

# 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: One process is the O.S. representation of a program during its execution
  
- Why?
  - One user program is something static: it is just a sequence of bytes stored in a “disk”
  - When user ask 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...



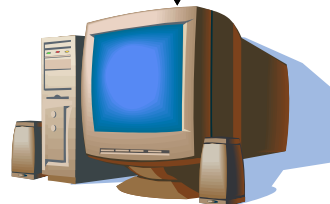
shell

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

Create new process Execute "p1" Finish process	p1.c"
--	-------

System calls

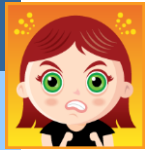
Kernel



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

# 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

# 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)

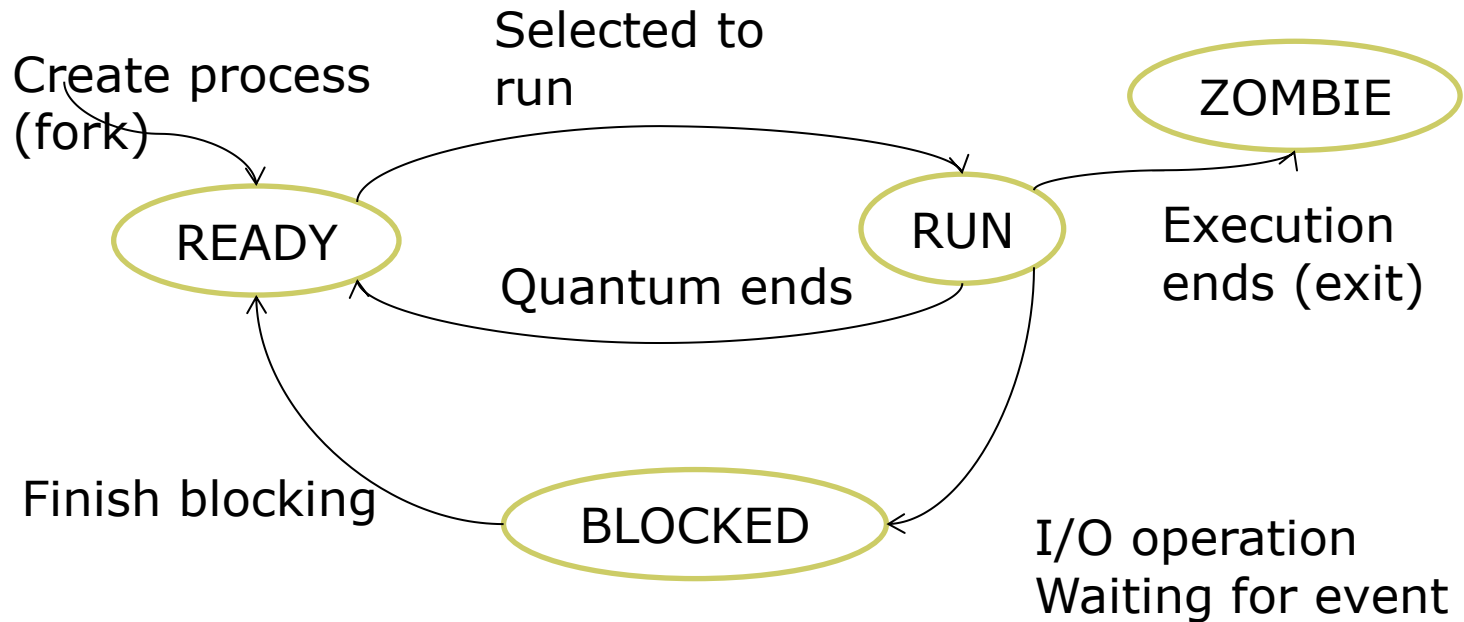


# Process state

---

- 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

# State graph example



# 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 availables 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.

# 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);
```

---

# 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

## Basic set of system calls for process manager X)

Description	Syscall
Process creation	fork
Changing executable file	exec (execvp)
End process	exit
Wait for a child process( <b>can block</b> )	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 actual process changes form RUN to BLOCK
  - ▶ The selected to run process goes from READY to RUN

# Process creation

```
int fork();
```

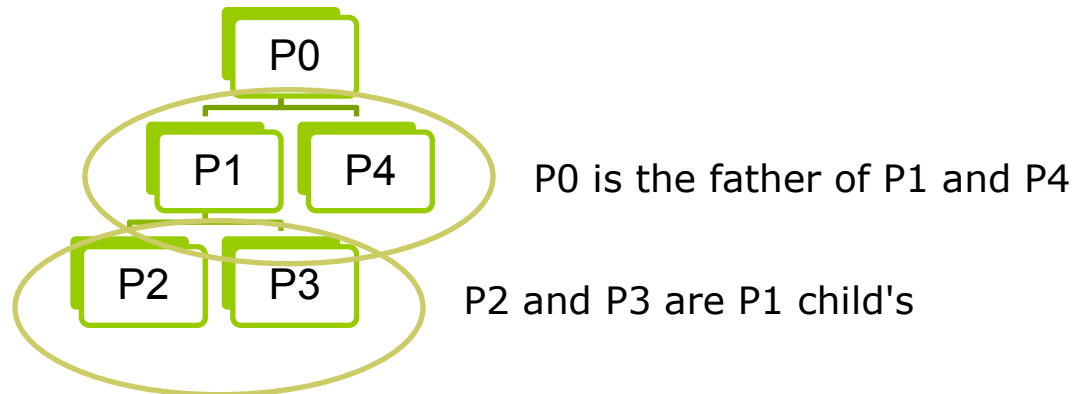


- “Creates a new process by duplicating the calling process. The new process, referred to as the *child*, is an exact duplicate of the calling process, referred to as the *parent*, except for some points:”
  - ▶ The child has its own unique process ID
  - ▶ The child's parent process ID is the same as 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

# 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

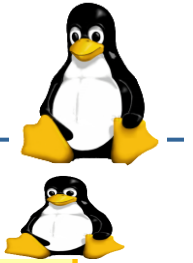
# Terminates process

```
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

# 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

# Changing the executable file



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



- 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...



shell

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

```
pid=fork();  
if (pid==0) execlp("p1","p1", (char *)null);  
waitpid(...);
```

```
p1.c", (char *)null);
```

System calls

Kernel

Reserve and initialize PCB

Reserve memory

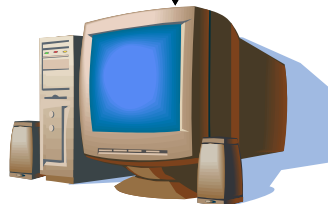
Copy program into  
memory

Assign cpu (when  
possible)

....

Release *resources*

Release PCB



# Process management

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

```
1. int ret=fork();
2. if (ret==0) {
3.     // These code lines are executed by the child, we have 2
    processes
4. }else if (ret<0){
5.     // The fork has failed. We have 1 process
6. }else{
7.     // These code lines are executed by the parent, we have 2
    processes
8. }
9. // These code lines are executed by the two processes
```

Ex 2: Both processes execute the same code lines

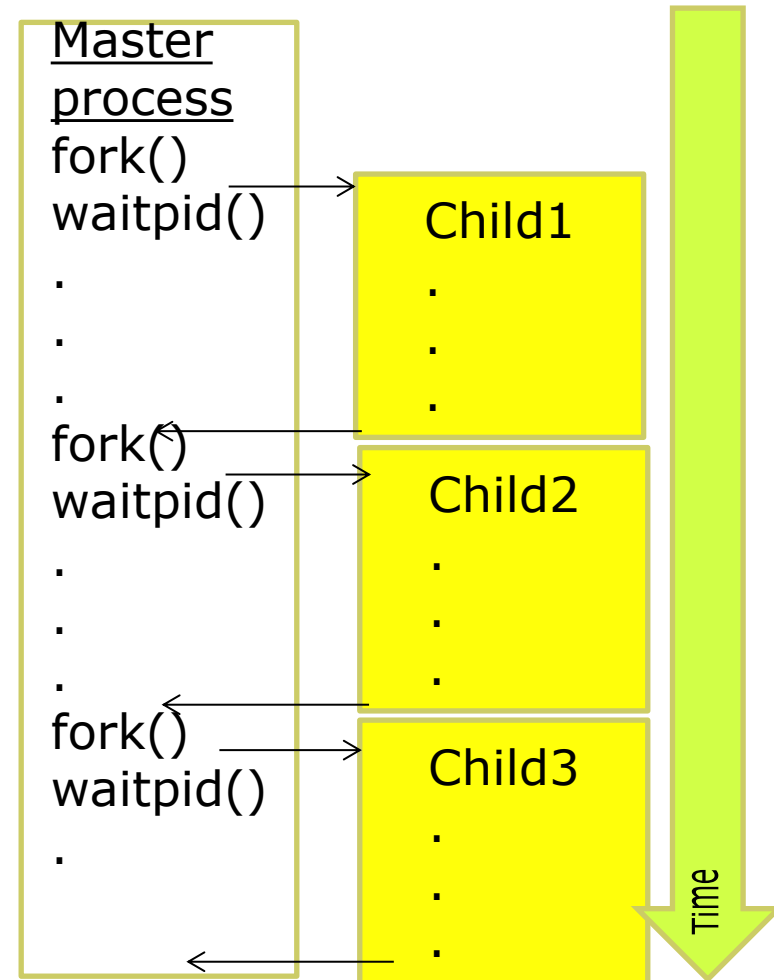
```
1. fork();
2. // We have 2 processes (if fork doesn't fail)
```



# Sequential scheme

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

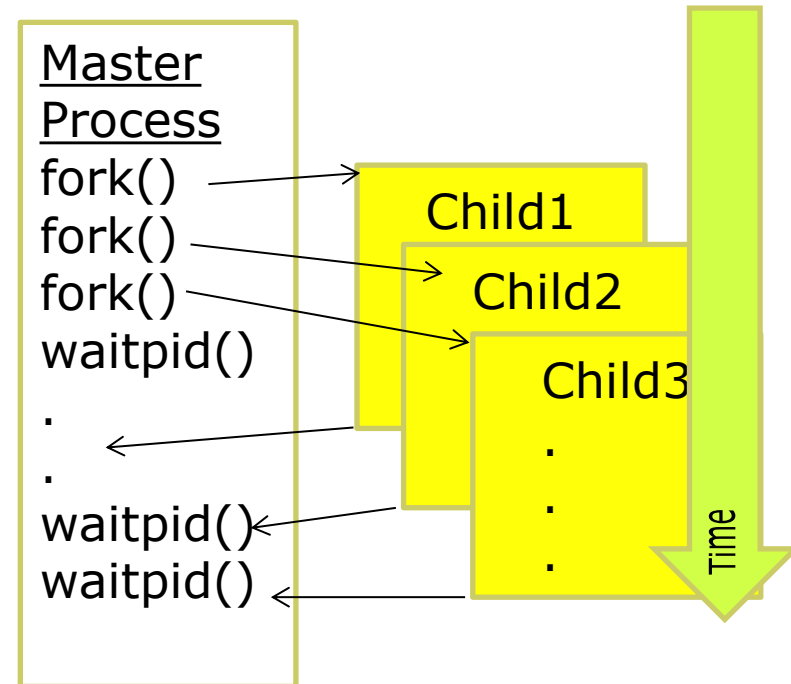
```
1. #define num_procs 3
2. int i,ret;
3. for(i=0;i<num_procs;i++){
4.     if ((ret=fork())<0) control_error();
5.     if (ret==0) {
6.         // CHILD
7.         ChildCode();
8.         exit(0);
9.     }
10.    waitpid(-1,NULL,0);
11.}
```



# Concurrent scheme

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

```
1. #define num_procs 3
2. int ret,i;
3. for(i=0;i<num_procs;i++){
4.     if ((ret=fork())<0) control_error();
5.     if (ret==0) {
6.         // CHOLD
7.         ChildCode();
8.         exit(0);
9.     }
10.}
11.while( waitpid(-1,NULL,0)>0);
```



# Examples



## ■ Analyze this code

```
1. int ret;  
2. char buffer[128];  
3. ret=fork();  
4. sprintf(buffer,"fork returns %d\n",ret);  
5. 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();
sprintf(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();
sprintf(buffer, "id1 value: %d; ret value %d", id1, ret);
write(1, buffer, strlen(buffer));
```



# Example (this is a real exam question)

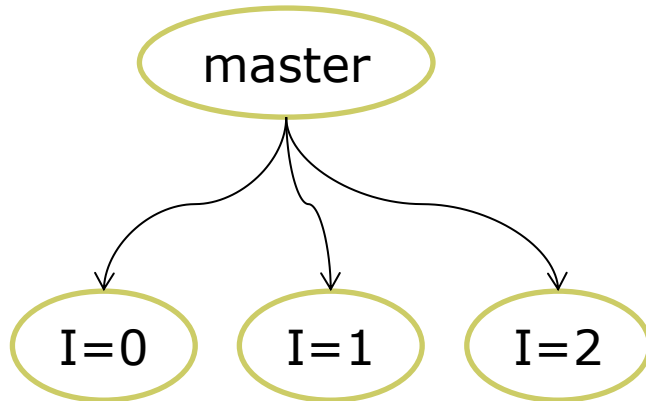


```
void main()
{   char buffer[128];
    ...
    sprintf(buffer,"My PID is %d\n", getpid());
    write(1,buffer,strlen(buffer));
    for (i = 0; i < 3; i++) {
        ret = fork();
        if (ret == 0)
            Dork();
    }
    while (1);
}

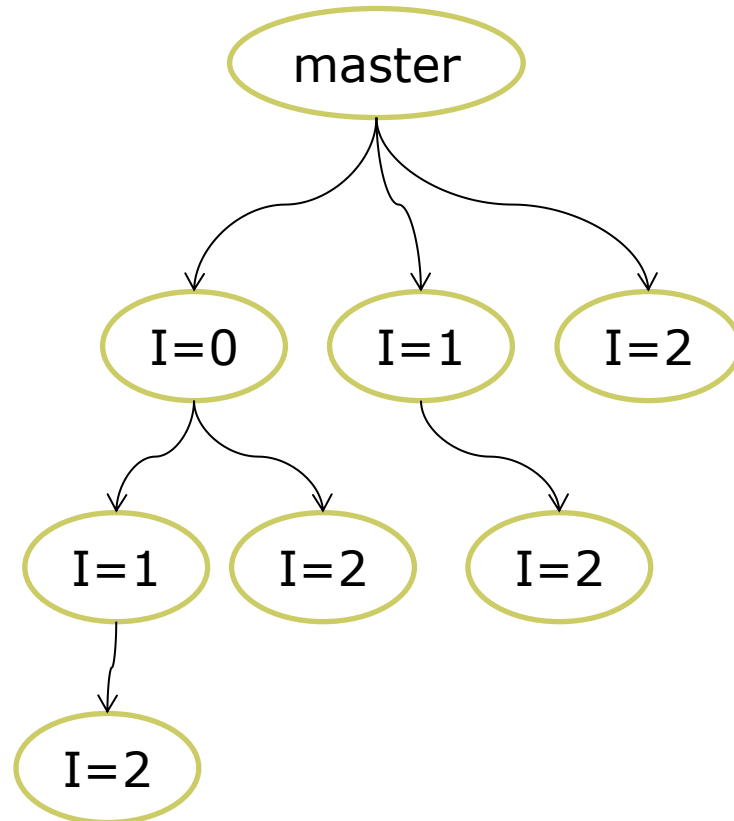
void Work()
{   char buffer[128];
    sprintf("My PIDies %d an my parent PID %d\n", getpid(), getppid());
    write(1,buffer,strlen(buffer));
    exit(0);   ← Now, remove this code line
}
```

# Process hierarchy

With exit system call

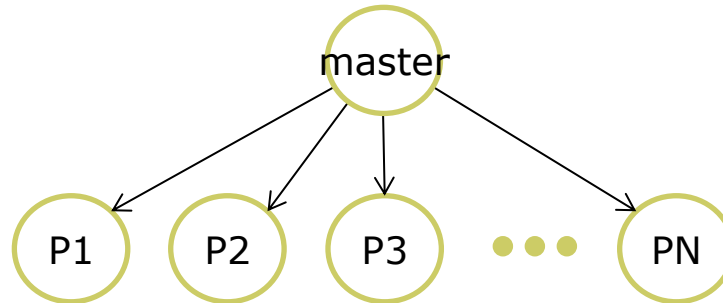


Without exit

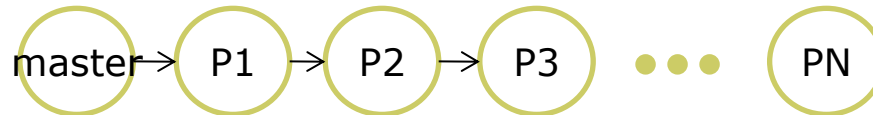


# More examples

- Write a C program that creates this process scheme:



- Modify the code to generate this new one :





# Example


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

progA

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


progB

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];  
...}
```

# Example

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


progA

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


progB

P1

P2



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



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

# 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

# Example: exit

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

A

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

It is get from PCB

kernel

PCB (process "A")

pid=...

exit\_code=

...

...

Exit status is stored at PCB

However, exit\_code value isn't 4. We have to apply some masks

# 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
...
```

# 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));
    }
}
```

---

# PROCESS COMMUNICATION

# 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 ☹



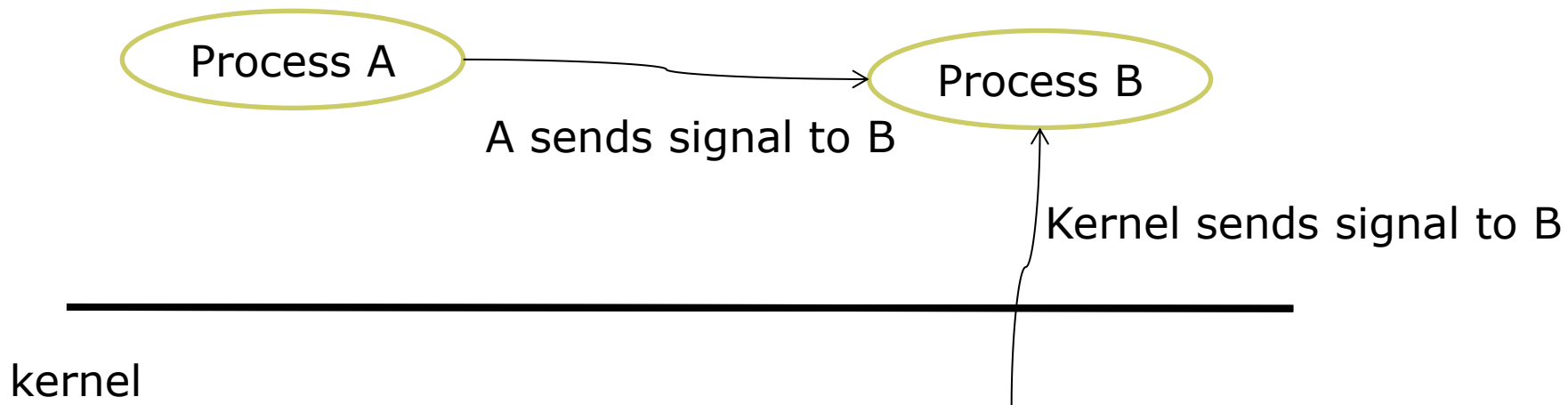
# IPC in Linux

---

- 
- **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

# Signals: Send and Receive

Usage: One process can receive signals from the kernel and from processes of the same user (including itself). One process can send signals to any process of the same user (including itself)



# Linux: Signals (3)

---

Description	System call
Catch a signal. Modifies the action associated to the signal reception.	signal
Sends a signal	kill
Blocks the process until a not ignored signal is received	pause
Programs the automatic delivering of signal SIGALARM after N seconds	alarm

- Test `kill -l` to see the list of signals

# Linux: Signals



- The kernel takes into account a list of predefined events, each one with an associated signal
  - ▶ Ex: SIGALRM, SIGSEGV, SIGINT, SIGKILL, SIGSTOP, ...
    - It is implemented internally like a unsigned int, but the kernel defines constants to make it easy the use
  - There are two valid but *undefined* signals: SIGUSR1 and SIGUSR2
- We will use them mainly for:
  - Time control
  - Process synchronization, waiting for an specific event

# Signal handling

```
sighandler_t signal(int signum, sighandler_t handler);
```



- The user can specify the action to be taken at signal reception
- Signum is one of the valid signal numbers: SIGALRM, SIGUSR1, etc
- *Handler* values:
  - SIG\_IGN: To ignore the signal
  - SIG\_DFL: To execute the default kernel action
  - Address of a programmer-defined function (a "signal handler")
- A signal handler has an specific API:
  - void function\_name(int sig\_num);
  - Where sig\_num is the signal number of the received signal
    - One signal handler can be used for more than one signal

# Signal handling: Default kernel actions

---

- Some signals are ignored by default
- Some signals cause the process ends
- Some signals cause the process ends and the full memory content is stored in a special file (called core)
- Some signals cause the process stop its execution

# Send signal to process

---

```
int kill(pid_t pid, int sig_num);
```



- pid is the PID of a valid process
- sig is a valid signal number: SIGALRM, SIGUSR1, etc

# Wait for an event

---

```
int pause();
```



- Causes the calling process to block until a not ignored signal is received.
- WARNING: The signal must be received after the process blocks, otherwise, the process will block indefinitely.



# Alarm clock



```
unsigned int alarm(unsigned int sec);
```

- After *sec* seconds, a SIGALRM is sent to the calling process
- If *sec* is zero, no new **alarm()** is scheduled.
- Any call to alarm cancels any previous clock
- It returns the number of seconds remaining until any previously scheduled alarm was due to be delivered

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

# Examples

- Process A sends a signal SIGUSR1 to process B. Process B executes function f\_sigusr1 when receiving it

## Process A

```
.....  
Kill( pidB, SIGUSR1);  
.....
```

## Process B

```
void f_sigusr1(int s)  
{  
...  
}  
int main()  
{  
signal(SIGUSR1,f_sigusr1);  
....  
}
```

It is similar to an interrupt handler. If f\_sigusr1 doesn't include an exit system call, process B will be interrupted and then it will continue its execution

# Examples

- Process A sends SIGUSR1 to process B. Process B waits for signal reception. What happens at each process B option?

## Process A

```
.....  
Kill( pidB, SIGUSR1);  
....
```

## Process B:option1

```
void f_sigusr1(int s)  
{  
  //f_sigusr1 code  
}  
int main()  
{  
  signal(SIGUSR1,f_sigusr1);  
  ....  
  pause();  
  ....  
}
```

## Process B:option2

```
void f_sigusr1(int s)  
{//empty}  
int main()  
{  
  signal(SIGUSR1,f_sigusr1);  
  ....  
  pause();  
  ....  
}
```

## Process B:option3

```
int main()  
{  
  ....  
  pause();  
  ....  
}
```

# Signals: Kernel internals

---



- Signal management is per-process. Information management is stored at PCB. Main data structures:
  - Table with per-signal defined actions
  - 1 bitmap of pending signals (1 bit per signal)
  - 1 alarm clock
    - ▶ If a process programs two times the alarm clock, the last one cancels the first one
  - 1 Bitmask that indicates enabled/disabled signals

# Example: alarm clock



```
void main()
{   char buffer[128];
    ...
    //That means: execute function f_alarma at signal SIGALRM reception
    signal(SIGALRM, f_alarma);
    // We ask to the kernel for the automatic sending of
    // SIGALRM after 1 second
    alarm(1);

    while(1) {
        sprintf(buffer, "Doing some work\n");
        write(1, buffer, strlen(buffer));
    }
}

void f_alarma(int s)
{
    char buffer[128];
    sprintf(buffer, "TIME, 1 second has passed!\n");
    write(1, buffer, strlen(buffer));
    // If we want to receive another SIGALRM, we must re-program the
    // alarm clock
    alarm(1);
}
```

You can find this code in <http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EjemplosProcesos.tar.gz>

# Some signals

---

Name	Default	Means...
<b>SIGCHLD</b>	IGNORE	A child process has finish
<b>SIGSTOP</b>	STOP	Stop process
<b>SIGTERM</b>	FINISH	Interrupted from keyboard (Ctr-C)
<b>SIGALRM</b>	FINISH	Alarm clock has reached zero
<b>SIGKILL</b>	FINISH	Finish the process
<b>SIGSEGV</b>	FINISH+CORE	Invalid memory reference
<b>SIGUSR1</b>	FINISH	User defined
<b>SIGUSR2</b>	FINISH	User defined

# Signals and process management

---

- FORK: New process but same executable
  - Table with per-signal defined actions: duplicated
  - 1 bitmap of pending signals: reset
  - 1 alarm clock: reset
  - 1 Bitmask that indicates enabled/disabled signals: duplicated
- EXECLP: Same process, new executable
  - Table with per-signal defined actions: reset
  - 1 bitmap of pending signals: not modified
  - 1 alarm clock: not modified
  - 1 Bitmask that indicates enabled/disabled signals: not modified

# Process synchronization

---

- Process block (pause)
  - Programmer must guarantee the signal will be received after the block
  - It is efficient, the process doesn't use the CPU just to wait for something
- Otherwise....Busy waiting
  - The process will wait by checking some kind of boolean or counter variable.
  - It isn't efficient, however, some times is the only way since we can block indefinitely when using pause



# Example: Waiting for process....and doing work

---

- Requirements:
  - We have to create a process every 10 seconds → alarm + wait for alarm (that means some kind of “wait for the alarm clock ends”)
  - We have to get process exit code as fast as they finish to show a message with exit status → (that means “wait for process finalization”)
- If we have two “blocks”, we have to do in a different way than using pause or waitpid
- We can use:
  - Busy waiting for alarm clock
  - SIGCHLD for process finalization capture

# Example: waiting for pocesses

```
int clock=0; // It MUST be global
void main()
{   int pid,exit_status;
    // We capture SIGALRM and SIGCHLD signals
    signal(SIGALRM,f_alarm);
    signal(SIGCHLD,end_child);
    alarm(10);
    for(work=0;work<10;work++){
        //BUSY WAITING
        while (clock==0);
        clock=0;
        do_work();
    }
    // Once we finish, we can wait processes using blocking waitpid
    signal(SIGCHLD,SIG_IGN);
    while((pid=waitpid(-1,&exit_status,0))>0)
        TrataExitCode(pid,exit_status);
}
```

# Example: waiting for pocesses

```
// Alarm clock function, we just program the next one
void f_alarm()
{
    clock=1;
    alarm(10);
}
// The kernel send a SIGCHLD signal, when a child process ends...
// however, it is not so easy
void end_child(int s)
{
    int exit_status,pid;
    while((pid=waitpid(-1,&exit_status,WNOHANG))>0)
        TrataExitCode(pid,exit_status);
}
// Work creation
void do_work()
{
    int ret;
    ret=fork();
    if (ret==0){
        execute_work();
        exit();
    }
}
```

- 
- Data structures
  - Scheduling policies
  - Kernel mechanisms for process management

# **KERNEL INTERNALS**

# Kernel Internals

---

- Main data type to store per-process information → Process Control Block (PCB)
- Data structures to group or organize PCB's, usually based on their state
  - In a general purpose system, such as Linux, Data structures are typically queues, multi-level queues, lists, hash 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: to decide which process/processes must run, how many time, and the mapping to cpus (if required)
- Mechanism to make effective scheduling decisions

# 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>

# 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

# 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 our 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)



# 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

# 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

# Process characterization

---

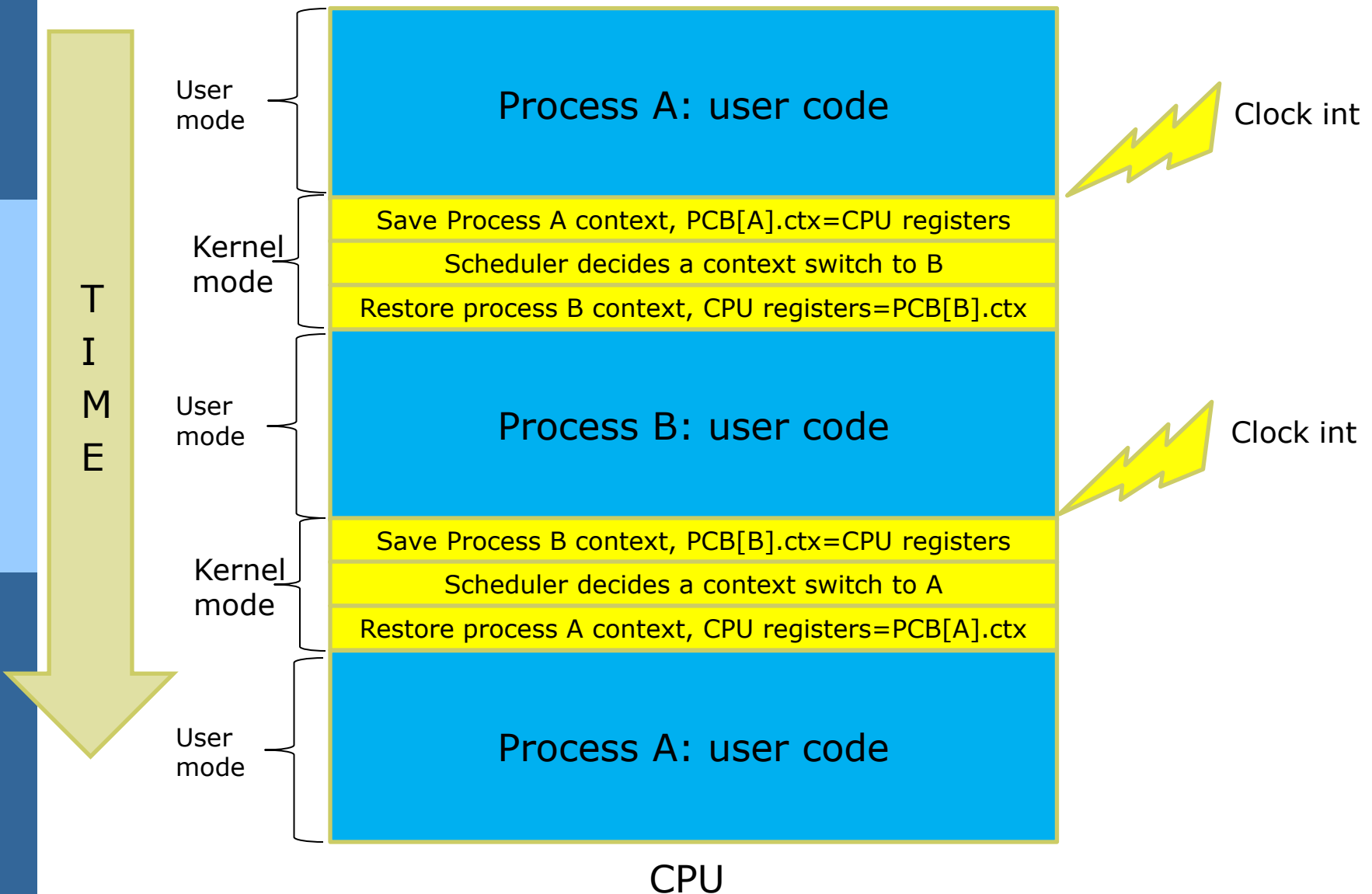
- Processes alternates the cpu usage with the devices accesses
- These periods are called cpu bursts and I/O bursts
- Based on the ratio of number and duration of the different burst we will refer to:
  - CPU processes. They spent more time on CPU bursts than on I/O bursts
  - I/O processes. They spent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

# 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 !!

# Context switch



# 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



---

## **PUTTING ALL TOGETHER**

# 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 physical 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

# 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 or 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.

# 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>