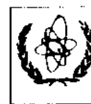


INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

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/474 - 8

COLLEGE ON
"THE DESIGN OF REAL-TIME CONTROL SYSTEMS"
1 - 26 October

UNIX

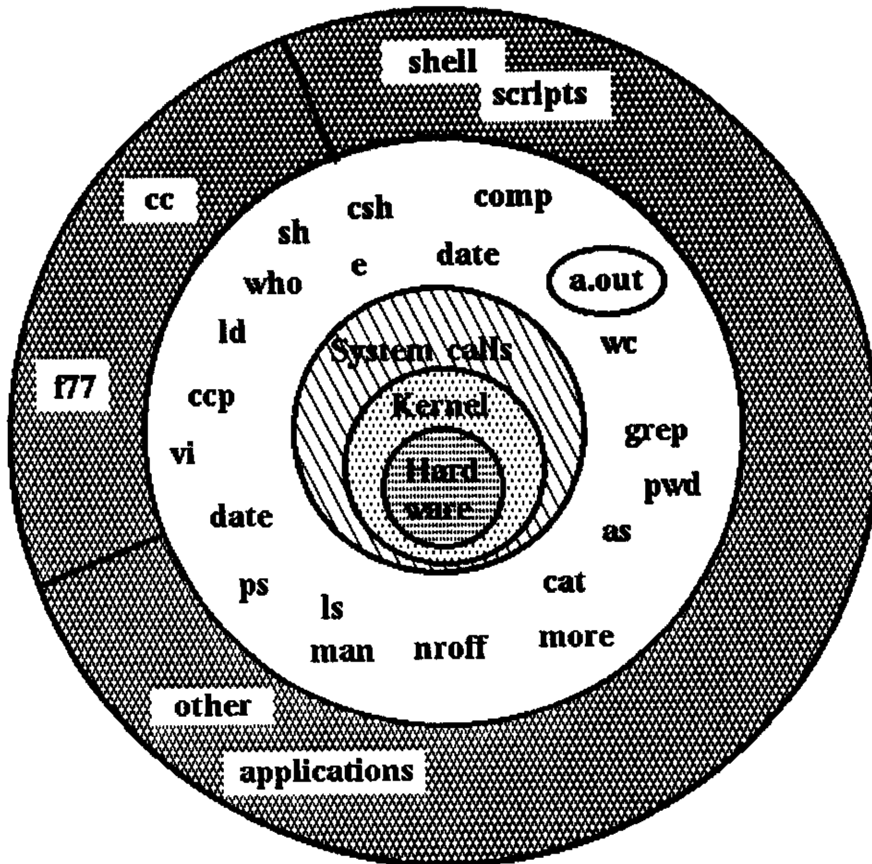
U. RAICH
C.E.R.N.
E.P. Division
CH-1211 Geneva 23
Switzerland

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

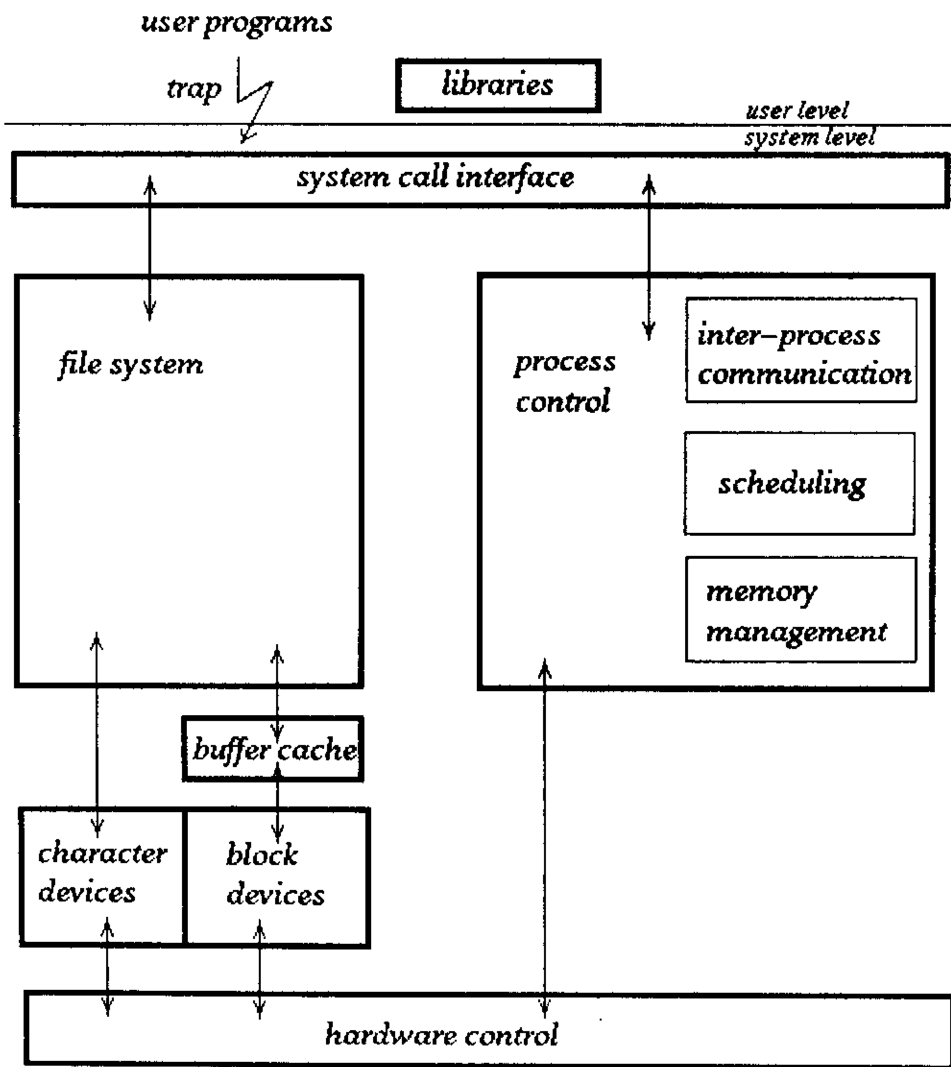
Topic: Advantages of Unix

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



Theme: The Unix Kernel (Internals) Slide no: 1.4

Topic: The Unix File System

Steps to be executed when reading/writing a file:

Opening the file

generate entries into tables

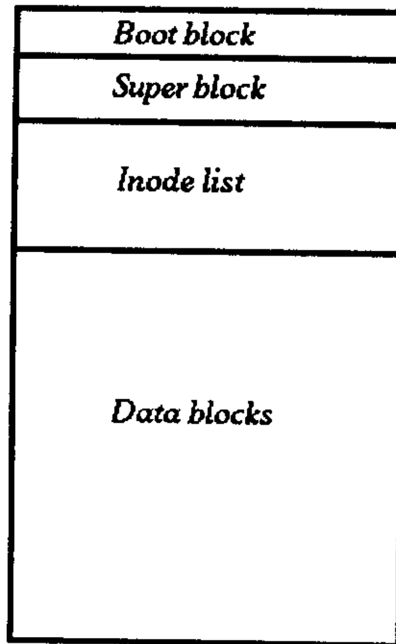
- *allowing a process to reference the file*
- *and allowing the kernel to know which file in the system is open for read, write or both*
- *convert the filename into a more easily accessible structure describing the file (inodes)*
- *allocate new inodes*
- *allocate data blocks on disk*

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*

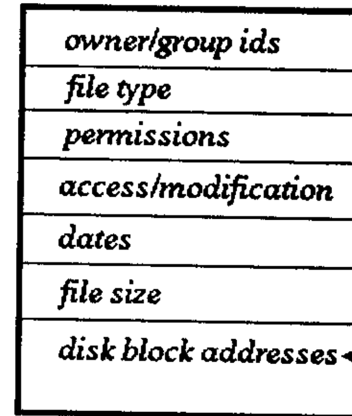
Theme: The Unix Kernel (Internals) Slide no: 1.6

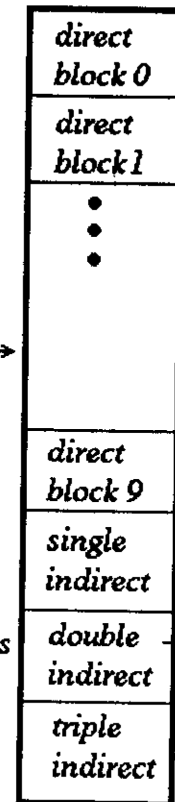
Topic: The Unix File System

Reading/writing a Unix file

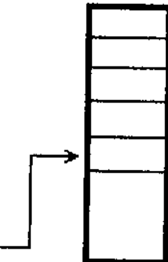
The inode structure:

data blocks





256 block numbers

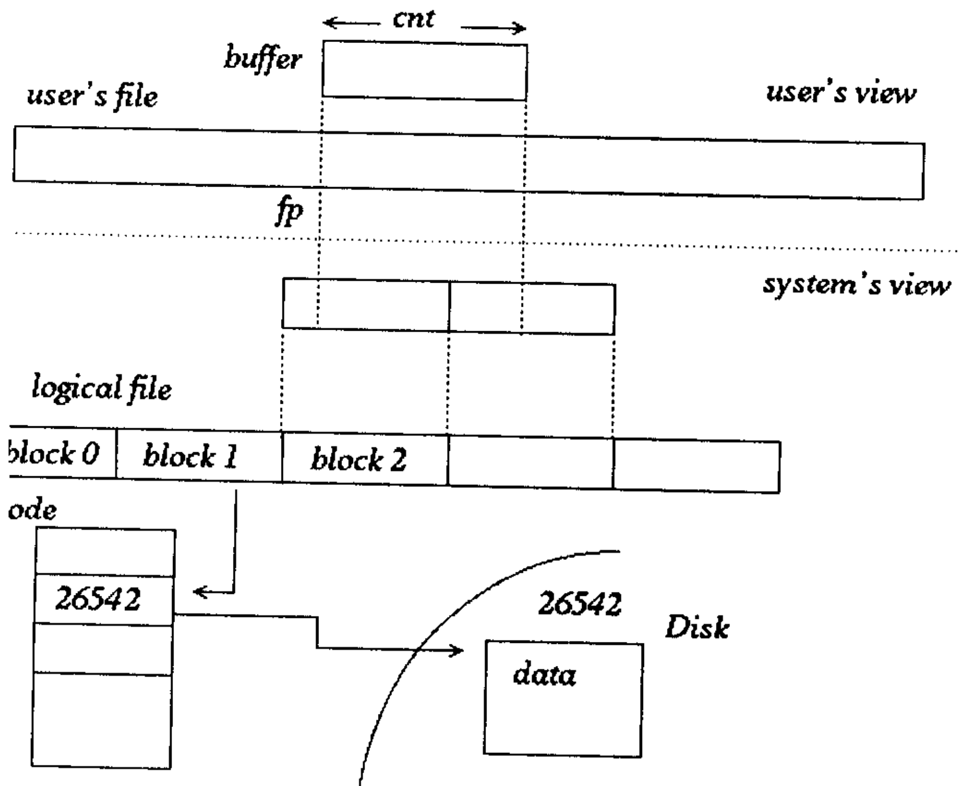


max file sizes:

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

if 1 block 1024 bytes

Writing or reading a Unix file

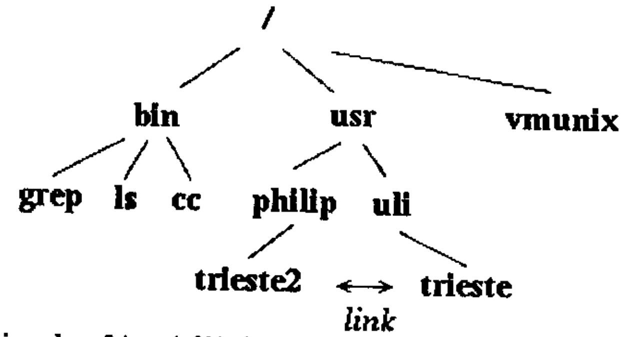


example:

write buffer of size 1850
starting from file pointer at 2740

Converting a filename to an inode

The layout of a directory file



example: get inode of /usr/uli/trieste

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

118	.
17	..
204	uli
23	philip

204	.
118	..
193	trieste

23	.
118	..
193	trieste2

absolute reference:

/ is known to the system in a global variable

relative reference:

The current directory can be found in the process descriptor

The Super Block

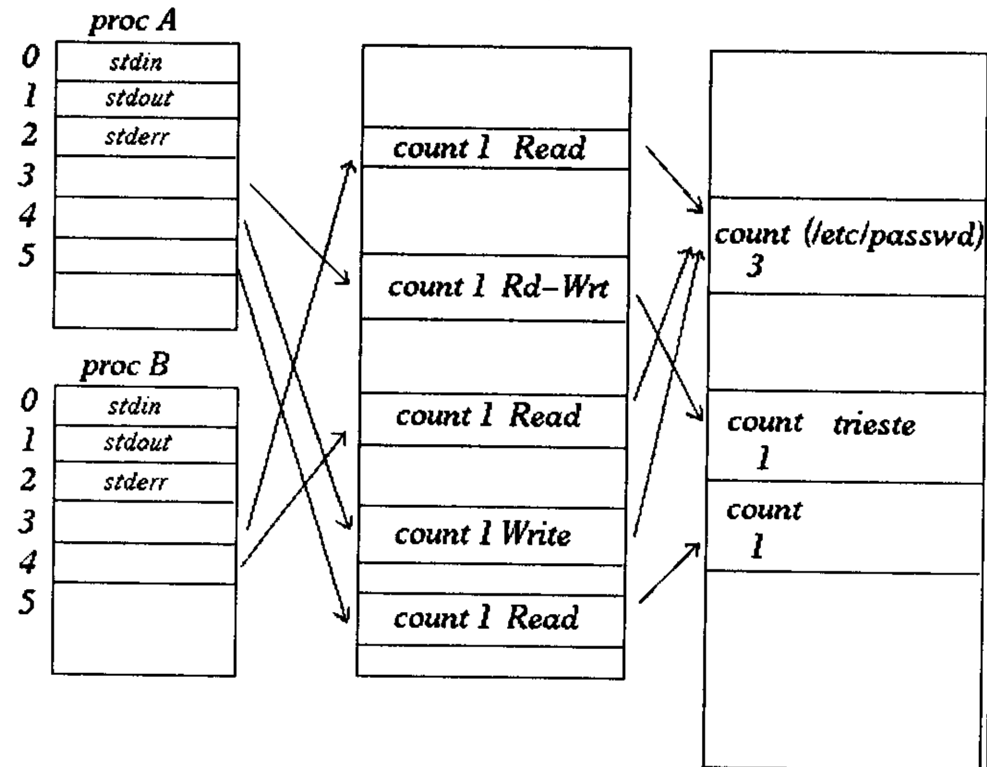
Allocating inodes (when creating a new file)

Super block fields	Purpose
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*

Descriptor tables of "open" files

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

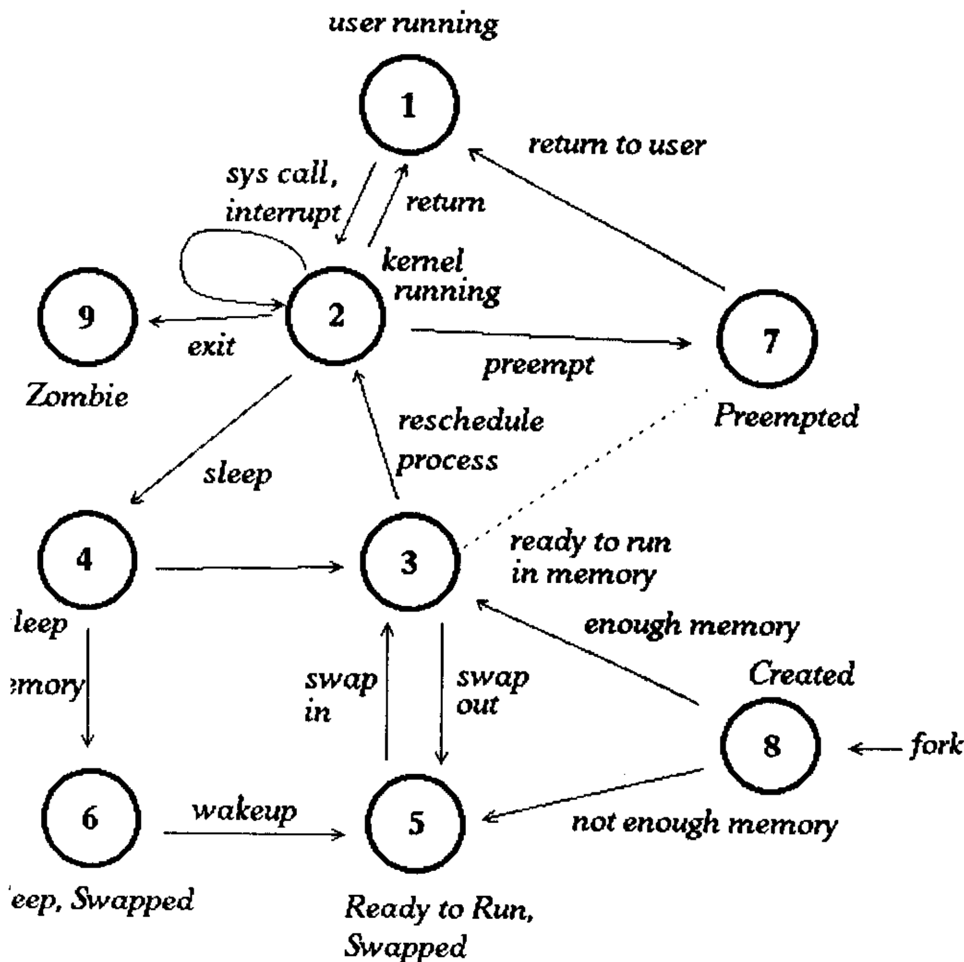


fd is the index into the user file descriptor table

heme: The Unix Kernel (Internals) Slide no: 1.11

opic: Process Management

Process State Diagram

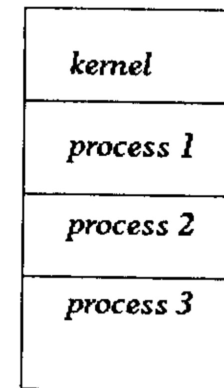


Theme: The Unix Kernel (Internals) Slide no: 1.12

Topic: Process Management

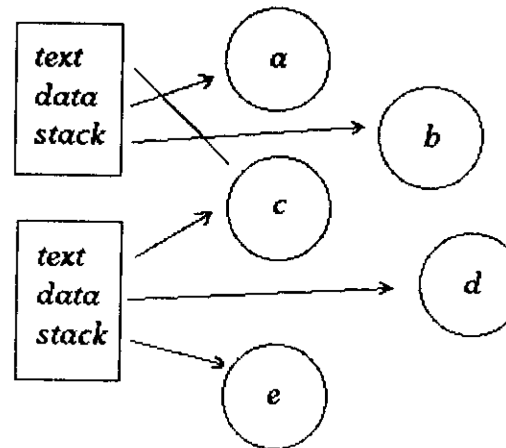
Layout of system memory

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**



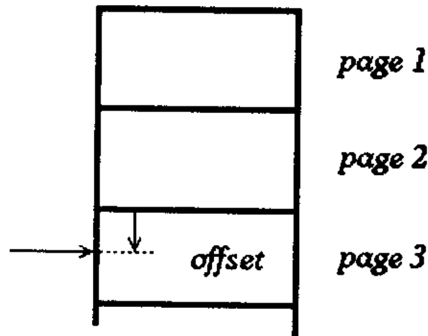
Theme: The Unix Kernel (Internals) Slide no: 1.13

Topic: Process Management

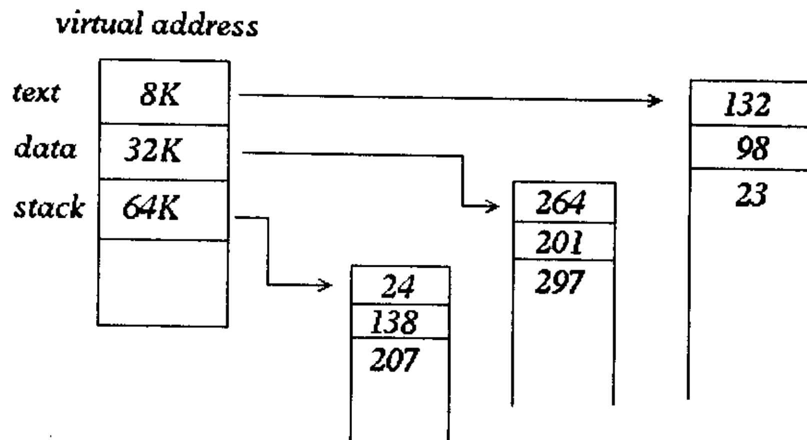
The paging system

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

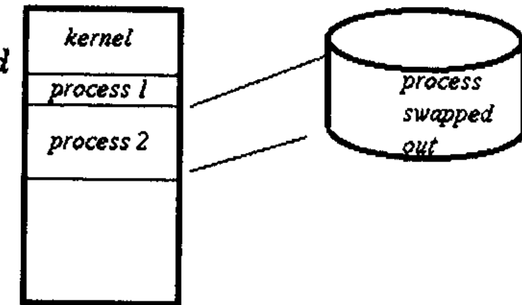
Topic: Process Management

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*

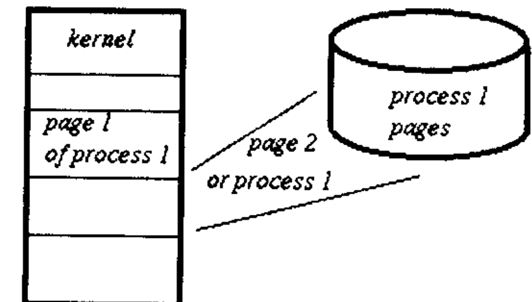


- **Demand Paging**

  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.



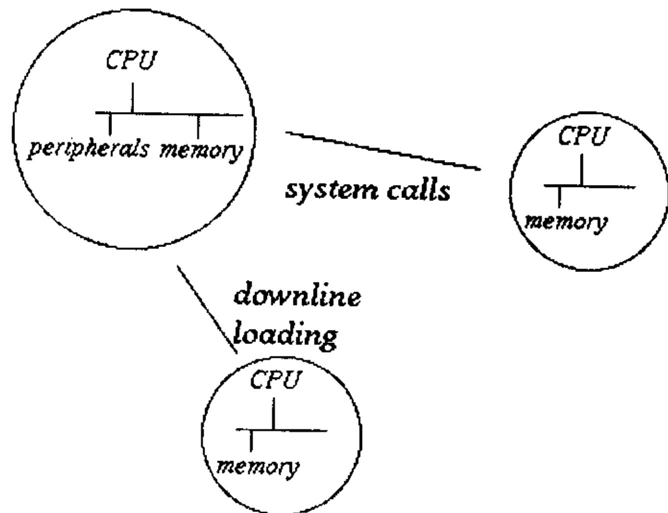
- **Hybrid systems**

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

- **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*



The Newcastle connection

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

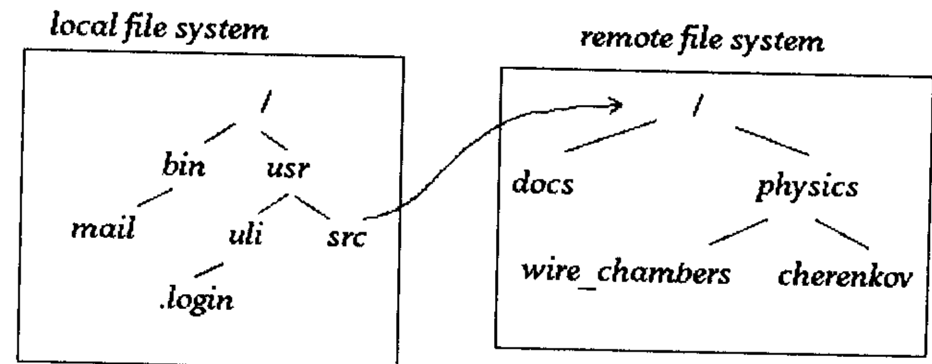
trieste//usr/uli/course

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

Needs special C library in order to parse file names

Transparent 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

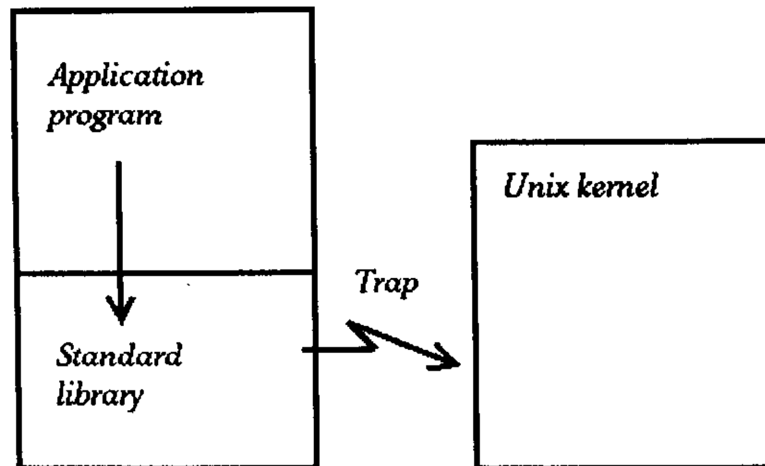
Generalities on system calls

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.

Theme: **The Unix System Calls** Slide no: **2.4**

Topic: **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 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

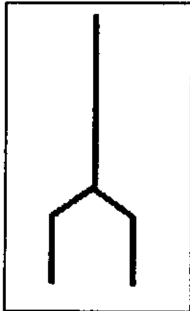
Theme: The Unix System Calls **Slide no:** 2.5

Topic: Process Control

Process Creation

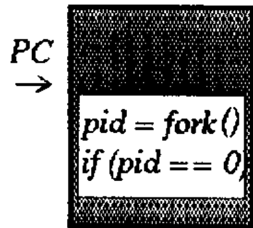
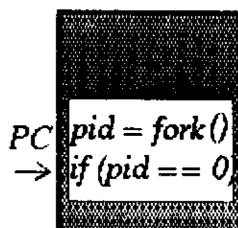
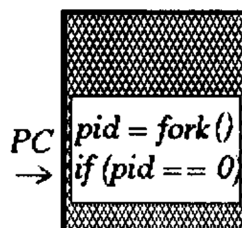
All new processes are created through a **fork** *system call*

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"

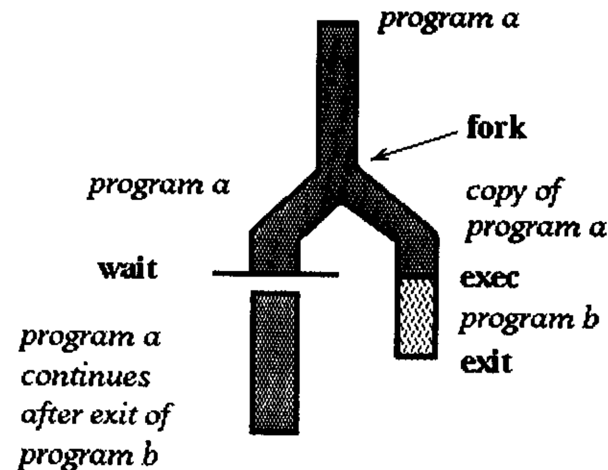
The "exec" family of system calls

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:

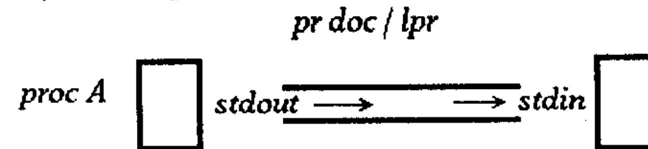
kill(pid,SIGUSR1) — *since pid is needed signals can only be sent to parent or offspring*

(getppid returns pid of parent)

Theme: **The Unix System Calls** Slide no: **2.8**

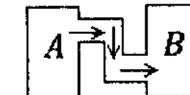
Topic: **Pipes**

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: **2.10**

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
```


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);
}
```

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_RNMID,&msq_stat)

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 objecy from the system

heme: The Unix Shell Slide no: 3.1

opic: 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

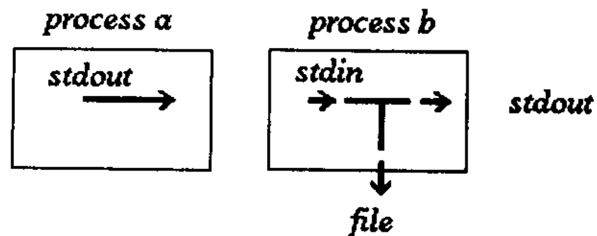
Stdout of one program can be connected to stdin of another one through a pipe
Example: We want to know the number of login processes on our system . This can be found by counting the number of lines output by **who**

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

Theme: The Unix Shell — Slide no: 3.3

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
$ _
```


Theme: The Unix Shell — Slide no: 3.4

Topic: Shell Scripts

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
$ _
```

Theme: **The Unix Shell** Slide no: **3.5**

Topic: **Shell Scripts**

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
```


Theme: **The Unix Shell** Slide no: **3.10**

Topic: **Flow of Control**

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
$ _
```

Theme: **The Unix Shell** Slide no: **3.11**

Topic: **Flow of Control**

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
```


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

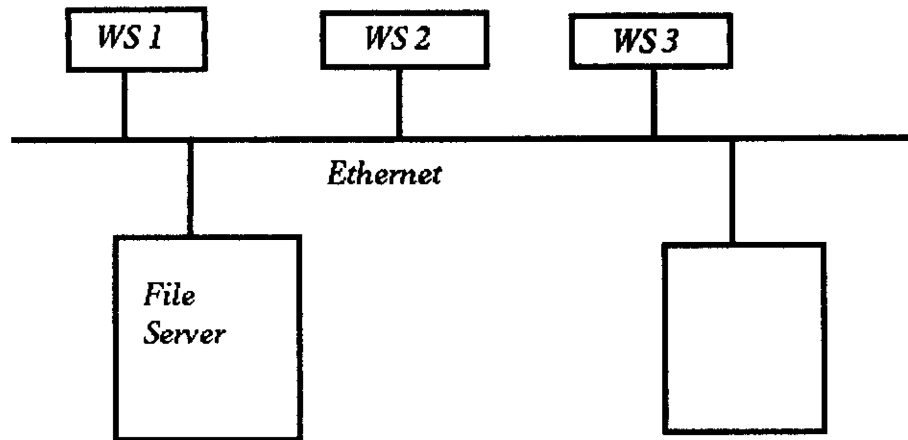
trap sequence of commands signal number

```
new=/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.

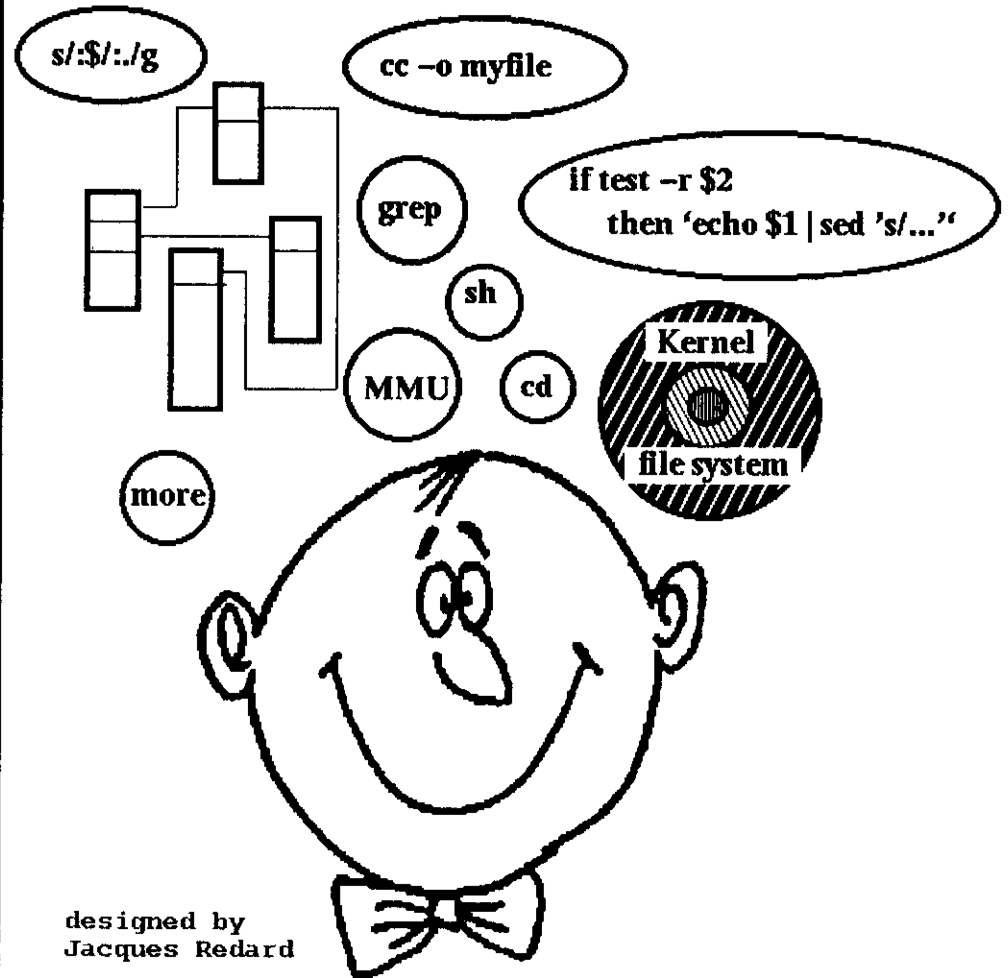
The workstation's file systems are mounted on the file server (transparent distributed system) The swap space may also be remote (diskless workstation).

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**



That's all folks !