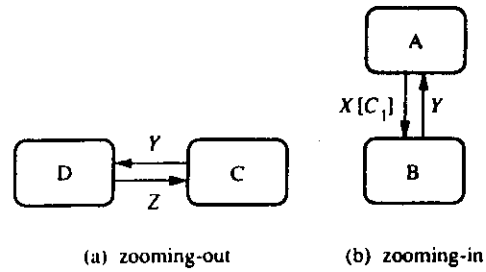


Figure 32: Zooming in and out.

Zooming in and out (figures 32) enables one to have different levels' views of the system.

8.5.3 Petri Nets

The complexity of real-time systems is essentially due to the interactions between tasks, the access conflicts and the temporal evolution of the system. It is necessary to use powerful tools to represent the evolution of such a system at the conception level. The *Petri net representation* is a very powerful tool, which enables to represent the interactions between processes and the evolution of processes.

A Petri net is a quadruple $C = (P, T, I, O)$ including N *places* $p_i \in P$ and L *transitions* $t_i \in T$. The structure is described by two matrices I and O of dimension $L \times N$ specifying *inputs* and *outputs* viewed by the *transitions.*

The elements of those matrices are integers specifying the weight of the link between a place and transition. The absence of a link is obviously described by a weight $w = 0$.

A Petri net can be represented by a *Petri graph*, with two types of nodes: places and transitions. The directed edges may only link nodes of different type. As an example, a Petri net described by

$$
\begin{aligned}
C &= (P, T, I, O) \\
P &= |p_1, p_2, p_3, p_4, p_5| \\
T &= |t_1, t_2| \\
I &= \begin{array}{c|ccccc|c} & p_1 & p_2 & p_3 & p_4 & p_5 & \\ \hline & 1 & 1 & 2 & 0 & 0 & t_1 \\ & 0 & 0 & 0 & 0 & 1 & t_2 \end{array} \\
O &= \begin{array}{c|ccccc|c} & p_1 & p_2 & p_3 & p_4 & p_5 & \\ \hline & 0 & 0 & 0 & 1 & 2 & t_1 \\ & 0 & 0 & 1 & 0 & 0 & t_2 \end{array}
\end{aligned}
$$

is represented by the graph of figure 33

This definition of a Petri net enables only the static representation of a system. To modelize the temporal evolution, the Petri net is completed by *marking.* A marked Petri net represents a state of the system. Marking *tokens* are represented by dots on the graphs (fig. 34).

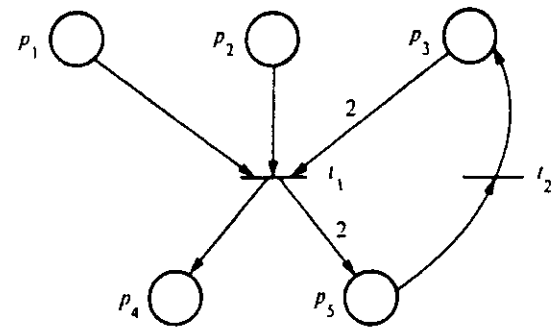


Figure 33: Petri graph with weighted arcs

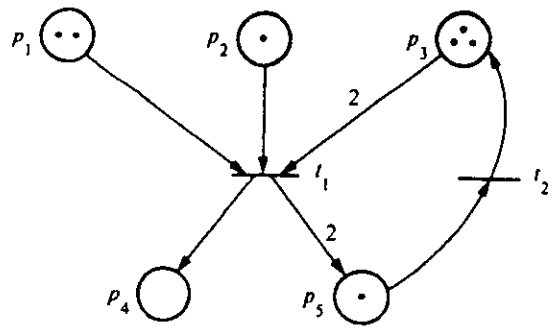


Figure 34: Marked Petri graph

A marking is a N-dimensional vector specifying the numbers of tokens in each place. The system becomes dynamic when the tokens travel through the net. The travelling is done through *transition firing.* A transition may be fired only if all the preceding places are marked (active). This transition is said to be enabled.

Only one transition is fired at a time, randomly chosen between enabled transitions. A firing has the following effects on the places preceding and succeeding the transition:

- w token is removed from each preceding place.
- w token is put in each following place.

Firing is:

Voluntary An enabled transition may be fired, but it is not mandatory.

Instantaneous All the operations related to a firing occur simultaneously, and take no time.

Complete All the operations related to a firing do occur.

The figure 35 shows the result of firing transition t_1 in figure 34.

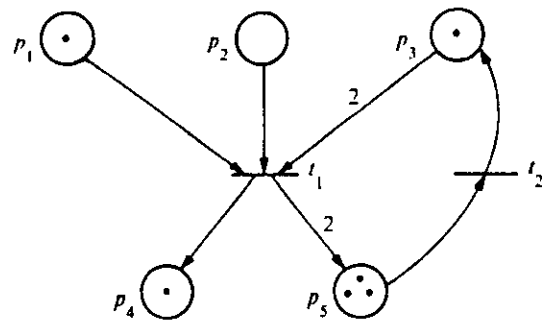


Figure 35: Petri graph after the firing of t_1.

A Petri net may be annotated as shown in the figure 36 illustrating the allocation of a processor: As soon as the processor is idle (p_2 marked) and there is a task waiting in the queue (p_1 marked), the processing may begin (t_1). The task is executed (p_3 marked). At the end (t_2), the task is completed (p_4 marked), and the processor is deallocated (p_2 marked).

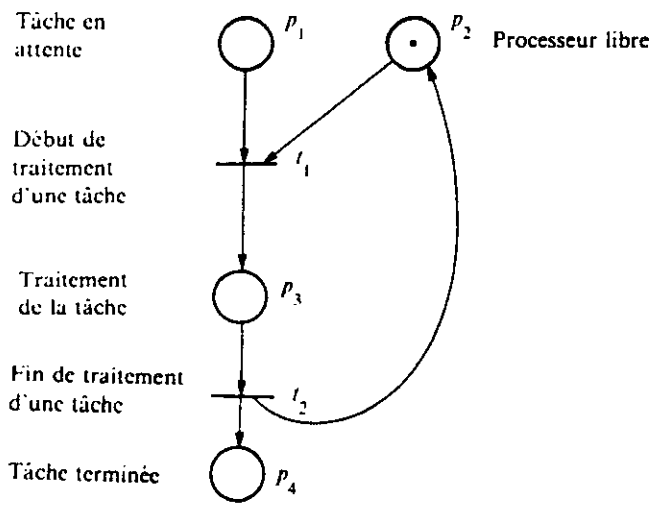


Figure 36: Petri net modelizing a processor allocation.

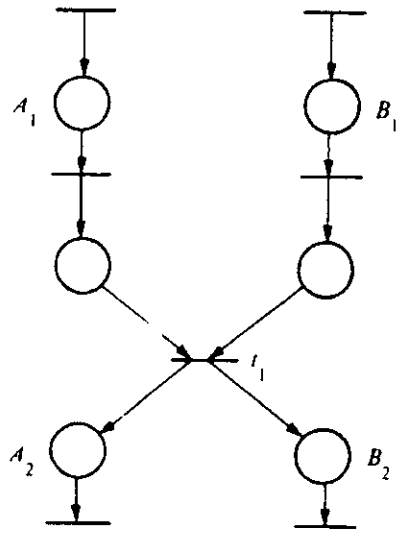


Figure 37: Petri net modelizing a *rendez-vous* type synchronisation.

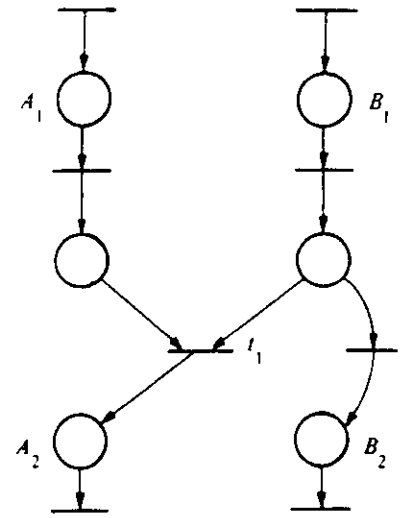


Figure 38: Petri net modelizing a *mailbox* type synchronisation.

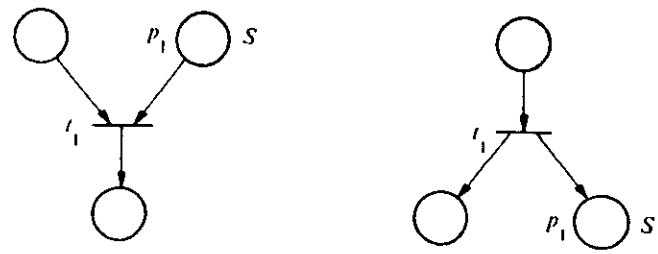


Figure 39: Petri net modelizing the semaphores primitives $P(s)$ (left) and $V(s)$ (right).

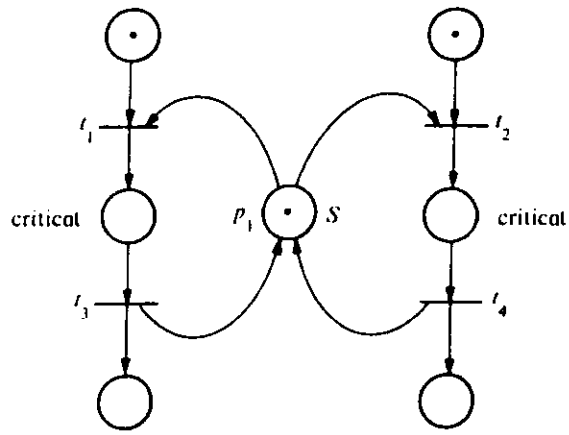


Figure 40: Petri net modelizing mutual exclusion by semaphore.

Without going into the details of the Petri net model, we can say that conditions are associated to places, and events to transitions. The figures 37-40 show Petri nets representing some real-time issues.

The Petri net model may be used by the designer in a kind of top-down structured approach (figs. 41-44) :

- Start with a global Petri net model of the system (fig. 41).
- Stepwise refine it by substituting (fig. 43) the transitions by *well-formed blocks* (fig. 42) . A well-formed block should have only one input and one output (fig. 44).

The Petri nets can be transformed to flowcharts. The nodes of the flowcharts are associated to the Petri net transitions, while the arcs will replace the places (figs. 45 and 46).

8.6 Structured design of Real-Time Systems

In addition to the concepts of structured design, we have to address the notions of timing constraints and interprocess communications. *DARTS* (Design Approach for Real-Time Systems) was developed by General Electric to extend the notion of structured design to include *process decomposition* and *process interfacing.*

First, an analysis of the system has to be done in terms of functions: The system is then viewed as a *data flow* transformed by *functions.*

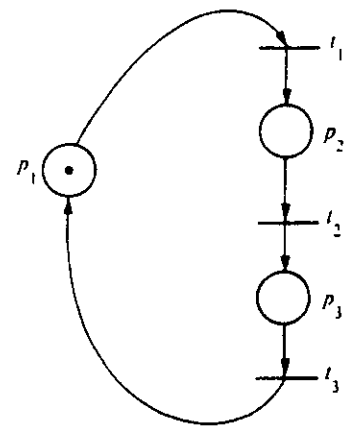


Figure 41: Initial step for structured design.

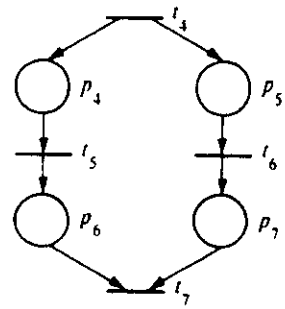


Figure 42: Block example.

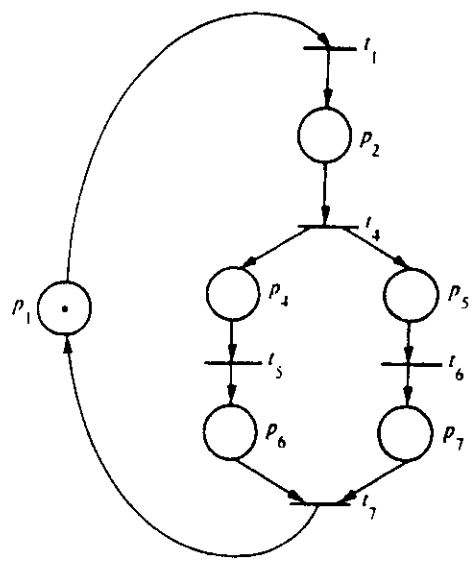


Figure 43: Replacement of t_2, p_3, t_3 in fig. 41 by the block of fig. 42 .

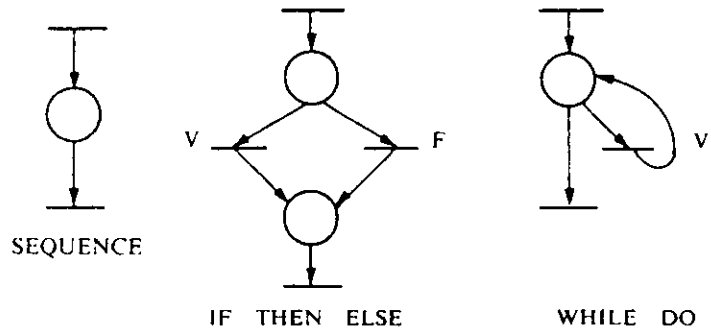


Figure 44: Well-formed blocks.

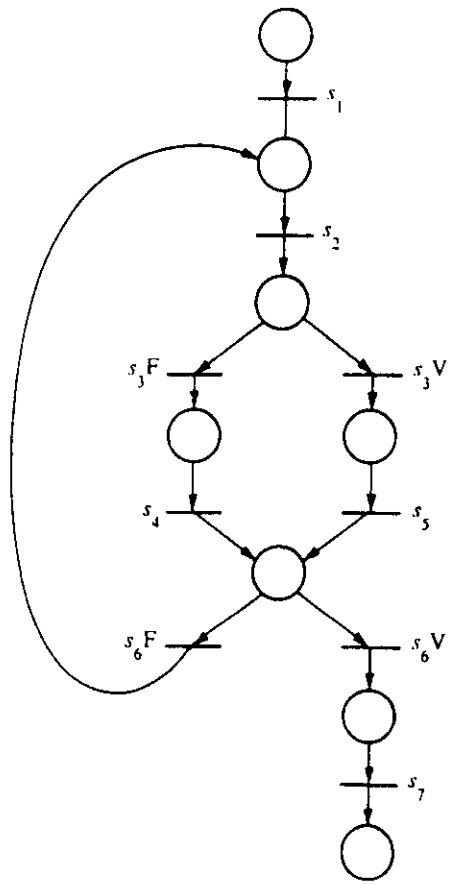


Figure 45: Petri net example.

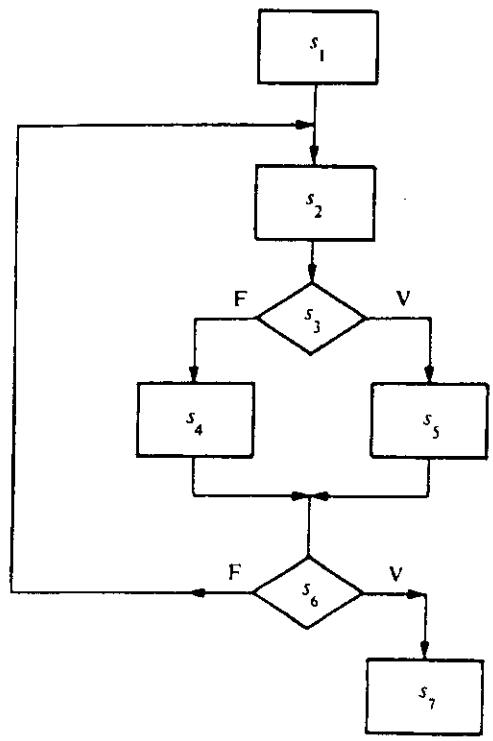


Figure 46: Flowchart for the Petri net of the figure 45.

8.6.1 Process Decomposition

When the functions have been identified and described, they must be assigned to processes. DARTS defines criteria to assign a function to a separate process, or to group it in a process with other functions:

I/O dependency If a slow peripheral dictates the speed of execution of a function, this function should be put in a separate process.

Time-critical functions High priority functions should be kept in a separate process.

Computational requirements Intensive computation functions should receive a separate process.

Functional cohesion Closely related functions should be grouped in a process.

Temporal cohesion Functions triggered by the same stimulus should also be grouped.

Periodic execution Periodically executed functions should be kept in a separate process.

So we see that functional and temporal cohesion are a criterion to group function in a single process, where they can still be separated and distinguished by creating modules inside the process. Timing constraints and special requirements justify on the other hand separate processes.

8.6.2 Interprocess Communication

DARTS provides two types of modules for the communication between processes:

- Message communications modules (MCM).
- Information hiding modules (IHM). It is used mainly in cases of shared data. IHM defines the data structure in a hidden way, with procedures to access it.

The figure 47 shows three processes P_1, P_2 and P_3 communicating through the data they share, and which is defined in the module IHM, with the data hidden in structures B and C, accessed only through the procedure a.

Please notice how close this approach is from the object concept.

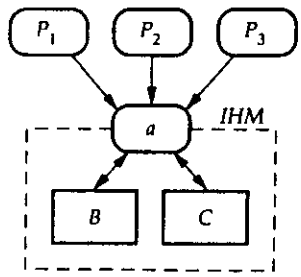


Figure 47: IHM module

9 References and Bibliography

Structured Programming

- **Dahl O., Dijkstra E.W. and Hoare C.A.R.,** Structured programming, *Academic Press 1972*

- **Dijkstra E.W.,** A discipline of programming, *Prentice-Hall 1976*

- **Kruse R.L.,** Data structures and program design, *Prentice-Hall 1984*

- **Wirth N.,** Program development by stepwise refinement, *CACM* **14**, *221-227 (1971)*

- **Wirth N.,** Systematic programming, *Prentice-Hall 1973*

Algorithms & Data Structures

- **Knuth D.E.,** The art of computer programming, vol. 1 : Fundamental algorithms, *Addison-Wesley*

- **Knuth D.E.,** The art of computer programming, vol. 2 : Seminumerical algorithms, *Addison-Wesley*

- **Knuth D.E.,** The art of computer programming, vol. 3 : Sorting and searching, *Addison-Wesley*

- **Krob D.,** Algorithmique et structures de données, Programmation, *Ellipses 1989*

- **Lipschutz S.,** Data Structures, *McGraw-Hill 1986*

- **Wirth N.,** Algorithms & Data Structures, *Prentice-Hall 1986*

- **Sedgewick** Algorithms *Addison-Wesley 1983*

Object Orientation

- **Aubert J.-P. and Dixneuf P.,** Conception et programmation par objet, *Masson 1991*

- **Blaschek G., Pomberger G. and Strizinger A.,** A comparison of object-oriented programming languages, *Structured programming* **4**, *187-198 (1989)*

- **Booch,** Object-oriented design with applications, *Benjamin/Cummings 1991*

- **Quément B.,** Conception objet des structures de données, *Masson 1992*

- **Voss G.,** Object-oriented programming, *McGraw-Hill 1991*

- **Reiser M.** The Oberon System *Addison-Wesley 1991*

- **Reiser M. and Wirth N.** Programming in Oberon, Steps beyond Pascal and Modula *Addison-Wesley, ACM Press 1992*

- **Object behavior analysis,** Rubin K.S. and Goldberg A., *CACM* **9** *(1992)*

Concurrent and Real-Time Programming

- **Levi S.-T. and Agrawala A.K.,** Real-Time system design, *McGraw-Hill 1990*

- **Nussbaumer H.,** Informatique industrielle, vol.2: Introduction à l'informatique du temps réel, *Presses Polytechniques Romandes 1986*

- **Schiper A.,** Programmation concurrente, *Presses Polytechniques Romandes 1986*

Languages

- **Darnell P.A. and Margolis P.E.** C, A Software Engineering Approach *Springer-Verlag 1991*
- **Oualline Steve** Practical C textslO'Reilly & Associates 1993
- **Hanly, J.R. and Koffman E.B.** Problem Solving and Program Design in C *Addison-Wesley 1996*
- **King K.N.**, Modula-2, *D.C. Heath and Company 1988*
- **Lippman S.B.**, C++ Primer, *Addison-Wesley 1989*
- Borland C++ Documentation, *Borland International 1989*
- **Thorin M.**, Ada, Manuel complet du langage avec exemples, *Eyrolles 1981*

UNIX Tools

- **Kernighan B.W. and Plauger P.J.** Software Tools *Addison-Wesley 1976*
- **DuBois Paul** Type Less, Accomplish More Using csh & tcsh *O'Reilly & Associates 1995*
- **Bolinger D. and Bronson T.** Applying RCS and SCCS *O'Reilly & Associates 1995*
- **Miller W.** A Software Tools Sampler *Prentice-Hall 1987*
- **Manis R. Schaffer E. Jørgensen** UNIX Relational Database Management, Application Development in the UNIX Environment *Prentice-Hall 1988*

Use of man pages and apropos

One should not forget all the man pages, either interactively on the screen, or in printed form. The man pages for gcc, in particular, are very detailled.

When printed pages are realy needed, they can be produced with

```
man command  | lpr
```

or, if troff is installed,

```
man -t command
```

`man -k keyword` and `apropos keyword` can be used to retrieve command names that are related to some keywords.

Here is an example:

```
obssq18:~ 551> apropos administration
admind              admind (1m)       - distributed system administration daem
admintool           admintool (1m)    - system administration with a graphical
dispadmin           dispadmin (1m)    - process scheduler administration
nis_checkpoint      nis_ping (3n)     - misc NIS+ log administration functions
nis_ping            nis_ping (3n)     - misc NIS+ log administration functions
nisgrpadm           nisgrpadm (1)     - NIS+ group administration command
nistbladm           nistbladm (1)     - NIS+ table administration command
nlsadmin            nlsadmin (1m)     - network listener service administratior
pmadm               pmadm (1m)        - port monitor administration
sacadm              sacadm (1m)       - service access controller administratic
obssq18:~ 552>
```

10 Think

Think !

- think before doing
- think while doing
- think after having done
- your are responsible, you are the master
  never give it to μP
- μP must obey, not dictate

Think small !

- 'Small is beautiful'
- keep things manageable, under control
- use small modules

Think with others !

- do not reinvent the wheel
- make your work shareable
- build-up libraries
- accept help, call for help
- the others can and must think too

Think on your own !

- do not accept buzz words for granted
- adapt to your own country
- do not destroy your richness
- never accept dogma