INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

UNITED NATIONS INDUSTRIAL DEVELOPMENT ORGANIZATION
INTERNATIONAL CENTRE FOR SCIENCE AND HIGH TECHNOLOGY
c/o INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS 34100 TRIESTE (ITALY) VIA GRIGNANO, 9 (ADRIATICO PALACE) P.O. BOX 586 TELEPHONE 040-224572 TELEFAX 040-224575 TELEX 460449 APH I

SMR/643 - 11

**SECOND COLLEGE ON
MICROPROCESSOR-BASED REAL-TIME CONTROL -
PRINCIPLES AND APPLICATIONS IN PHYSICS
5 - 30 October 1992**

PROGRAM DESIGN PRINCIPLES

**D. MEGEVAND
Observatoire de Geneve
Ch. des Maillettes 51
1290 Sauverny
Switzerland**

Program design principles.

A good program should be

- simple, understandable
- reliable
- adaptable

If possible, without interference on the previous points, it should be

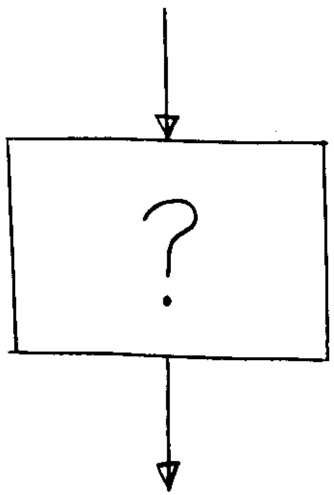
- efficient
- portable

A good design is the key to the respect of these points!

Structured Block

Piece of code with 1 input and 1 output (at most).

May be further decomposed.



Complexity of an algorithm

$f(n)$ = number of operations for a given algorithm

$f(n) \sim$ computation time for the implementation of the algorithm.

$f(n)$ is the complexity of the algorithm.

O(): Big O notation

$$|f(n)| \leq k \cdot |g(n)| \Rightarrow f(n) \text{ is } O(g(n))$$

$f(n)$ has order at most $g(n)$

$f(n)$ grows no more rapidly than $g(n)$

$g(n)$ should be as simple as possible.

Ex: $f(n)$ is $O(n^2)$ means that the algorithm complexity is quadratic.

The inequality defining $O(g(n))$ introduces an ambiguity : A quadratic algorithm such as $f(n) = \frac{n^2}{2}$ is limited as well by $g(n) = n^2$ as by $g(n) = n^7$

$\Theta(\,)$: Big Theta notation

$f(n)$ has the same order than $g(n)$ if $f(n)$ is $\Theta(g(n))$

which is verified if there exist two positive integers k and h such that

$$|f(n)| \leq k \cdot |g(n)|$$

and

$$|g(n)| \leq h \cdot |f(n)|$$

Search algorithms

- One can show that the binary search is the most efficient algorithm within the "key comparison" algorithms.
- Algorithms for searching a string in another bigger string may be more efficient.
  - Straight string search : bad
    worst case : $O(M \cdot N)$
  - Knuth-Morris-Pratt : better
    worst case : $O(M+N)$
  - Boyer-Moore : even better
    worst case : $O(N)$
- Other combinations of these algorithms may be imagined for certain special situations.

Sorting algorithms

Efficiency of different algorithms (N-elements array)

	Worst Case	Average Case
Insertion sort	$\frac{N(N-1)}{2} = O(n^2)$	$\frac{N(N-1)}{4} = O(n^2)$
Selection sort	$\frac{N(N-1)}{2} = O(n^2)$	$\frac{N(N-1)}{2} = O(n^2)$
Bubble sort	$\frac{N(N-1)}{2} = O(n^2)$	$\frac{N(N-1)}{2} = O(n^2)$
Quick sort	$\frac{N(N+3)}{2} = O(n^2)$	$1.4 N \log N = O(n \log n)$
Heap sort	$3N \log N = O(n \log n)$	$3N \log N = O(n \log n)$
Merge-sort	$N \log N = O(n \log n)$	$N \log N = O(n \log n)$

Records

- Records are linear heterogeneous structures
- In C language, they are called structures.
- Each element is defined with its type.

may itself be a record.

Example :

1. Declarations

```
TYPE   Name = RECORD
                  first : ARRAY [15] OF CHAR
                  m_i :   CHAR
                  last :  ARRAY [15] OF CHAR
              END

TYPE   Area = RECORD
                  city :  ARRAY [10] OF CHAR
                  zip :   LONG
                  state : ARRAY [10] OF CHAR
              END
```

Records (continued)

```
TYPE Adress = RECORD
                  no:     INTEGER
                  street: ARRAY[15] OF CHAR
                  area:   Area
              END

TYPE Person = RECORD
                  name:   Name
                  adress: Adress
              END
```

2. Instantiation

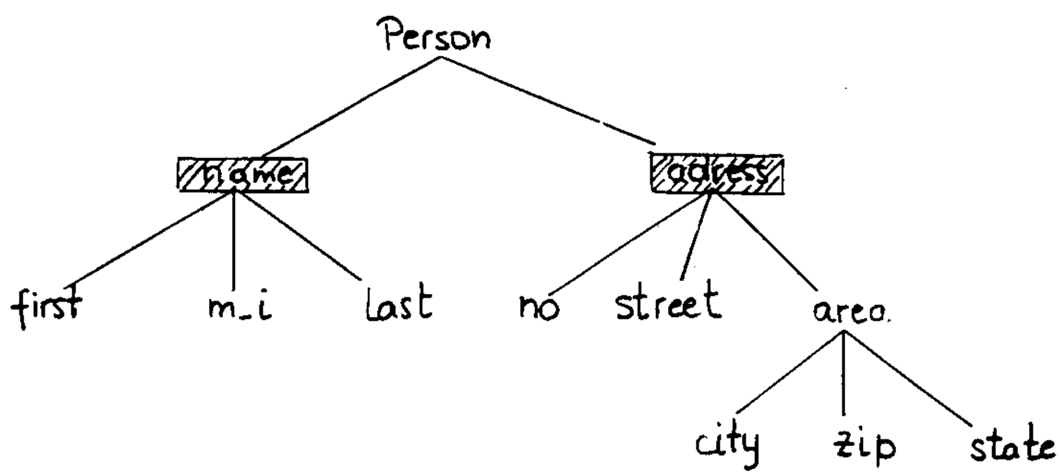
```
Person  Brown, Jack
```

3. Usage

```
Jack.name.last := Smith
Brown.adress.area.zip := 80137
Brown.name.m_i := Jack.name.first [1]
```

Records (continued)

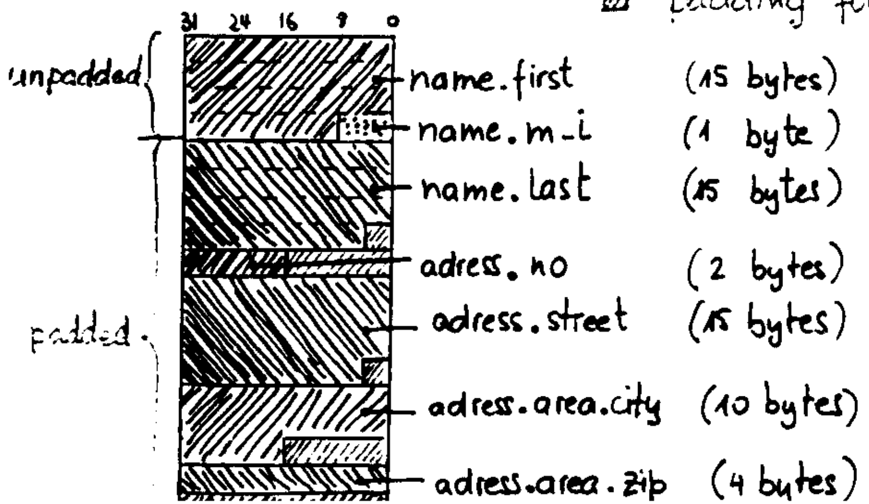
Tree - representation



Internal representation

(ex: 32 bit memory)

▨ padding for alignment



Variants of records

- To describe related, but slightly different record, one may define variants of a record:
  - One or more fields may be defined by a "case ... of" expression

Example:

```
            TYPE  Sex    = (male, female)
            TYPE  Person = RECORD
Common part {   name:   Name
                adress: Adress
                CASE s:Sex OF
Variant part {    male   : weight: REAL
                  female : size  : ARRAY[3] OF INTEGER
                END
            END
```

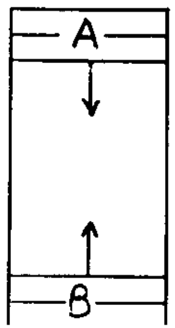
The fields of common parts are always present.
The fields of variant parts are discriminated by the tag field s.

Errors with stacks

Stacks related errors can be classified in two categories:

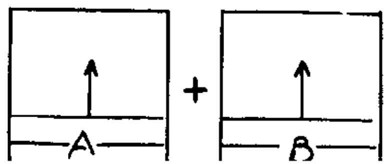
- Stack underflows depend on the data in the stack and the procedure accessing those data.
- Stack overflow are not mandatorily due to an error, but are consequences of the limitations of the stack.

One can try to minimize their effects by special implementations, for example

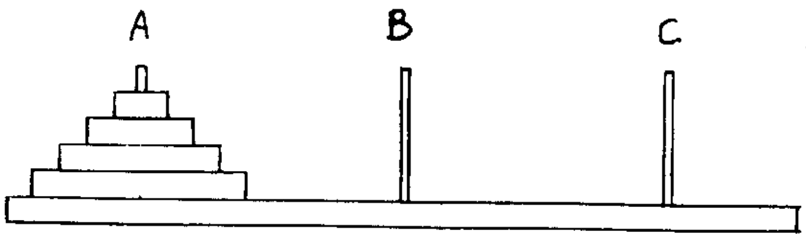


Stacks A and B grow in opposite directions. This technique is saving space, while minimizing overflows

is better than



Towers of Hanoi



The poblem :

- Move all the discs from departure peg A to arrival peg C, with the following rules
  - Move only one disc at a time.
  - Never put a larger disc on a smaller one.

We can use the peg B as an auxiliary peg at any time.

The solution by recursion :

```
Move (N, A, B, C)
    IF (N = 1) THEN
        A → C
    ELSE
        Move (N-1, A, C, B)
        A → C
        Move (N-1, B, A, C)
```

Hashing

- Hashing is a searching technique whose complexity is independant of n.
- The idea is to give to each element of a list a code, and to use this code as a pointer to the element.
- We can see immediately that, as the code is derived from the element and points to it, the element has to be unique.
- So, this technique is to be used for indexing arrays of unique keys
- For example, it can be used for maintaining an index of words, a table of keywords (compilers use it!), etc.

Hashing (continued)

- A hashing function is a transform between keys and pointers.
- A hashing function should be simple and quick to compute.
- A hashing function should distribute uniformly the pointers over the memory area.

Example: Suppose we want to have a large file of records for the personnel of a company, and we want a very quick access to these records.

We can use the name of each employee to construct a hash code, with a function

$$H(\text{name}) = \sum_i \text{name}(i) \cdot i \pmod{m}$$

where m is the size of the memory allocated to the records pointers.

Hashing (continued)

- Each name of employee is transformed in a hash-code, which is a pointer to the record keeping the information relative to this person.
- We cannot avoid that the function transforms two different names in a same hash-code.
- That case is called a collision, and some method has to solve it.

Collision resolution

- The load factor is the ratio between the number of keys (K) and the number of hash-adresses (m = size of allocated memory)

$$\lambda = \frac{K}{m}$$

- The load factor should be kept sufficiently small to limit the number of collisions, but cannot guarantee there won't be any

Collision resolution (continued)

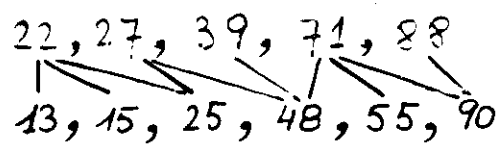
- Thus, we must provide a way to solve those.
- One way is to assign to the key getting an already busy H-code the next available memory.
- Other methods may be used, to minimize the clustering obtained by this technique.
- A good way is to have linked lists containing the pointers to the records, and a hash-table pointing to the linked lists. This is called Chaining and can be very efficient.

Merge - sort

To merge two arrays in the action of combining the elements of both arrays in a single sorted array.

When both arrays are already sorted, the easier way is to compare the first element of each list and to pick-up the smaller, then compare the first element of each remaining list, etc...

Example:



13, 15, 22, 25, 27, 39, 48, 55, 71, 88, 90

Merge-sort (continued)

- The idea is to split an array in two parts, sort these parts, and then merge them.
- Each of the part is in its turn sorted by the same algorithm.
- We can do that without supplementary space requirement to keep the sub-arrays:

  ① Merge each pair of elements
  ⇒ sorted pairs

  ② Merge each pair of sorted pairs
  ⇒ sorted quadruples

  ③ Merge each pair of sorted quadruple
  ⇒ sorted 8-uples

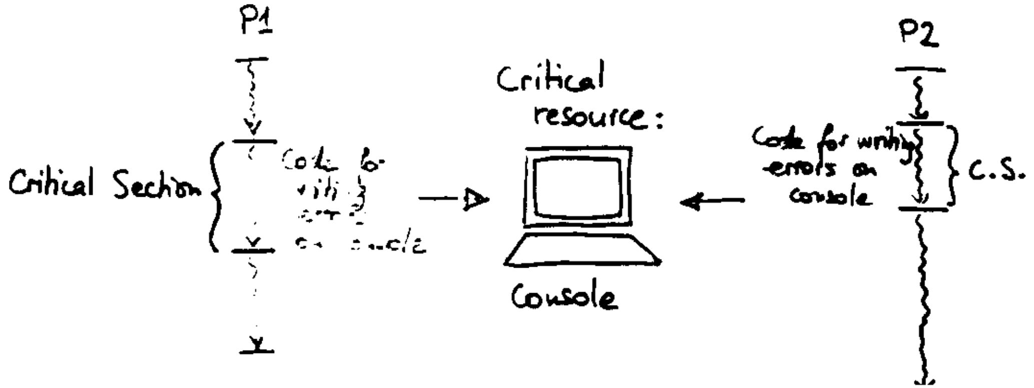
  ⋮

  ⓝ Merge the two sorted subarrays.
  ⇒ sorted array

Mutual exclusion

- Resources accessed by more than 1 process should be protected against simultaneous access.
- Such resources are called critical resources.
- The access protection is realized by mutual exclusion.
- The part of a process code accessing a critical resource is called a critical section.

Example: A process P1 is accessing a printer, a process P2 is accessing a memory disk. Both processes should report errors on a console.



Active waiting

Critical resource busy ⇒ Process has to wait in a loop until the resource becomes available.

- This should never be implemented with only one boolean variable:

```
P1 : boolean busy
     :
     while busy do end
     busy := true
```

Processor is attributed to process P2

P2 can access legally the resource (busy = false)
P1 also can access the resource, because even when P2 sets busy true, it doesn't check it anymore!

- Implementation through two variables can be safe. There are different techniques.
- Not use anyway, because wasting computing time!

Interrupt masking

- Disable interrupt system while in critical section.
- No more interrupts ⇒ no process commutation.

Many problems arise at this point:

- Input/Output operations in interrupt mode are impossible in the critical section.
- Interrupts can be lost: If more than one interrupt arrive during the critical section from one peripheral, they will be lost.

In shared memory multiprocessors systems, you cannot realize mutual exclusion by masking the interrupts, because processes are really executing simultaneously.

Locks

- Record containing:
  - Boolean variable
  - List of waiting processes. (queue)

- 2 manipulation procedures: lock, unlock
  - Mutual exclusion primitives
  - Never interrupted.

lock:
- called at the beginning of a critical section.
- lock open → close it
- lock closed → process put in the queue

unlock:
- called at the end of a C.S.
- queue empty → open lock
- queue not empty → schedule first process

Primitives can be executed with interrupts masked,

because:
- very short
- No I/O operations.

Semaphores

- N critical resources allocation: locks don't fit.
- Semaphores presented by Dijkstra in 1968.
- Record containing:
  - Integer counter
  - Processes queue
- 2 manipulation procedures: P, V

P:
- called at the beginning of a critical section.
- decrements counter
- if counter negative → process is put in the queue

- Negative counter:
  - no more available resource.
  - number of processes in the queue.

V:
- called at the end of a critical section
- increments counter
- if counter non positive → schedule first process

- As for locks, primitives can be executed with interrupt system disabled.

- Semaphore initialized to 1: mutual exclusion.

Process synchronization.

- Process synchronization is needed when a certain order is required in the sequence of operations

- Some tools help managing synchronization.

  - Events : Record containing a boolean variable and a process queue.

    Three procedures are used to manipulate the events : wait, set, reset.

  - Semaphores : Initialized to ∅ for 2-process synchronization

  - Monitors : Modules containing the data structures and procedures implementing the synchronization mechanisms.

    Procedures are protected against simultaneous access. (Mutual exclusion.)

Process synchronization (continued)

Synchronization by the data.

- Mailboxes : A variable on which are defined two procedures : send , receive

  - The receive procedure is put in a sleeping state if the mailbox is void.

- Rendez-vous : A variable, and two procedures manipulating it : P!e , P?v

  where P2!e sends expression e to processus P2.

  P1.?v receives in variable v a value from processus P1.

  - The receive procedure P1?v blocks the process P2 until the variable v gets the message.

  - The send procedure P2!e blocks the process P1 until procedure P1?v has been executed in process P2.

Example of a concurrent problem.

We want to implement a chronometer-watch that will display on a terminal screen:

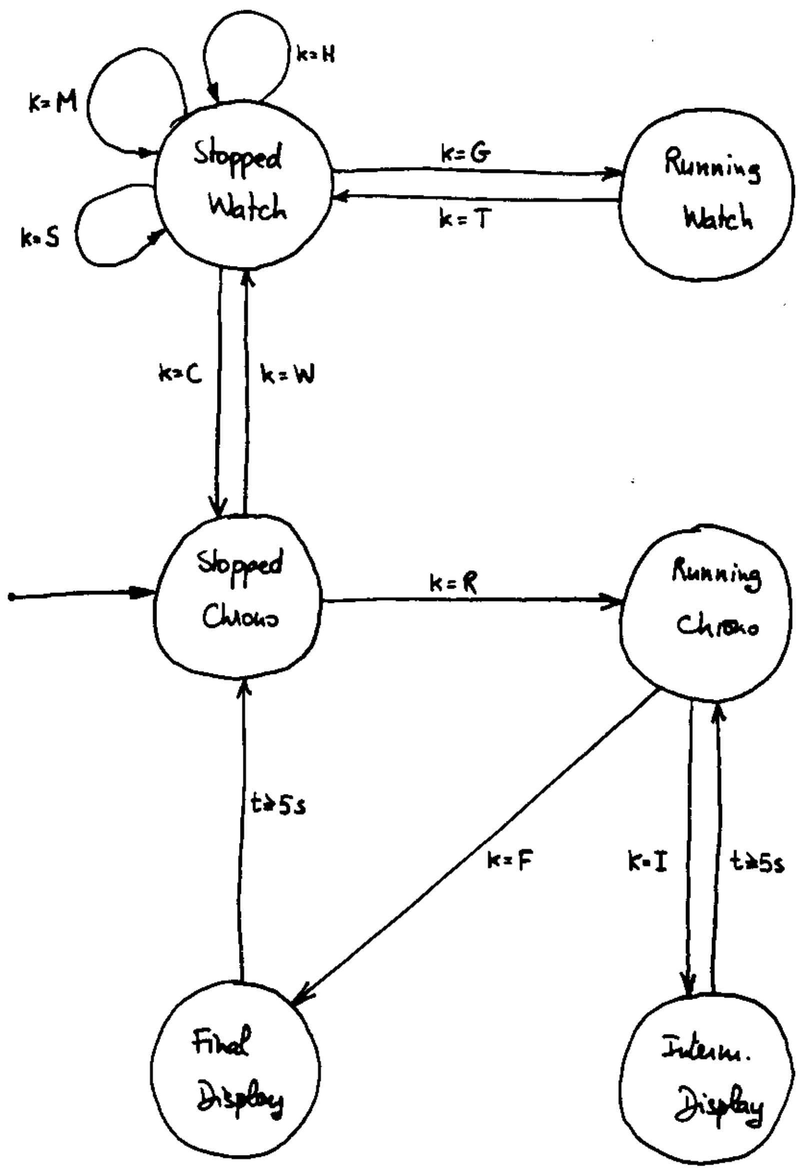
- At initialization: 00:00:00

Then, keyboard "one-key" command are driving the instrument:

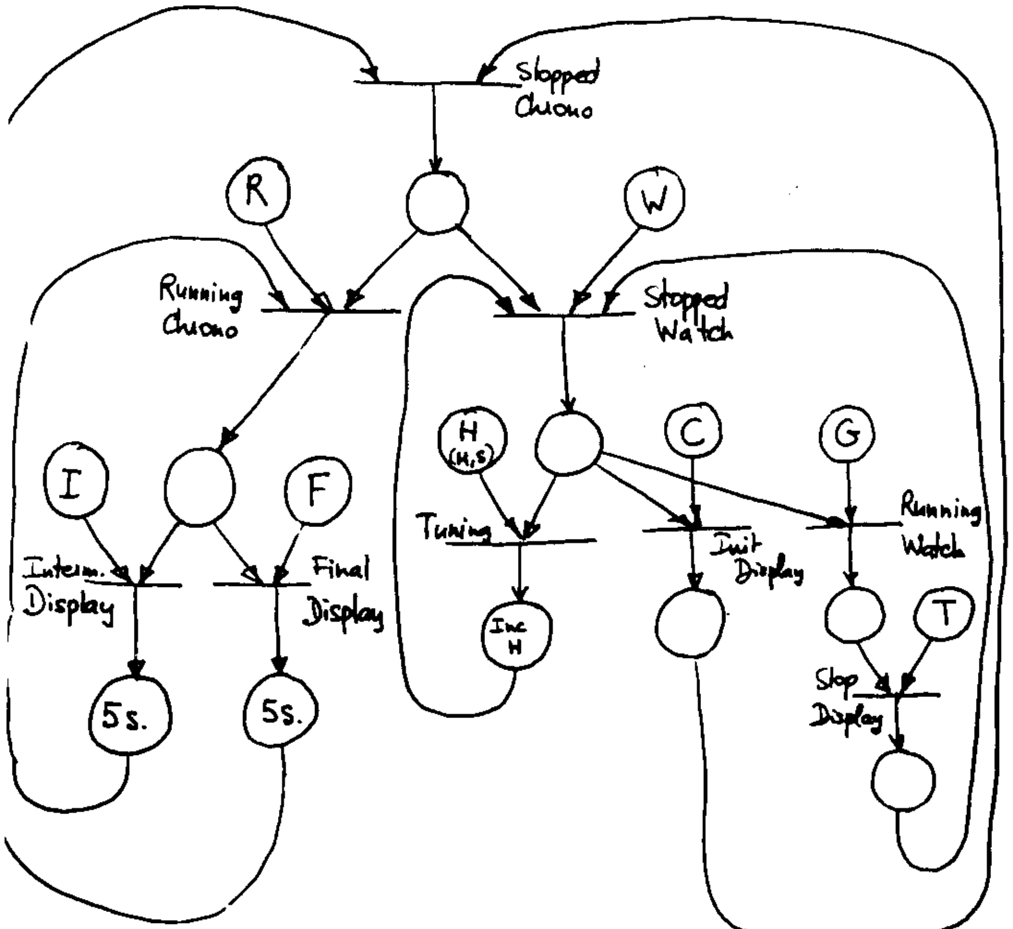
W : s.chrono → s.watch mode selection
C : s.watch → s.chrono mode selection
H : s.watch → increments hours
M : s.watch → " minutes
S : s.watch → " seconds
G : s.watch → r.watch
T : r.watch → s. watch
R : s.chrono → r. chrono
I : r. chrono → i. chrono
F : r. chrono → f. chrono

s. watch , s. chrono : stopped watch & chrono modes
r. watch , r. chrono : running watch & chrono modes
i. chrono : intermediate display for 5 seconds
f. chrono : final display for 5 seconds.

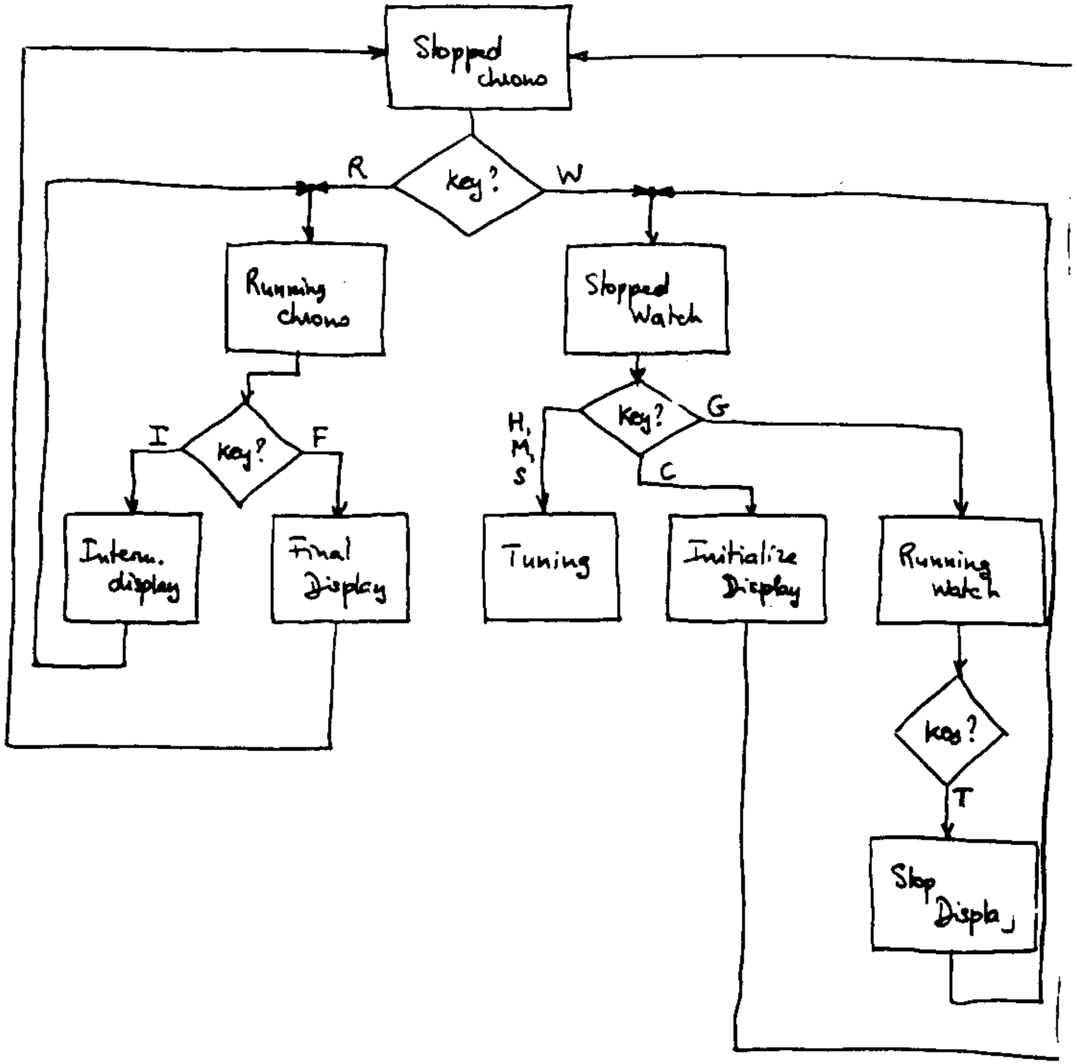
Chrono Statechart



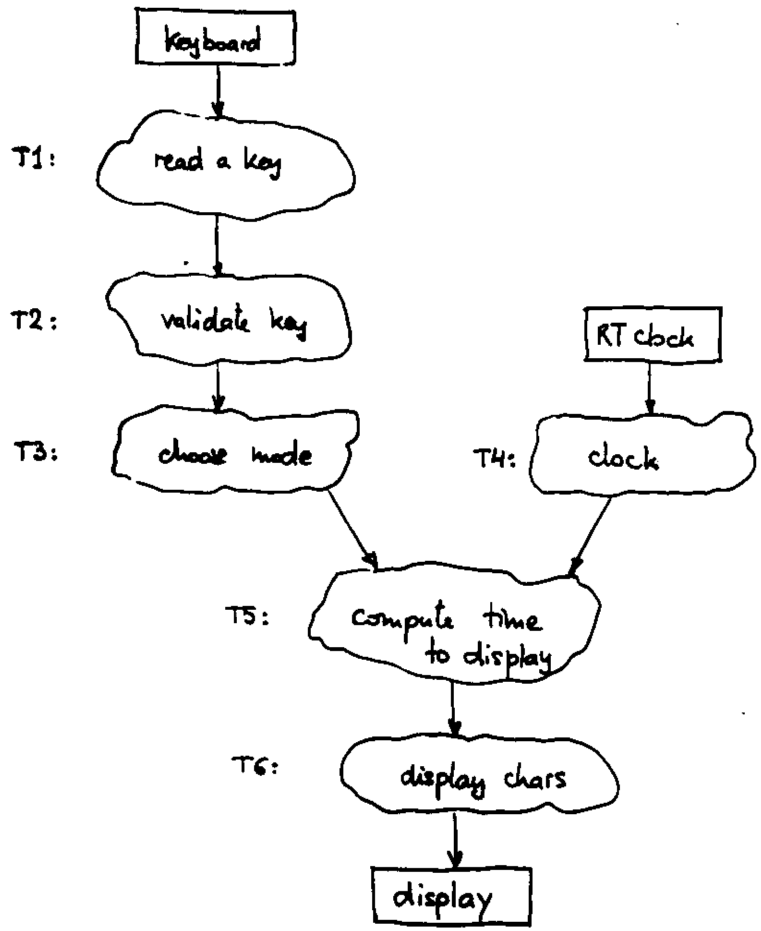
Petri Graph for the Chrono



Flowchart of the chrono



Data flow study



Process decomposition.

- According to DARTS, we have to select the data flow transforms which will receive a separate process.

- I/O dependancy : a process should be given to transform T1 (keyboard input) and to transform T6 (display output).
- Time critical functions : none
- Computational requirements : none
- Functional cohesion : Same process for T2 (validate key) and T3 (choose mode)
- Temporal cohesion : T5 (compute time to display) is to be put in the same process as T2 & T3, or as T4. Mode dependant.
- Periodic execution : T4 (clock) is to be executed at each RT clock tick. Should be kept in a separate process.

INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

UNITED NATIONS INDUSTRIAL DEVELOPMENT ORGANIZATION
NTERNATIONAL CENTRE FOR SCIENCE AND HIGH TECHNOLOGY
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS 34100 TRIESTE (ITALY) VIA GRIGNANO, 9 (ADRIATICO PALACE) P.O. BOX 586 TELEPHONE 040-224572 TELEFAX 040-224575 TELEX 460449 APH I

SMR/643 - 12

SECOND COLLEGE ON
MICROPROCESSOR-BASED REAL-TIME CONTROL -
PRINCIPLES AND APPLICATIONS IN PHYSICS
5 - 30 October 1992

UNIX and LynXOS

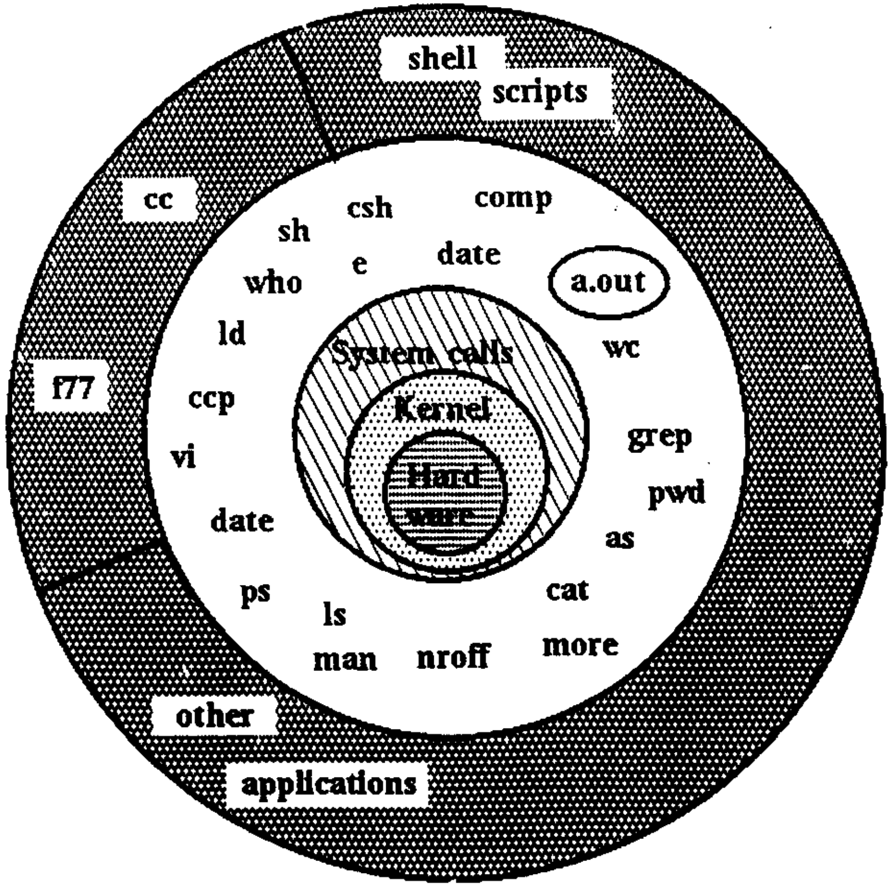
U. RAICH
E.P. Division
CERN
CH-1211 Geneva
Switzerland

These are preliminary lecture notes, intended only for distribution to participants.

Theme: **The Unix System Calls** Slide no: 1.1

Topic: **Lecture Overview**

Architecture of the Unix System



Theme: Course Introduction **Slide no:** 1.2

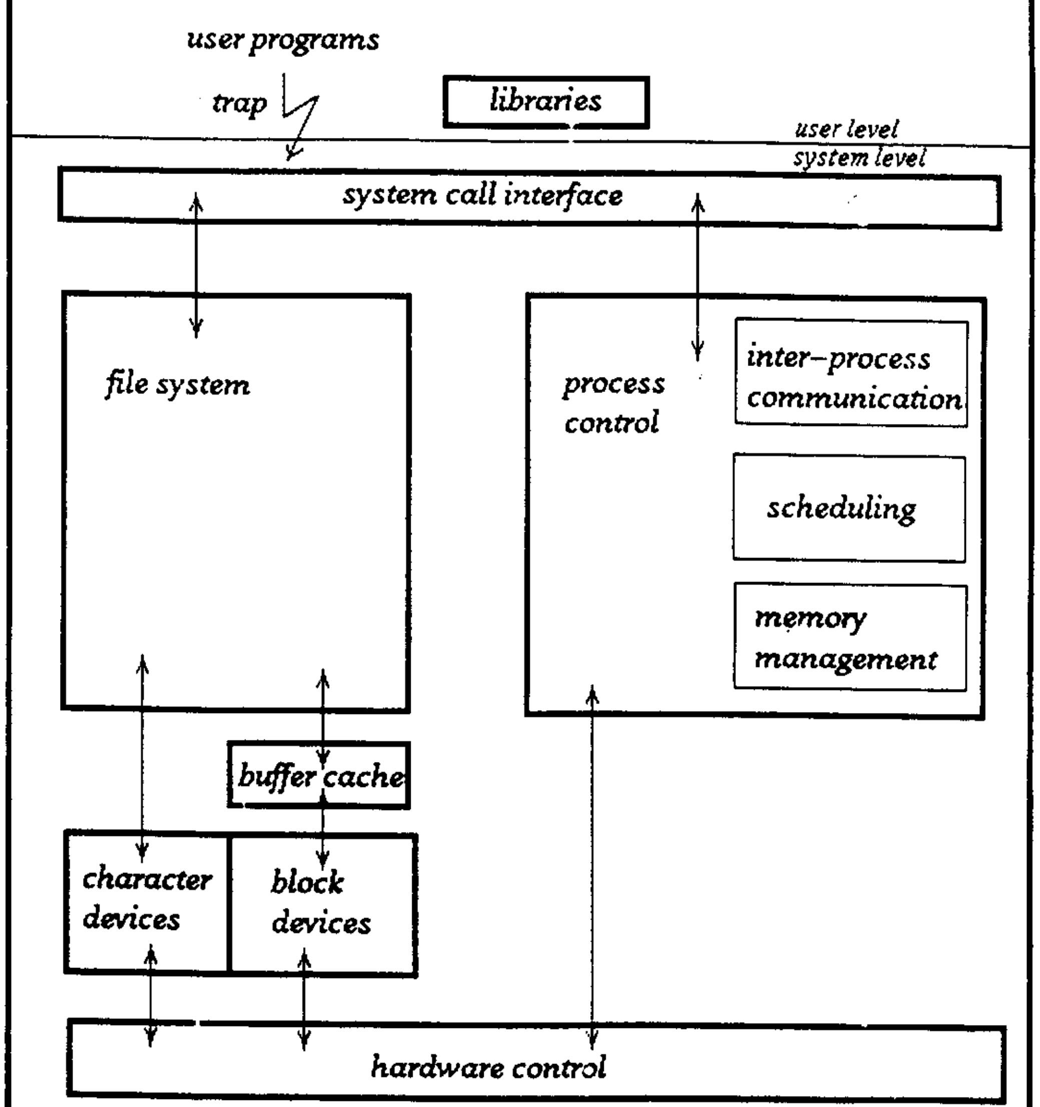
Topic: Advantages of Unix

What makes Unix Systems so popular?

- *System is written in high level language, thus portable. (Less than 3% of the kernel in assembly)*
- *Simple yet powerful user interface*
- *Hierarchical file system allowing easy implementation and maintainance*
- *Consistent file format (the byte stream)*
- *Simple consistent interface to peripheral devices*
- *Multiuser, multiprocess system*
- *Provides primitives to permit complex programs to be built from simpler programs*
- *Hides machine architecture*

Theme: The Unix Kernel **Slide no:** 1.3

Topic: Overview of Internals



eme: The Unix Kernel (Internals) slide no: 1.4

pic: The Unix File System

Steps to be executed when reading/writing a file:

Opening the file

generate entries into tables

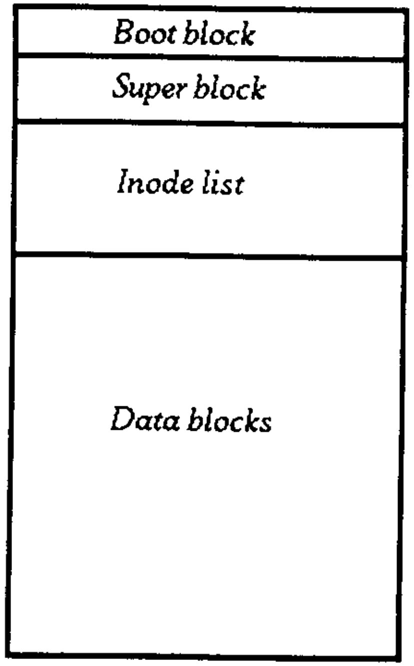
- *allowing a process to reference the file*
- *and allowing the kernel to know which file in the system is open for read, write or both*
- *convert the filename into a more easily accessible structure describing the file (inodes)*
- *allocate new inodes*
- *allocate data blocks on disk*

Writing/Reading a file

- *Convert the user's view of a file into a systems view*
- *Convert location inside the file to disk block numbers*

Theme: The Unix Kernel (Internals) slide no: 1.5

Topic: The Unix File System



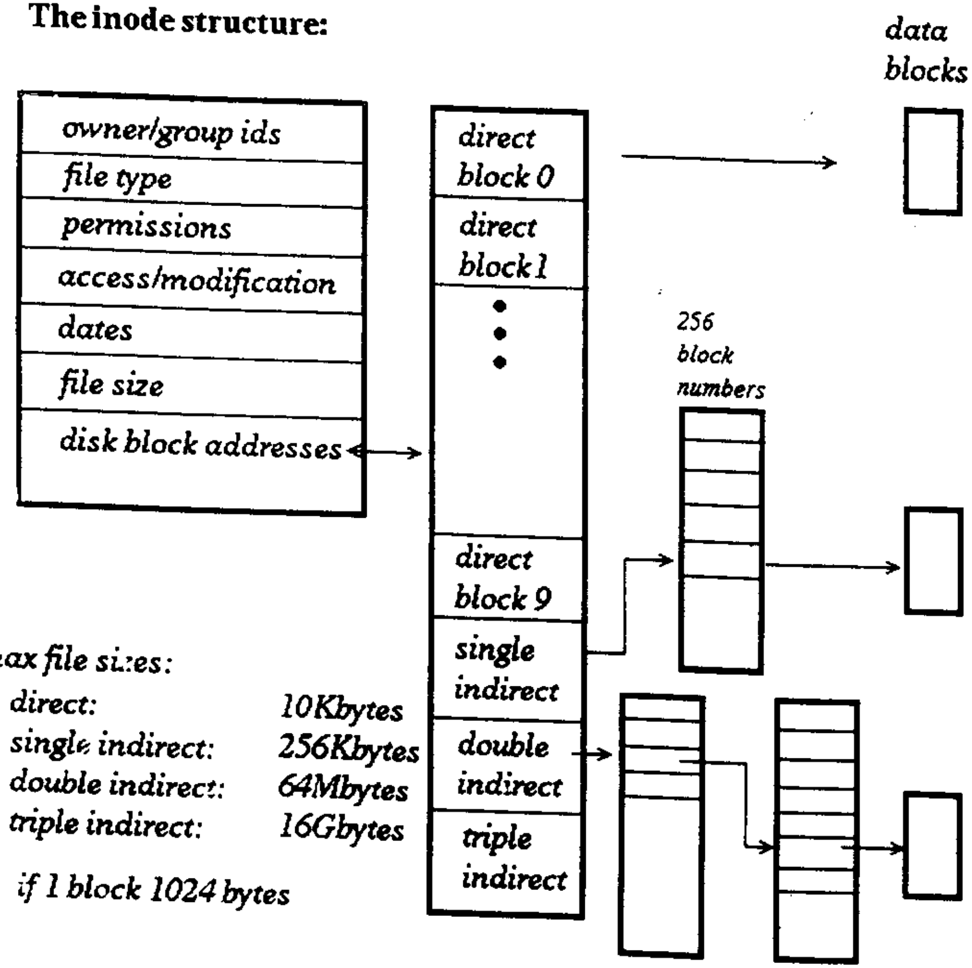
Boot block:	*Needed to load and start /vmunix (the operating system image)*
Super block:	*Describes the file system on a disk partition*
Inode list:	*An inode describes a file. The length of the inode list determines the maximum number of files in the file system*
Data blocks:	*Space available for user data*

Theme: The Unix Kernel (Internals) Slide no: 1.6

Topic: The Unix File System

Reading/writing a Unix file

The inode structure:



max file sizes:

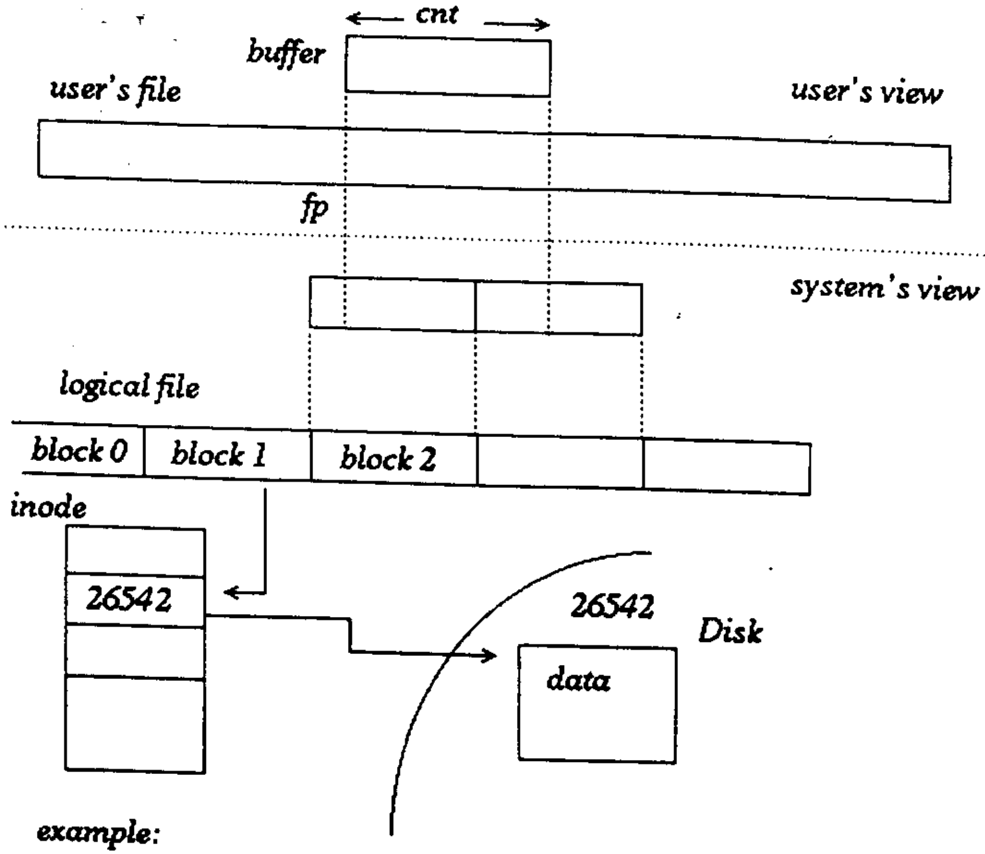
direct:	*10Kbytes*
single indirect:	*256Kbytes*
double indirect:	*64Mbytes*
triple indirect:	*16Gbytes*

if 1 block 1024 bytes

Theme: The Unix Kernel (Internals) Slide no: 1.7

Topic: The Unix File System

Writing or reading a Unix file



example:

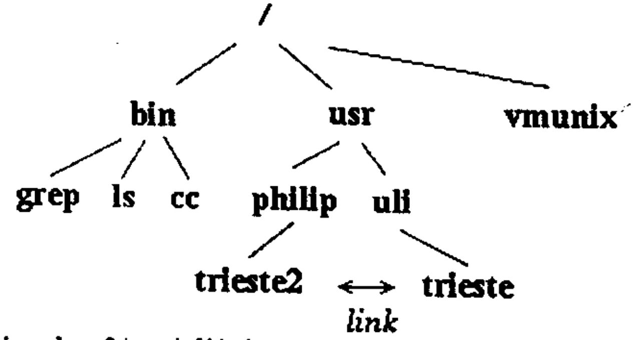
write buffer of size 1k
starting from file pointer at 1500

eme: The Unix Kernel (Internals) slide no: 1,8

pic: The Unix File System

Converting a filename to an inode

The layout of a directory file



ample: get inode of /usr/uli/trieste

inode	*filename (14 chars)*
17	.
17	..
207	vmunix
118	usr
34	bin

absolute reference:

/ is known to the system in a global variable

118	.
17	..
204	uli
23	philip

relative reference:

The current directory can be found in the process descriptor

204	.
118	..
193	trieste

23	.
118	..
193	trieste2

Theme: The Unix Kernel (Internals) slide no: 1.9

Topic: The Unix File System

The Super Block

Allocating inodes (when creating a new file)

Super block fields	
file system size	
no of free data blocks *list of free data blocks* *index of next free block*	*allocation of disk blocks*
size of inode list *no of free inodes* *list of free inodes* *index to next free inode*	*allocation of inodes*
lock field for free block and free inode lists *modified flag*	

Algorithm:

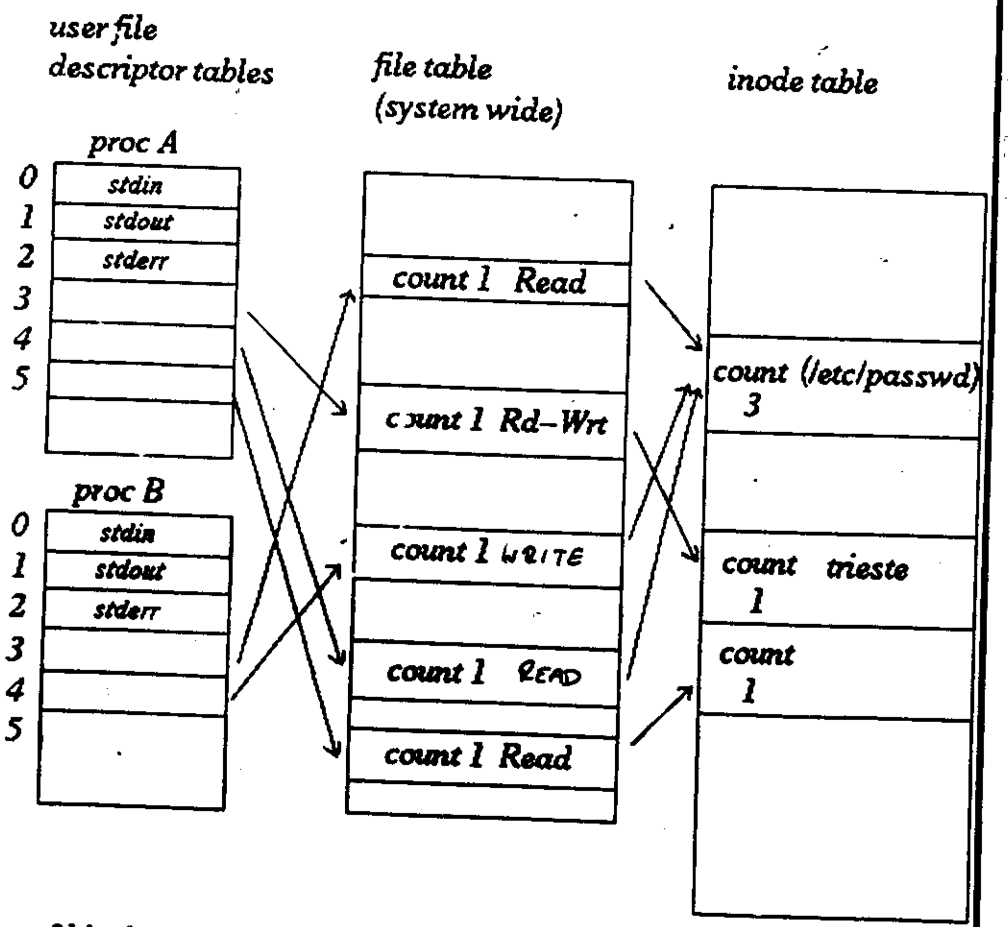
- *read free inodes from disk*
- *build a free inode table in memory (type field = 0 means free, remember last free inode on disk)*
- *allocate inode from memory list until exhausted, then read inodes from disk starting a remembered position*

Theme: The Unix Kernel (Internals) **Slide no: 1.10**

Topic: The Unix File System

Descriptor tables of "open" files

fd = open("myfile.dat",O_RDONLY)

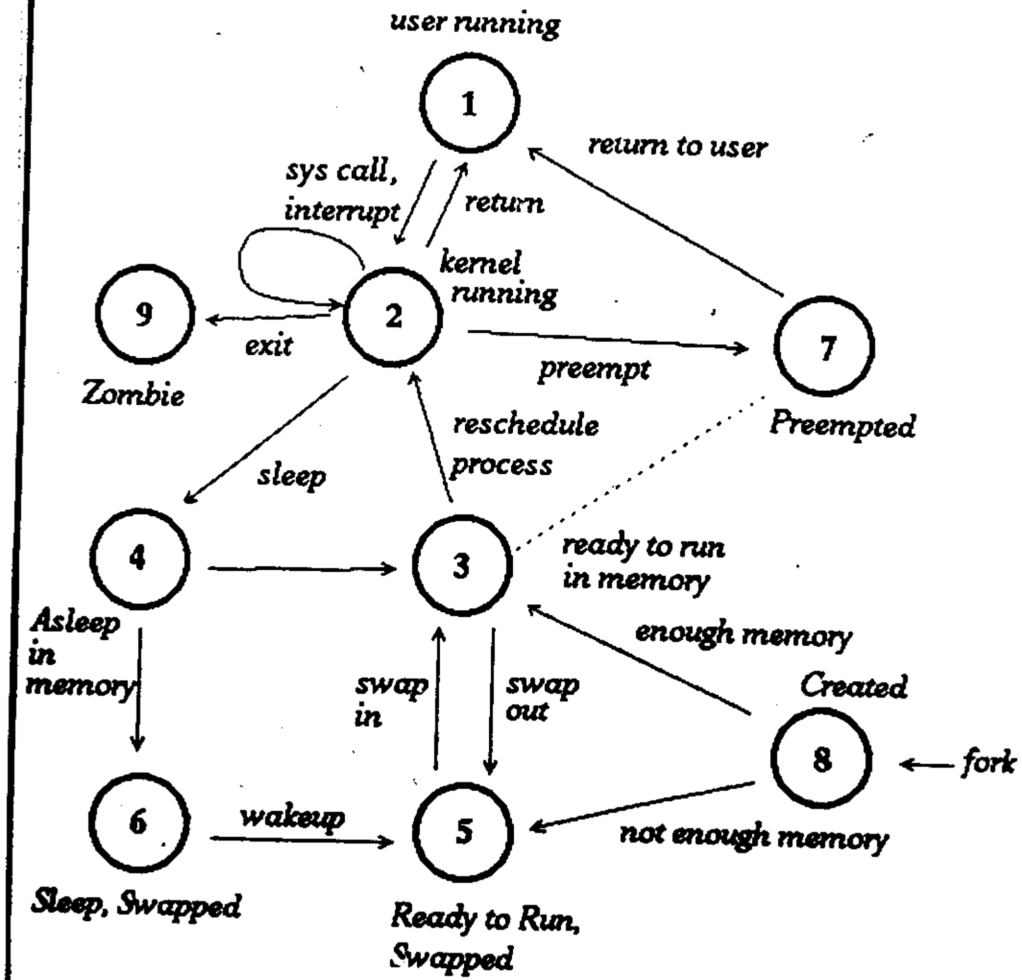


fd is the index into the user file descriptor table

Theme: The Unix Kernel (Internals) **Slide no: 1.11**

Topic: Process Management

Process State Diagram

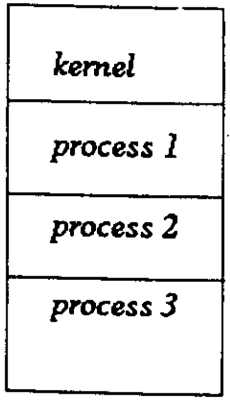


Theme: The Unix Kernel (Internals) Slide no: 1.12

Topic: Process Management

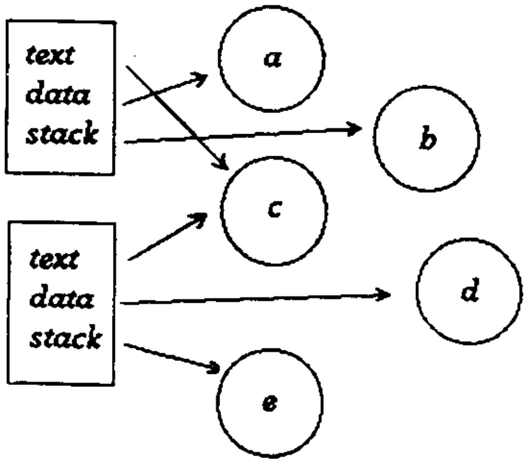
Layout of system memory

Possibility: *Compiler generates absolute addresses but: impractical*



Solution adopted: Compiler generates **virtual** *addresses which a memory management mechanism transforms into (real)* **physical** *addresses*

The virtual address space is subdivided into **regions**



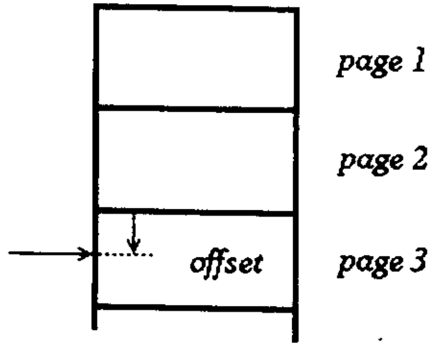
Theme: The Unix Kernel (Internals) Slide no: 1.13

Topic: Process Management

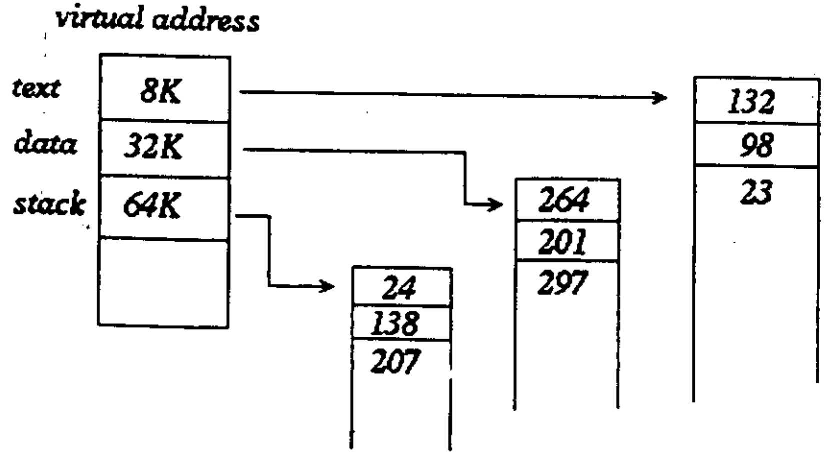
The paging system

Physical memory is divided into equally sized **pages**

A virtual address is converted into a **page number** *and an* **offset**



The **region tables** *contain pointers to* **page tables**



Theme: The Unix Kernel (Internals) Slide no: 1.14

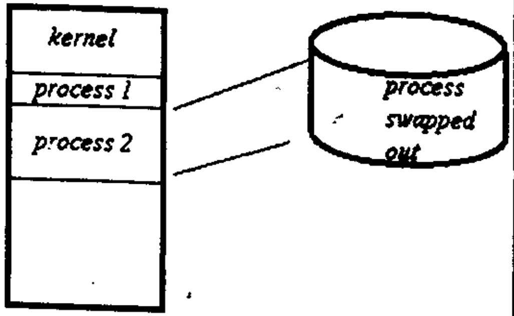
Topic: Process Management

Memory management policies

- **Swapping**

  The entire process is copied from memory to disk

  When
  - *creating new process*
  - *increasing process region*
  - *increasing stack space*
  - *swapping in a process*

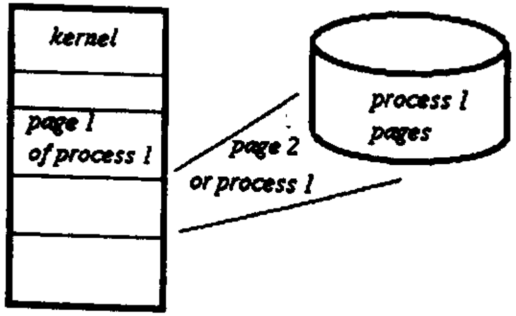


- **Demand Paging**

  Machines whose memory architecture is based on pages and whose CPU allows to rerun failed instructions can support a kernel with demand paging

  Accessing a virtual address whose page is not resident in memory generates a **page fault**

  The missing page is read from memory and the faulty instruction is rerun.



- **Hybrid systems**

  Both, demand paging and swapping.
  When the kernel cannot allocate enough memory pages a complete process is swapped out.

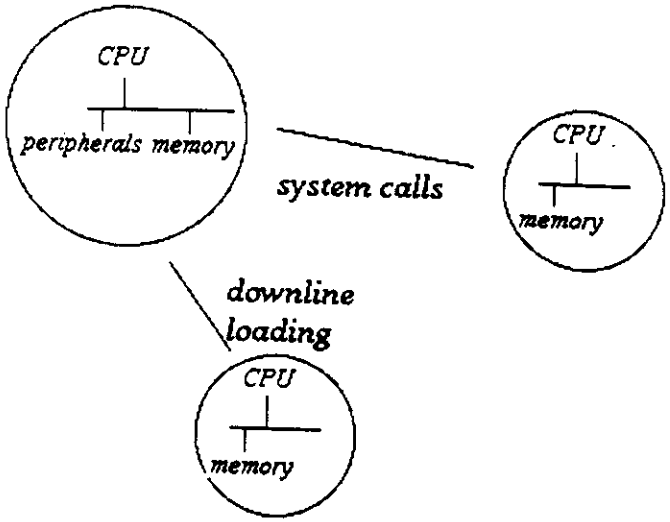
Theme: The Unix Kernel (Internals) Slide no: 1.15

Topic: Distributed Unix Systems

- **Satellite systems**

  One main processor containing CPU, memory and peripherals and several satellites with CPU and memory (+ communications) only.

  Programs and a (stripped down) operating system are downline loaded.
  Each satellite has an associated **stub process** *running in the main processes treating requests for system calls*



ame: The Unix Kernel (Internals) Slide no: 1.16

ic: Distributed Unix Systems

he Newcastle connection

Each machine runs the full kernel (including treatment of system calls. File sharing is implemented trough an extension to the file name:

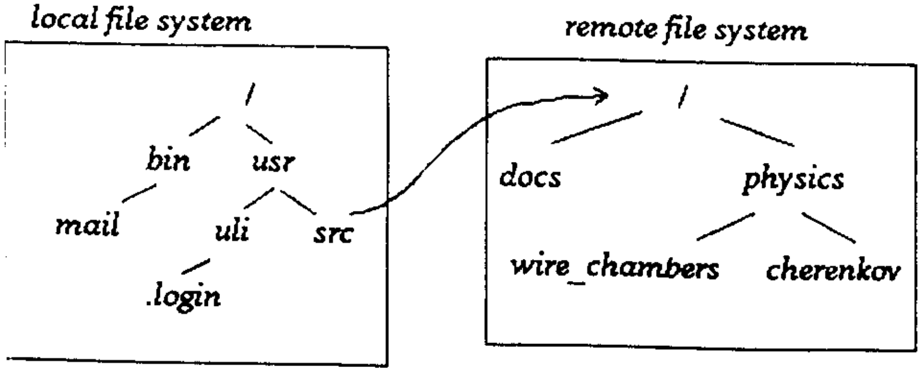
trieste!/usr/uli/course

specifies file /usr/uli/course on machine "trieste"

Needs special C library in order to parse file names

ransparent distributed systems (example: NFS)

A remote file system is **mounted** *on a mount point of the local file system*



/usr/src/physics/cherenkov accesses the file on the remote file system

---

Theme: The Unix System Calls Slide no: 2.1

Topic: Introduction

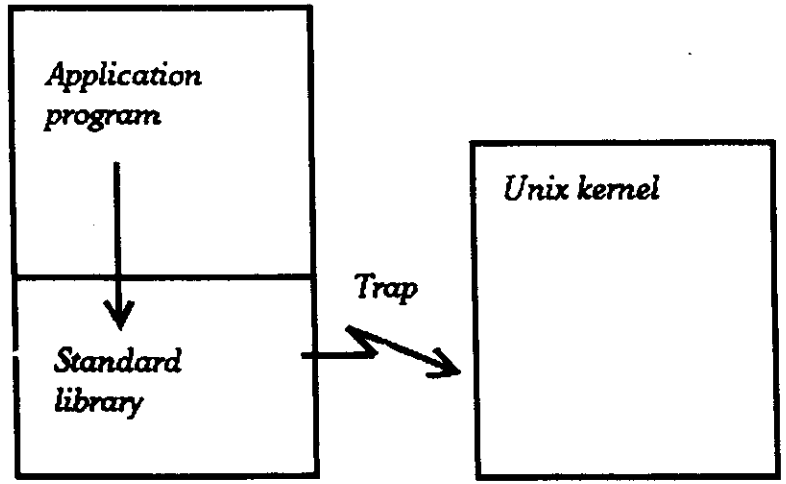
Generalities on system calls

System calls form an integral part of the Unix kernel and are therefore

- *executed in supervisor mode*
- *cannot be preempted*

They are accessed through a "trap mechanism" (software interrupt)

Access to system calls



Theme: The Unix System Calls   Slide no: 2.2

Topic: An Overview of the system calls

- **Access to the file system:**

  open,creat
  close
  read,write
  lseek
  unlink

- **Process handling**

  fork
  exec
  exit
  wait

- **Interprocess communication**

  signals
  signal,kill,alarm
  pipes,fifos
  IPC **package** (*Inter Process Communication*)
  messages
  semaphores
  shared memory segments

Theme: The Unix System Calls   Slide no: 2.3

Topic: System Calls for File System Access

The same routines allow to access

- *disk files*
- *pipes/fifos*
- *"special files" (device drivers)*

open *opens a file for reading or writing*

creat *creates an empty file (shrinks an existing file to size zero)*
(in earlier versions of Unix "open" worked only on existing files)

fildes = open(pathname, flags, [mode])

flags:	O_RDONLY	O_CREAT
	O_WRONLY	O_TRUNC
	O_RDWR	O_EXCL
		O_APPEND ...

mode: *access permissions*

fildes = open("myfile",O_WRONLY|O_CREAT|O_APPEND,0644)

opens "myfile",

if non existant:

creates with permission

user	group	world
rwx	rwx	rwx
110	100	100

else

sets file pointer to end of file.

eme: The Unix System Calls slide no: 2.4

pic: System Calls for File System Access

Writing and reading data to and from files

n_written = write(fildes,buffer,bufsiz)
n_read = read(fildes,buffer,bufsiz)

eof is detected by n_read = 0

increments file pointer by bufsiz

for efficiency reason use

- *rather big buffers (limits the number of system calls)*
- *buffers sizes being multiples of the natural disk blocking factor (mostly 1024 bytes)*

Random access to files

newpos = lseek(fildes,offset,direction)

long	*offset:*	*specifies new position in file*
int	*direction:*	*0: offset=nr of bytes from start of file*
		1: offset added to current position of file pointer
		2: offset added to pos. of last byte in file

example:

filsiz = lseek(fildes,0L,2)
returns size of file

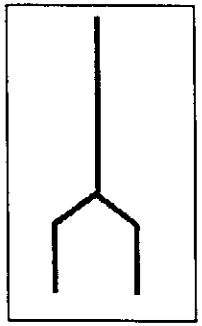
Theme: The Unix System Calls slide no: 2.5

Topic: Process Control

Process Creation

All new processes are created through a **fork** *system call*

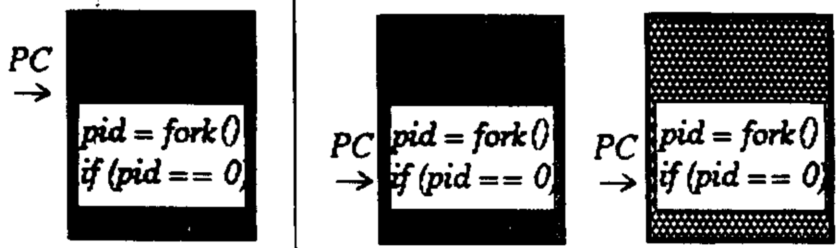
example:



```
main()
{
   int  pid;
   printf("Before fork \n");
   pid = fork() ;
   if (pid == 0)
      printf("child process\n");
   else if (pid > 0)
      printf("parent process\n");
   else
      perror("Fork returned error:\n");
}
```

Fork creates a second instance of the same process. The program code as well as the variables are identical in both processes.

before fork	*after fork*



parent process *parent process pid = child's pid* *pid = 0*

Theme: The Unix System Calls — Slide no: 2.6

Topic: Overlaying the Child Process with "exec"

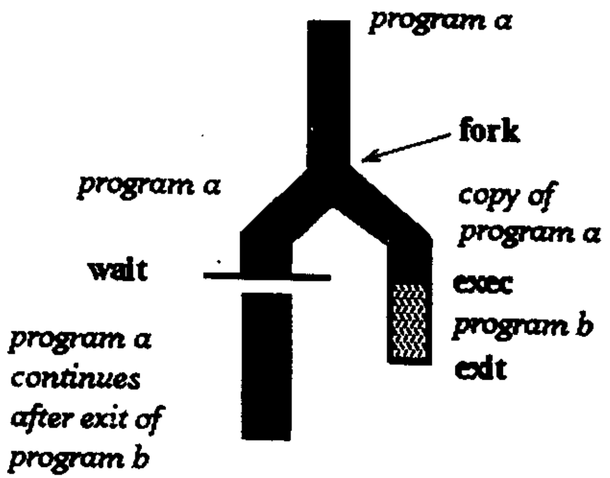
The "exec" family of system calls

The **exec** *calls load a new program into the calling process memory space. The old program is oblitered by the new*

```
ret = execl(path,  arg0,  arg1,...(char *)0)
ret = execv(path,argv)
ret = execlp(file,  arg0,  arg1,...(char *)0)
ret = execvp(file,argv)
```

path: must be a true program
file: may be a true program or a shell script

Sequence of fork,exec,wait,exit calls



usage of wait and exit:

pid = wait(&status)
exit(status)

With this knowledge we are able to create a shell !!!(CLI)

Theme: The Unix System Calls — Slide no: 2.7

Topic: Signals

Sending and receiving signals

On exception events (^C,illeagal instr.,floating point exception etc.) the kernel sends a signal to the process. This normally exits the process. However a process may decide to catch the signal and treat it. Processes may also send signals to other processes.

	SIGINT,SIGQUIT	*user interrupt*
send by kernel	*SIGILL*	*illeagal instr.*
	SIGKILL	*forced exit (cannot be caught)*
	SIGPIPE	*write to pipe without end*
	SIGALRM	*time elapsed*
send by process	*SIGTERM*	*terminate child*
	SIGUSR1,SIGUSR2	*for free use by process*

Catching a signal:

int catchit(); — *define an exception handler;*

signal (SIGUSR1,catchit); — *connect the handler with the signal*

Each time the signal SIGUSR arrives "catchit" will be executed.

Sending a signal:

kill(pid,SIGUSR1) — *since pid is needed signals can only be sent to parent or offspring*

(getppid returns pid of parent)

me: The Unix System Calls slide no: 2.7a

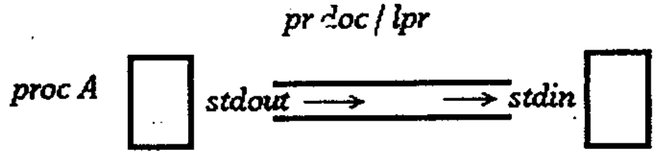
ic: Signals

```
emonstrates interprocess communication
sing signals

lude <signal.h>
lude <stdio.h>

()


it pid,papa;  /* process identifier */
it n_char;
iar cherac[100];
it catchint();

intf("Creating a second process\n");
.d = fork();

 (pid > 0)  {/* parent process */
signal(SIGUSR1,catchint);
wait((int *)0);
exit(0);

 (pid == 0)  /* child process */
papa = getppid();
while (1) {
  n_char = read(0,charac,100); /* wait for character from stdin
                                                                */
  kill (papa,SIGUSR1);
};
end of main */

atchint()

f ("Saw a User 1 signal! \n");
```

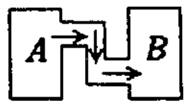
Theme: The Unix System Calls slide no: 2.8

Topic: Pipes

A pipe is a one way communications channel which couples one process to another and is yet a generalisation of the Unix file concept.




```
/* pipe implementation */
#include <stdio.h>
#define MSGSIZE=16

char *msg="Hi Trieste!";
main()
{
  char inbuf[MSGSIZE];
  int  p[2],pid; /* pipe file descriptors */
/* open the pipe */
  if (pipe(p) <0) {
    perror("pipe call ");
    exit(1);
  };
  if ((pid=fork()) < 0) {
    perror("fork call ");
    exit(2);
  };
  if (pid == 0) { /* child process */
    close(p[1]); /* close write section */
    read(p[0],inbuf,MSGSIZE);
    printf("Child read \"%s\" from pipe\n",inbuf);
  }
  if (pid > 0) { /* parent process */
    close(p[0]); /* close read section */
    write(p[1],msg,MSGSIZE);
  }
  exit(0);
}
$ _
```



Theme: The Unix System Calls   Slide no: 2.9

Topic: Pipes

Fifos or named pipes

Pipes can only be used between strongly related processes (e.g. parent child) because the pipe id is needed for reading and writing.
Named pipes remedy this problem:

A named pipe can be generated using the mknod program. The pipe is the opened as any normal file

```
$ mknod fifo p
$ ls -l fi*
prw-r--r--  1 uli                0 Oct 12 18:47 fifo
$
```

We have two entirely separate programs one opening the fifo for writing the othe one for reading:

reading program

```
/* pipe implementation */
#include <fcntl.h>
#include <stdio.h>
#define MSGSIZE=16

main()
{
  char inbuf[MSGSIZE];
  int  fd,pid; /* pipe file descriptors */
/* open the pipe */
  if ((fd= open("fifo",O_RDONLY)) < 0) {
    perror("pipe call ");
    exit(1);
  };
  read(fd,inbuf,MSGSIZE);
  printf("Child read \"%s\" from pipe\n",inbuf);
  close(fd);
  exit(0);
}
```

Theme: The Unix System Calls   Slide no: 2.10

Topic: Pipes

Here is the writing program:

```
/* pipe implementation */
#include <fcntl.h>
#include <stdio.h>
#define MSGSIZE=16

char *msg="Hi Trieste!";
main()
{
  char inbuf[MSGSIZE];
  int  fd,pid; /* pipe file descriptors */
  if ((fd=open("fifo",O_WRONLY)) < 0) {
    perror("pipe call ");
    exit(1);
  };
  write(fd,msg,MSGSIZE);
  close(fd);
  exit(0);
}
```

and the result:

```
$ fif1&
2898
$ fif2&
2899
$ Child read "Hi Trieste!" from pipe
```

me: The Unix System Calls Slide no: 2.11

ic: IPC Facilities

Inter process communication facilities (IPC)

3 IPC constructs are provided by the kernel:

- *Message passing*
- *Semaphores*
- *Shared memory*

IPC facilities are identified by unique **keys** *just as files are identified by file names*

A set of similar routines is available for each of the 3 mechanisms

The IPC **get** *operation :*

takes the user specified key and returns an id (similar to open/creat) If there is no IPC object with the specified key it may be created.

example: msg_qid = msgget((key_t)0100,IPC_CREAT)

The IPC **op** *calls:* *They do the essential work*

example: err_code = msgsnd(msg_quid,&message,size,flags)

The IPC **ctl** *calls:* *get or set status information for the IPC object specified or allow to remove it*

example: err_code = msgctl(msg_qid,IPC_RMID,&msq_stat)

Theme: The Unix System Calls Slide no: 2.12

Topic: IPC Facilities

Sending and receiving messages

A message has the form:

```
struct my_msg {
  long mtype;
  char mtext[LENGTH];
}
```

Such a message can be sent to a message queue who's identifier has been determined by a msgget call:

retval = msgsnd(msg_qid,&message,size,flags)

it can be read by:

retval = msgrcv (msg_qid,&message,size,msg_type, flags)

msg_type = 0:	*first entry in queue*
msg_type > 0;	*first entry of this type*
msg_type < 0;	*first entry with lowest msg_type*

Theme: The Unix System Calls Slide no: 2.13

Topic: IPC Facilities

Shared memory segments

Normally data regions of different processes are separated. The IPC shared memory facility allows several processes to share a section of physical memory.

shmid = shmget((key,size,permflags)

creates such a shared memory section in physical memory

memptr = shmat(shm_id,daddr,shmflags)

attaches the shared memory section to the process. memptr is a pointer in virtual addresses where the process can access the section

**memptr = "hello Trieste"*

will write this memory section.

err_code = smctl(semid,IPC_RMID, &shm_stat)

removes the shared memory section from the system

Theme: The Unix System Calls Slide no: 2.14

Topic: IPC Facilities

Shell commands supporting IPC facilities

There a two shell level commands treating IPC facilities:

ipcs: showing the state of all IPC objects in the system

```
IPC status from /dev/kmem as of Sat Oct 13 17:31:18 1990
Message Queues:
T      ID      KEY         MODE          OWNER     GROUP
q      0           64 --rw-rw-rw-         uli      users

Shared Memory
T      ID      KEY         MODE          OWNER     GROUP
m      0            0 --rw-------         uli      users

Semaphores
T      ID      KEY         MODE          OWNER     GROUP
*** No semaphores are currently defined ***
```

iprm: allows to remove an IPC object from the system

Theme: The Unix Shell — Slide no: 3.1

Topic: Introduction

What is a shell ?

A shell is a command string interpreter reading user input from stdin and executing commands.
However shell commands may also come from a file.

The standard Unix shells (ex. Bourne shell) provides:

- *I/O statements*
- *I/O redirection*
- *pipes*
- *variables & assignment statements*
- *conditional statements*
- *loops*
- *subshells*

→ *Full blown programs may be written using only shell commands (shell scripts)*

Theme: The Unix Shell — Slide no: 3.2

Topic: Simple Shell Commands

Simple commands:

Single word, no parameters

who: prints all login processes
ps: prints all processes started by the user on the standard output device (stdout)

newline or ";" are separation characters

```
$ who
uli       ttyp0   Oct  4 08:08   (:0.0)
uli       ttyp1   Oct  4 08:08   (:0.0)
uli       console Oct  4 08:07
$ ps
  PID TT STAT  TIME COMMAND
22692 co I     0:50 /usr/bin/X11/mwm
22693 p0 S    15:29 /usr/bin/dxterm -ls
22697 p0 I     0:05  (csh)
24984 p0 S     0:00  (sh)
24986 p0 R     0:00  (ps)
22694 p1 I    19:05 /usr/bin/dxterm -ls -n dxterm1
22698 p1 I     0:09  (csh)
24966 p1 I     0:52  (dxpaint)
$ _
```

Pipes

Stdout of one program can be connected to stdin of another one through a pipe

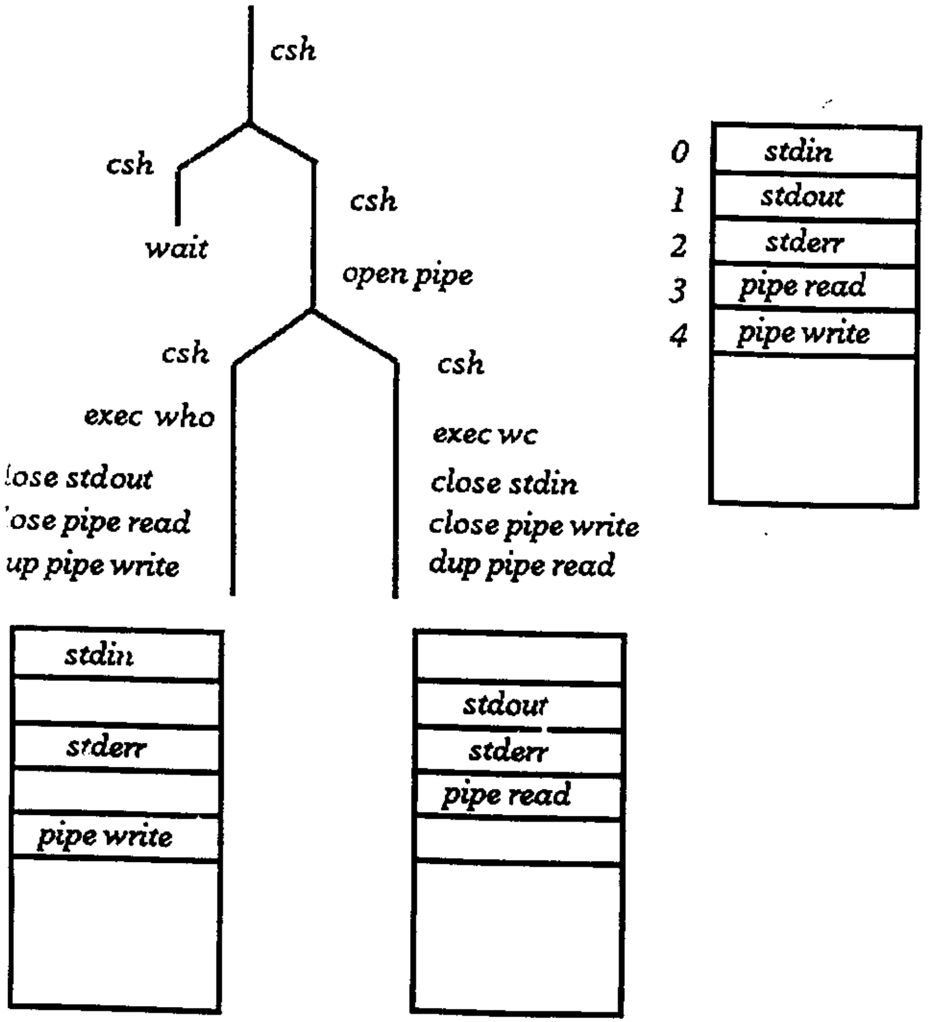
Example: We want to know the number of login processes on our system . This can be found by counting the number of lines output by **who**

```
$ who |wc -l
      3
$ _
```

Theme: The Unix Shell Slide no: 3.2a

Topic: Simple Shell Commands

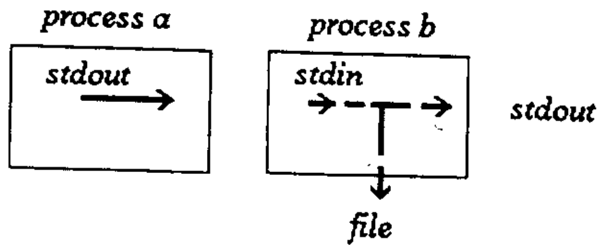
example: who | wc –l



Theme: The Unix Shell Slide no: 3.3

Topic: Simple Shell Commands

The tee command:




```
$ (date;who) |tee save |wc
       4       23      133
$ cat save
Tue Oct  9 17:23:05 MET 1990
uli      ttyp0   Oct  9 15:23   (:0.0)
uli      ttyp1   Oct  9 15:23   (:0.0)
uli      console Oct  9 15:21
$ _
```

Running commands in background:

```
$ (date;who) |tee save |wc >count &
923
$ cat save
Wed Oct 10 11:47:58 MET 1990
uli      ttyp0   Oct  9 15:23   (:0.0)
uli      ttyp1   Oct  9 15:23   (:0.0)
uli      console Oct  9 15:21
$ cat count
       4       23      133
$ _
```

Creating new commands

The shell is a user program as any other one provided by the system or written by you. It's name is **sh**

Since sh accepts input from stdin and we can redirect input to it from a file we execute shell commands from a file:

```
$ cat no_users          this is the contents of file
who |wc -l              no_users
$ sh <no_users          here we execute it
       3
```

If the shell is given an argument it interprets it as the file from which commands are to be read:

```
$ sh no_users
       3
```

We can even make the text file executable and call the shell implicitly:

```
$ chmod +x no_users
$ no_users
       3
$ _
```


Passing parameters into shell scripts

Write a shell script that adds execute permission to a file:

```
$ ls cx
cx: No such file or directory
$ echo 'chmod +x $1' >cx
$ ls -l cx
-rw-r--r--   1 uli          12 Oct 10 12:27 cx
$ sh cx cx
$ ls -l cx
-rwxr-xr-x   1 uli          12 Oct 10 12:27 cx
$ echo 'echo Hello fans !' >hello
$ hello
hello: cannot execute
$ cx hello
$ hello
Hello fans !
$ _
```

$0 : script name
$n: contents of nth parameter
$#: number of parameters
$: all parameters*
$?: exit status of last command executed

Theme: The Unix Shell **Slide no:** 3.6

Topic: Simple Shell Commands

Program output used as arguments

The output of programs can be used as arguments into other programs:

```
$ echo 'echo At the tone ^Gthe time will be exactly `date`' >tim
$ cat tim
echo At the tone the time will be exactly `date`
$ chmod +x tim
$ tim
At the tone the time will be exactly Thu Oct 11 16:29:54 MET 1990
$ _
```

Shell variables and environment variables

Variables can be defined and assigned strings
The environment variables are known to the shell

```
$ myvar=whatever
$ echo $myvar
whatever
$ echo $PATH
.:/usr/local/bin:/user1/uli/bin:/usr/ucb:/bin:/usr/bin:/usr/bin/X11
local/unix:/usr/new:/usr/hosts:/usr/local/unix:/usr/local/priam
$ _
```

Theme: The Unix Shell **Slide no:** 3.7

Topic: Filters

Programs that read input, perform some simple transformation and produce some output are called **filters**

examples: grep,tail,sort,wc,sed,awk...

grep: *searches files for a certain pattern and prints out lines containing it*

```
$ cat telephone
philip          2587
mark            3860
evelyn          1275
peter           6530
$ grep mark telephone
mark            3860
$ _
```

special meanings in grep:

^	*beginning of line*
.	*a single character*
[...]	*any character in ..., ranges allowed*
[^...]	*any character not it ...,ranges allowed*
e*	*any occurences of e*

grep '^[^:] ::' /etc/passwd*

passwd entry:
name:password:other information
name::other information means: no password was set!

Theme: The Unix Shell — Slide no: 3.8

Topic: Filters

The stream editor sed

Takes a stream of characters from stdin or from a file, transforms it using line editor commands and outputs it on stdout.

sed 'list of editor commands' filenames

example: sed 's/Mr Miller/Miss Smith/g' letter >new letter

```
$ cat letter
Dear Mr. Brown,
after the Trieste course I would like to invite you for a drink
at Mr. Miller's home. I think we all earned it. Mr. Miller
will be glad to welcome you all.
Best regards, the Trieste course organizers.
$ sed 's/Mr. Miller/Miss Smith/g' letter >new_letter
$ cat new_letter
Dear Mr. Brown,
after the Trieste course I would like to invite you for a drink
at Miss Smith's home. I think we all earned it. Miss Smith
will be glad to welcome you all.
Best regards, the Trieste course organizers.
$ _
```

Even more tricky: The list of editor commands may come from a file:

sed -f cmdfile

Theme: The Unix Shell — Slide no: 3.9

Topic: Flow of Control

Loops in shell programs

There are 3 loop constructs in the shell:

The **for** *loop*

```
for var in list of words
do
   commands
done
```

The **while** *loop*

```
while command
do
  loop body executed as long as command
  returns true
done
```

The **until** *loop*

```
until command
do
  loop body executed as long as command
  returns false
done
```

example:

```
until   who | grep uli
do
   sleep 60
done
```

Conditional statements

case word in
  pattern 1)   commands;;
  pattern 2)   commands;;
  ...
esac.

The **case** *is very often used to check the syntax of a command and to assign default values to optional parameters*

```
$ cat asm
incl=`echo $1 |sed 's/\..*//'`
out=$incl.o
incl=$incl.m
case $# in
 0) echo usage: $0 infile \[macro file\] \[outfile\]
    exit 2;;
 2) incl=$2;;
 3) out=$3;;
 *) ;;
esac
echo m6809 $1 $incl $out
exit 0
$ asm
usage: asm infile [macro file] [outfile]
$ asm z.a
m6809 z.a z.m z.o
$ asm z.a d.m
m6809 z.a d.m z.o
$ _
```


if ... then ... else

if command
  then cmds
  else cmds
fi

The if statements tests the exit status of 'command' ($?) and if successful (exit status = 0) executes the then clause.

In if statements the **test** *program is often used*

test −r file	*tests if file is readable*
test −f file	*tests if file exists*
test −w file	*tests if file is writable*
test s1=s1	*tests if two strings are equal*
test n1 −eq−n2	*tests if two numbers are equal*
...	

if test −r $1
then
  do something
else
  echo Cannot find file $1
fi

eme: **The Unix Shell** Slide no: 3.12

pic: **Flow of Control**

A shell script demonstrating conditional and loop constructs

The script replaces the ":" in the PATH environment variable by blanks and the checks for each resulting directory name if the file 'command' exits.

```
$ cat where
case $# in
0) echo 'Usage: whereis command' 1>&2; exit
esac
for i in `echo $PATH | sed 's/^:/.:/
                          s/::/:.:/g
                          s/:$/:./
                          s/:/ /g'`
do
  if test -f $i/$1
    then
      echo $i/$1
  fi
done
$ where where
./where
$ where ls
/bin/ls
$ _
```

Theme: **The Unix Shell** Slide no: 3.13

Topic: **Signals**

Catching signals

Typing ^C sends an interrupt signal to all processes run from your terminal. This will normally will terminate the processes.

The shell protects processes started in background from being terminated through ^C.

Shell scripts working with temporary files which are removed at the end of the script should do this cleanup also when terminated by ^C.

We can **trap** *signals and execute a 'trap handler' or we can ignore signals*

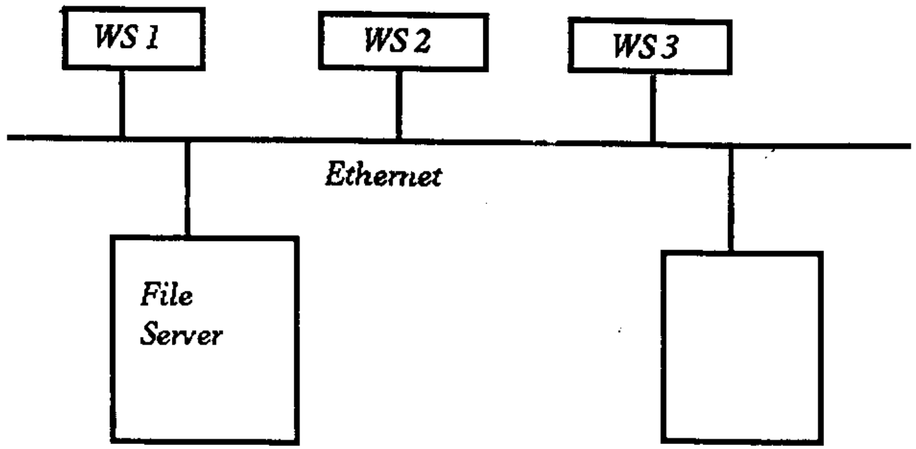
trap sequence of commands signal number

```
nzw=/tmp/temp.$$
cat >$new
trap 'rm -f $new; exit 2' 2 15
```

signal numbers:	
0	shell exit
2	interrupt
9	kill (cannot be caught)
15	terminate

Theme: The Unix Shell Slide no: 3.14

Topic: Workstations



On startup the workstation sends a boot request down the ethernet containing the requesting node's hardware address. It's server recognizes the request and downline loads the kernel image corresponding to the workstation's hardware configuration.

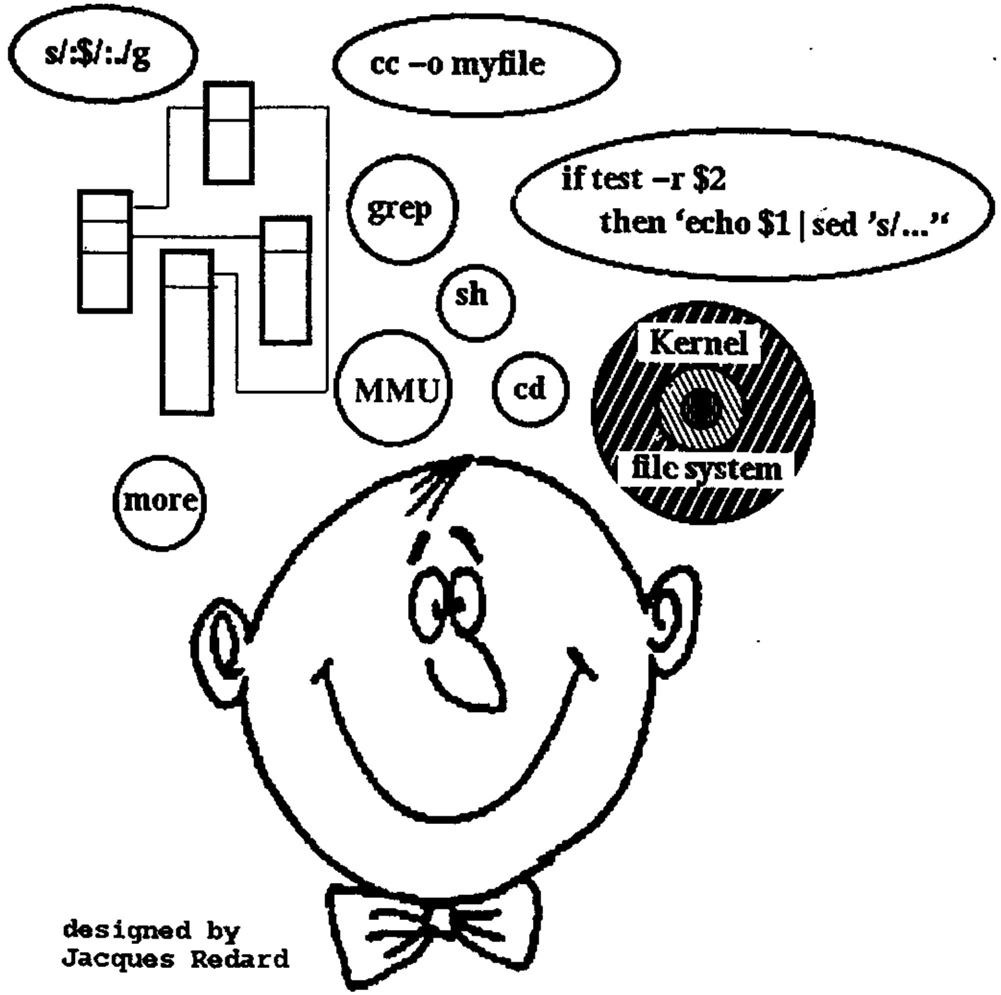
The workstation's file systems are mounted on the file server (transparent distributed system) The swap space may also be remote (diskless workstation).

The system starts up a window system (X–Windows/Motif) and allows login.

On login a terminal emulator window is brought up and allows the user to communicate with the shell.

Theme: The Unix Shell Slide no: 3.15

Topic: Good Bye

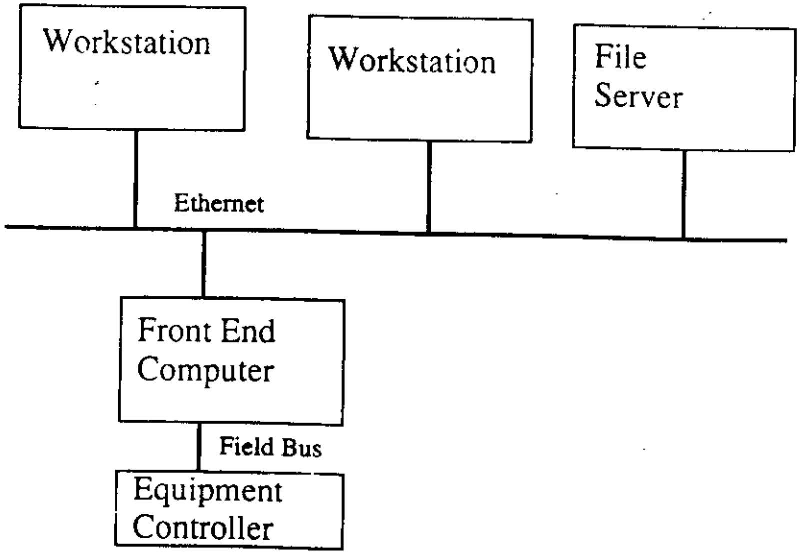


designed by
Jacques Redard

That's all folks !

LynxOS Slide 1

The architecture of the accelerator control system:



System needed in the front end computer:

- System with Unix user interface and Unix system calls
- Real time features
- Front End Computers are situated near the equipment in a harsh environment -> diskless system
- Bootable over the network

LynxOS Slide 2

Advantage of Unix: Operating Systems running on several hardware platforms. But ... several Unix variants make protability more difficult.

POSIX: defines standards:

- POSIX.1:
  — defines the interface between portable applications and the operating system, based on historical UNIX system models. Consists of a library of functions frequently implemented as system calls
- POSIX.2
  — specifies a shell command language based on the System V shell with some features from the C Shell and the Korn Shell.
- POSIX.3
  — provides detailed testing and verification requirements for the POSIX family
- POSIX.4
  — is a set of real-time extensions to POSIX.1. The standard contains:
  - * Binary semaphores
  - * Process memory locking
  - * Memory mapped files and shared memory
  - * Real-tim signal extensions
  - * Clocks and timers

What is Real Time Performance?

An application is real time if it must generate a response to an external event within a bounded time interval in order to function correctly

LynxOs compared to Unix:

- The internal coding of the kernel is completely different
- User shell is similar to Unix shell (anyway many different shells are used under Unix)
- Uses mainly GNU compiler / debugger tools
- Has all Unix system calls (and many more)
- Different (Real Time) Scheduler
- Threads
- Different types of semaphores
- Real task priorities
- Allows interrupt handling through device drivers
- Allows to selectively enable and disable paging
- Access to physical memory may be granted

Parameters influencing the Real Time Performance

Task response time

Driver response time

Interrupt latency

Critital Section Completion Time

Task Completion time

Interrupt dispatch time

Task switch time

- **Task response time:**

  The time it takes the application to be notified of the interrupt occurence
- **Driver response time:**

  The time it takes the device driver to be notified of the occurence of the interrupt.
- **Interrupt latency**

  The time interrupt acknowledgement is disabled due to a critical kernel operation or an interrupt service already in progress.
- **Interrupt dispatch time**

  The time it takes the hardware to acknowledge the interrupt and the operating systrem to dispatch to the appropriate driver.
- **Task completion time**

  The time it takes the application to finish its time critical operations.
- **Task switch time**

  Time it takes to schedule and perfom a context switch to the highest priority task.

LynxOS Slide 5

Task completion time

Task execution time

Interrupt execution time

Priority inversion time

- **Task execution time:**

  The actual amount of CPU time needed by the task to carry out its functions
- **Interrupt execution time:**

  The time the task is suspended because the system is servicing interrupts
- **Priority inversion time:**

  The time the highest priority task is blocked waiting for a resource held by a lower priority task to be freed.

LynxOS Slide 6

Real time Scheduling

3 types of scheduling:

- **Fixed Priority**

  —Only the highest priority runnable tasks will be scheduled

  —Priorities are only changed through explicit directives

  —minimizes priority inversion time
- **Fifo**

  —A task runs until it completes, blocks, voluntarily yields the processor or is preemptied by a higher priority task
- **Round robin**

  —Same as Fifo, but task may be preempted by another task of same priority
- **Priority inheritence:**

  —A low priority task holding a resource needed by a high priority task will have its priority boosted temporarily to the high priority until it frees the resource.

Example:

- Data aquisition and critical control -> high priority Fifo
- Status display update and user interface -> medium priority round robin
- printer log -> low priority

There are calls to change the type of scheduling and the task priority:

get/setprio get/setscheduler yield (forces the process to release the CPU)

LynxOS Slide 7

Signals and Events

Signals work the same way as in Unix however ...

Events are extensions to signals: (numbers 32-64).

Events are queued, so cannot be lost. They also transfer a data word.

Definition of a set of events:

```
#include <signal.h>
sig_set_T      set;
/* create an empty set of signals */
sigemptyset(&set)
/* now fill the set */
sigaddset(&set,SIGFPE);
```

Definition of an Event Handler:

```
#include <signal.h>
#include <events.h>
void handler(Int signum, int data);
struct sigaction act, old_act;
act.sa_handler = handler;
act.sa_mask = mask_while _handling;      /* a sig_set */
int success;
success = sigaction(signum, &act, &old_act);
where:        handler = SIG_DFL:         default handler
              handler = SIG_IGN:         ignore
              else address of a handler routine
```

LynxOS Slide 8

Shared memory segments

Creation of a shared memory segment:

```
#include <sys/shmmap.h>
mkshm("myshared",0666 | SHM_PERSIST,4096)
arguments:      filename
                access rights
                size
```

Opening of the shared memory segment:

```
fd = open("myshared",O_RDWR,0)  /* 0: mode only used with creat */
```

Attaching to a shared memory segment

```
address_of_sm = shmmap(fd,NULL,length,offset,flags)
flags: SHM_READ,SHM_WRITE,SHM_EXEC
```

Detaching from the segment

```
shmunmap(address_of_sm,length);
```

Closing the segment:

```
close(fd)
```

The use of the shared memory ressembles very much normal file access

LynxOS Slide 9

Accessing physical addresses:

Each process has its own memory segments and can only access memory outside its private area if it attaches to a shared memory segment.

How can we access registers of an I/O device e.g. an ADC or I/O register ?

#include <smem.h>

char * smem_create(name, phys_address, size, perm)

if the shared memory segment "name" does not exist, it is created and the base address is returned to the caller. Otherwise the address of the existing segment is returned.

The segment is not owned by the caller, it is valid for any process in the system.

To get rid of it:

smem_remove(name)

LynxOS Slide 10

Semaphores

There are 2 types of semaphores:

- Binary semaphore
- Counting semaphore

The mechanism is similar to the shared memory concept:

Open a semaphore

fd = open ("mysem",O_RDWR,0)

Wait for the semaphore

semwait(fd) waits until semaphore is freed

semifwait(fd) returns with error if semaphore is blocked

Release a semaphore

sempost(fd)

semifpost(fd) if no process waits for the semaphore: error

Threads

Advantages of threads over processes

All threads have a common memory zone and therefore all data are accessible to all threads. It is however also possible to create a small amount of private data.

Creation of a thread:

Create attributes

```
#include <pthread.h>
pthread_attr_t attr;
```

pthread_attr_create(&attr);

Now the attributes for the thread can be set:

pthread_attr_set/getstacksize

pthread_attr_set/getsched

pthread_attr_set/getprio

Once all attributes are setup we create the thread:

pthread_create(tidp,attr,routine,arg)

where:
- tidp: thread identifier tid
- attr: attributes setup before
- routine: address of the code to be executed by this thread
- arg: arguments passed to the thread.

pthread_exit(status)

finished the thread

Treatment of interrupts:

Interrupts must be treated in supervisor (system) mode.

From user point of view: 2 solutions:

- read system call — waits until a blocking read is finished
- select system call — waits until input is received on a fd

Both use **device drivers.**

The kernel communicates with device drivers through the entry points:

- XYZinstall(): installs a new major device
- XYZuninstall(): removes a majpr device
- XYZioctl(): control operations and status information
- XYZselect(): needed for select system call
- XYZread(): reads data from the device
- XYZwrite() writes data to the device

The device_info_structure passed to the install routine contains all essential information of the device (physical address, interrupt vector ...)

LynxOS Slide 13

The driver has no access to system calls. However a number of calls are provided for interprocess communication and debugging:

Semaphores: swait, ssignal

System threads: Needed when an interrupt treatment would take too much time

cprintf and kkprintf for debugging