Process Management

Outline

- Main concepts
- Basic services for process management (Linux based)
- Inter process communications: Linux Signals and synchronization
- Internal process management
  - Basic data structures: PCB. list/queues of PCB's
  - Basic internal mechanism. Process context switch.
  - Process scheduling
- Effect of system calls in kernel data structures

Process definition
Concurrency
Process status
Process attributes

PROCESSES

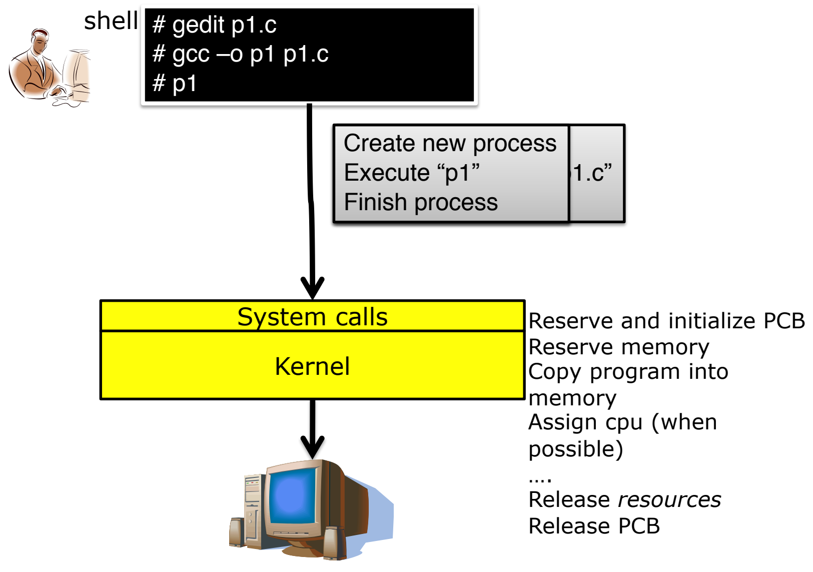
Program vs. Process

- Definition: a process is the O.S. representation of a program during its execution
- Why?
  - A user program is something static: it is just a sequence of bytes stored in a "disk"
  - When user asks the O.S. for executing a program, in a multiprogrammed-multiuser environment, the system needs to represent that particular "execution of X" with some attributes
    - Which regions of physical memory is using
    - Which files is accessing
    - Which user is executing it
    - What time it is started
    - How many CPU time it has consumed
    - …

Processes

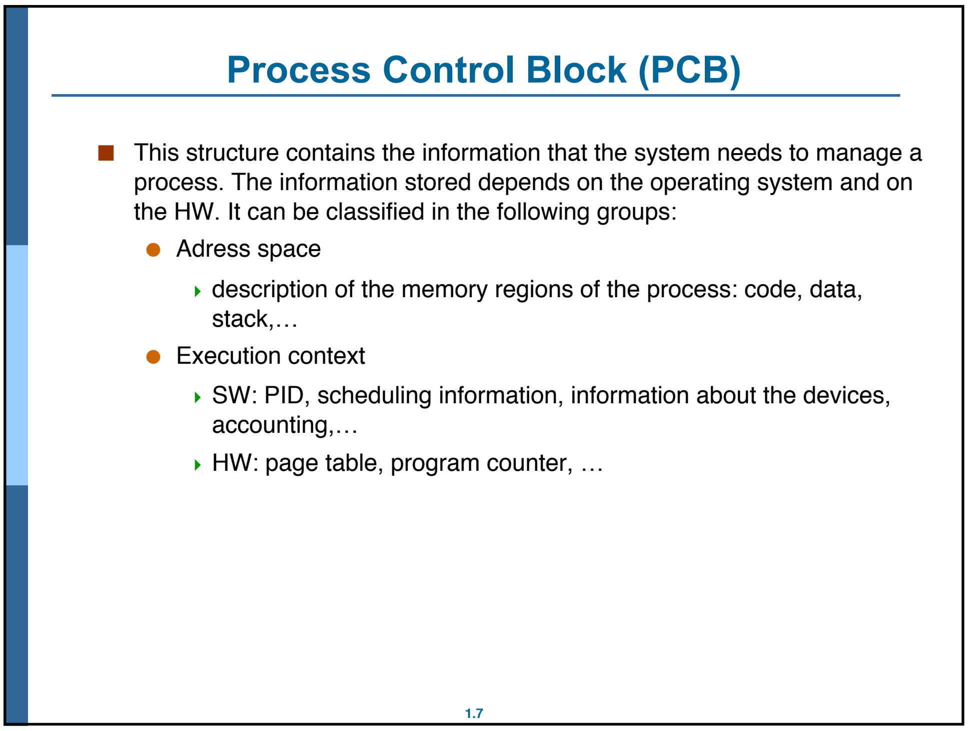
- Assuming a general purpose system, when there are many users executing… each time a user starts a program execution, a new (unique) process is created
  - The kernel assigns resources to it: physical memory, some slot of CPU time and allows file access
  - The kernel reserves and initializes a new process data structure with dynamic information (the number of total processes is limited)
    - Each O.S. uses a name for that data structure, in general, we will refer to it as **PCB** (Process Control Block).
    - Each new process has a unique identifier (in Linux its a number). It is called **PID** (Process Identifier)

How it's made? It is always the same…



```
# gedit p1.c
# gcc –o p1 p1.c
# p1
```

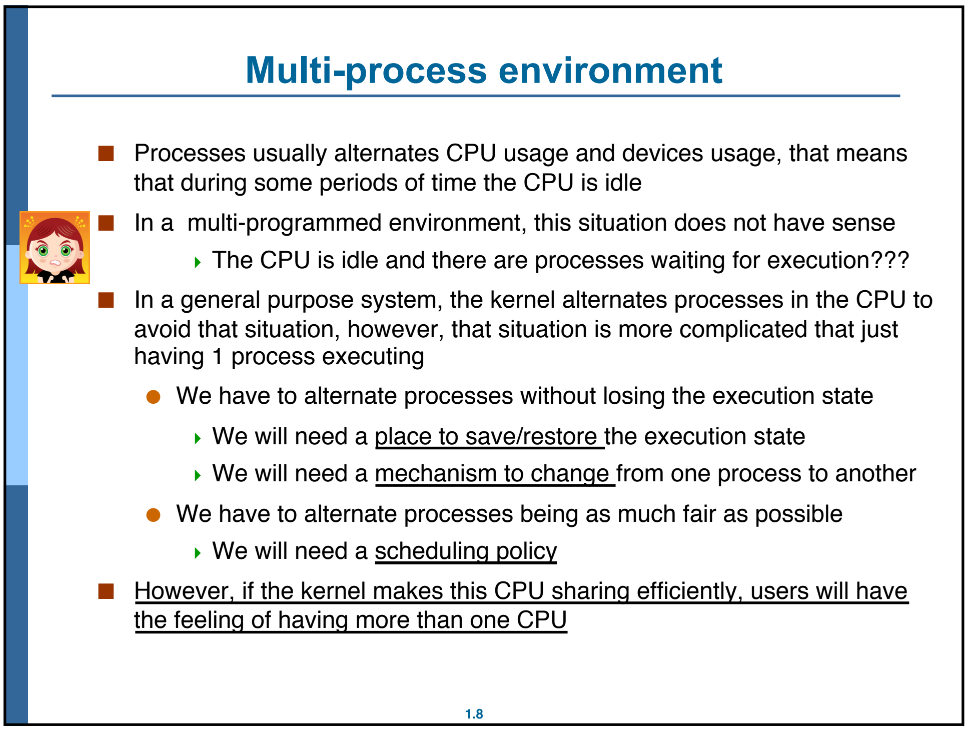
Reserve and initialize PCB
Reserve memory
Copy program into memory
Assign cpu (when possible)
….
Release *resources*
Release PCB



Process Control Block (PCB)

- This structure contains the information that the system needs to manage a process. The information stored depends on the operating system and on the HW. It can be classified in the following groups:
  - Adress space
    - description of the memory regions of the process: code, data, stack,…
  - Execution context
    - SW: PID, scheduling information, information about the devices, accounting,…
    - HW: page table, program counter, …

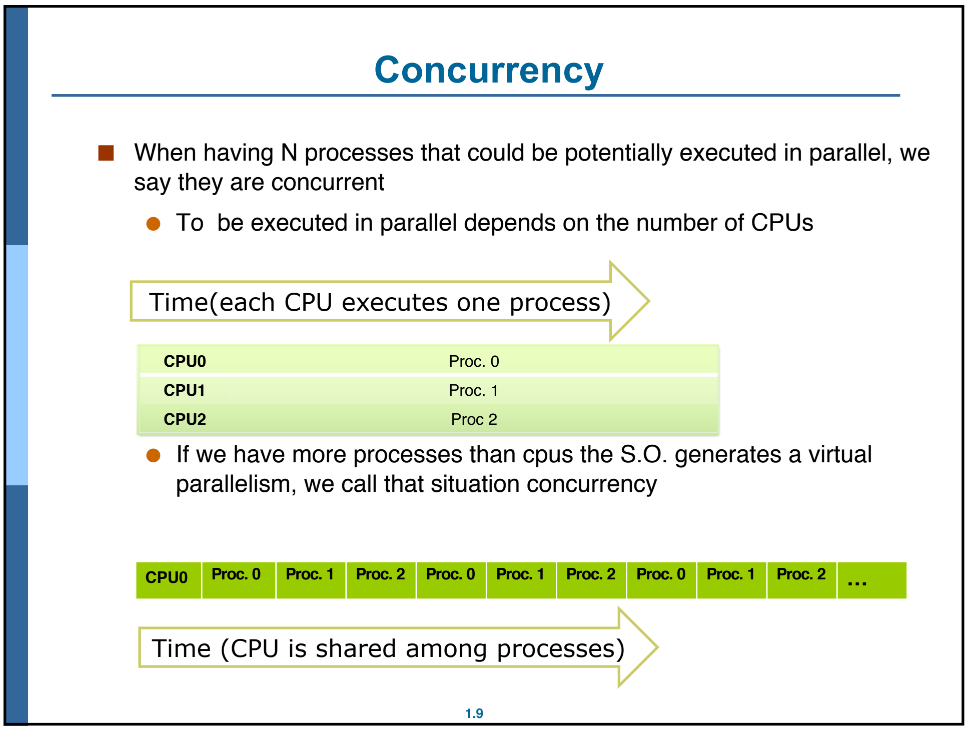
1.7



Multi-process environment

- Processes usually alternates CPU usage and devices usage, that means that during some periods of time the CPU is idle
- In a multi-programmed environment, this situation does not have sense
    - The CPU is idle and there are processes waiting for execution???
- In a general purpose system, the kernel alternates processes in the CPU to avoid that situation, however, that situation is more complicated that just having 1 process executing
  - We have to alternate processes without losing the execution state
    - We will need a place to save/restore the execution state
    - We will need a mechanism to change from one process to another
  - We have to alternate processes being as much fair as possible
    - We will need a scheduling policy
- However, if the kernel makes this CPU sharing efficiently, users will have the feeling of having more than one CPU

1.8



Concurrency

- When having N processes that could be potentially executed in parallel, we say they are concurrent
  - To be executed in parallel depends on the number of CPUs

Time(each CPU executes one process)

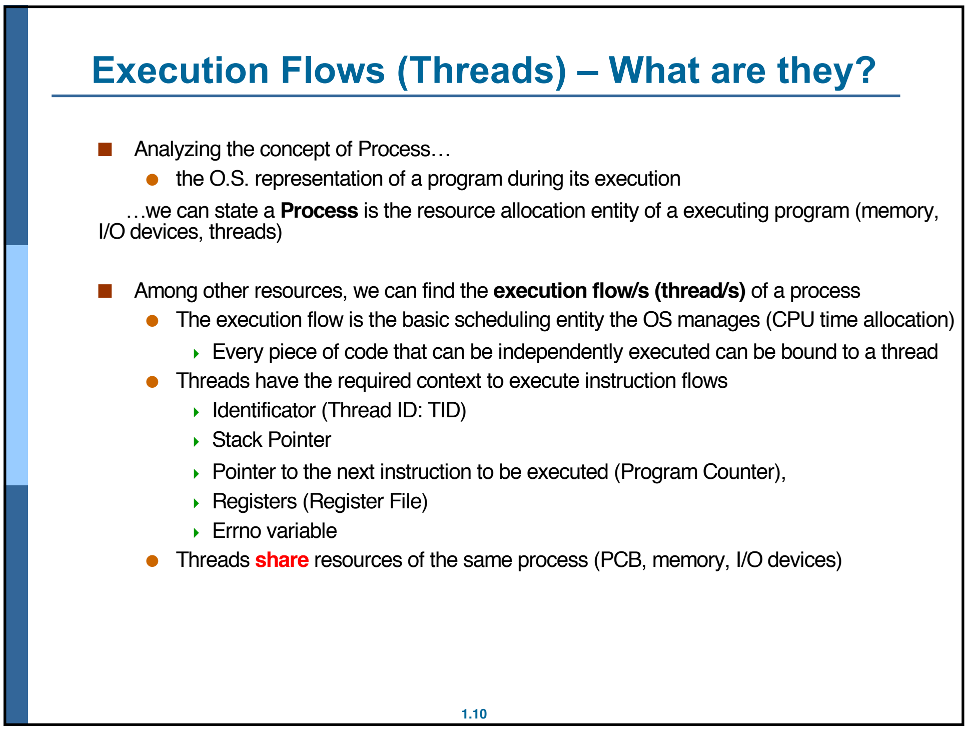
CPU0	Proc. 0
CPU1	Proc. 1
CPU2	Proc 2

  - If we have more processes than cpus the S.O. generates a virtual parallelism, we call that situation concurrency

CPU0	**Proc. 0**	**Proc. 1**	**Proc. 2**	**Proc. 0**	**Proc. 1**	**Proc. 2**	**Proc. 0**	**Proc. 1**	**Proc. 2**	**…**

Time (CPU is shared among processes)

1.9



Execution Flows (Threads) – What are they?

- Analyzing the concept of Process…
  - the O.S. representation of a program during its execution

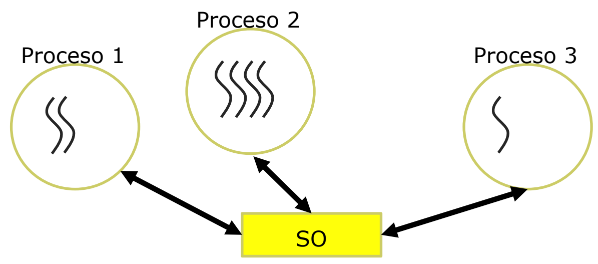
…we can state a **Process** is the resource allocation entity of a executing program (memory, I/O devices, threads)

- Among other resources, we can find the **execution flow/s (thread/s)** of a process
  - The execution flow is the basic scheduling entity the OS manages (CPU time allocation)
    - Every piece of code that can be independently executed can be bound to a thread
  - Threads have the required context to execute instruction flows
    - Identificator (Thread ID: TID)
    - Stack Pointer
    - Pointer to the next instruction to be executed (Program Counter),
    - Registers (Register File)
    - Errno variable
  - Threads **share** resources of the same process (PCB, memory, I/O devices)

1.10

Execution Flows (Threads)

- A process has a single thread when it is launched
- A process can have several threads
  - E.g.: current high-performance videogames comprise **>50 threads**; Firefox/Chrome show **>80 threads**
- The next figure depicts: Process1 has 2 threads; Proceso2 has 4 threads; Process3 has 1 thread
- The management of multi-threaded processes depends on the OS support
  - **User Level Threads** vs **Kernel Level Threads**



Execution Flows (Threads) – Why?

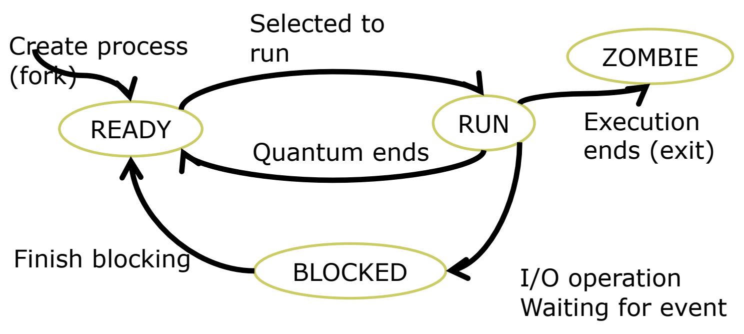
- When and what are threads used for…
  - Parallelism exploitation (code and hardware resources)
  - Task encapsulation (modular programming)
  - I/O efficiency (specific threads for I/O)
  - Service request pipelining (keep required QoS)
- Pros
  - Threads present lower cost at creation/termination and at context switching (among threads of the same process) compared to processes
  - Threads can exchange data without syscalls, since they share memory
- Cons
  - Hard to code and debug due to shared memory
    - Synchronization and mutual exclusion issues
      - Incoherent executions, wrong results, infinite blocks (stalls), etc.

Process state

- Because of concurrency, processes aren't always using CPU, but they can be doing different "things"
  - Waiting for data coming from a slow device
  - Waiting for a signal
  - Blocked for a specific period of time
- The O.S. classify processes based on what their are doing, this is called the process state
- It is internally managed like a PCB attribute or grouping processes in different lists (or queues)
- Each kernel defines a state graph

1.13

State graph example



- This is just a generic example. The diagram depends on OS internals
- OSs with multithreading support need internal structures to differentiate execution states of every particular thread of a process
  - E.g.: Light Weight Process (LWP) in Linux/UNIX based OSs

1.14

Linux: Process characteristics

- Any process attribute is stored at its PCB
- All the attributes of a process can be grouped as part of:
  - **Identity**
    - Combination of PID, user and group
    - Define resources available during process execution
  - **Environment**
    - Parameters (argv argument in main function). Defined at submission time
    - Environment variables (HOME, PATH, USERNAME, etc.). Defined before program execution.
  - **Context**
    - All the information related with current resource allocation, accumulated resource usage, etc.

1.15

Linux environment

- Parameters

```
# add 2 3
The sum is 5
```

```
void main(int argc,char *argv[])
{
    int firs_num=atoi(argv[1]);
```

  - The user writes parameters after the program name
  - The programmer access these parameters through argc, argv arguments of main function

- Environment variables:
  - Defined before execution
  - Can be get using getenv function

```
# export FIRST_NUM=2
# export SECOND_NUM=2
# add
The sum is 5
```

```
char * first=getenv("FIRST_NUM");
int first_num=atoi(first);
```

1.16

PROCESS MANAGEMENT

Main services and functionality

- System calls allow users to ask for:
  - Creating new processes
  - Changing the executable file associated with a process
  - Ending its execution
  - Waiting until a specific process ends
  - Sending events from one process to another
- In this course we **DO NOT** go into details of thread related syscalls

Basic set of system calls for process managemen[illegible]X)

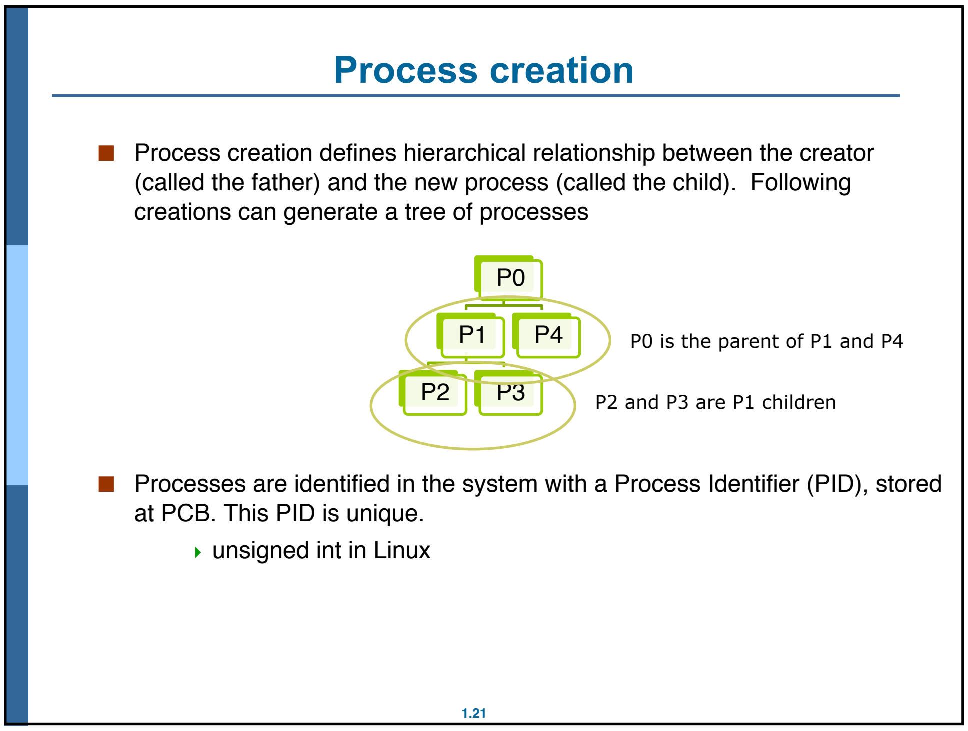
Description	Syscall
Process creation	fork
Changing executable file	exec (execlp)
End process	exit
Wait for a child process(**can block**)	wait/waitpid
PID of the calling process	getpid
Father's PID of the calling process	getppid

- When a process ask for a service that is not available, the scheduler reassigns the CPU to another ready process
  - The current process changes form RUN to BLOCK
  - The selected to run process goes from READY to RUN

Process creation

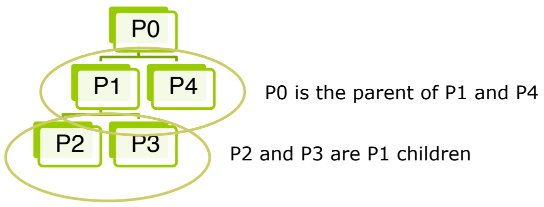
```
int fork();
```

- "Creates a new process by duplicating the calling process. The new process, referred to as the *child*, is an exact duplicate of the calling process, referred to as the *parent*, except for some points:"
  - The child has its own unique process ID
  - The child's parent process ID is the same as the parent's process ID.
  - Process resource and CPU time counters are reset to zero in the child.
  - The child's set of pending signals is initially empty
  - The child does not inherit timers from its parent (see *alarm*).
  - Some advanced items not introduced in this course



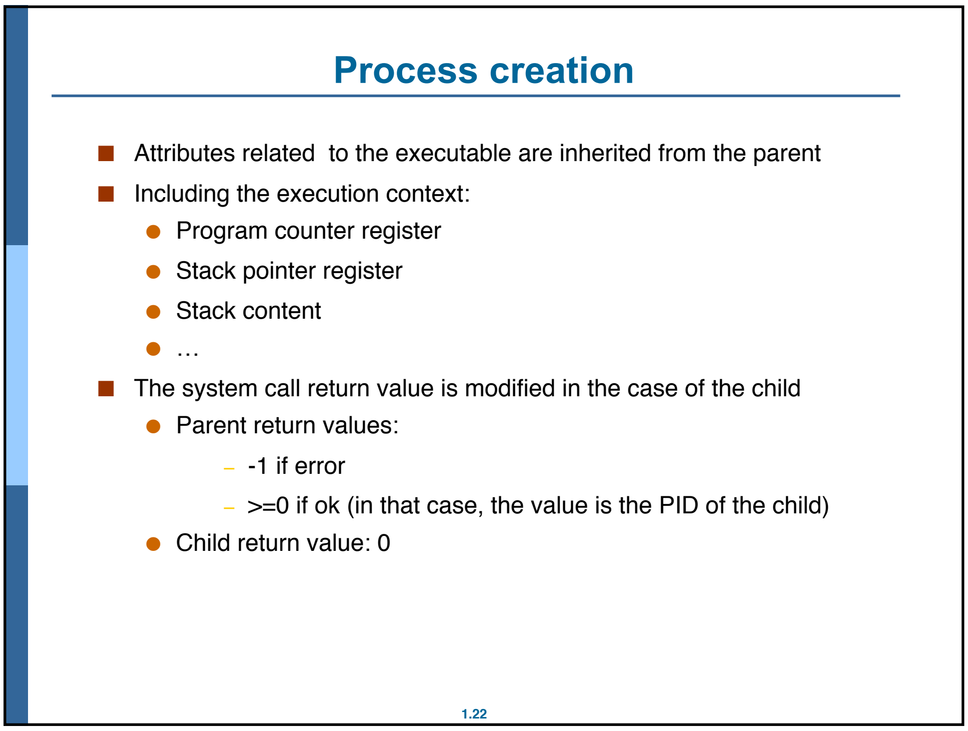
Process creation

- Process creation defines hierarchical relationship between the creator (called the father) and the new process (called the child). Following creations can generate a tree of processes



- Processes are identified in the system with a Process Identifier (PID), stored at PCB. This PID is unique.
    - unsigned int in Linux

1.21



Process creation

- Attributes related to the executable are inherited from the parent
- Including the execution context:
    - Program counter register
    - Stack pointer register
    - Stack content
    - …
- The system call return value is modified in the case of the child
    - Parent return values:
        - -1 if error
        - >=0 if ok (in that case, the value is the PID of the child)
    - Child return value: 0

1.22

Terminates process

```
void exit(int status);
```

- Exit terminates the calling process "immediately" (voluntarily)
  - It can finish involuntarily when receiving a signal
- The kernel releases all the process resources
  - Any open file descriptors belonging to the process are closed (T4)
  - Any children of the process are inherited by process 1, (*init process)*
  - The process's parent is sent a **SIGCHLD** signal.
- Value *status* is returned to the parent process as the process's exit status, and can be collected using one of the wait system calls
  - The kernel acts as intermediary
  - The process will remain in ZOMBIE state until the exit status is collected by the parent process

1.23

Terminates process

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- It suspends execution of the calling process until one of its children terminates
- *Pid* can be:
  - waitpid(-1,NULL,0) →Wait for any child process
  - waitpid(pid_child,NULL,0)→ wait for a child process with PID=pid_child
- *If status* is not *null, the kernel* stores status information in the *int* to which it points.
- *Options* can be:
  - 0 means the calling process will block
  - WNOHANG means the calling process will return immediately if no child process has exited

1.24

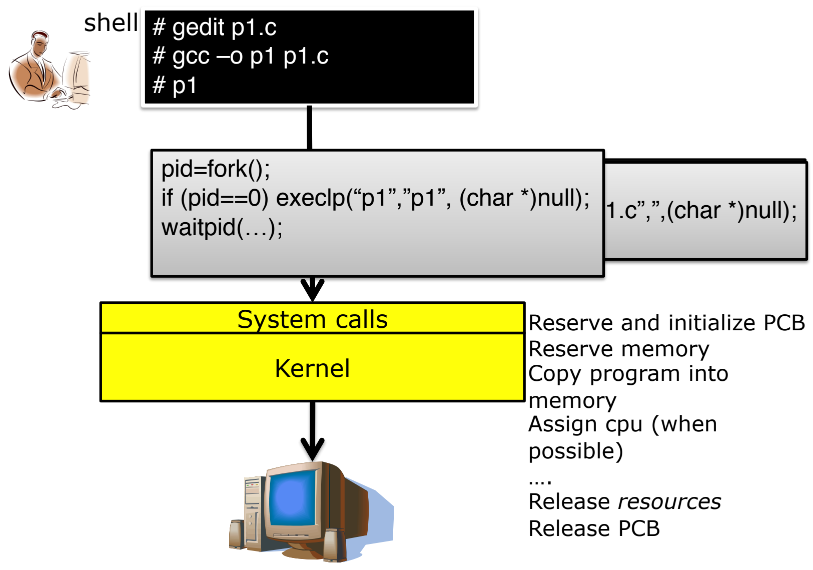
Changing the executable file

```
int execlp(const char *exec_file, const char *arg0, ...);
```

- It replaces the current process image with a new image
  - Same process, different image
  - The process image is the executable the process is running
- When creating a new process, the process image of the child process is a duplicated of the parent. Executing an execlp syscall is the only way to change it.
- Steps the kernel must do:
  - Release the memory currently in use
  - Load (from disc) the new executable file in the new reserved memory area
  - PC and SP registers are initialized to the main function call
- Since the process is the same….
  - Resource usage account information is not modified (resources are consumed by the process)
  - The signal actions table is reset

SOME EXAMPLES

How it works…



```
# gedit p1.c
# gcc –o p1 p1.c
# p1
```

```
pid=fork();
if (pid==0) execlp(“p1”,”p1”, (char *)null);
waitpid(…);
```

```
1.c”,”,(char *)null);
```

Reserve and initialize PCB
Reserve memory
Copy program into memory
Assign cpu (when possible)
….
Release *resources*
Release PCB

1.27

Process management

Ex 1: We want parent and child execute different code lines

```
int ret=fork();
if (ret==0) {
   // These code lines are executed by the child, we have 2
processes
}else if (ret<0){
    // The fork has failed. We have 1 process
}else{
  // These code lines are executed by the parent, we have 2
processes
}
// These code lines are executed by the two processes
```

Ex 2: Both processes execute the same code lines

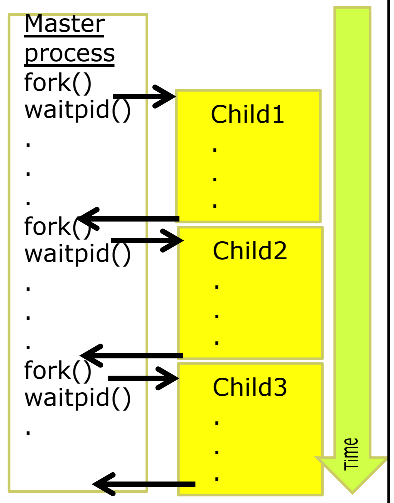
```
fork();
// We have 2 processes (if fork doesn’t fail)
```

1.28

Sequential scheme

The code imposes that only one process is executed at each time

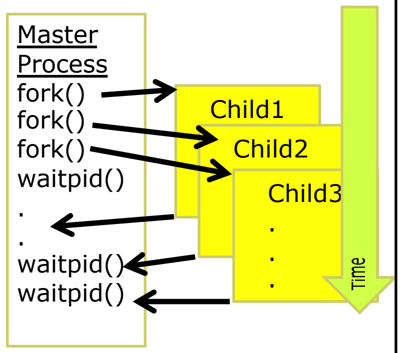
```
#define num_procs 2
int i,ret;
for(i=0;i<num_procs;i++){
   if ((ret=fork())<0) control_error();
   if (ret==0) {
      // CHILD
     ChildCode();
     exit(0); //
   }
   waitpid(-1,NULL,0);
}
```



Concurrent scheme

This is the default behavior. All the process are execute “at the same time”

```
#define num_procs 2
int ret,i;
for(i=0;i<num_procs;i++){
   if ((ret=fork())<0) control_error();
   if (ret==0) {
      // CHOLD
     ChildCode();
     exit(0); //
   }
}
while( waitpid(-1,NULL,0)>0);
```



Examples

- Analyze this code

```
int ret;
char buffer[128];
ret=fork();
sprintf(buffer,”fork returns %d\n”,ret);
write(1,buffer,strlen(buffer));
```

- Output if fork success?
- Output if fork fails?
- Try it!! (it is difficult to try the second case)

Examples

- How many processes are created with these code?

```
...
fork();
fork();
fork();
```

- And this one?

```
...
for (i = 0; i < 10; i++)
    fork();
```

- Try to generate the process hierarchy

Examples

- If parent PID is 10 and child PID is 11…

```
int id1, id2, ret;
char buffer[128];
id1 = getpid();
ret = fork();
id2 = getpid();
srintf(buffer,"id1 value: %d; ret value: %d; id2 value: %d\n", id1, ret,
id2);
write(1,buffer,strlen(buffer));
```

- Which messages will be written in the standard output?
- And now?

```
int id1,ret;
char buffer[128];
id1 = getpid();
ret = fork();
id1 = getpid();
srintf(buffer,"id1 value: %d; ret value %d", id1, ret);
write(1,buffer,strlen(buffer));
```

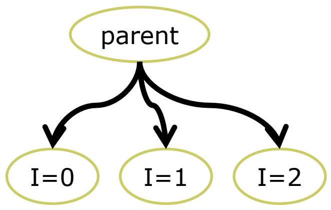
Example (this is a real exam question)

```
void main()
{    char buffer[128];
     ...
     sprintf(buffer,"My PID is %d\n", getpid());
     write(1,buffer,strlen(buffer));
     for (i = 0; i < 3; i++) {
          ret = fork();
          if (ret == 0)
               Dork();
     }
     while (1);
}

void Work()
{    char buffer[128];
     sprintf("My PIDies %d an my parent PID %d\n", getpid(), getppid());
     write(1,buffer,strlen(buffer));
     exit(0);     <------ Now, remove this code line
}
```

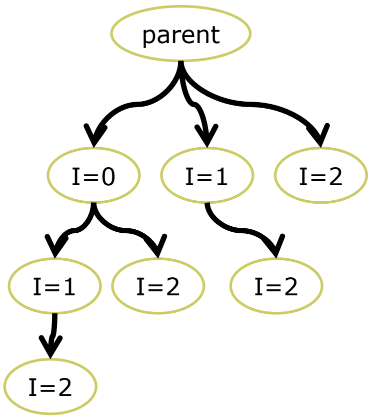
http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EjemplosProcesos.tar.gz

Process hierarchy

With exit system call

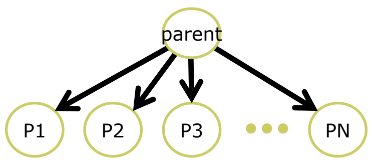


Without exit

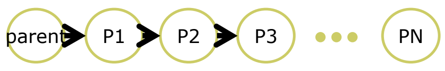


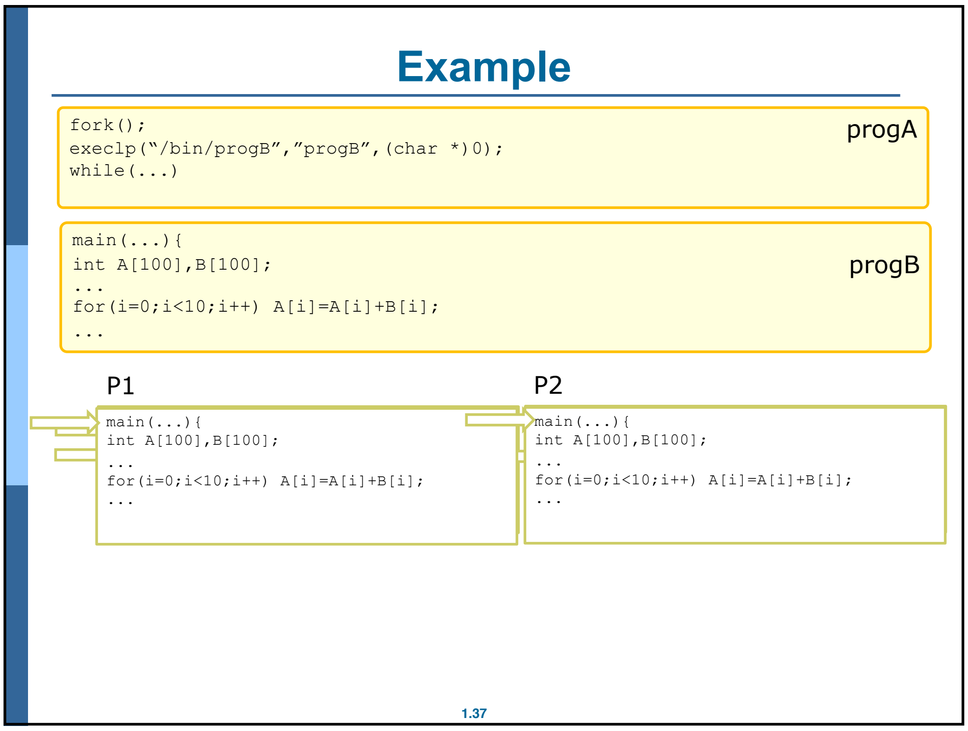
More examples

- Write a C program that creates this process scheme:



- Modify the code to generate this new one :





Example

progA

```
fork();
execlp("/bin/progB","progB",(char *)0);
while(...)
```

progB

```
main(...){
int A[100],B[100];
...
for(i=0;i<10;i++) A[i]=A[i]+B[i];
...
```

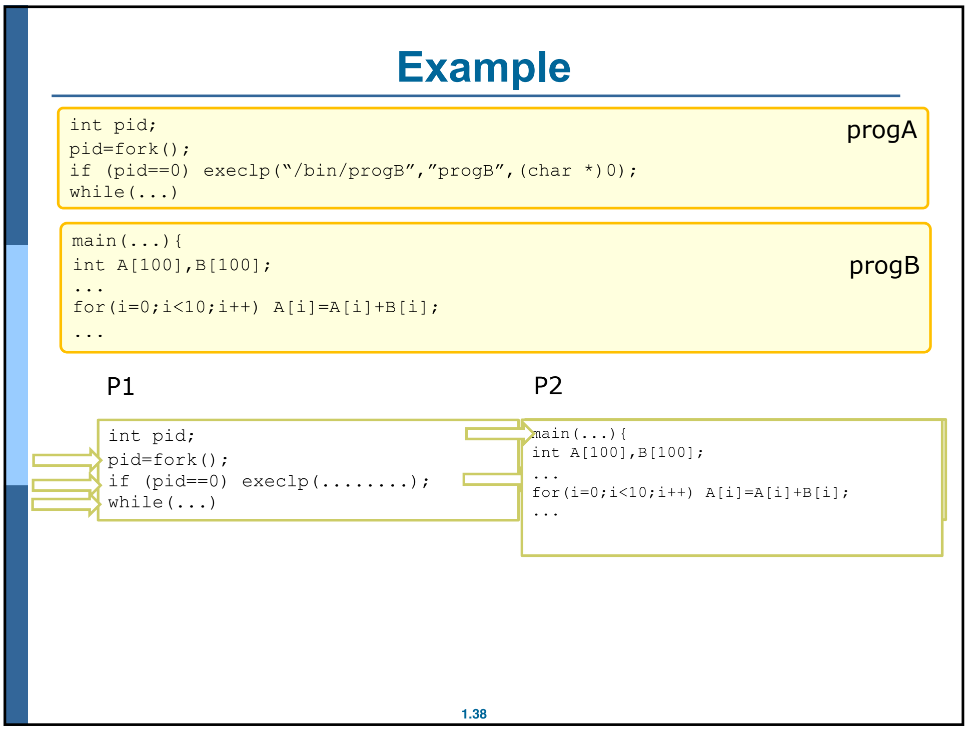
P1

```
main(...){
int A[100],B[100];
...
for(i=0;i<10;i++) A[i]=A[i]+B[i];
...
```

P2

```
main(...){
int A[100],B[100];
...
for(i=0;i<10;i++) A[i]=A[i]+B[i];
...
```

1.37



Example

progA

```
int pid;
pid=fork();
if (pid==0) execlp("/bin/progB","progB",(char *)0);
while(...)
```

progB

```
main(...){
int A[100],B[100];
...
for(i=0;i<10;i++) A[i]=A[i]+B[i];
...
```

P1

```
int pid;
pid=fork();
if (pid==0) execlp(........);
while(...)
```

P2

```
main(...){
int A[100],B[100];
...
for(i=0;i<10;i++) A[i]=A[i]+B[i];
...
```

1.38

Example

- When executing the following command line...

```
% ls -l
```

1. The shell code creates a new process and changes its executable file

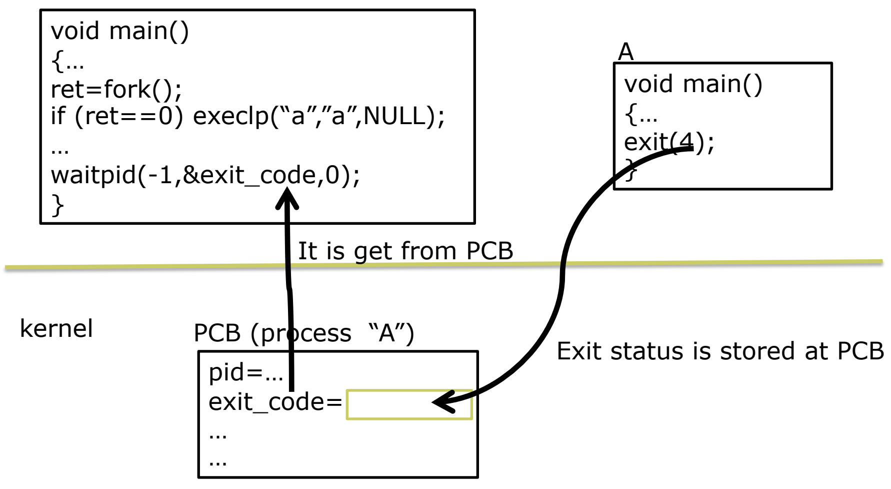
- Something like this

```
...
ret = fork();
if (ret == 0) {
    execlp("/bin/ls", "ls", "-l", (char *)NULL);
}
waitpid(-1,NULL,0);
...
```

- Waitpid is more complex (status is checked), but this is the idea

1.39

Example: exit

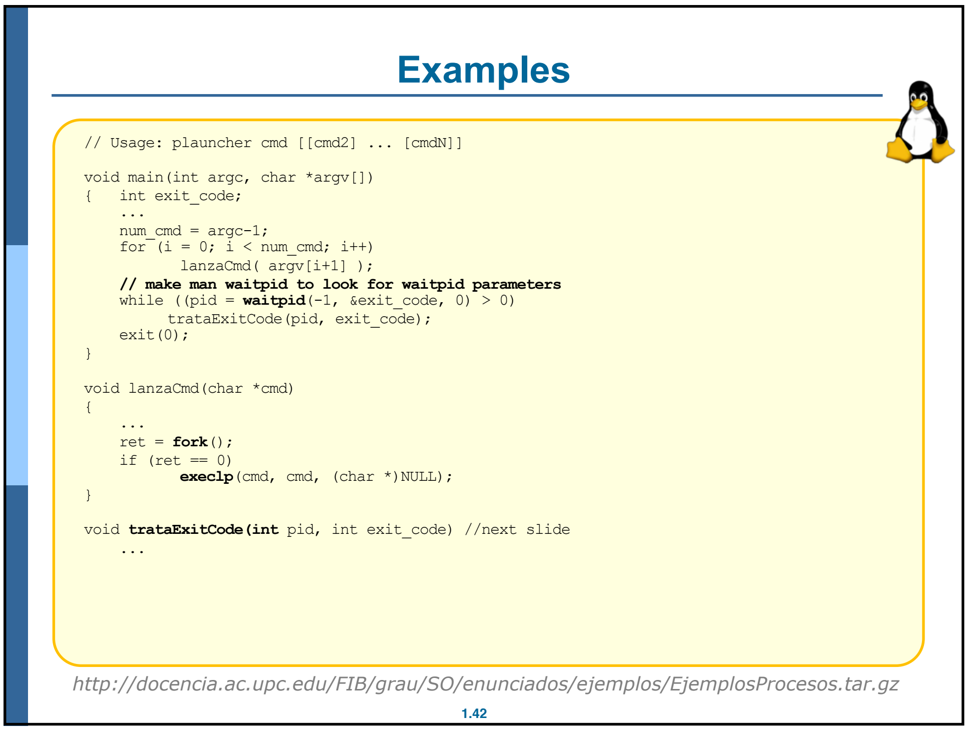


```
void main()
{...
ret=fork();
if (ret==0) execlp("a","a",NULL);
...
waitpid(-1,&exit_code,0);
}
```

```
A
void main()
{...
exit(4);
}
```

However, exit_code value isn't 4. We have to apply some masks

1.41

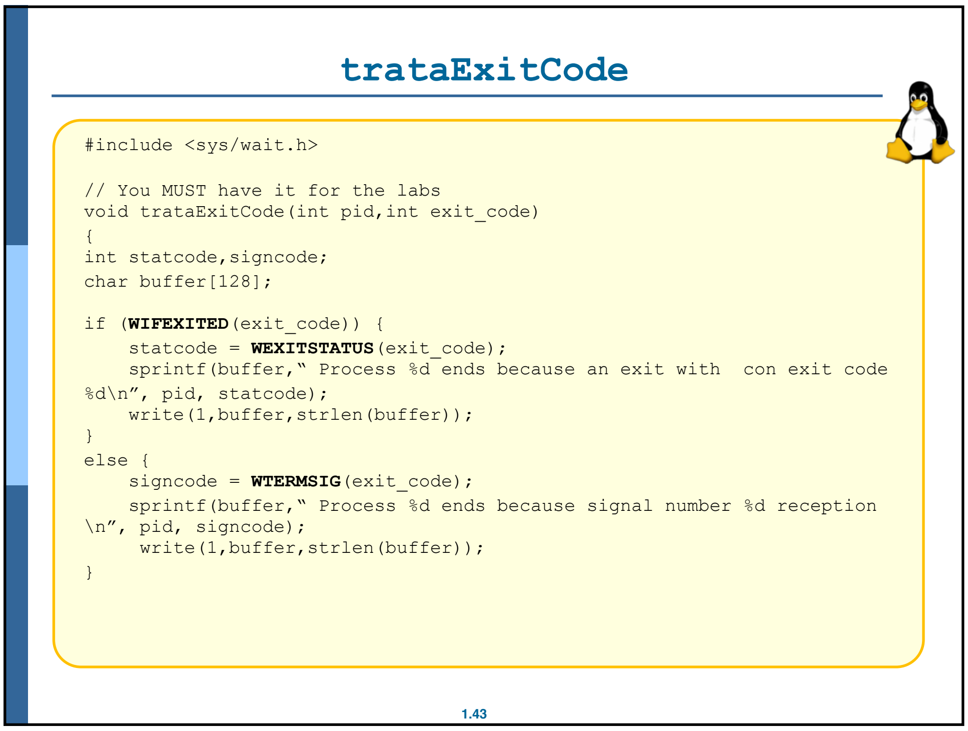


Examples

```
// Usage: plauncher cmd [[cmd2] ... [cmdN]]

void main(int argc, char *argv[])
{   int exit_code;
    ...
    num_cmd = argc-1;
    for (i = 0; i < num_cmd; i++)
           lanzaCmd( argv[i+1] );
    // make man waitpid to look for waitpid parameters
    while ((pid = waitpid(-1, &exit_code, 0) > 0)
          trataExitCode(pid, exit_code);
    exit(0);
}

void lanzaCmd(char *cmd)
{
    ...
    ret = fork();
    if (ret == 0)
           execlp(cmd, cmd, (char *)NULL);
}

void trataExitCode(int pid, int exit_code) //next slide
    ...
```

http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EjemplosProcesos.tar.gz

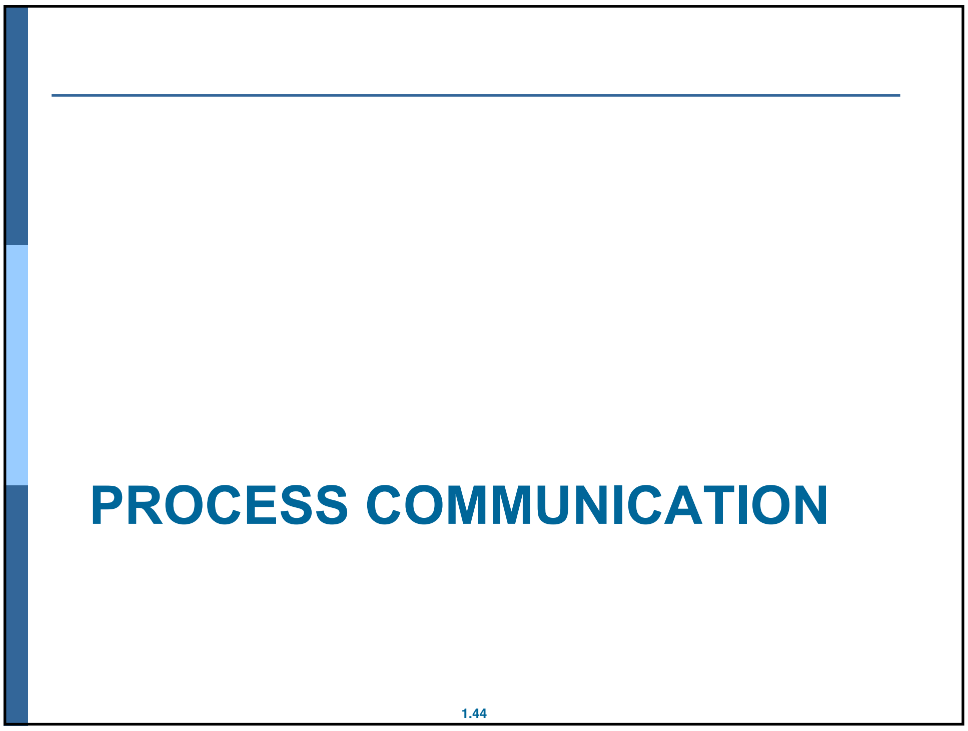
1.42



trataExitCode

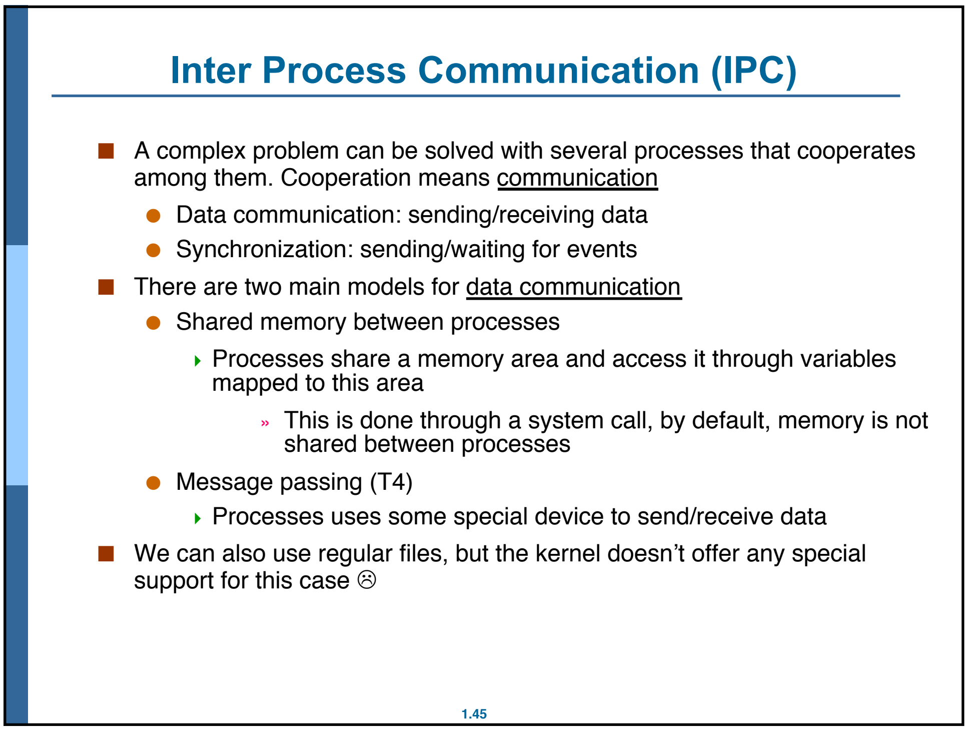
```
#include <sys/wait.h>

// You MUST have it for the labs
void trataExitCode(int pid,int exit_code)
{
int statcode,signcode;
char buffer[128];

if (WIFEXITED(exit_code)) {
    statcode = WEXITSTATUS(exit_code);
    sprintf(buffer,“ Process %d ends because an exit with  con exit code
%d\n”, pid, statcode);
    write(1,buffer,strlen(buffer));
}
else {
    signcode = WTERMSIG(exit_code);
    sprintf(buffer,“ Process %d ends because signal number %d reception
\n”, pid, signcode);
     write(1,buffer,strlen(buffer));
}
```

1.43



PROCESS COMMUNICATION

1.44



Inter Process Communication (IPC)

- A complex problem can be solved with several processes that cooperates among them. Cooperation means communication
  - Data communication: sending/receiving data
  - Synchronization: sending/waiting for events
- There are two main models for data communication
  - Shared memory between processes
    - Processes share a memory area and access it through variables mapped to this area
      - This is done through a system call, by default, memory is not shared between processes
  - Message passing (T4)
    - Processes uses some special device to send/receive data
- We can also use regular files, but the kernel doesn't offer any special support for this case ☹

1.45

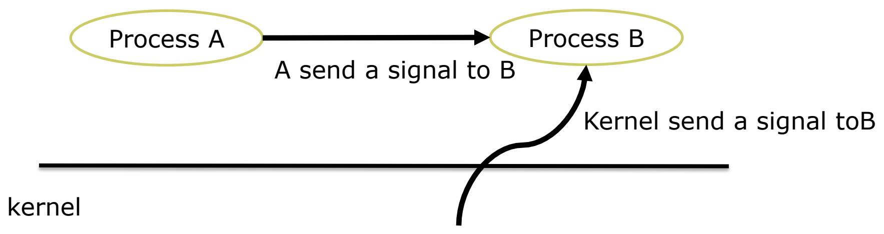
IPC in Linux

- **Signals – Events send by processes belonging to the same user or by the kernel**
- **Pipes /FIFOs: special devices designed for process communication. The kernel offers support for process synchronization (T4)**
- Sockets – Similar to pipes but it uses the network
- Shared memory between processes – Special memory area accessible for more than one process

1.46

Signals: idea

- Signals: notification that an event has occurred
- Signals received by a process can be sent by the kernel or by other processes of the same user



1.47

Type of signals and management (I)

- Each type of event has an associated signal
  - Type of events and associated signals are defined by the kernel
    - The type of signal is a number, but there exists constants that can be used inside programs or in the command line
  - There are two signals that are not associated to any event, so the programmer can assign any meaning to them → SIGUSR1 y SIGUSR2
- **Each process** has associated a **management** to **each signal**
  - Default managements
  - A process can ***catch*** (change the associated management) to all type of signal **except** SIGKILL and SIGSTOP

Type of signals and management (2)

- Some signals

Name	Default management	Event
SIGCHLD	IGNORE	A child process has finished the execution or has been stopped
SIGCONT		To continue a stopped process
SIGSTOP	STOP	To stop a process
SIGINT	END	Interrupted from the keyboard (Ctrl-C)
SIGALRM	END	timer programmed by alarm has expired
SIGKILL	END	Finish the process
SIGSEGV	CORE	Invalid memory access
SIGUSR1	END	Defined by the process
SIGUSR2	END	Defined by the process

- Main uses in this course:
  - Process synchronization
  - Time control (alarms)

Type of signals and management (3)

- Reaction of a process to a signal delivering is similar to the reaction to an interrupt:
  - When a process receives a signal, it stops the code execution, executes the management associated to that signal and then (if it survives) continues with the execution of the code.
- Processes can **block/unblock** the delivery of **each signal** except SIGKILL and SIGSTOP (signals SIGILL, SIGFPE and SIGSEGV cannot be blocked when they are generated by an exception).
  - When a process blocks a signal, if that signal is sent to the process it will not be delivered until the process unblocks it.
    - The system marks the signal as pending to be delivered
    - Each process has bitmap of pending signals: it only can remember one pending delivery for each type of singal
  - When a process unblocks a signal, it will receive the pending signal and will execute the associated management

Linux: signals basic interface

Service	System call
Send a signal	kill
Catch a signal	sigaction
Block/unblock signals	sigprocmask
Wait for a signal (this system call is blocking)	sigsuspend
Timer setting (to send a SIGALRM to the process)	alarm

- **Definition of signals constants: /usr/include/bits/signum.h**
  - Check man 7 signal
- There exists several interfaces to deal with signals, all of them incompatibles and each of them with different limitations. Linux implements the POSIX interface.

Interface: sending and catching signals

- Sending:

```
int kill(int pid, int signum)
```

  - signum→ SIGUSR1, SIGUSR2,etc
  - Requirement: to know the pid of the target process
- Catching signals: defining the management of the signal for the current process

```
int sigaction(int signum, struct sigaction *mngt,
struct sigaction *old_mngt)
```

  - signum→SIGUSR1, SIGUSR2,etc
  - mngt→struct sigaction that describes how to behave when the process receives the signal signum
  - old_mngt→ struct sigaction that describes the former management for this signal. This parameter can be NULL if this information is not necessary

1.52

A sends a signal to B

- Process A sends (at any time) a signal to B and B executes an action when it receives it

```
Process A
…..
Kill( pid, event);
….
```

```
Process B
int main()
{
struct sigaction mng,old_mng;
/* code to initialize trat */
sigaction(event, &mng, &old_mng);
….
}
```

1.53

Definition of struct sigaction

- struct sigaction: several fields. In this course we will only use the following 3 fields:
  - sa_handler: can take 3 different values
    - SIG_IGN: ignore the signal when it is received
    - SIG_DFL: use the default management for this signal
    - user function with the following signature: void function_name(int s);
      - IMPORTANT: this function will be called by the kernel. Parameter *s* will take as a value the delivered signal (SIGUSR1, SIGUSR2, etc), So you can associate the same function to several signals and make a differential treatment inside it.
  - sa_mask: signals to add to the mask of blocked signals of the process just during the execution of the signal manangement
    - If this mask is empty, the only signal added is the signal catched by the sigaction system call
    - When the signal management ends, the mask of blocked signals used before the signal management is restaured
  - sa_flags: to configure the behaviour of the signal management (if this field is 0, the default configuration is used). Some flags:
    - SA_RESETHAND: Signal catching is only valid for one signal delivering. After managing one signal, the default signal management is restored.
    - SA_RESTART: if a process blocked in a system call receives a signal, the blocking system call is restarted after the signal management (we will talk about this in lecture 4)

1.54

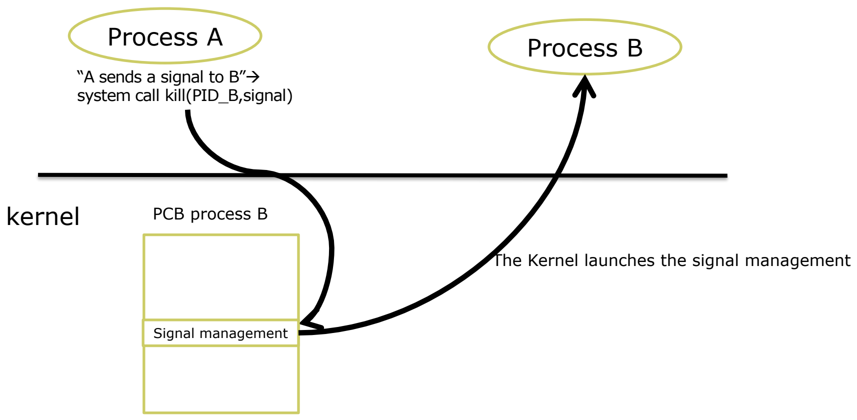
Kernel data structures

- Signal management is per process: the management information is in the PCB. Each process has:
  - A **table of signals** (1 entry per signal),
    - Action to perform when the signal is received
  - A bitmap of **pending events** (1 bit per signal)
    - It is not a counter, is like a boolean
  - A **timer** to manage alarms
    - If a process programs two timers, the only valid is the last one
  - A mask to indicate which signals the process want to receive and manage

1.55

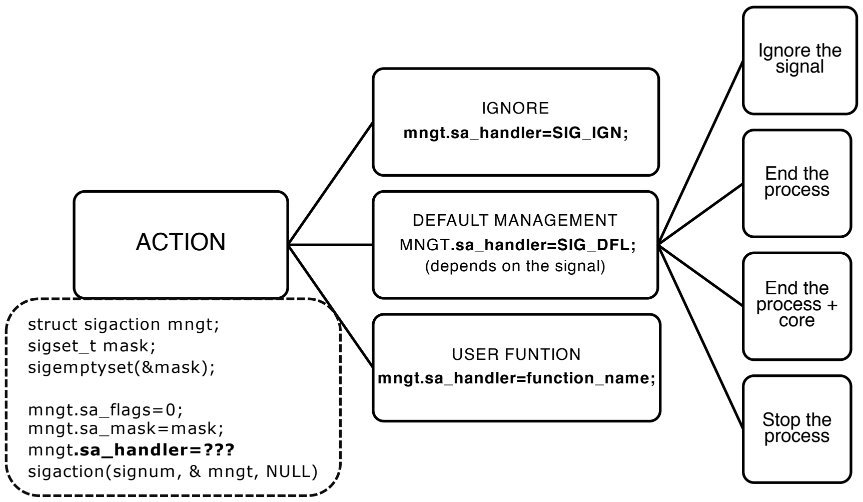
Signals: Sending and delivering

What really happens?: the kernel offers the service to pass the information.


Process A
"A sends a signal to B"→ system call kill(PID_B,signal)
Process B
kernel
PCB process B
Signal management
The Kernel launches the signal management

1.56

Possible signal managements



```
struct sigaction mngt;
sigset_t mask;
sigemptyset(&mask);

mngt.sa_flags=0;
mngt.sa_mask=mask;
mngt.sa_handler=???
sigaction(signum, & mngt, NULL)
```

1.57

Signal mask management

- sigemptyset: initializes a mask without any signal

```
int sigemptyset(sigset_t *mask)
```

- sigfillset: initializes a mask with all signals

```
int sigfillset(sigset_t *mask)
```

- sigaddset:add a signal to the mask parameter

```
int sigaddset(sigset_t *mask, int signum)
```

- sigdelset:removes a signal from the mask parameter

```
int sigdelset(sigset_t *mask, int signum)
```

- sigismember: returns true if signal is set in mask

```
int sigismember(sigset_t *mask, int signum)
```

Example: catching signals

```
void main()
{   char buffer[128];
    struct sigaction trat;
    sigset_t mask;
    sigemptyset(&mask);
    trat.sa_mask=mask;
    trat.sa_flags=0;
    trat.sa_handler = f_sigint;

    sigaction(SIGINT, &trat, NULL); // when the process receives SIGINT
                                    //  it will execute f_sigint
    while(1) {
        sprintf(buffer,"Doing something\n");
        write(1,buffer,strlen(buffer));
    }
}

void f_sigint(int s)
{
    char buffer[128];
    sprintf(buffer,"SIGINT received!\n");
    exit(0);
}
```

This code is in: signal_basico.c

Blocking/Unblocking signals

- A process can control in which part of the code can receive and manage each type of signal

```
int sigprocmask(int operacion, sigset_t *mask, sigset_t *old_mask)
```

  - Operation can be:
    - SIG_BLOCK: **adds** the signals in *mask* to the mask of blocked signals of the process
    - SIG_UNBLOCK: **removes** the signals in *mask* to the mask of blocked signals of the process
    - SIG_SETMASK: *mask* is the new mask of blocked signals of the process

1.60

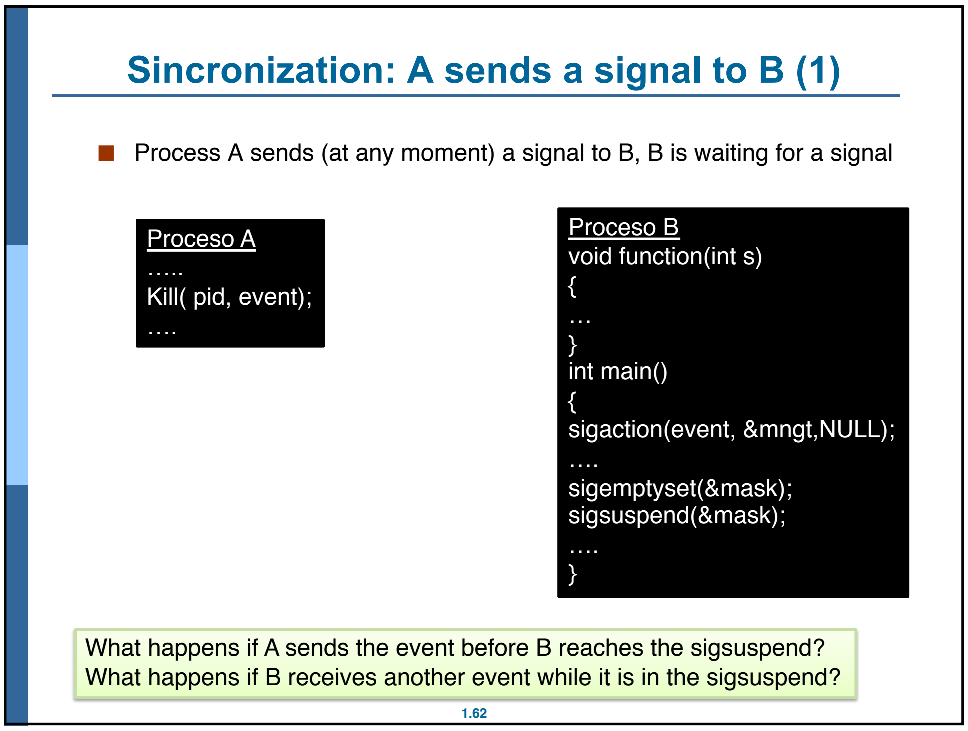
Wait for an event

- Wait (blocked) for a signal

```
int sigsuspend(sigset_t *mask)
```

  - Blocks the process until an event whose treatment is not SIG_IGN arrives
  - **While** the process is blocked in the sigsuspend, *mask* is the current mask of blocked signals
    - This allows to control which signals will unblock the process
  - When the process exits the sigsuspend, the former mask of blocked signals is restaured and all pending signals are managed that got unblocked

1.61



Sincronization: A sends a signal to B (1)

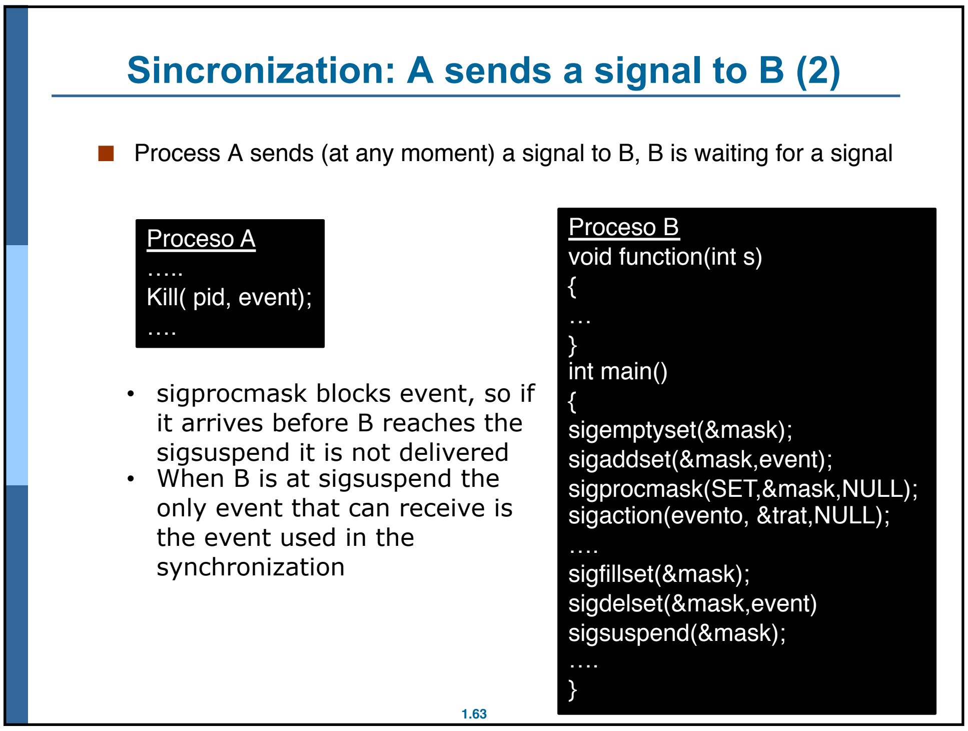
- Process A sends (at any moment) a signal to B, B is waiting for a signal

```
Proceso A
…..
Kill( pid, event);
….
```

```
Proceso B
void function(int s)
{
…
}
int main()
{
sigaction(event, &mngt,NULL);
….
sigemptyset(&mask);
sigsuspend(&mask);
….
}
```

What happens if A sends the event before B reaches the sigsuspend?
What happens if B receives another event while it is in the sigsuspend?

1.62



Sincronization: A sends a signal to B (2)

- Process A sends (at any moment) a signal to B, B is waiting for a signal

```
Proceso A
…..
Kill( pid, event);
….
```

- sigprocmask blocks event, so if it arrives before B reaches the sigsuspend it is not delivered
- When B is at sigsuspend the only event that can receive is the event used in the synchronization

```
Proceso B
void function(int s)
{
…
}
int main()
{
sigemptyset(&mask);
sigaddset(&mask,event);
sigprocmask(SET,&mask,NULL);
sigaction(evento, &trat,NULL);
….
sigfillset(&mask);
sigdelset(&mask,event)
sigsuspend(&mask);
….
}
```

1.63

Choices to implement process syncrhonization

- **Choices** to “wait” for an event
  1. **Active waiting**: The process consumes CPU to check if the event has arrived (usually checking the value of some variable)
     - Example: while(!received);
  2. **Blocking**: The process leaves the CPU (gets blocked) and the kernel willl unblock it when the waiting event arrives
     - Example: sigsuspend
- For short waiting times, the recommendation is to use active waiting
  - It does not compensate for the overhead required to execute the process blocking and the context switch
- For long waiting times, the recommendation is to use blocking
  - The CPU can be used to advance with the execution of the rest of processes (including the one that is going to cause the waiting event)

Timing control: programming the timer

- Programming the delivery of a SIGALRM after a period of time (the kernel will send to us the signal)
  - int alarm(num_secs);

```
ret=rem_time;
si (num_secs==0) {
        enviar_SIGALRM=OFF
}else{
        enviar_SIGALRM=ON
        rem_time=num_secs,
}
return ret;
```

Timing control: timer usage

- The process programs a timer of 2 seconds and gets blocked until the timer expires

```
void function(int s)
{
…
}
int main()
{
sigemptyset(&mask);
sigaddset(&mask, SIGALRM);
sigprocmask(SET,&mask, NULL);
sigaction(SIGALRM, &mngt, NULL);
….
sigfillset(&mask);
sigdelset(&mask,SIGALRM);
alarm(2);
sigsuspend(&mask);
….
}
```

The process gets blocked at sigsuspend. After 2 seconds the SIGALRM is delivered, the process executes the function in the mngt associated and continues after the sigsuspend

1.66

Relation with fork and exec

- FORK: **new process**
  - Child inherit from parent
    - the signal table
    - the mask of blocked signals
  - Child resets
    - The bitmap of pending events
    - Timers
  - Events and timers are associated to a particular pid (the pid of the parent) and children are new processes with new pids.
- EXECLP: **same process**, new image
  - Process keeps
    - The bitmap of pending events
    - The mask of blocked signals
    - Timers
  - Process resets
    - The signal table→ the code of the process is different so the handle code of the signals is set to SIG_DFL

1.67

Example 1: management of 2 signals (1)

```
void main()
{
   sigemptyset(&mask1);
   sigaddset(&mask1,SIGALRM);
   sigprocmask(SIG_BLOCK,&mask1,NULL);

   trat.sa_flags=0;
   trat.sa_handler = f_alarma;
   sigemptyset(&mask2);
   trat.sa_mask=mask2;
   sigaction(SIGALRM, &trat, NULL);

   sigfillset(&mask3);
   sigdelset(&mask3,SIGALRM);

   for(i = 0; i < 10; i++) {
     alarm(2);
     sigsuspend(&mask3);
     crea_ps();
   }
}
```

```
void f_alarma()
{
}
void crea_ps()
{
    pid = fork();
    if (pid == 0)
         execlp("ps", "ps",
(char *)NULL);
}
```

This code is in : cada_segundo.c

Example 2: active waiting vs blocking (1)

```
void main()
{
    configurar_esperar_alarma()
    trat.sa_flags = 0;
    trat.sa_handler=f_alarma;
    sigsetempty(&mask);
    trat.sa_mask=mask;
    sigaction(SIGALRM,&trat,NULL);
    trat.sa_handler=fin_hijo;
    sigaction(SIGCHLD,&trat,NULL);
    for (i = 0; i < 10; i++) {
         alarm(2);
         esperar_alarma(); // Which options do we have?
         crea_ps();
    }
}
void f_alarma()
{
    alarma = 1;
}
void fin_hijo()
{
    while(waitpid(-1,NULL,WNOHANG) > 0);
}
```

```
void crea_ps()
{
    pid = fork();
    if (pid == 0)
         execlp("ps", "ps,
               (char *)NULL);
}
```

This code is in: cada_segundo_sigchld.c

Example 2: active waiting vs blocking (2)

Choice 1: active waiting

```
void configurar_esperar_alarma() {
   alarma = 0;
}
void esperar_alarma(){
   while (alarma!=1);
   alarma=0;
}
```

Choice 2: blocking

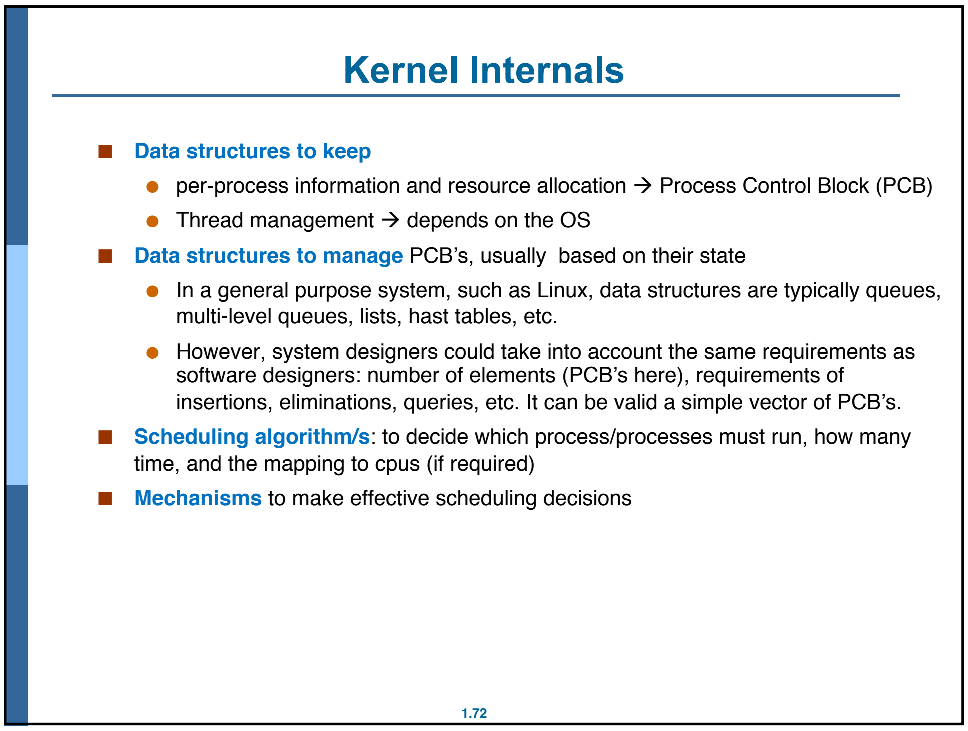
```
void configurar_esperar_alarma() {
   sigemptyset(&mask);
   sigaddset(&mask, SIGALRM);
   sigprocmask(SIG_BLOCK,&mask, NULL);
}

void esperar_alarma(){
        sigfillset(&mask);
        sigdelset(&mask,SIGALRM);
        sigsuspend(&mask);
}
```

-Data structures

-Scheduling policies

-Kernel mechanisms for process management

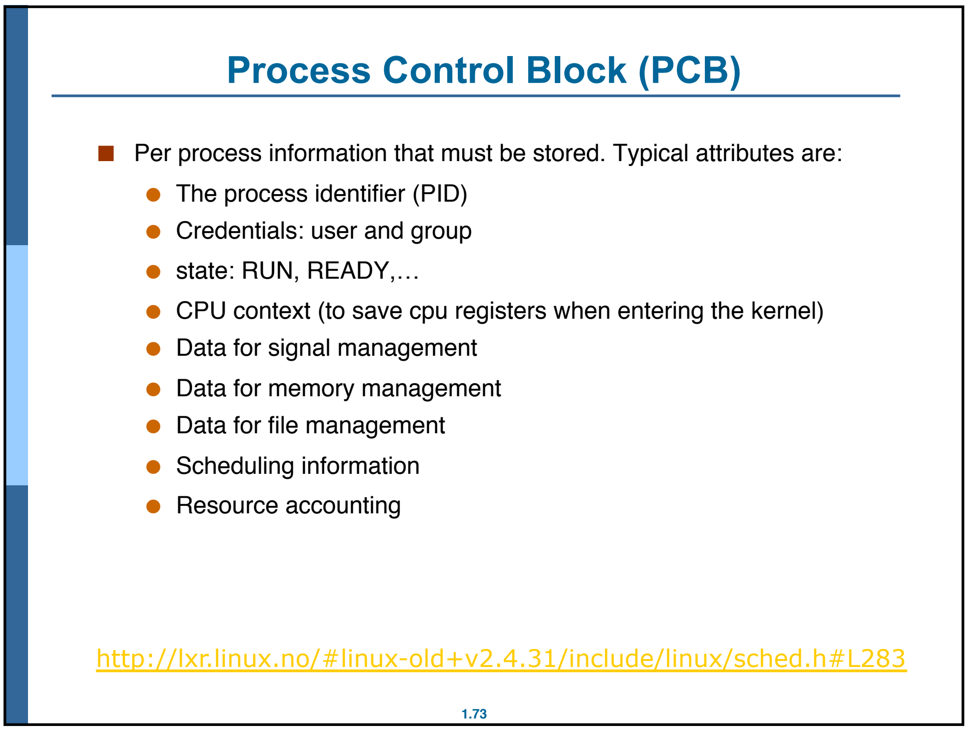
KERNEL INTERNALS



Kernel Internals

- **Data structures to keep**
  - per-process information and resource allocation → Process Control Block (PCB)
  - Thread management → depends on the OS
- **Data structures to manage** PCB's, usually based on their state
  - In a general purpose system, such as Linux, data structures are typically queues, multi-level queues, lists, hast tables, etc.
  - However, system designers could take into account the same requirements as software designers: number of elements (PCB's here), requirements of insertions, eliminations, queries, etc. It can be valid a simple vector of PCB's.
- **Scheduling algorithm/s**: to decide which process/processes must run, how many time, and the mapping to cpus (if required)
- **Mechanisms** to make effective scheduling decisions

1.72

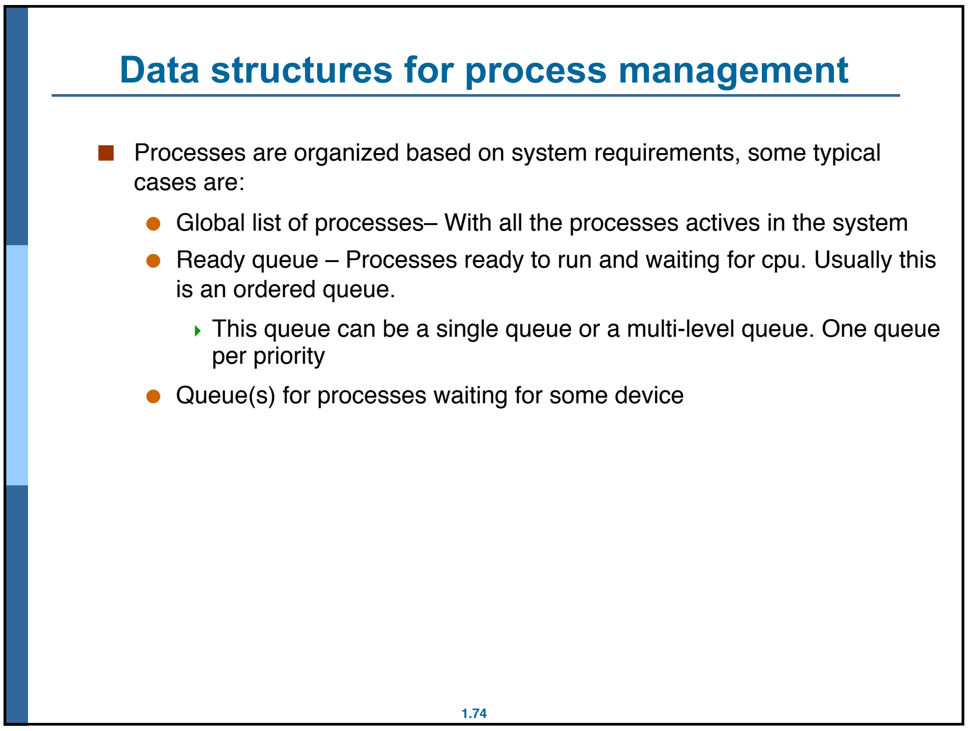


Process Control Block (PCB)

- Per process information that must be stored. Typical attributes are:
  - The process identifier (PID)
  - Credentials: user and group
  - state: RUN, READY,…
  - CPU context (to save cpu registers when entering the kernel)
  - Data for signal management
  - Data for memory management
  - Data for file management
  - Scheduling information
  - Resource accounting

http://lxr.linux.no/#linux-old+v2.4.31/include/linux/sched.h#L283

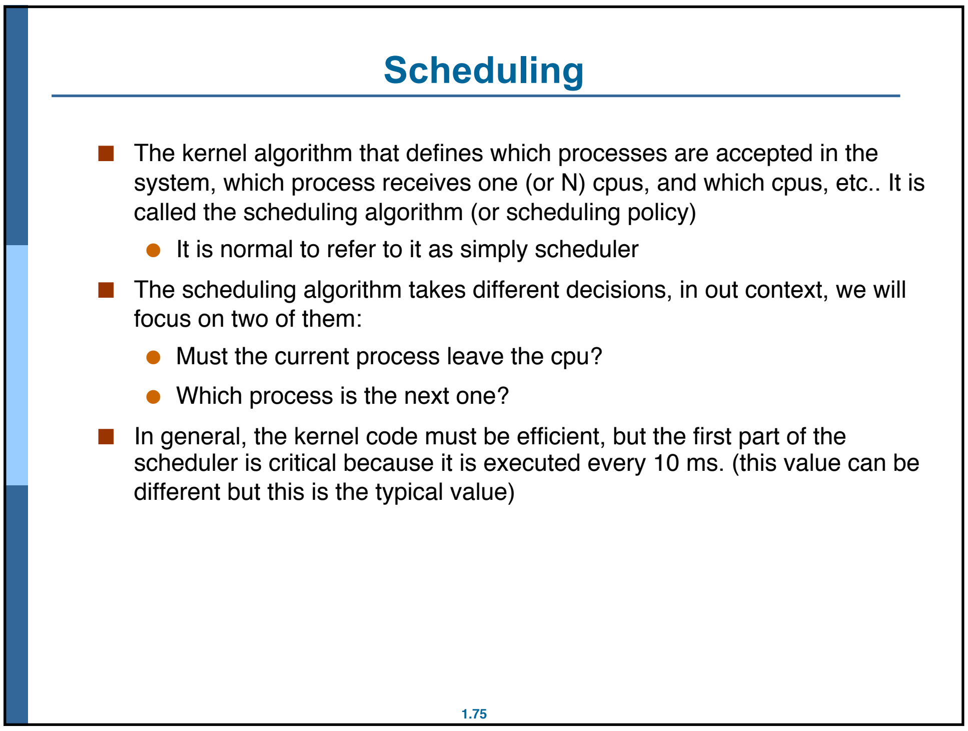
1.73



Data structures for process management

- Processes are organized based on system requirements, some typical cases are:
  - Global list of processes– With all the processes actives in the system
  - Ready queue – Processes ready to run and waiting for cpu. Usually this is an ordered queue.
    - This queue can be a single queue or a multi-level queue. One queue per priority
  - Queue(s) for processes waiting for some device

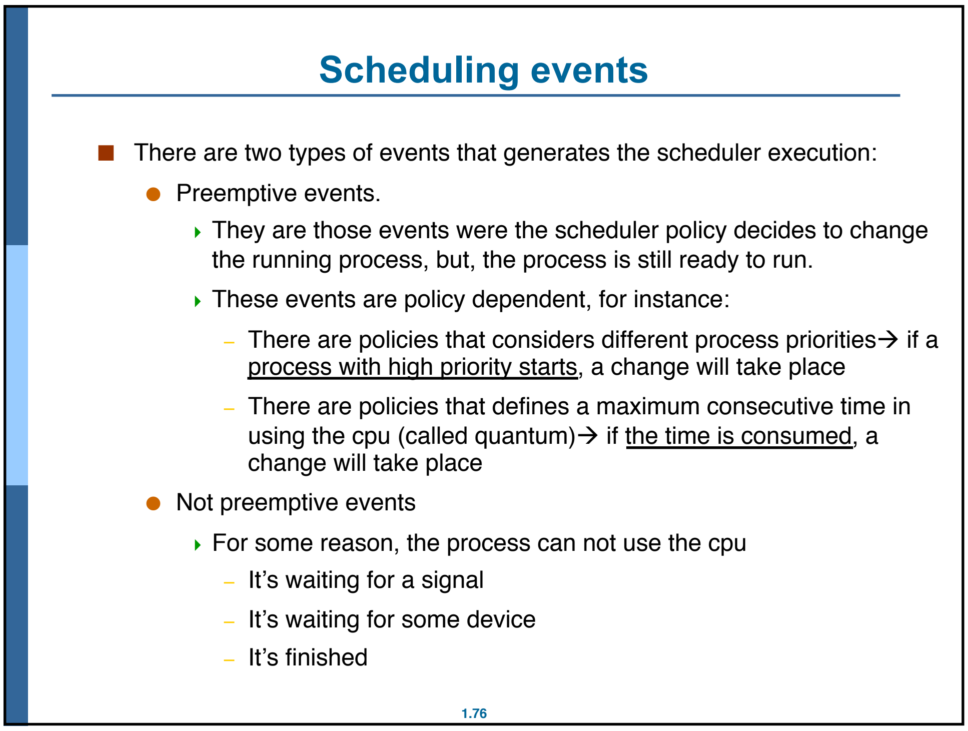
1.74



Scheduling

- The kernel algorithm that defines which processes are accepted in the system, which process receives one (or N) cpus, and which cpus, etc.. It is called the scheduling algorithm (or scheduling policy)
  - It is normal to refer to it as simply scheduler
- The scheduling algorithm takes different decisions, in out context, we will focus on two of them:
  - Must the current process leave the cpu?
  - Which process is the next one?
- In general, the kernel code must be efficient, but the first part of the scheduler is critical because it is executed every 10 ms. (this value can be different but this is the typical value)

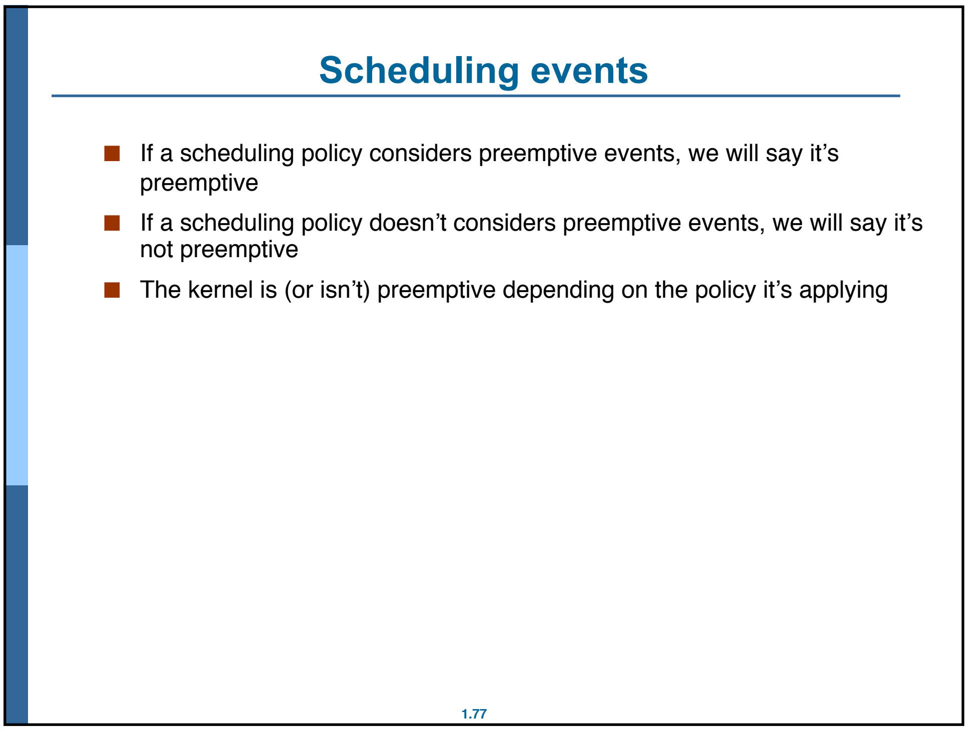
1.75



Scheduling events

- There are two types of events that generates the scheduler execution:
  - Preemptive events.
    - They are those events were the scheduler policy decides to change the running process, but, the process is still ready to run.
    - These events are policy dependent, for instance:
      - There are policies that considers different process priorities→ if a process with high priority starts, a change will take place
      - There are policies that defines a maximum consecutive time in using the cpu (called quantum)→ if the time is consumed, a change will take place
  - Not preemptive events
    - For some reason, the process can not use the cpu
      - It's waiting for a signal
      - It's waiting for some device
      - It's finished

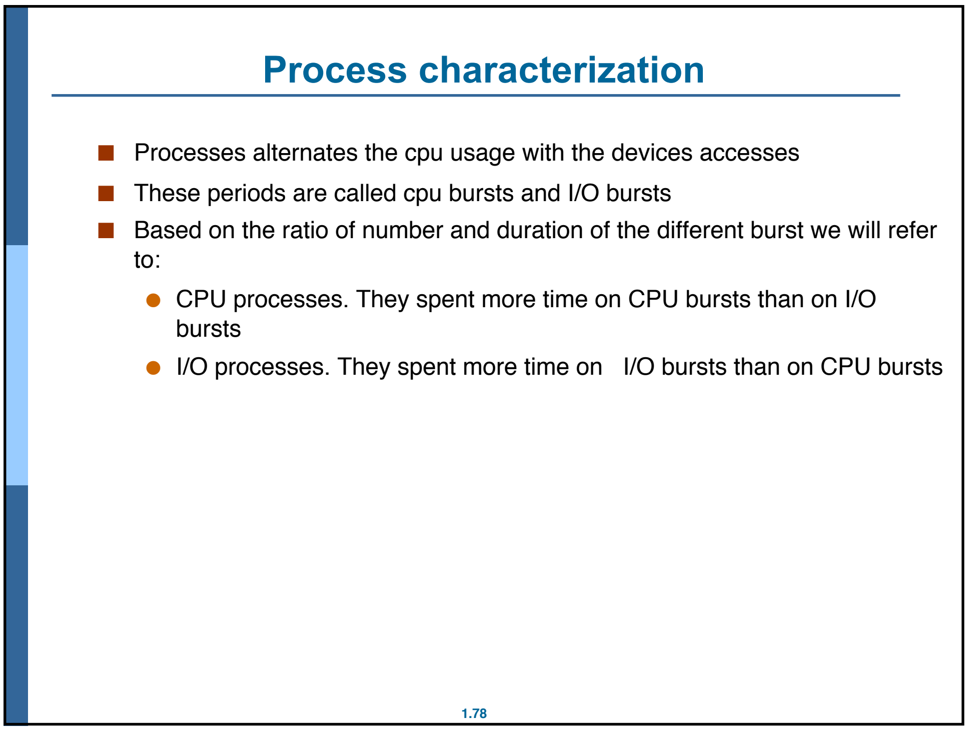
1.76



Scheduling events

- If a scheduling policy considers preemptive events, we will say it's preemptive
- If a scheduling policy doesn't considers preemptive events, we will say it's not preemptive
- The kernel is (or isn't) preemptive depending on the policy it's applying

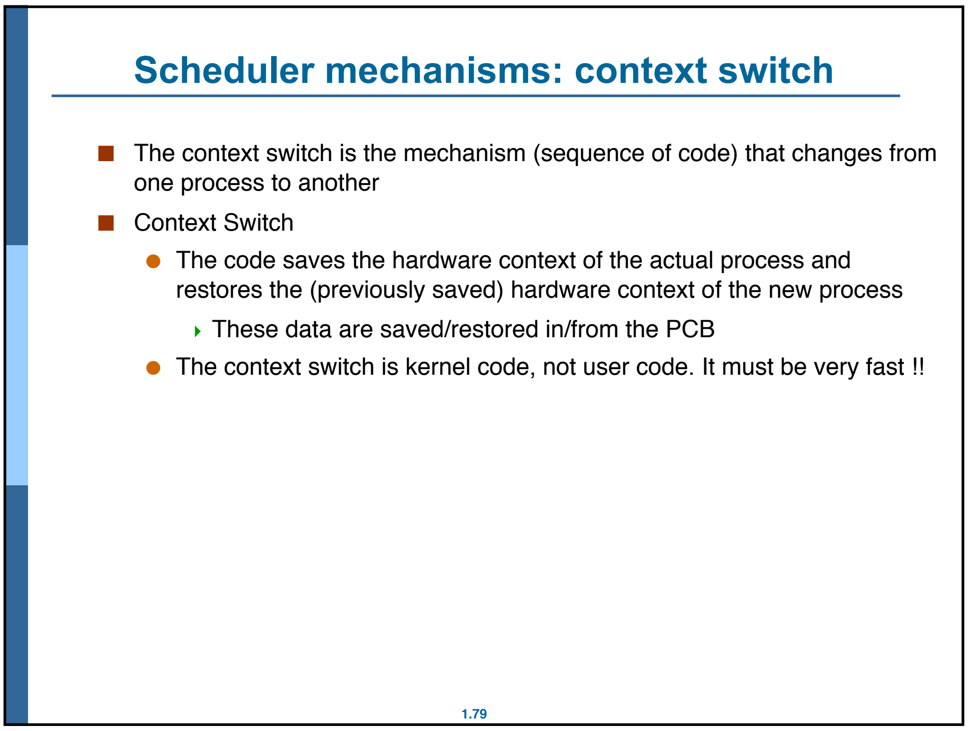
1.77



Process characterization

- Processes alternates the cpu usage with the devices accesses
- These periods are called cpu bursts and I/O bursts
- Based on the ratio of number and duration of the different burst we will refer to:
  - CPU processes. They spent more time on CPU bursts than on I/O bursts
  - I/O processes. They spent more time on I/O bursts than on CPU bursts

1.78

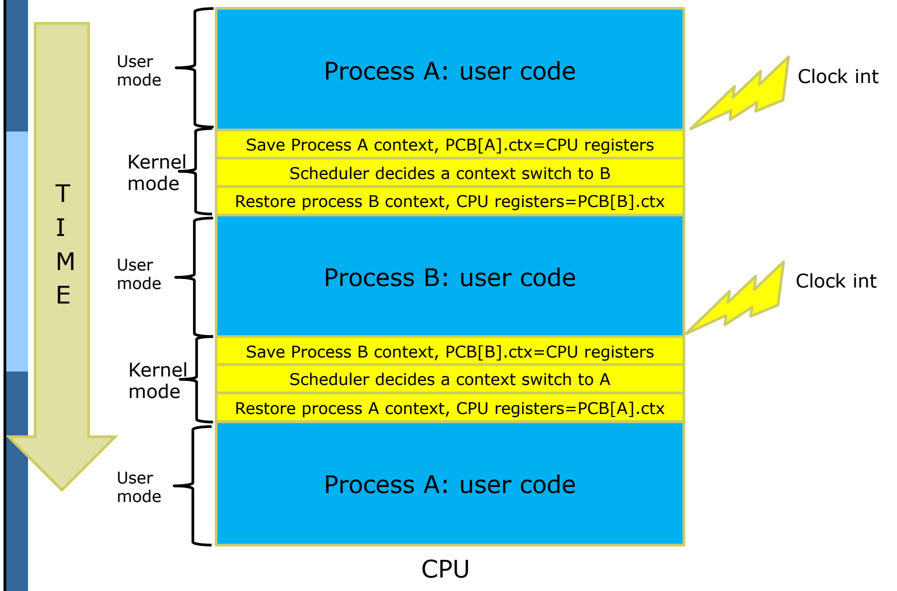


Scheduler mechanisms: context switch

- The context switch is the mechanism (sequence of code) that changes from one process to another
- Context Switch
  - The code saves the hardware context of the actual process and restores the (previously saved) hardware context of the new process
    - These data are saved/restored in/from the PCB
  - The context switch is kernel code, not user code. It must be very fast !!

1.79

Context switch


T I M E
User mode
Process A: user code
Clock int
Kernel mode
Save Process A context, PCB[A].ctx=CPU registers
Scheduler decides a context switch to B
Restore process B context, CPU registers=PCB[B].ctx
User mode
Process B: user code
Clock int
Kernel mode
Save Process B context, PCB[B].ctx=CPU registers
Scheduler decides a context switch to A
Restore process A context, CPU registers=PCB[A].ctx
User mode
Process A: user code
CPU

Scheduling metrics

- Scheduling policies are designed for different systems, each one with specific goals and metrics to evaluate the success of these goals
- Some basic metrics are:
  - Execution time: Total time the process is in the system
    - End time- submission time
    - It includes time spent in any state
    - It depends on the process itself, the scheduling policy and the system load
  - Wait time: time the process spent in ready state
    - It depends on the policy and the system load

Scheduling policy: Round Robin

- Round Robin policy goal is to fairly distribute CPU time between processes
- The policy uses a ready queue
  - Insertions in the data structure are at the end of the "list"
  - Extractions are from the head of the "list"
- The policy defines a maximum consecutive time in the CPU, once this time is consumed, the process is queued and the first one is selected to run
  - This "maximum" time is called Quantum
  - Typical value is 10ms.

Round Robin (RR)

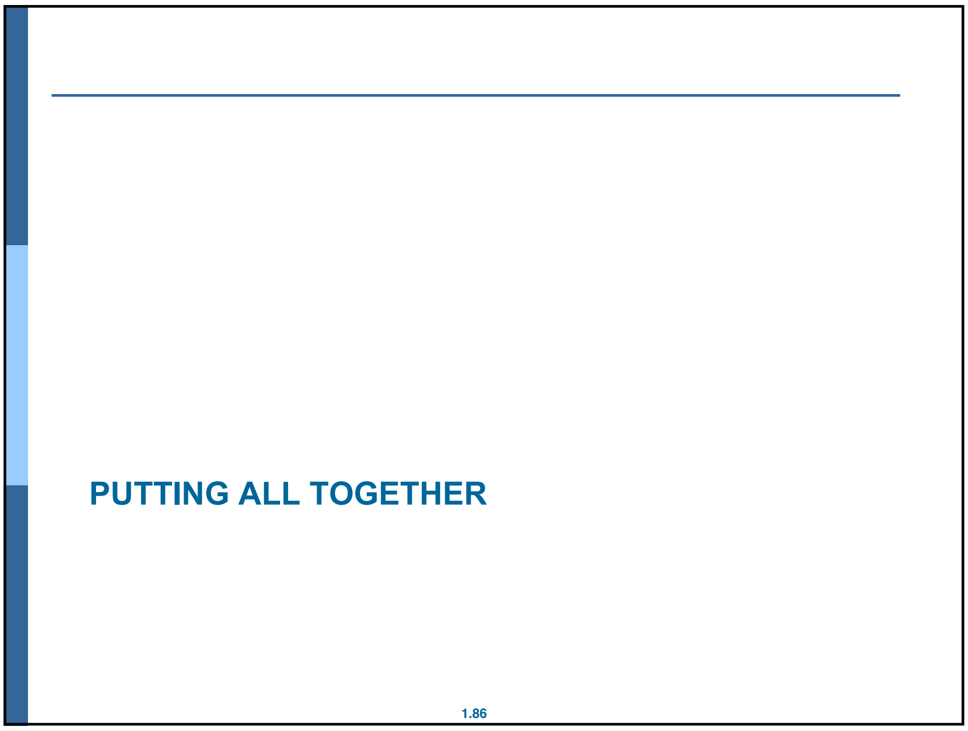
- Round Robin events:
  1. The process is blocked (not preemptive)
  2. The process is finished (not preemptive)
  3. The Quantum is finished (preemptive)
- Since it considers preemptive events, it is a preemptive policy
- Depending on the event, the process will be in state….
  1. Blocked
  2. Zombie
  3. Ready

Round Robin (RR)

- **Policy evaluation**
  - If there are N processes in the READY queue, and the quantum is Q milliseconds, each process gets 1/n parts of cpu time in blocks of Q milliseconds at most.
    - No process waits more than (N-1) Q milliseconds.
  - Policy works different depending on the quantum
    - *q* too large ⇒ RR scheduling degenerates to FCFS scheduling
    - *q too small* ⇒ scheduling overhead in the form context-switch time becomes excessive.

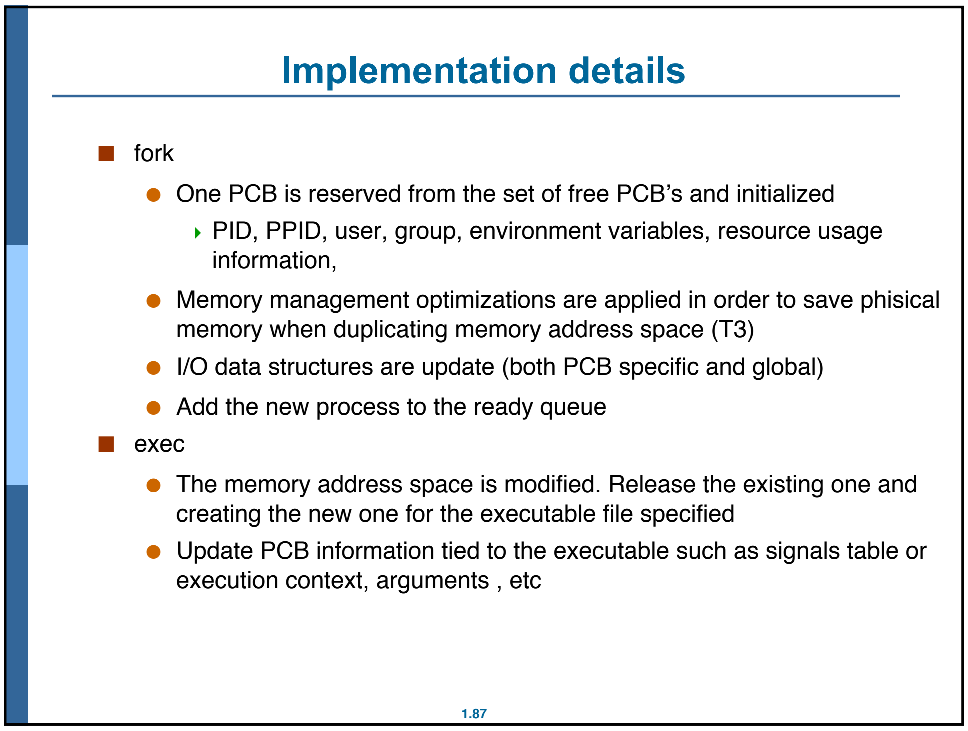
Completely Fair Scheduler

- Linux algorithm since kernel 2.6.23
- Objective metric: CPU usage time of all processes has to be equivalent
  - Round Robin penalizes I/O intensive processes
- CPU time slice (~quantum) is variable
  - Theoretical cpu time consumed for each process should be the result of divide the current process execution time by all runnable processes
  - Each process gets cpu until it blocks, finish or gets its theoretical time slice
- Priority: theoretical CPU time distance. The farther, the more priority
- Process groups: instead of all tasks being treated fairly, the spawned tasks with their parent share their theoretical runtimes across the group (in a hierarchy).
  - Goal: to prevent a user running many processes from monopolizing the machine



PUTTING ALL TOGETHER

1.86



Implementation details

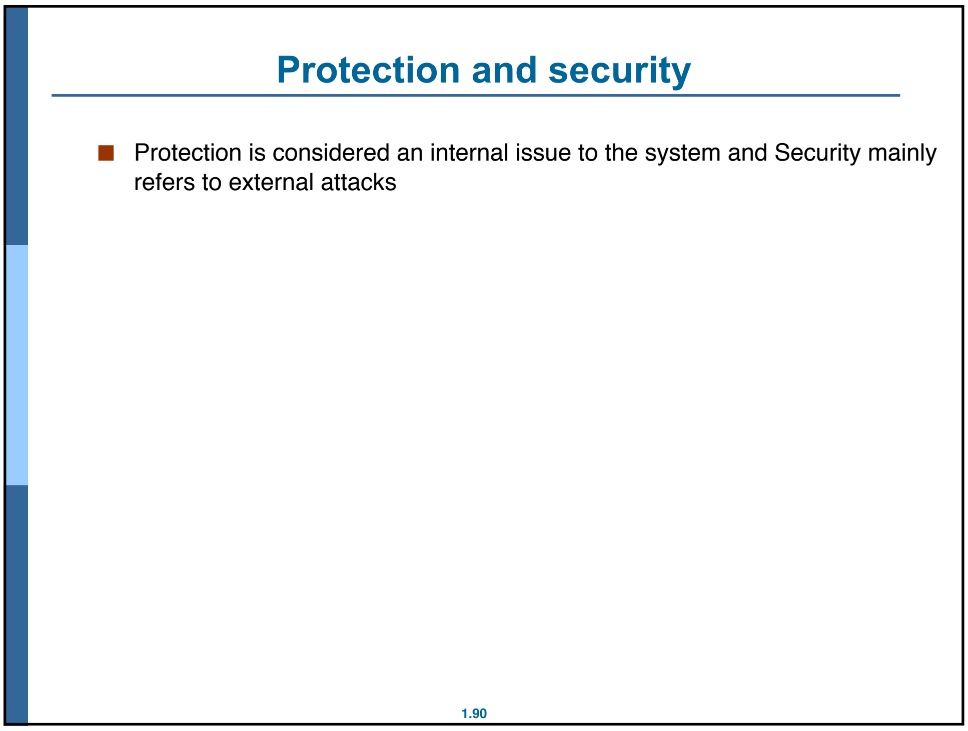
- fork
  - One PCB is reserved from the set of free PCB's and initialized
    - PID, PPID, user, group, environment variables, resource usage information,
  - Memory management optimizations are applied in order to save phisical memory when duplicating memory address space (T3)
  - I/O data structures are update (both PCB specific and global)
  - Add the new process to the ready queue
- exec
  - The memory address space is modified. Release the existing one and creating the new one for the executable file specified
  - Update PCB information tied to the executable such as signals table or execution context, arguments , etc

1.87

Implementation details

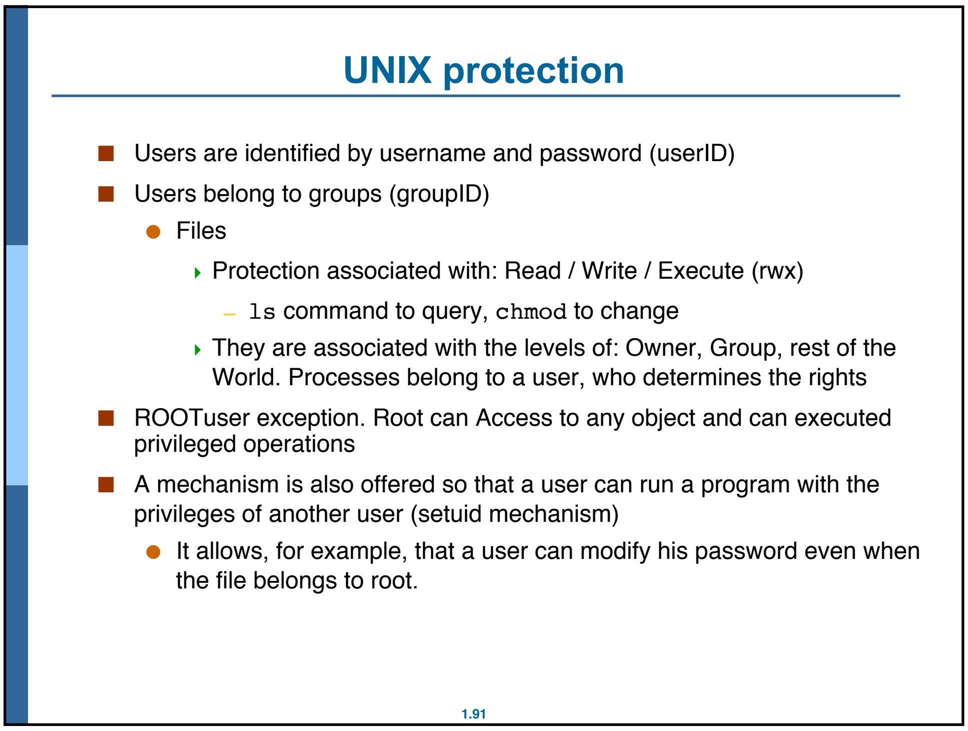
- exit
  - All the process resources are released: physical memory, open devices, etc.
  - Exit status is saved in PCB and the process state is set to ZOMBIE
  - The scheduling policy is applied to select a new process to run
- waitpid
  - The kernel looks for a specific child o some child in ZOMBIE state of the calling process
  - If found, exit status will be returned and the PCB is released (it is marked as free to be reused)
  - Otherwise, the calling process is blocked and the scheduling policy is applied.

PROTECTION AND SECURITY



Protection and security

- Protection is considered an internal issue to the system and Security mainly refers to external attacks



UNIX protection

- Users are identified by username and password (userID)
- Users belong to groups (groupID)
  - Files
    - Protection associated with: Read / Write / Execute (rwx)
      - `ls` command to query, `chmod` to change
    - They are associated with the levels of: Owner, Group, rest of the World. Processes belong to a user, who determines the rights
- ROOTuser exception. Root can Access to any object and can executed privileged operations
- A mechanism is also offered so that a user can run a program with the privileges of another user (setuid mechanism)
  - It allows, for example, that a user can modify his password even when the file belongs to root.

Security

- Security must be considered at four levels:
- Physical
  - Machines and access terminals must be in a safe room/building.
- Human
  - It is important to control who is granted access to the systems and make users aware of not facilitating that other people can access their user accounts
- OS
  - Prevent a process (s) from saturating the system
  - Ensure that certain services are always working
  - Ensure that certain access ports are not operational
  - Check that processes cannot access outside their own address space
- Network
  - Most data today moves through the network. This component of the systems is normally the most attacked.

References

- Examples: *http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EjemplosProcesos.tar.gz*
- http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EjemploCreacionProcesos.ppsx