Real-Time Operating Systems

C. Verkerk[†*]

[†] *International Centre for Theoretical Physics*
Trieste, Italy

Lecture given at the:
Second Workshop on
Distributed Laboratory Instrumentation Systems
Trieste, 20 October — 14 November 2003

LNS

[*]Rinus.Verkerk@cern.ch

Abstract

In these lecture notes the operating system support needed for a real-time application is examined. Various features of Linux are described to provide the necessary background for a discussion on its suitability for hard real-time applications. The efforts to adapt Linux to the requirements imposed on a hard real-time operating system are presented.

Keywords: Linux
PACS numbers: 64.60.Ak, 64.60.Cn, 64.60.Ht, 05.40.+j

Contents

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 may well 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 the behaviour of some apparatus, machinery, or even an entire factory. We call these *real-time control systems*. We are literally 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 and or running 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 alpha-numeric 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 the latter 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 a little further what the components and constraints 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 – and rather superficially – 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 much 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 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 in order to detect any anomaly. These anomalies could then be reported im-

[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.

mediately. 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 of running trains. Likewise, when it arrives at B, it has to be removed from this 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 an entire section of single track, as long as there is still a north-bound train somewhere in this entire section, and vice-versa. A section consists of several blocks and inside a section there are no switches enabling a train to move from one track

to another, nor to put it on a side-track. 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 (north-bound or south-bound) of train should be allowed to use the resource. As long as it is used, the other contender must be excluded until the resource has become free again. If the stretch of single track is long enough, and comprises several 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 northern 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 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 profit from features provided 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 it, 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.

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-9000* for *Motorola 68000 machines*, and *QNX* or *LynxOS* for *Intel machines*, *Solaris* for *Sparc processors*, to mention only a few of the older ones. 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 systems, which you *compile and link into your application*. *VxWorks, MCX11* and *μCOS* are rather oldish examples. They are much cheaper —or practically free: *MCX11* and *μCOS*—, 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 or time constraints, of course.

When we analyse 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 organise and control 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, 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 in the system. 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 process that is using the CPU 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[1].

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 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 before some event happens*. 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 first to complete an output operation, 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

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

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 our suspended 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[1].

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 this will cause the process that will update these three bytes to be woken up. 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 occurs. As the process that updates the clock has a higher priority than the running process, the latter is suspended. The clock process now updates the time, setting it to 11.00.00. Control then 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

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

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. The test and the locking are done in a single **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 process 1 previously had found the lock to be open and it 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 this solution is excluded.

Some machines (for instance the Motorola m68000 series) 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, making it a candidate for use as a lock.

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, for instance by sending it a *signal*. 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 truly go to sleep, forever!

The solution to the problem was given in 1965 by Dijkstra, when he defined the **semaphore**. A semaphore can count the number of such "*lost wakeups*", without trying to wake up a process that is not sleeping. It can only have a positive value, or "0". Two **atomic operations** are defined on a semaphore, which we will call **up** and **down**[1]. Once an operation on a semaphore is started, no other process can access the same semaphore, until the operation is finished. 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 *down* operation checks the value of the semaphore. If it is greater than zero, it decrements the value and the calling process just continues execution. On the contrary, If the *down* operation finds that the semaphore value is zero, the calling process is suspended and the semaphore value left to zero. The *up* operation on a semaphore will always succeed and increment its value. If one or more processes were suspended on this semaphore (i.e. its value was equal to zero), one of them is selected by the operating system. The selected process will then be allowed to run and it can now complete its *down*, which had failed earlier. 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

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

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[1] excellent book[1]. The reader should study carefully the listing of the *Producer-Consumer* problem given here. 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 */
      }
}
```

[1]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.

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 (orprocesses). It has been designed to give a *fair share of theresources*, 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[1]. 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 did not have this feature, but *another scheduler*, designed for mixed time-sharing and real-time use is available and *is usually compiled into the kernel*.

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

[1]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.

ization 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. Originally Linux did not have the possibility of locking processes into memory, but again, *memory locking* is now compiled into all recent kernels.

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 *library implementations do exist* to implement the standard POSIX pthreads.

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[1] devices. They have to be written by the application programmer. In a standard Unix system, such a new device driver

[1]With exotic I really mean *very weird non-standard devices*. The list of devices supported by Linux is indeed incredibly long!

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 each can 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*, usually 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 1003.1a standard, which defines a set of system calls, and POSIX 1003.1b, which defines a standard set of Shell commands. Both were approved by IEEE and by ISO and thus gained international acceptance. Also *real-time extensions to operating systems* have been defined in the *POSIX 1003.1c-1994* (called POSIX-4 in ancient times) standard, which has also been accepted by ISO. All the points discussed above are part of this POSIX 1003.1c standard. *multiple threads and mutexes* are defined in a later extension. The so-called *pthreads* are part of the international POSIX 1003.1cstandard.

In summary, Linux used to be weak on the following, and may still be on a few items:

- **Mutexes**. A simple mutex did not exist in the original Linux kernel. 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 could possibly be used, but extreme care should be exercised (danger of *priority inversion*). Mutexes are defined in the **pthreads** package. They will work between user threads inside a single process, and for some implementations also between threads and another 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 1003.1c compliant scheduler for Linux exists and is part of the kernel in most *Linux distributions*. The perfect scheduler will be part of the 2.6 kernel, expected to be available soon.

- **Memory Locking**. Memory locking is part and parcel of the more recent Linux kernels (at least above 2.0.x and maybe earlier).

- **Multiple User Threads**. A few library implementations exist. The more recent Linux distributions have Leroy's Pthread library, which makes use of a particular feature of Linux : the "*clone*" system call. It is entirely compliant with the POSIX standard. You will soon get into close contact with it. The new 2.6 kernel should have support for a much more efficient implementation of pthreads.

- **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 to my knowledge.

- **Fine-grained real-time clocks and interval timers**. They are part 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* taken from Bill O. Gallmeister's book[2]. The reader is invited to study this example in detail.

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

```
#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 */
  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:

```
#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!\n", 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(), 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 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 standard, except where the facilities are not implemented in Linux İn that case we will describe the mechanism Linux makes available.

6.1 Unix and POSIX 1003.1a 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*, *SIGKILL* or *SIGALRM*, 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 interceptable signal. Certain signals cannot be caught this way, 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.

A very serious limitation to the use of signals in real-time applications is the time that may elapse between the sending of a signal and its *delivery* to the target process. In fact the signal, when sent, is registered by UNIX. When the scheduler decides that the target process will be the next to run, UNIX will check if there are signals pending for that process. If there is one and a handler has been defined for it, UNIX will run the handler first — thus delivering the signal — and then proceed to run the process from where it had left off previously.

Linux is compliant with this POSIX 1003.1a definition of signals.

6.2 POSIX 1003.1c signals

From the description above, we have seen that the POSIX 1003.1a signals are a rather complicated business (in Unix jargon this is called flexibility). The POSIX 1003.1c extensions to the signal mechanism introduces even more flexibility.

POSIX 1003.1c really defines an entirely *new set of signals*, which can peacefully co-exist with the old signals of POSIX 003.1a. 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 1003.1c 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, not all features of the POSIX 1003.1c signals are part of standard Linux , so we will not dwell on them any further.

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 the pipe */
  if (pipe(pipe_ends) < 0) {
    perror("pipe");
    exit(1);
  }
/* Then fork the child process */
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[1], 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 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 1003.1a *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 1003.1c standard. We will nevertheless describe them briefly, as they are the only ones we have at present.

In system V *struct msqid_ds* describes the **message resource**, 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[l]; }
```

To set up a message queue, the creator process executes a *msgget* system call: *msqid = msgget(key_t key, int msgflg);* The *msqid* is a unique identification of the particular message queue which ensures that messages are delivered to the correct destination. The exact role of the key is complicated; in most cases the key can be chosen to be *IPC_PRIVATE*. The use of *IPC_PRIVATE* will create a new message queue if none exist already. If you want to do unusual things or make full use of the built-in *flexibility*, you may fabricate your own key with the *ftok(char* pathname, char proj)* library call and play with *msgflg*.

A process 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:

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

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 a complicated business again. 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 *(IPC_NOWAIT | sem_flg)* is 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.

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*.

The shared memory facility available in Linux comes from System V, and may be **not** conforming to POSIX.1c. 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:

shmid = 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 original scheduling algorithm of Linux aimed at giving a *fair share of the resources* to each user. It therefore was 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*[1].

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

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

priority scheduling. Other jobs run at lower priorities and have to accept the time-sharing scheduler. This aspect of System V has not been ported to Linux.

A POSIX.1c 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.1c scheduler. At the time these notes were prepared, we had not yet had a chance to try it.

The advantage of a POSIX.1c scheduler is, of course, that your application program will be portable between different platforms.

What does a POSIX.1c scheduler do? Here is what it provides[1]:

```
#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);
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

[1] Again from Gallmeister's book.

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 the three following 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*, which is defined in <*sys/param.h*>. On Linux machines, *HZ=100*, so the resolution of the interval timers is 10000 microseconds.

POSIX.1c 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.1c 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 the argument *length* 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) standardises 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 concurrently the same function. The services needed to *create threads, schedule their execution, communicate and synchronise 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[1].

Lightweight Processes, as in Solaris or SunOS 4.x, are somewhere mid-way: 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 *synchronisation* 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

[1] at present for the standard kernels at least.

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, many 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*.

Threads can be implemented as a *library of user functions*. The standard set of functions is defined in POSIX.1c, but other implementations also exist. The most commonly used package[1] implements the **POSIX.1c pthreads**. There are some 50 service requests defined. They are described in 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 data 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`

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 the man pages for the complete list. The *scheduling policy* can be one of: `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`, as for the POSIX.1c 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 func-

[1] Xavier Leroy's implementation, called **LinuxThreads**, which is part of many recent Linux distributions (Xavier.Leroy@inria.fr).

tions. 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`.

An important function is:

`int pthread_join()` 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 synchronising 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*. A condition variable is used to *signal* a thread that a particular condition has been satisfied in another thread. The first thread — the one receiving the signal — will then be allowed to proceed if it had blocked on the condition variable (CV). It works as shown in table 1.

Translated into code, this becomes as shown in the next table, 2

Note that *pthread_cond_wait()* will automatically 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

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	

Table 1: Using condition variables, algorithm

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);`	

Table 2: Using condition variables, code example

many — *clients* is easy.

We close this section with a complete code example[1]. The example concerns an *Automatic Teller Machine*, e.g. one of those machines that distribute banknotes. The main program, which is the **server**, receives requests over a communication line from ATMs scattered all over town. For each request received, the server spawns a *worker* or *client thread* which undertakes the actions necessary to satisfy the request. This example mainly illustrates the creation of several threads.

```
typedef struct workorder {
        int conn;
        char req_buf[COMM_BUF_SIZE];
        } workorder_t;

main(int argc, char **argv)
{
  workorder_t *workorderp;
  pthread_t   *worker_threadp;
  int  conn, trans_id;

  atm_server_init(argc, argv);

  for(;;) {
    /*** Wait for a request ***/
    workorderp = (workorder_t *)malloc(sizeof(workorder_t));
    server_comm_get_request(&workorderp->conn,
            &workorderp->req_buf);

    sscanf(workorderp->req_buf, "%d", &trans_id);
    if (trans_id == SHUTDOWN) {
          . . .
      break;
    }

    /*** Spawn a thread to process this request ***/
    worker_threadp = (pthread_t *)malloc(sizeof(pthread_t));
    pthread_create(worker_threadp, NULL, process_request,
          (void *)workorderp);

    pthread_detach(*worker_threadp);
    free(worker_threadp);
  }
  server_comm_shutdown();
}
```

[1]This and the next example are from *B. Nichols et.al., Pthreads Programming*, See Appendix C, item ii

The *worker thread* (the **client**) looks as follows:

```
void process_request(workorder_t *workorderp)
{
  char resp_buf[COMM_BUF_SIZE];
  int trans_id;
  sscanf(workorderp->req_buf, "%d", &trans_id);

  switch(trans_id) {
    case WITHDRAW_TRANS:
         withdraw(workorderp->req_buf, resp_buf);
         break;

    case BALANCE_TRANS:
         balance(workorderp->req_buf, resp_buf);
         break;
         .
         .
    default:
         handle_bad_trans_id(workorderp->req_buf, resp_buf);
  }

  server_comm_send_response(workorderp->conn, resp_buf);
  free(workorderp);
}
```

There are two points to note in this example. The first concerns the passing of arguments to a child thread. The standard allows a single argument only. Encapsulating several arguments in a single structure and passing a pointer to this structure to the child is a way to program around the restriction. The second point is a subtle one and concerns the use of *malloc*. Using static storage for the workorder does not work: for every newly created thread the workorder would be overwritten and most threads would work with a corrupted workorder.

The following example, taken from the same source, illustrates the use of a *mutex* and a *condition variable*. Two of the threads created in this example simply increment a counter and check if it has reached a limit value. In that case they *signal the condition variable*. The third thread waits on the condition variable and prints its value. The main thread will exit when all three threads it created have "*joined*".

```
#include <pthread.h>
#define TCOUNT 10
#define WATCH_COUNT 12

int count = 0;
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_cond_t count_threshold_cv = PTHREAD_COND_INITIALIZER;
int thread_ids[3] = {0, 1, 2};

main()
{
  pthread_t threads[3];

  pthread_create((&threads[0], NULL, inc_count, &thread_ids[0]);
  pthread_create((&threads[1], NULL, inc_count, &thread_ids[1]);
  pthread_create((&threads[2], NULL, watch_count, &thread_ids[2]);
  for(i = 0; i < 3; i++)
    pthread_join((&threads[i], NULL);
}


void watch_count(int *idp)
{
  pthread_mutex_lock(&count_mutex);
  while(count <= WATCH_COUNT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch: Thread %d, Count is %d\n", *idp, count);
  }
  pthread_mutex_unlock(&count_mutex);
}

void inc_count(int *idp)
{
  int i;
  for(i = 0; i < TCOUNT;, i++)
    pthread_mutex_lock(&count_mutex);
    count++;
    printf("inc: Thread %d, count is %d\n", *idp, count);
    if(count == WATCH_COUNT)
      pthread_cond_signal(&count_threshold_cv);
    pthread_mutex_unlock(&count_mutex);
  }
}
```

In this example the reader should note that two threads share identical code and that only one copy of this code is present in memory. The program counter and the stack are of course private property of each individual thread.

11 Real-time Enhancements to Linux

During the last eight years or so the community of Linux developers have put considerable efforts into making enhancements to Linux in order to make it better suitable for *hard real-time* applications. Spurred by an increasing demand for *embedded systems*, industry also made considerable contributions to this effort. The result is that versions of Linux are now available that have response times better than a few tens of microseconds when running on —rather oldish— processors such as a 133 MHz Pentium.

One of the he reasons why standard Linux cannot pretend to be a *Real-time Operating System* is that when a low priority task is running, a prohibitively long delay may occur between an external stimulus (*hardware interrupt*) to a sleeping high-priority real-time task and the moment that this real-time task wakes up. On the same Pentium a delay (**latency**) of 100 milliseconds is no exception. The average response is much shorter and in fact Linux can be used “as is” in certain *soft real-time* applications, such as playing music or video.

The *latency* can be broken down into two main components. First there is the delay between the raising of an electrical interrupt signal by an external device and the moment the processor “feels” the interrupt. This is due to the fact that the Linux kernel may run for rather long periods with external interrupts disabled. Secondly, the Linux kernel itself may introduce long delays between the moment the high priority real-time task becomes *runnable* and the instant the running lower priority task is *preempted*. For instance the scheduler will not be run when the kernel is executing a *system call*, thus allowing a low priority task to delay execution of higher priority tasks. The kernel is also generous in protecting access to its own data structures for periods longer than strictly necessary.

The different nature of these two causes of long latency times resulted in two approaches to solving the problem. The first solution consists of adding a second kernel to the system, which is small and fast. This **real-time kernel** is in fact controlling the entire system. Its main duty is to run the real-time task at highest priority, but when this task is idle the real-time kernel runs the entire Linux system (including its kernel) at low-priority. The *real-time kernel* intercepts **all** hardware interrupts and analyses them. Interrupts which are related to the activities of the Linux kernel (such as normal I/O) are passed on to Linux , the ones related to the real-time task are handled directly by the additional kernel. Vice-versa, when Linux decides to disable interrupts the real-time kernel intercepts this event and will stop passing on external interrupts to Linux . These interrupts are not lost; they are kept in stock to be handed over to the Linux kernel when it enables again its interrupts. This approach was appropriately called by Tim Bird [3] **Interrupt Abstraction**. The first implementation was **RTLinux** from the New Mexico Institute of Technology, followed by **RTAI** (*Real-Time Applications Interface*) from the “Dipartimento di Ingegneria Aereospaziale, Politecnico di Milano”. **KURT** (*Kansas University Real-Time*) uses a different technique, less effective in reducing latency. These three systems will be described in some more detail below.

The second approach is to improve **preemption** in the Linux kernel. This problem was tackled in two different ways. The first consists in introducing into the Linux kernel so-called **preemption points**, where the running kernel thread may give up control or make itself available to being preempted. These improvements are generally called *low-latency patches* to the kernel. The second is to make use of macros for *Symmetric Multi-Processor* (**SMP**) operation which exist already in the Linux kernel. The trick is to use these macros also in the case of a *Uni-Processor*. The first solution was implemented mainly by Ingo Molnar, one of the Linux kernel developers; the second was initiated by MontaVista. Both solution paths will be described in some more detail below.

The two approaches to improving latency briefly described above are valid for Linux running on a desktop machine as well as for *embedded systems*. For the latter there is the additional requirement of reducing the size of the system so as to fit into the limited storage available on most embedded systems. Embedded systems are adequately covered in other lecture series in this Workshop, so we will not dwell on them any longer.

11.1 Implementations based on a second kernel

11.1.1 RTLinux

RTLinux was developed in the mid nineties at the *New Mexico Institute of Technology* by Barabanov and Yodaiken [4] As briefly outlined above it consists of an additional kernel (or *Real-time Executive* as they call it), which intercepts all hardware interrupts and reacts immediately to those concerning the real-time application that is running. This small kernel has total control over the machine and normal Linux can only be run as a *low priority* process, during the periods in which the real-time application has no work to do. All Linux facilities are thus available to the user, via the normal keyboard. The real-time application has the highest priority for this executive, and will therefore run immediately when it is ready to do so, the entire Linux system being preempted to free the processor.

The real-time application should only take care of the *time-critical tasks*; the others, such as recording data to disk, are left to be done under normal Linux . This implies that there must be a mechanism for communication between the time-critical part of the application program, running in *kernel memory space* and the other, non-critical parts, which run as normal Linux processes in *user space*. Normal pipes cannot be used (because of the two different memory spaces), so another mechanism was developed: RT-FIFOs. They behave as pipes really, but the interface to the task running in kernel space is different.

The real-time application program communicates to the *real-time executive* via a special API (*Application Programmer's Interface*), which is different from the normal Linux API.

RTLinux is built by applying patches to a number of kernel source files and recompiling the kernel. It can then be arranged that at boot-up time a choice can be made between booting standard Linux or RTLinux. To change from one to the other a reboot is necessary.

The development of RTLinux started as a research project, but a few years ago the techniques used in it were the subject of a patent and a version called *RTLinuxPro* is commercialised. Another version, RTLinuxFree is still available for developers and the GPL (GNU Public License) seems to apply to its use. There also seems to be a distribution list for this GPL version, but I was unable to locate it (http:// www.rtlinux.org is immediately redirected to the commercial site, where no technical information is directly available).

11.1.2 RTAI

RTAI was developed at the Politecnico di Milano, where work is going on to extend it and to make it more universal. It is distributed for free under the GNU Public License. The principle is the same as for RTLinux, but the implementation is different: the changes to the Linux kernel sources are very minimal: 20 lines of the kernel source code are modified and 50 or so other lines are added. This greatly improves maintainability and adaptation to newer kernel versions.

These relatively small changes to the kernel establish a *Hardware Abstraction Layer*. This **HAL** performs the same function of selecting and redirecting hardware interrupts as in RTLinux. It consists of a table of pointers to interrupt vectors and the two functions necessary to enable/disable interrupts. These pointers can be re-established to their original values, thus allowing a return from real-time mode of operation to normal mode.

All the rest of RTAI is implemented in the form of *kernel modules* which can be dynamically loaded at runtime. The structure is very modular and the user can arrange to load only those modules he needs, leaving the others aside. In February 2003 eight different modules were available.

The basic frame of RTAI is implemented in the *rtai* module. If the only thing you need is interrupt handlers for some devices, then this is the unique module that must be loaded. When you wish to run a real-time task that is properly scheduled, you must also load the module *rtai_sched*. The module *rtai_fifo* adds FIFOs, if the real-time task needs to communicate with normal Linux processes. Another way of communicating with a normal Linux process is via *shared memory*, which necessitates loading of *rtai_shm*. This last module is not restricted to sharing memory between a real-time task and a Linux process, but it can also be used for communication between two Linux processes, replacing in fact some of the System V IPC mechanisms.

Two further modules, *rtai_pthread* and *rtai_pqueue* implement POSIX 1003.1c compatible *real-time threads* and *message queues*.

A very interesting module is *lxrt* (LinuXRealTime). It in fact allows a hard real-time task to run in *user space*, as opposed to kernel space. I cite from an article available on the RTAI Website [5] : “Since the initial RTAI programs ran in kernel space, LXRT is seen as an extension that brings this kernel API to userland programs. The advantages are significant. It allows to define realtime and non-realtime threads in the same program so that one (thread) could read/write a file (=non-realtime) while the other runs with a deterministic period (=realtime). It also allows IPC with standard Linux processes or allows your realtime program

to have a GUI integrated. LXRT extensions allow to communicate in realtime with hardware from these programs."

LXRT requires another scheduler to be loaded as a kernel module: *rtai_sched_newlxrt*, which schedules both kernel and *lxrt* tasks. Running *lxrt* implies scheduling the realtime task in user space, introducing some overhead. This overhead is however minimal and does not overshadow the benefits of running the real-time task in user space. On a Pentium PII at 300 MHz an extra 3 μs overhead has been measured, on top of the 3 μs latency of the scheduler itself.

It should be noted that for the recent versions of RTAI after loading the *rtai* module in the usual way (with '*insmod rtai*') nothing happens. *rtai* is in fact a dormant module; it must be '*mounted*' by calling *rt_mount_rtai()* to activate it. Similarly, it can be de-activated by calling *rt_umount_rtai()*.

Obviously the user has to write a program that cares for his real-time application. In older versions of RTAI this user program also had to be written in the form of a kernel module and therefore had to run in kernel space. With *lxrt* the application can be written as a user program.

Some rules are to be obeyed however if one wants to avoid bad surprises. For instance all memory needed for the real-time process should be allocated before the task starts running and no call to 'malloc' should be made in the course of its execution. Also system calls which are known to introduce large delays must be avoided (such as 'fork' and 'exec').

A few more points concerning RTAI are worth mentioning. Semaphores, mutexes and conditional variables are available to the real-time task and so are real-time FIFOs, shared memory and message queues. These allow to share and transmit data between RTAI real-time tasks and Linux processes. RTAI can run on a SMP (Symmetric Multi-Processor), where each real-time task can be assigned to any subset of the processors or to a single processor. In addition the real-time interrupt service can be assigned to a single processor. In order to be able to obtain information about the state of RTAI, the *proc* filesystem is available. The real-time task can make use of the Floating Point Unit (FPU), so that it can perform floating point operations (see the example below).

For more details, including descriptions of the internal workings of RTAI, the reader is refered to the website: `http://www.aero.polimi.it/projects/rtai` or also: `www.rtai.org/index.html`. Some performance figures of RTAI are given in an article [6] in *Linux Journal* of August 2000. They refer to a Pentium II at 233 Mhz. The maximum possible real-time task iteration rate, with Linux running simultaneously under heavy load, was measured to be 125 KHz. The measured *jitter* on this rate was 0–13 μs. The *context switching time* was approximately 4 μs.

The appendices A and B show simple real-time tasks to run with LXRT and RTAI.

11.1.3 KURT

KURT, for *Kansas University Real Time* Linux was developed by Balaji Shrinavasan[7] of the Information and Telecommunications Technology Centre (ITTC) of the

University of Kansas. The author calls it a *firm* real-time system, somewhere between *hard* and *soft* real-time. KURT was developed with *multi-media* applications in mind, which are characterised by the fact that, besides having rather stringent time constraints, they have to rely heavily on services provided by the normal Linux , such as handling network trafic and reading Compact Disks. RTLinux does (did?) not provide access to normal operating system services from within a real-time task.

The implementation of KURT is based on a different principle from *RTLinux*.

KURT allows the *explicit scheduling by the applications programmer* of real-time **events**, instead of just *processes*. The event scheduling is done by the system, which will call the appropriate *Event Handler* module.

Once KURT has been installed, Linux has acquired a second mode of operation. The two modes are: *normal-mode* and *real-time mode*. In the first the system behaves as normal Linux , but when the kernel is running in *real-time mode*, it executes only *real-time processes*. All system resources are then dedicated to the real-time tasks. There is a *system call* that toggles between the two modes.

During the *setup phase* the schedule of events to be executed in *real-time mode* is established and the processes that must run in this mode are marked. The kernel is then switched to the *real-time mode*. When all tasks have finished, the kernel is switched back to *normal-mode*.

In order to obtain this behaviour, KURT consists of a **Real-Time Framework** which takes care of scheduling any *real-time event*. When such an event is to be handled, the *real-time framework* calls the **event handler** of the associated **RTMod** (Real-Time Module). The *RTMod*s can be very simple; calling them according to a defined schedule is the responsibility of the *real-time framework*. This framework also provides the system calls that switch the kernel between the two modes.

The *RTMod*s are *kernel modules*, which are loaded at runtime. Every *RTMod* registers itself with the *real-time framework*, to which it provides pointers to its functions: the *event handler* proper and —as required for a kernel module— its *initialisation* and *clean-up* routines.

The instant in time when a *RTMod* must be invoked is defined in the **Real-Time Schedule**, which is just a **file**. This file must be built beforehand. It can be copied entirely into memory, or it can remain on disk. In the latter case, the timing of events may become distorted by disk access times. Schedules read into memory can be executed *periodically*.

Events can be scheduled with a high time resolution, when another package has been installed: **UTIME**, for μ*second time*. This package was developed at the same Institute as KURT. It makes use of the *time-stamp counter* which is present in most PCs. If it is not installed then the time resolution is only the usual 10 ms.

To install **UTIME** and/or **KURT**, the Linux kernel must be recompiled.

A more detailed description and *sample programs* are available from the KURT Web sites listed in Appendix C.

11.2 Implementations based on kernel patches

Whereas two of the implementations described sofar reduced the *interrupt latency time*, the other potential source of long delays —scheduling of processes in the Linux kernel— has also undergone close scrutiny, which resulted in spectacular improvements. These improvements were gradually introduced into the "development kernels" from release 2.5.4 onward. The release of the "stable" kernel version 2.6 should make these a standard feature of Linux . The improvements to the kernel are nicely described by Love[8].

11.2.1 Reducing scheduler latency

In the original implementations of the kernel, the latency of the *process scheduler* grew proportionnally to the number of processes running on the system. The scheduler effected a loop over all *runnable* processes on the system to find the process that best merits to be run. The new scheduler needs a *constant* and *predictable* time to find that lucky process. As described by Love[8], the new algorithm is:

```
get the highest priority level that has runnable processes
get the first process in the list at that priority level
run that process
```

The algorithm, instead of having to search, simply looks up the things specified in the first two lines. The kernel keeps a record of these two properties whenever a process becomes runnable. The scheduler latency now is strictly constant and independent of the total number of processes on the system. Apparently, on modern machines, scheduler latency has been measured to be less than 500 nanoseconds!

11.2.2 Low-latency patches

With the classical Linux kernel, whenever a task is executing a *system call* it will continue to run until the system call terminates, even beyond its time-slice and irrespective of its priority. This rather primitive behaviour was an easy solution to avoid concurrent access to shared data structures in the kernel. The unfortunate consequence is that a low-priority task may block a real-time or other high priority task for periods which may exceed hundreds of milliseconds. The response can be improved by analysing the kernel code and finding those pieces that execute for long periods, but which could be broken down into pieces where no access is made to shared structures. During the execution of these pieces no harm would be done if the task making use of this particular kernel service would be preempted in favour of a higher priority task.

This approach was followed by Ingo Molnar and a number of kernel patches which greatly reduced latency were the result. Some test results will be given below. The problem with this approach is that it may involve patches spread

over large parts of the kernel, impairing maintainability. And, how can one be sure to spot all the potentially dangerous code segments?

Great progress has been made nevertheless, specially in the parts of the kernel dealing with Virtual Memory and Virtual File Systems.

11.2.3 Introducing preemption points

The problem of preventing simultaneous access to shared data also exists on Symmetric Multi-Processors (SMP)[1]. The Linux kernel was adapted for SMP machines already some time ago, by introducing *spin locks* [2]. At MontaVista, a firm in Florida, the idea was born to modify these spin locks so as to use them to signal to the scheduler that the task executing in the kernel could be preempted if desired. If the code holds a lock, it cannot be preempted; if it has no lock activated, it can be safely replaced by another task.

This mechanism to profit of the spin locks in the kernel works also in the case of a mono-processor machine. In this situation, a task must of necessity release a lock before another task can run. Thus no task can ever block on a spin lock in a single processor machine.

11.2.4 Some measured results

Back in March 2002, Clark Williams[9] of RedHat Inc. made some measurements on the effect of the low-latency patches versus the preemptive kernel. He set up a "stress-kernel" package which heavily loads the kernel and puts it under considerable stress. It is not necessary to describe here the details of his procedure, nor the exact configuration used for his tests. The numbers below therefore do not have an absolute significance, but are representative of the improvements that can be obtained with the kernel patches. Each test run lasted some 40 minutes during which 2048 interrupts per second were generated by a real-time clock. At the end of a run a histogram file was output. The histogram recorded the number of times the measured latency fell in a given interval. The latency values were measured in milliseconds and tenths of a millisecond.

The first run was with an off-the-shelf kernel, release 2.4.17. The maximum latency found was 232.6 ms, the average was 0.088 ms, with a standard deviation of 2.119 ms. 92.84% of the measured values were below 0.1 ms, whereas 99.988% of the samples had a latency below 100 ms.

The same test was repeated with a 2.4.17 kernel which had the preemption patches applied to it. The maximum latency now was 45.2 ms, the average 0.053 ms with a standard deviation of 0.046 ms. 97.95% of the samples showed a latency below 100 μs, and 99.999% below 10 ms.

---

[1] Here access can be truly simultaneous and not simply concurrent.

[2] A spin lock is simply a flag, set by one of the processors when it enters a critical section of code. Code running in another processor can repeatedly test the value of the flag (hence its name: *spin lock*) and access the critical section when allowed again by the first processor.

With the low-latency patches applied, the same kernel showed the following results: maximum latency 1.3 ms, average 0.054 $\pm$ 0.025 ms. 99.62% of the samples had a latency of less than 100 μs, and 99.999% were below 1 ms.

The tests show that the low-latency patches were more effective than the preemption patch. Having noted that the two types of patches are practically independent of each other, the author applied them both together to the same 2.4.17 kernel. This did not cause an impressive improvement however over the low-latency patches alone, but it nevertheless decreased the maximum latency from 1.3 to 1.2 ms.

In a run of approximately 14.5 hours duration with the low-latency patch applied, one occurrence of a latency of 215.2 ms did happen!

11.2.5 Other improvements beneficial to real-time applications

A stable kernel version 2.6 is expected to be released in autumn 2003. The improvements mentioned above will almost certainly be incorporated, together with several others. All of them aim at boosting responsiveness. Thus disk access will be improved and speeded up considerably. Also the Virtual Memory system will be much improved.

Another great stride forward will be a much better kernel support for *pthreads*, which results in drastically speeding up the execution of pthread related system calls. In one test[8] the time to create and then kill a thread was measured. In this test, ten threads were running initially, and each of these create a new child thread and then kill it. The time to do this was reduced from 140 μs with the old *pthreads library* to a bare 20 μs with the new *NPTL (Native POSIX Threading Library)*. When each initial thread created and then destroyed ten child threads, the measured times were 170 μs and 20 μs.

Although these improvements may not all have a significant bearing on running real-time applications, every stone that can be added to the construction will certainly be welcome. In particular there should be no a-priori obstacle to use RTAI or RTLinux with the new 2.6 kernel. Avoiding excessive delays in those parts of the real-time application that run under the supervision of Linux instead of the additional real-time kernel, can only be beneficial.

12 Conclusion

We have tried in this course to give an overview of the requirements of a real-time application and we have investigated to what extent “standard” Linux could do the job. This led us to cover in some detail —at least in these written notes— mechanisms internal to the Linux kernel and which could directly influence the response time of a real-time application or have a bearing on its structure. We have shown that the standard releases of Linux lacked the responsiveness needed for a hard real-time application.

Over the last seven or eight years great improvements have been made by a number of developers, which however did not make their way into the Linux

kernel releases or the so-called *Linux distributions*, such as RedHat, Debian, SuSE, etc. The situation now seems to change and the forthcoming release of Linux version 2.6 will in all likelihood incorporate a number of improvements, several of them aimed at reducing delays due to latency.

On the other hand, efforts to reduce the interrupt latency have led to the development of, among others, RTLinux and RTAI. These supplements to Linux , when installed on a modern, fast PC will reach interrupt latency times of below a few microseconds, thus allowing to run true hard real-time applications. It is however doubtful if these supplements will make their way into the standard kernel releases, so they might always require some extra development effort: building a modified kernel.

The new Linux together with RTAI will in all likelihood represent a totally satisfactory —and **free**— solution for the vast majority of problems that may arise in industrial control or laboratory automation. Only in exceptional cases, such as those that arise for instance in Particle Physics, would one have to resort to specialised hardware or other frontend processors. It also becomes more and more difficult to imagine in what respect a commercial real-time operating system would do really better than the Linux + RTAI combination.

The important thing to remember is that you should analyse your problem very carefully and thoroughly, before setting out to implement a solution. You should single out the parts of your application that are really time critical and put those under control of RTAI, whereas the rest of the application may run under Linux . You should also take some precautions: allocate in the initialisation phase of your application all memory it needs (including stack space) and lock your code into memory. Also create from the start all processes and threads the application will need. As a last recourse, eliminate all other processes and deamons from your machine and let your application run alone, to avoid encountering after tens of hours of perfect operation an unexpected latency time of 100 ms.

These lectures did not cater for those people who may be just interested in hooking up existing laboratory instruments, using the bus they are already equipped with: serial or USB, or GPIB or maybe Camac. In many cases a NIC (Network Interface Computer) would be the ideal solution as was shown in this workshop. Those who wish to hook up this sort of instruments directly to a PC, should look for suitable packages running under Linux which are available on the Web. For instance packages for controlling instruments via a GPIB or CAMAC bus exist. The first parts of these were released already in 1995. They have graphical interfaces, use X11 and are extensible[1]. And **they are free!**. These packages are probably an ideal solution for laboratories using standard equipment. Device drivers for *VME crates and modules* are part of some off-the-shelf Linux distributions (*SuSE 8.1 for instance*).

Hopefully, this course has shown you the points to consider when you want to build your real-time application, and where to search for existing and acceptable solutions.

[1]You can ftp these packages from koala.chemie.fu-berlin.de.

To end, I wish you a happy time programming your real-time applications! Now that many more and much better tools are available than a few years ago, there is a good chance that this wish comes true.

Enjoy!

A RTAI example 1

This program is developed for the RTAI-1.4 version. You can find a Makefile, one kernel module (rt_process.c) and a Linux process (scope.c). By means of the scipt Run you can run the program. The program simply generates a sine signal and displays the instantaneous values on the screen.

```
------------- MAKEFILE --------------------
all: scope rt_process.o
LINUX_HOME = /usr/src/linux
RTAI_HOME = /home/rtai-1.4
INCLUDE = -I/include -I/include
MODFLAGS = -D__KERNEL__ -DMODULE -O2 -Wall
scope: scope.c
    gcc -o $@ $<
rt_process.o: rt_process.c
    gcc -c -o $@ $<
clean:
    rm -f rt_process.o scope

-------------- RT_PROCESS.C ------------------
#include <linux/module.h>
#include <asm/io.h>
#include <math.h>
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>
#define TICK_PERIOD 1000000
#define TASK_PRIORITY 1
#define STACK_SIZE 10000
#define FIFO 0

static RT_TASK rt_task;

static void fun(int t)
{
    int counter = 0;
    float sin_value;
    while (1) {
        sin_value = sin(2*M_PI*1*rt_get_cpu_time_ns()/1E9);
        rtf_put(FIFO, &counter, sizeof(counter));
        rtf_put(FIFO, &sin_value, sizeof(sin_value));
        counter++;
        rt_task_wait_period();
    }
}
```

```
int init_module(void)
{
    RTIME tick_period;
    rt_set_periodic_mode();
    rt_task_init(&rt_task, fun, 1, STACK_SIZE, TASK_PRIORITY, 1, 0);
    rtf_create(FIFO, 8000);
    tick_period = start_rt_timer(nano2count(TICK_PERIOD));
    rt_task_make_periodic(&rt_task, rt_get_time() +
                                    tick_period, tick_period);
    return 0;
}
void cleanup_module(void)
{
    stop_rt_timer();
    rtf_destroy(FIFO);
    rt_task_delete(&rt_task);
    return;
}
-------------------- SCOPE.C ----------------------
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>

static int end;
static void endme(int dummy) { end=1; }

int main (void)
{
    int fifo, counter;
    float sin_value;
    if ((fifo = open("/dev/rtf0", O_RDONLY)) < 0) {
        fprintf(stderr, "Error opening /dev/rtf0\n");
        exit(1);
    }
    signal(SIGINT, endme);
    while (!end) {
        read(fifo, &counter, sizeof(counter));
        read(fifo, &sin_value, sizeof(sin_value));
        printf(" Counter : %d Sine : %f \n", counter, sin_value);
    }
    return 0;
}
```

```
-------------------- RUN ----------------------
sync
insmod /home/rtai-1.4/modules/rtai.o
insmod /home/rtai-1.4/modules/rtai_sched.o
insmod /home/rtai-1.4/modules/rtai_shm.o
insmod /home/rtai-1.4/modules/rtai_fifos.o
insmod rt_process.o
./scope
rmmod rt_process
rmmod rtai_shm
rmmod rtai_fifos
rmmod rtai_sched
rmmod rtai
```

B RTAI example 2

This example is like the first one, but it uses the shared memory, instead of fifos, to communicate between Linux and the RTAI-layer. The common data space is declared in the header file parameters.h.

```
------------------ MAKEFILE -------------------------
all: scope rt_process.o
LINUX_HOME = /usr/src/linux
RTAI_HOME = /home/rtai-1.4
INCLUDE = -I/include -I/include
MODFLAGS = -D__KERNEL__ -DMODULE -O2 -Wall
scope: scope.c
    gcc -o $@ $<
rt_process.o: rt_process.c parameters.h
    gcc -c -o $@ $<
clean:
    rm -f rt_process.o scope


------------------- RT_PROCESS.C ---------------------

#include <linux/module.h>
#include <asm/io.h>
#include <math.h>
#include <rtai.h>
#include <rtai_shm.h>
#include <rtai_sched.h>
#include "parameters.h"
```

```
static RT_TASK rt_task;
static struct data_str *data;

static void fun(int t)
{
    unsigned int count = 0;
    float seno,coseno;
    while (1) {
        data->indx_counter = count;
        seno = sin(2*M_PI*1*rt_get_cpu_time_ns()/1E9);
        coseno = cos(2*M_PI*1*rt_get_cpu_time_ns()/1E9);
        data->sin_value = seno;
        data->cos_value = coseno;
        count++;
        rt_task_wait_period();
    }
}
int init_module(void)
{
    RTIME tick_period;
    rt_set_periodic_mode();
    rt_task_init(&rt_task, fun, 1, STACK_SIZE, TASK_PRIORITY, 1, 0);
    data = rtai_kmalloc(nam2num(SHMNAM), sizeof(struct data_str));
    tick_period = start_rt_timer(nano2count(TICK_PERIOD));
    rt_task_make_periodic(&rt_task, rt_get_time()
                                        + tick_period, tick_period);
    return 0;
}
void cleanup_module(void)
{
    stop_rt_timer();
    rt_task_delete(&rt_task);
    rtai_kfree(nam2num(SHMNAM));
    return;
}

---------------------- SCOPE.C ----------------------------
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
#include <rtai_shm.h>
```

```
#include "parameters.h"

static int end;
static void endme(int dummy) { end=1; }

int main (void)
{
    struct data_str *data;
    signal(SIGINT, endme);
    data = rtai_malloc (nam2num(SHMNAM),1);
    while (!end) {
        printf(" Counter : %d Sine : %f Cosine : %f \n",
                                        data->indx_counter,
                            data->sin_value, data->cos_value);
    }
    rtai_free (nam2num(SHMNAM), &data);
    return 0;
}


---------------- PARAMETERS.H -----------------------
#define TICK_PERIOD 1000000
#define TASK_PRIORITY 1
#define STACK_SIZE 10000
#define SHMNAM "MIRSHM"
struct data_str
{
    int indx_counter;
    float sin_value;
    float cos_value;
};


----------------------- RUN ------------------------
sync
insmod /home/rtai-1.4/modules/rtai.o
insmod /home/rtai-1.4/modules/rtai_sched.o
insmod /home/rtai-1.4/modules/rtai_shm.o
insmod /home/rtai-1.4/modules/rtai_fifos.o
insmod rt_process.o
./scope
rmmod rt_process
rmmod rtai_shm
rmmod rtai_fifos
rmmod rtai_sched
rmmod rtai
```

Fri Feb 14 15:17:03 2003

C For Further Reading

This is an annotated bibliography of books I found useful and which can give you more insight or may be helpful when you run into a problem with Linux . The bibliography is limited to books in English, but several of them have been translated and many others have been published directly in another language. The list starts with a few books published in the mid-nineties. They may not contain up-to-date information on the newest kernel releases or application packages (although more recent editions than the ones indicated may have been published in the meantime), but they still constitute a good base for a collection of books on Linux .

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, now at its 4th edition.

2 Olaf Kirch, *Linux Network Administrator's Guide*, Sebastopol CA95472, 1995, O'Reilly & Associates, Inc; ISBN 1-56592-087-2
An 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. A 2nd edition has been published.

3 Alessandro Rubini, *Linux Device Drivers*, 1998, O'Reilly & Associates, ISBN 1-56592-292-1. This book is a **real must** for anyone wanting to write or modify a device driver for Linux. Before publishing this book, the author had written many articles in the *kernel corner* of *Linux Journal*. The book leads the reader step by step through every corner of a Linux device driver. No secrets are left unveiled. The book is at its 2nd edition.

4 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.

5 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.

6 John O'Gorman, *Operating Systems with Linux*, 2001, Houndmills, Basingstoke, UK, Palgrave; ISBN 0-333-94745-2.
A recent book, introducing operating system principles. All examples are taken from Linux . Each chapter ends with a summary and discussion questions, including sometimes hands-on experiments.

7 M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner, *Linux Kernel Internals*, 1996, Addison Wesley, ISBN 0-201-87741-4.
There is at least a second edition: ISBN 0-201-33143-8, 1998. 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.

8 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 —like me— 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.

The following books concern **real-time**, **POSIX.1003.1c** and **Embedded Linux**:

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 Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell, *Pthreads Programming*, 1996, O'Reilly & Associates, Inc, ISBN 1-56592-115-1.
Probably the best book on Pthreads published so far, concentrating on the POSIX 1003.1c standard and written with a good didactical structure.

iii 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**.

iv S. Kleiman, Devang Shah, B. Smaalders, *Programming with Threads*, 1996, Sunsoft Press (Prentice Hall; ISBN 0-13-172389-8. This book contains a more in-depth treatment of threads programming than the previous title. It is also more pthreads-oriented.

v Craig Hollabaugh, *Embedded Linux: Hardware, Software and Interfacing*, 200?, Addison Wesley Professional, ISBN 0-672-32226-9 According to a number of detailed and favourable book reviews, the author of this book guides the reader step by step through the design and implementation of a fictive, but realistic project. It covers the use of three different microprocessor boards and a large part of the book is dedicated to interfacing.

vi Karim Yaghmour, *Building Embedded Linux Systems*, 2003, O'Reilly & Associates, Inc., ISBN 0-596-00222-X. A very recent book. The review I saw complains about the first four chapters, which seem to provide some superfluous information. The reviewer (Jerry Epplin) also critisises the "do it yourself" approach, not suitable for developers of commercial equipment, but which may be very profitable in an academic environment. A large part of the 416 pages are dedicated to interfacing to various buses and I/O ports.

vii John Lombardo, *Embedded Linux*, 2002, Indianapolis, New Riders Publishing. ISBN 0-7357-0998-X This book is more modest in size than the two listed before. It contains nevertheless a good amount of practical advice.

Having mentioned *Linux Journal*, I should add that you can subscribe via one of the following addresses: e-mail: subs@ssc.com, or on the web: www.linuxjournal.com, Fax: +1-206 297 7515 and by normal mail: SSC, Specialized System Consultants, Inc., PO Box 55549, Seattle, WA 98155-0549, USA. Also note that the last few years a number of periodicals on Linux have seen the light in languages other than english.

The following non-commercial Web sites contain a wealth of information on Linux for real-time and embedded applications:
`http://www.realtimelinux.org`
`http://www.aero.polimi.it/projects/rtai` or: `http://www.rtai.org`
`http://www.linuxdevices.com`
`http://www.ittc.ku.edu/kurt` (for KURT) and
`http://www.ittc.ku.edu/utime` (for UTIME)

To conclude, some mailing lists which may be useful. To subscribe to any of the lists, send an email to: **missdomo@realtimelinux.org** with **subscribe 'listname'** in the **body** of the email.

- **realtime** — realtime@realtimelinux.org — general discussion.
- **api** — api@realtimelinux.org — API discussion.

- **documentation** — documentation@realtimelinux.org
- **drivers** — drivers@realtimelinux.org — discussion of drivers for use with Realtime Linux.
- **kernel** — kernel@realtimelinux.org — Linux kernel modifications for use with Realtime Linux.
- **networking** — networking@realtimelinux.org — Realtime networks.
- **ports** — ports@realtimelinux.org — Porting of RTL/RTAI to other platforms.

- **testing** — testing@realtimelinux.org — discussion on testing

References

[1] Andrew S. Tanenbaum, *Modern Operating Systems* (Prentice Hall International Editions, 1992, ISBN 0-13-595752-4).

[2] Bill O. Gallmeister, *POSIX.4, Programming for the Real World*, (O'Reilly & Associates, Sebastopol, CA95472, 1995.)

[3] Tom Bird, *Comparing two approaches to real-time Linux* , `www.linuxdevices.org/articles/AT7005360270.html`.

[4] M. Barabanov and V. Yodaiken. Introducing Real-time Linux, *Linux Journal, 34, pp 19-23, 1997.*

[5] `http://www.aero.polimi.it/index.html`. The title of the article is: *HOWTO port your C++/Linux application to RTAI/LXRT*.

[6] P. Mantegazza, E. Bianchi, L. Dozio, S. Papacharalambous, S. Hughes and D. Beal, RTAI: Real-Time Application Interface, *Linux Journal, April 2000*.

[7] Balaji Srinavasan, KURT: the KU Real-Time Linux, `www.linuxdevices.org/articles/AT9514980326.html`.

[8] Robert Love, Introducing the 2.6 Kernel, *Linux Journal, 109, pp 52–57, May 2003.*

[9] Clark Williams, *Linux Scheduler Latency*, `www.linuxdevices.org/articles/AT8906594941.html`.

4 5 11 11.1.1 11.1.2 11.1.2 11.1.3 11.2, 11.2.1, 11.2.5 11.2.4