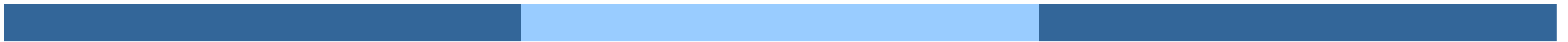


# T2-Processos



# Índex

---

- Conceptes relacionats amb la Gestió de processos
- Serveis bàsics per a gestionar processos (basat en Unix)
- Comunicació entre processos
  - Signals Linux i Sincronització
- Gestió interna dels processos
  - Dades: \*PCB
  - Estructures de gestió: Llistes, cues, etc., relacionades principalment amb l'estat del procés
- Mecanisme de canvi de context. Concepte i passos bàsics
- Planificació
- Relació entre les crides a sistema de gestió de processos i la gestió interna del S.O.
- Protecció i seguretat



---

Definició

Tasques del sistema operatiu

Concurrencia i paral·lelisme

Estats dels processos

Propietats d'un procés en Unix

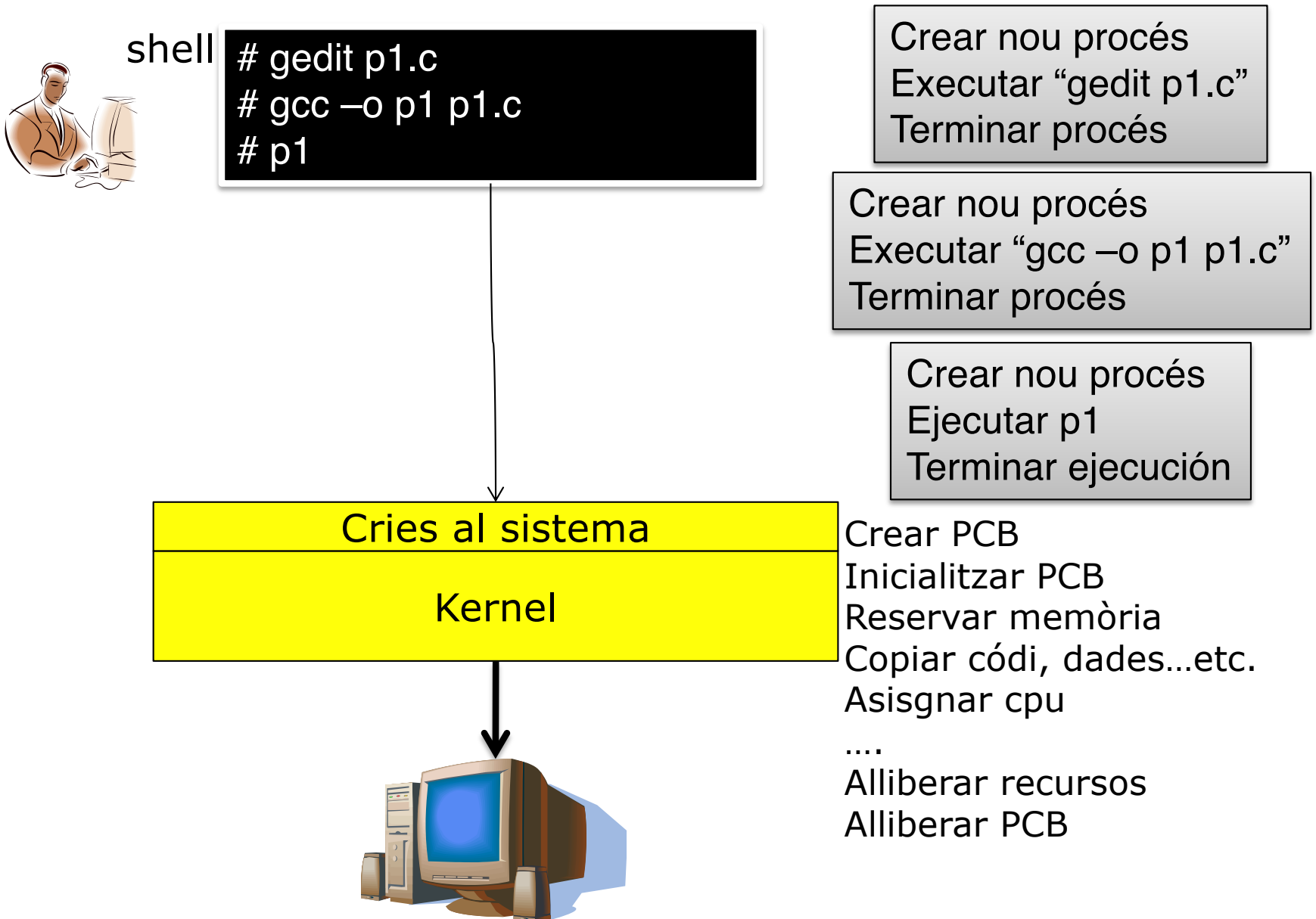
# CONCEPTES

# Concepte de procés

---

- Un procés és la representació de l'SO d'un programa en execució.
- Un programa executable bàsicament és un codi i una definició de dades, en posar-lo en execució necessitem:
  - Assignar-li memòria per al codi, les dades i la pila
  - Inicialitzar els registres de la cpu perquè es comenci a executar
  - Oferir accés als dispositius (ja que necessiten accés en manera kernel)
  - Moltes més coses que anirem veient
- Per a gestionar la informació d'un procés, el sistema utilitza una estructura de dades anomenada PCB (Process Control Block)
- Cada vegada que posem un programa a executar, es crea un nou procés
  - Poden haver-hi limitacions en el sistema

# Processos: Com es fa?



# Process Control Block (PCB)

---

- Conté la informació que el sistema necessita per a gestionar un procés. Aquesta informació depèn del sistema i de la màquina. Es pot classificar en aquests 3 grups
  - Espai d'adreces
    - ▶ descripció de les regions del procés: codi, dades, pila, ...
  - Context d'execució
  - SW: PID, informació per a la planificació, informació sobre l'ús de dispositius, estadístiques,...
  - HW: taula de pàgines, program counter, ...

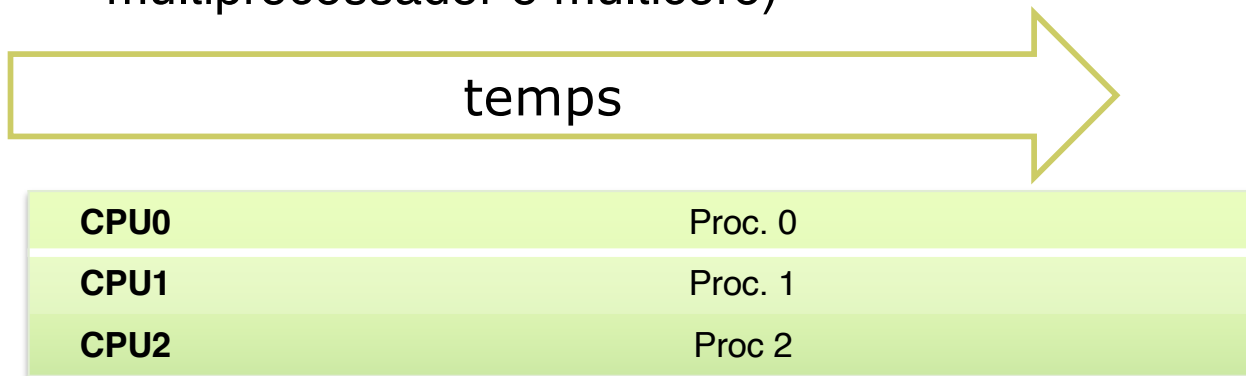
# Utilització eficient de la CPU

---

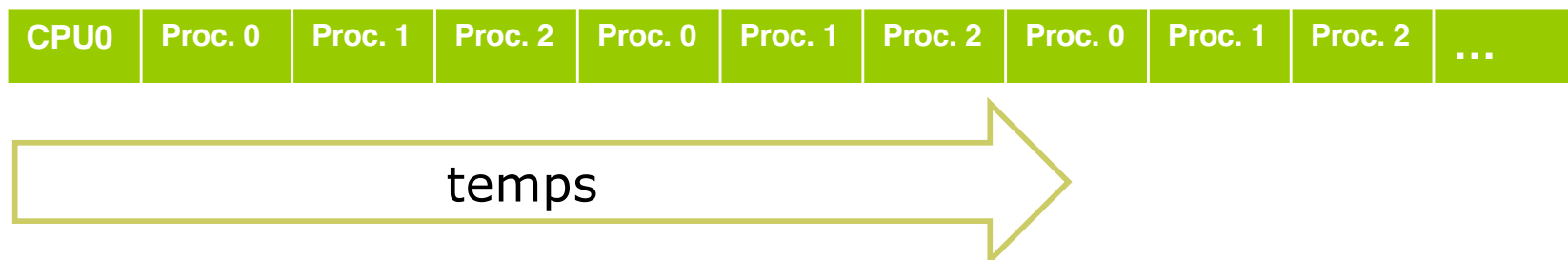
- En un sistema de propòsit general, l'habitual és tenir diversos processos alhora, de manera que s'aprofiti al màxim els recursos de la màquina
- Per què ens pot interessar executar múltiples processos simultàniament?
  - Si tenim diversos processadors podem executar més processos alhora, o un mateix usant diversos processadors
  - Aprofitar el temps d'accés a dispositius (Entrada/Sortida) d'un procés perquè altres processos usin la CPU
- Si el SO ho gestiona bé, aconsegueix la il·lusió que la màquina té més recursos (CPU) dels quals té realment

# Concurrencia

- Concurrencia és la capacitat d'executar diversos processos de manera simultània
  - Si realment hi ha diversos allora és paral·lelisme (arquitectura multiprocessador o multicore)



- Si és l'SO el qual genera un paral·lelisme virtual mitjançant compartició de recursos es parla de concurrència





# Concurrencia

---

- Es diu que diversos processos són concurrents quan **tenen la capacitat d'executar-se en paral·lel** si l'arquitectura ho permet
- Es diu que diversos processos són seqüencials si, independentment de l'arquitectura, s'executaran un després de l'altre (quan un acaba comença el següent). En aquest cas, és el programador el que força que això sigui així mitjançant sincronitzacions..
  - ▶ Posant un waitpid entre un fork i el següent
  - ▶ Mitjançant signals (esdeveniments)
- **Paral·lelisme** és quan diversos processos concurrents s'executen de forma simultània:
  - Depèn de la màquina
  - Depèn del conjunt de processos
  - Depèn de l'SO

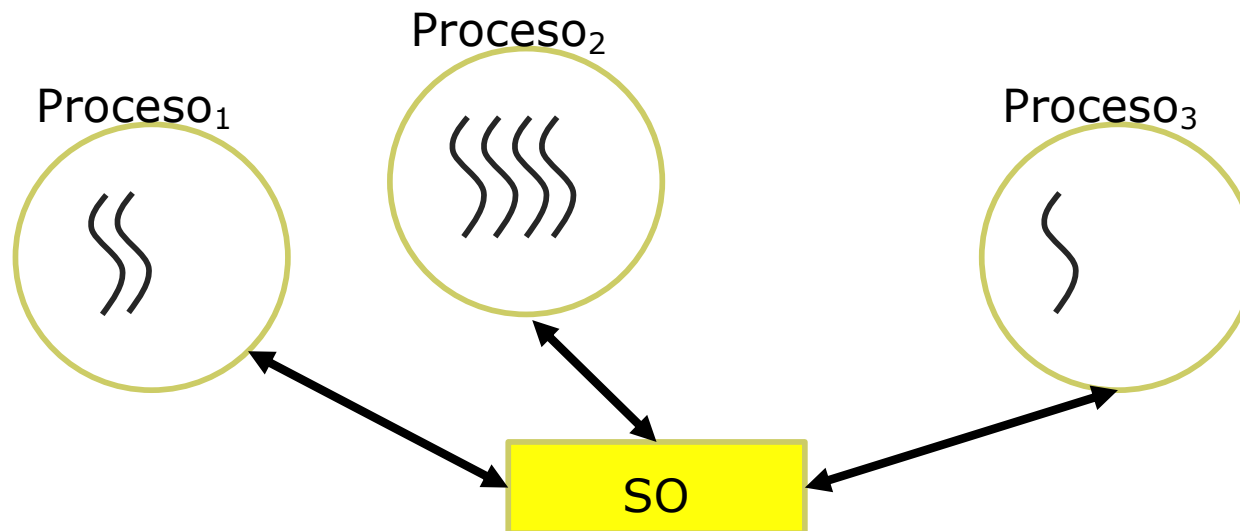
# Fils d'Execució (Threads) – Què són?

---

- Si entrem més detalladament en el concepte de Procés...
  - representació de l'SO d'un programa en execució
- ...podem afirmar que un **procés** és la unitat d'assignació de recursos d'un programa en execució (memòria, dispositius E/S, threads)
- Entre els recursos hi ha el/els fil/s d'execució (thread) d'un procés
  - Es tracta de la instància/flux d'execució d'un procés i és la mínima unitat de planificació del SO (assignació de temps de CPU)
    - ▶ Cada part del codi que es pot executar de forma independent se li pot associar un thread
  - Té associat el context necessari per seguir el flux d'execució de les instruccions
    - ▶ Identificador (Thread ID: TID)
    - ▶ Punter a Pila (Stack Pointer)
    - ▶ Punter a següent instrucció a executar (Program Counter),
    - ▶ Registres (Register File)
    - ▶ Variable `errno`
  - Els threads comparteixen els recursos del procés (PCB, memòria, dispositius E/S)

# Fils d'Execució (Threads)

- Un procés té un thread a l'inici de la execució
- Un procés pot tenir diversos threads
  - Ex.: videojocs actuals d'altres prestacions tenen **>50 threads**; Firefox/Chrome tenen **>80 threads**
- A la figura següent: Procés<sub>1</sub> té 2 threads; el Procés<sub>2</sub> té 4 threads; el Procés<sub>3</sub> té 1 thread
- La gestió de processos amb diversos threads dependrà del suport del SO
- User Level Threads vs Kernel Level Threads



# Fils d'Execució (Threads) – Per a què?

---

- Quan i per què es fan servir...
  - Explotar paral·lelisme (del codi i de recursos maquinari)
  - Encapsular tasques (programació modular)
  - Eficiència a l'E/S (Threads específics per a E/S)
  - Pipelining de sol·licituds de servei (per mantenir QoS de serveis)
  
- Avantatges
  - Els threads tenen menor cost en crear/acabar i en canviar de context (dins del mateix procés) que els processos
  - En compartir memòria entre threads d'un mateix procés, podeu intercanviar informació sense trucades al sistema
  
- Desavantatges
  - Difícil de programar i depurar a causa de la memòria compartida
  - Problemes de sincronització i exclusió mútua
  - Execucions incoherents, resultats erronis, bloquejos, etc.

# Estats d'un procés

---

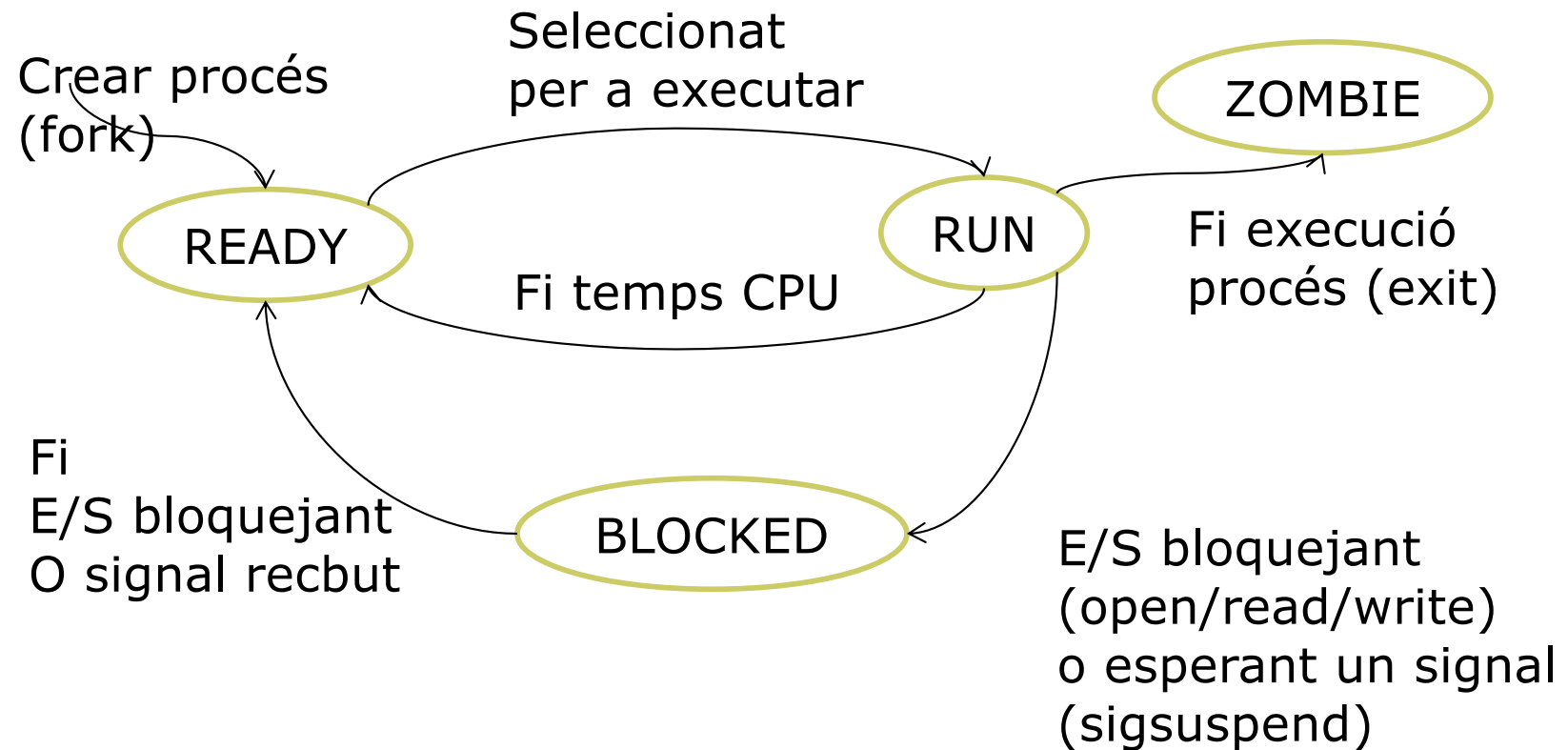
- No és normal que un procés estigui tot el temps utilitzant la CPU, durant períodes de la seva execució el podem trobar:
  - Esperant dades d'algun dispositiu: teclat, disc, xarxa, etc
  - Esperant informació d'altres processos
- Per aprofitar el HW tenim:
  - Sistemes multi programats: múltiples processos actius. Cada procés té la vostra informació al vostre PCB.
  - L'SO només assigna CPU a aquells processos que l'utilitzen.
- L'SO té “classificats” els processos en funció de “què estan fent”, normalment això s'anomena “ESTAT” del procés
- **L'estat** sol gestionar-se o amb un camp al PCB o tenint diferents llistes o cues amb els processos en un estat concret.

# Estats d'un procés

---

- Cada SO defineix un graf d'estats, indicant quins esdeveniments generen transicions entre estats.
- El graf defineix quines transicions són possibles i com es passa d'un a l'altre
- El graf d'estats, molt simplificat, podria ser:
  - run: El procés té assignada una cpu i s'està executant
  - ready: El procés està preparat per executar-se però està esperant que se li assigni una CPU
  - blocked: El procés no té/consum CPU, està bloquejat esperant un que finalitzi una entrada/sortida de dades o l'arribada d'un esdeveniment
  - zombie: El procés ha acabat la seva execució però encara no ha desaparegut de les estructures de dades del nucli
    - ▶ Linux

# Exemple diagrama d'estats



Els estats i les transicions entre ells depenen del sistema. Aquest diagrama és només un exemple.

# Exemple estats d'un procés

---

- Objectiu: Cal entendre la relació entre les característiques de l'SO i el diagrama d'estats que tenen els processos
  - Si el sistema és multi programat → READY, RUN
  - Si el sistema permet E/S bloquejant → BLOCKED
  - Etc.
  
- SO amb suport per a processos amb múltiples threads
  - Estructures per diferenciar threads del mateix procés i gestionar estats d'execució, entre altres coses
  - Ex.: Light Weight Process (LWP) en SO basats en Linux/UNIX



# Linux: Propietats d'un procés

---

- Un procés inclou, no només el programa que executa, sinó tota la informació necessària per a diferenciar una execució del programa d'una altra.
  - Tota aquesta informació se emmagatzema dins el kernel, al PCB.
- En Linux, per exemple, les propietats de un procés se agrupen en tres: **la identitat, l'entorn, i el context.**
- **Identitat**
  - Defineix qui és (identificador, propietari, grup) i què pot fer el procés (recursos als quals pot accedir)
- **Entorno**
  - Paràmetres (argv a un programa en C) i variables d'entorn (HOME, PATH, USERNAME, etc.)
- **Context**
  - Tota la informació que defineix l'estat del procés, tots els seus recursos que utilitza i que ha fet servir durant la seva execució.

# Linux: Propietats d'un procés (2)

---

- La IDENTITAT del procés defineix qui és i per tant determina què pot fer
  - Process ID (PID)
    - ▶ És un identificador únic per al procés. S'utilitza per identificar un procés dins del sistema. En crides al sistema identifica el procés al qual volem enviar un signal, modificar, etc.
    - ▶ El kernel en genera un de nou per a cada procés que es crea
  - Credencials
    - ▶ Cada procés està associat amb un usuari (userID) i un o més grups (groupID). Aquestes credencials determinen els drets del procés a accedir als recursos del sistema i dels fitxers.

- 
- Creació
  - Mutació (càrrega d'un executable nou)
  - Finalització
  - Espera

## **SERVEIS BÀSICS PER A GESTIONAR PROCESSOS**

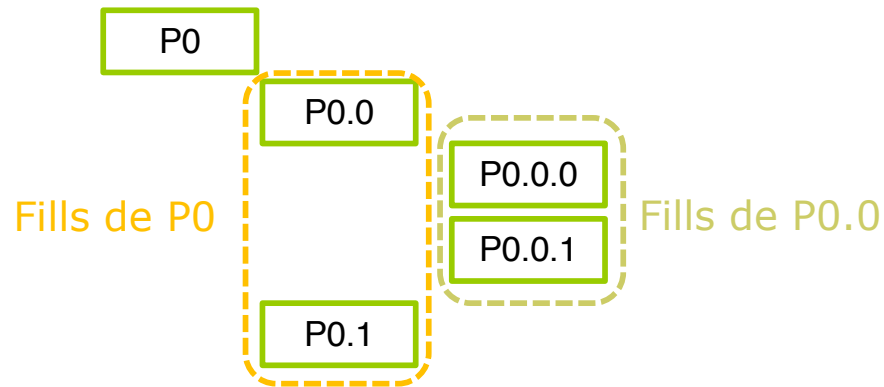
# Serveis i funcionalitat

---

- El sistema ens ofereix com a usuaris un conjunt de funcions (crides a sistema) per gestionar processos
  - Crear/Planificar/Eliminar processos
  - Bloquejar/Desbloquejar processos
  - Proporcionar mecanismes de sincronització
  - Proporcionar mecanismes de comunicació
    - ▶ Memòria compartida
    - ▶ Dispositius especials
    - ▶ Gestió de signals
- En aquest curs NO entrarem en detall en crides al sistema vinculades als threads

# Creació de processos

- Quan un procés en crea un altre, s'estableix una relació jeràrquica que s'anomena pare-fill. Alhora, el procés fill (i el pare) podrien crear altres processos generant-se un arbre de processos.



- Els processos s'identifiquen al sistema mitjançant un process identifier (PID)
- El SO decideix aspectes com ara:
- Recursos: El procés fill, comparteix els recursos del pare?
- Planificació: El procés fill s'executa abans que el pare?
- Espai d'adreces. Quin codi executa el procés fill? El mateix? Un altre?

# Creació de processos: opcions(Cont)

---

## ■ Planificació

- El pare i el fill s'executen concurrentment (UNIX)
- El pare espera fins que el fill acaba (es sincronitza)

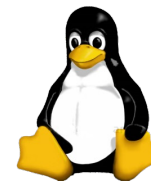
## ■ Espai d'adreces (rang de memòria vàlid)

- El fill és un duplicat del pare (UNIX), però cadascú té la seva pròpia memòria física. A més, el pare i el fill, en el moment de la creació, tenen el mateix context d'execució (els registres de la CPU valen el mateix)
- El fill executa un codi diferent

## ■ UNIX

- `fork system call`. Crea un nou procés. El fill és un clon del pare
- `exec system call`. Reemplaça (muta) l'espai d'adreces del procés amb un programa nou. El procés és el mateix.

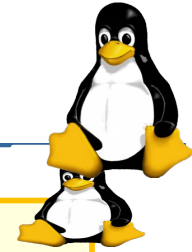
# Serveis bàsics (UNIX)



| Servei                                      | Crida a sistema |
|---|-----------------|
| Crear procés                                | fork            |
| Canviar executable=Mutar procés             | exec (execlp)   |
| Terminar procés                             | exit            |
| Esperar a procés fill ( <b>bloquejant</b> ) | wait/waitpid    |
| Retorna el PID del procés                   | getpid          |
| Retorna el PID del pare del procés          | getppid         |

- Una crida a sistema bloquejant és aquella que pot bloquejar el procés, és a dir, forçar que deixi l'estat RUN (abandona la CPU) i passi a un estat en què no es pot executar (WAITING, BLOCKED, ....., depèn del sistema)

# Crear procés: fork en UNIX

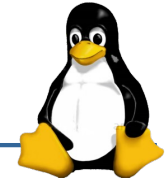


```
int fork();
```

- Un procés crea un procés nou. Es crea una relació jeràrquica pare-fill
- El pare i el fill s'executen de manera concurrent
- La memòria del fill s'inicialitza amb una còpia de la memòria del pare
- Codi/Dades/Pila
- El fill inicia l'execució al punt en què hi havia el pare en el moment de la creació
  - Program Counter fill = Program Counter pare
- Valor de retorn del fork és diferent (és la forma de diferenciar-los al codi):
  - Pare rep el PID del fill
  - Fill rep un 0.



# Creació de processos i herència



- El fill HERETA alguns aspectes del pare i d'altres no.
- HERETA (rep una còpia privada de....)
  - L'espai d'adreces lògic (codi, dades, pila, etc.).
    - ▶ La memòria física és nova, i conté una còpia de la del pare (al tema 3 veurem optimitzacions en aquest punt)
  - La taula de programació de signals
  - Els dispositius virtuals
  - L'usuari/grup (credencials)
  - Variables d'entorn
- NO HERETA (sinó que s'inicialitza amb els valors corresponents)
  - PID, PPID (PID del pare)
  - Comptadors interns d'utilització (Accounting)
  - Alarmes i signals pendents (són pròpies del procés)

# Terminar execució/Esperar que termini

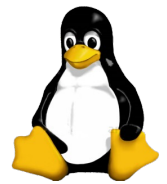


- Un procés pot acabar la seva execució voluntària (exit) o involuntàriament (signals)
- Quan un procés vol finalitzar la seva execució (voluntàriament), alliberar els seus recursos i alliberar les estructures de kernel reservades per a ell, s'executa la crida a sistema `exit`.
- Si volem sincronitzar el pare amb la finalització del fill, podem fer servir `waitpid`: el procés espera (si cal es bloqueja el procés) que acabi un fill qualsevol o un de concret
  - `waitpid(-1, NULL, 0)` → Esperar (amb bloqueig si cal) un fill qualsevol
  - `waitpid(pid_fill, NULL, 0)` → Esperar (amb bloqueig si cal) un fill amb `pid=pid_fill`
- El fill pot enviar informació de finalització (exit code) al pare mitjançant la crida a sistema `exit` i el pare la recull mitjançant `wait` o `waitpid`
  - El SO fa d'intermediari, l'emmagatzema fins que el pare la consulta
  - Mentre el pare no la consulta, el PCB no s'allibera i el procés es queda en estat ZOMBIE (defunct)
    - ▶ Convé fer `wait/waitpid` dels processos que creem per alliberar els recursos ocupats del nucli
  - Si un procés mor sense alliberar els PCB dels seus fills el procés init del sistema els allibera

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



# Mutació d'executable: exec en UNIX



- Quan feu `fork`, l'espai d'adreces és el mateix. Si volem executar un altre codi, el procés ha de **MUTAR** (Canviar el binari d'un procés)
- `exec1p`: Un procés canvia (muta) el seu propi executable per un altre executable (però el procés és el mateix)
  - **Tot el contingut de l'espai de adreces** canvia, codi, dades, pila, etc.
    - Es reinicia el comptador de programa a la primera instrucció (`main`)
  - **Es manté** tot allò relacionat amb la identitat del procés
    - Comptadors d'ús interns, signals pendents, etc.
  - Es modifiquen aspectes relacionats amb l'executable o l'espai d'adreces
    - Es defineix per defecte la taula de programació de signals

```
int execlp(const char *file, const char *arg, ...);
```





---

# **EXEMPLES GESTIÓ PROCESSOS**

# Processos: Ara ja sabem com es fa



shell

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

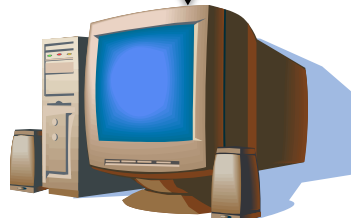
```
pid=fork();  
if (pid==0) execlp("p1","p1", (char *)null;  
waitpid(...);
```

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

crida a sistema

Kernel

Crear PCB  
Inicialitzar PCB  
Reservar memòria  
Copiar codi, dades,...etc.  
Assignar CPU  
....  
Alliberar recursos  
Alliberar PCB



# Creació processos

Cas 1: volem que facin línies de codi diferent

```
1. int ret=fork();
2. if (ret==0) {
3.     // aquestes línies només les executa el fill, tenim 2
    processos
4. }else if (ret<0){
5.     // En aquest cas ha fallat el fork, només hi ha 1 procés
6. }else{
7.     // aquestes línies només les executa el pare, tenim 2
    processos
```

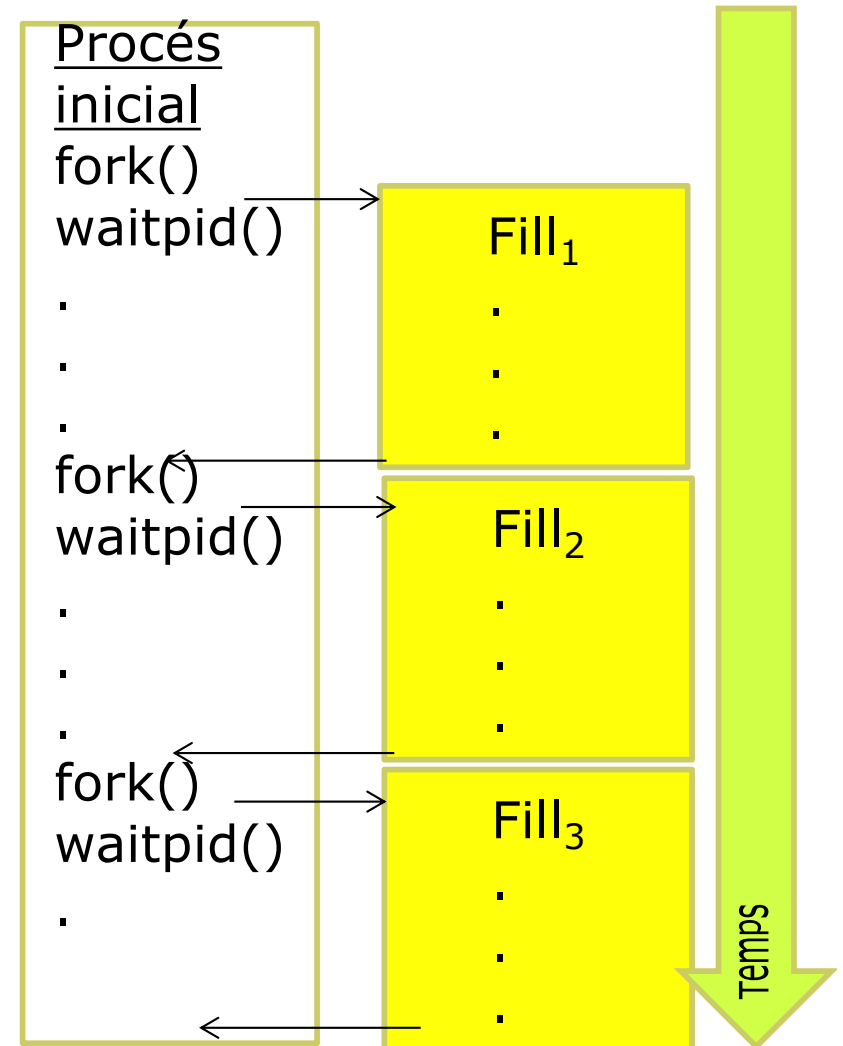
Caso 2: volem que facin el mateix

```
1. fork();
2. // Aquí, si no hi ha error, hi ha 2 processos
```

# Esquema Seqüencial

Seqüencial: forcem que el pare esperi que acabi un fill abans de crear el següent.

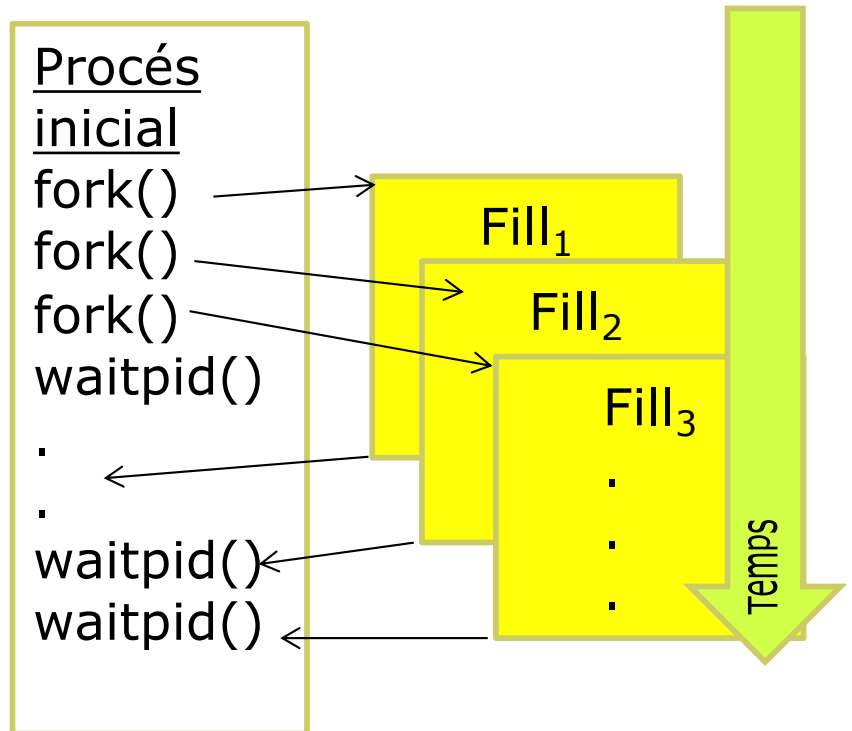
```
1. #define num_procs 2
2. int i,ret;
3. for(i=0;i<num_procs;i++){
4.     if ((ret=fork())<0) control_error();
5.     if (ret==0) {
6.         // aquestes línies només les
7.         // executa el fill
8.         codifill();
9.         exit(0); //
10.    }
11.    waitpid(-1,NULL,0);
12.}
```



# Esquema Concurrent

Concurrent; Primer creem tots els processos, que s'executen concurrentment, i després esperem que acabin.

```
1. #define num_procs 2
2. int ret,i;
3. for(i=0;i<num_procs;i++){
4.     if ((ret=fork())<0) control_error();
5.     if (ret==0) {
6.         // aquestes línies només
7.         // les executa el fill
8.         codigofill);
9.         exit(0); //
10.    }
11.}
12.while( waitpid(-1,NULL,0)>0);
```

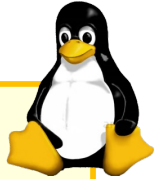




# Exemple amb fork

## ■ Què farà aquest codi?

```
1. int id;  
2. char buffer[128];  
3. id=fork();  
4. sprintf(buffer,"fork devuelve %d\n",id);  
5. write(1,buffer,strlen(buffer));
```



## ■ Què farà si el fork funciona?

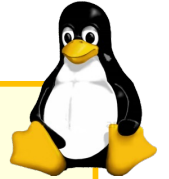
## ■ Què farà si el fork falla?

## ■ PROVEU-LO!!

# Exemples amb fork

- Quants processos es creen en aquest fragment de codi?

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



- I en aquest altre fragment?

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

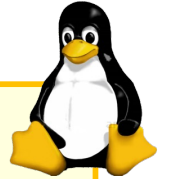


- Quin arbre de processos es genera?

# Exemples amb fork

- Si el pid del procés pare val 10 i el del procés fill val 11

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



- Quin missatge veurem en pantalla?
- I ara?

```
int id1, ret;
char buffer[128];
id1 = getpid(); /* getpid retorna el pid del proces que l executa */
ret = fork();
id1 = getpid();
sprintf(buffer, "Valor d id1: %d; valor de ret: %d", id1, ret);
write(1, buffer, strlen(buffer));
```



# Exemple: fork/exit (examen)



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

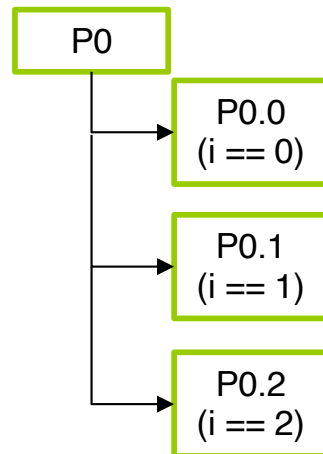
void ferTasca()
{   char buffer[128];
    sprintf("Meu PID es %d i el de mon pare %d\n", getpid(), getppid());
    write(1, buffer, strlen(buffer));
    exit(0);
```

**← Ara, proveu a treure aquesta instrucció,  
és molt diferent!!!**

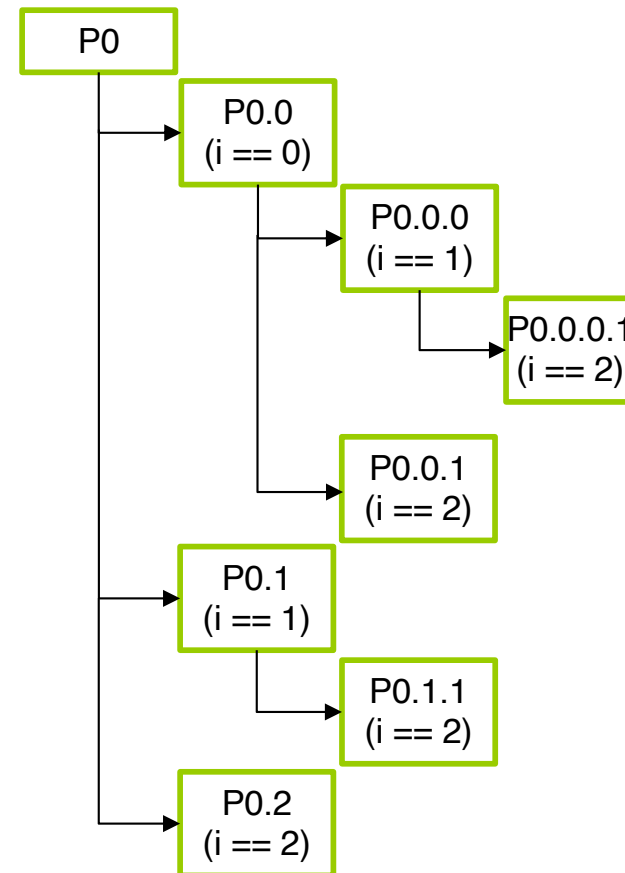
*Podeu trobar el codi complet a: Nprocesos.c y NprocesosExit.c*

# Arbre de processus (examen)

Amb exit

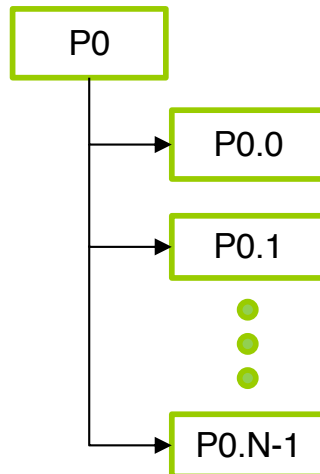


Sense exit

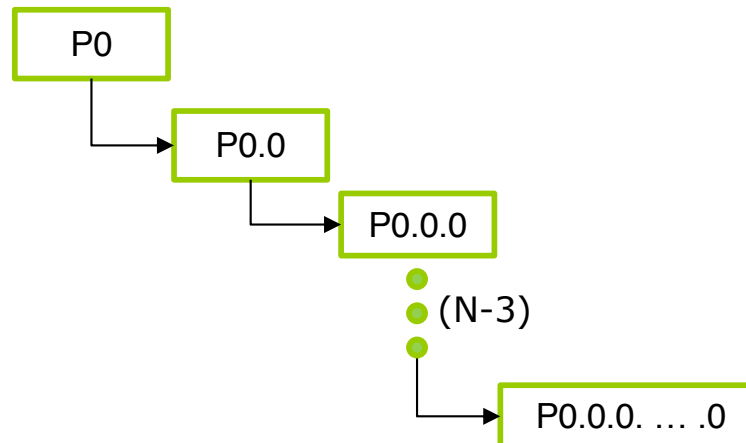


# Altres exemples amb fork

- Escriuiu un programa que creï N processos segons el següent arbre de processos:



- Modifiquen el codi anterior per a que els creï segons aquest altre arbre de processos:



# Example fork+exec

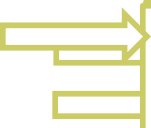
```
fork();  
execlp("/bin/progB", "progB", (char *)0);  
while(...)
```

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];  
...}
```

# Exemple fork+exec

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

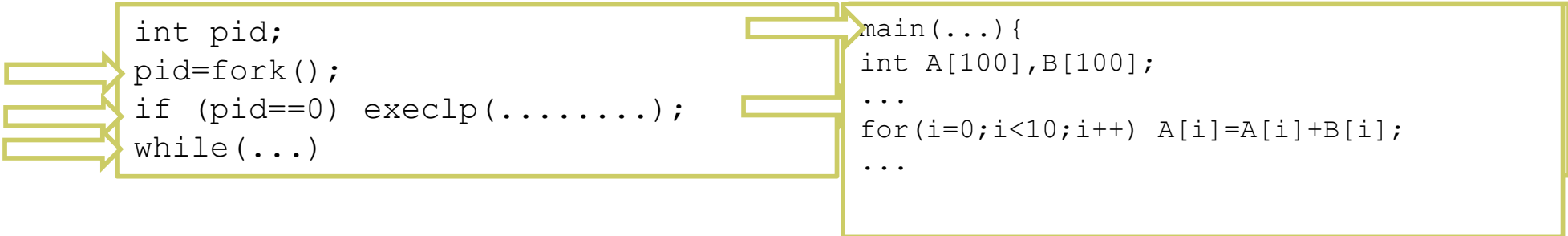
progA

```
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];  
...}
```



# Exemple: exec

- Quan al *shell* executem la següent comanda:

```
% ls -l
```

1. Es crea un nou procés (*fork*)
2. El nou procés canvia la imatge, i executa el programa ls (*exec*)

- Com s'implementa això?

```
...  
ret = fork();  
if (ret == 0) {  
    execvp("/bin/ls", "ls", "-l", (char *)NULL);  
}  
// A partir de aquí, este código sólo lo ejecutaría el padre  
...
```



- Fa falta posar un *exit* després de l'*execvp*?
- Què passa si l'*execvp* falla?

# Terminació de processos. exit

```
void main()
{...
ret=fork(); (1)
if (ret==0) execlp("a","a",NULL);
...
waitpid(-1,&exit_code,0); (3)
}
```

A

```
void main()
{...
exit(4); (2)
}
```

Es consulta del PCB

kernel

PCB (del procés "A")

pid=...

exit\_code=

...

...

Es guarda en el PCB

Tot i que exit\_code no val 4!!! Cal que processar el resultat

# Exemple: fork/exec/waitpid



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

void main(int argc, char *argv[])
{
    ...
    num_cmd = argc-1;
    for (i = 0; i < num_cmd; i++)
        execCmd( argv[i+1] );
    // waitpid format
    // ret: pid del procés que termina o -1
    // arg1== -1→ espera a un procé fill qualsevol
    // arg2 exit_code→variable on el kernel ens copiarà el valor de
    // finalizació
    // argc3==0 → BLOQUEJANT
    while ((pid = waitpid(-1, &exit_code, 0) > 0)
        trataExitCode(pid, exit_code);
    exit(0);
}

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

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

*Examineu les fitxers complets: plauncher.c y Nplauncher.c*

# handleExitCode



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

// PROGRAMEU-LA PER ALS LABORATORIOS
void HANDLEExitCode(int pid,int exit_code)
{
    int exit_code,statcode,signcode;
    char buffer[128];

    if (WIFEXITED(exit_code)) {
        statcode = WEXITSTATUS(exit_code);
        sprintf(buffer,"El proces %d termina amb exit code %d\n", pid,
statcode);
        write(1,buffer,strlen(buffer));
    }
    else {
        signcode = WTERMSIG(exit_code);
        sprintf(buffer,"El proces %d termina pel signal %d\n", pid,
signcode);
        write(1,buffer,strlen(buffer));
    }
}
```



---

-Linux:Signals

# COMUNICACIÓ ENTRE PROCESSOS

# Comunicación entre procesos

---

- Els processos poden ser independents o cooperar
- Per què pot ser útil que diversos processos cooperin?
  - Per a compartir informació
  - Per a accelerar la computació que realitzen
  - Per modularitat
- Per a poder cooperar, els processos necessiten comunicar-se
  - Interprocess communication (IPC) = Comunicació entre processos
- Per a comunicar dades hi ha 2 models principalment
  - Memòria compartida (Shared memory)
    - ▶ Els processos utilitzen variables que poden llegir/escriure
  - Pas de missatges (Message passing)
    - ▶ Els processos utilitzen funcions per a enviar/rebre dades

# Comunicación entre procesos en Linux

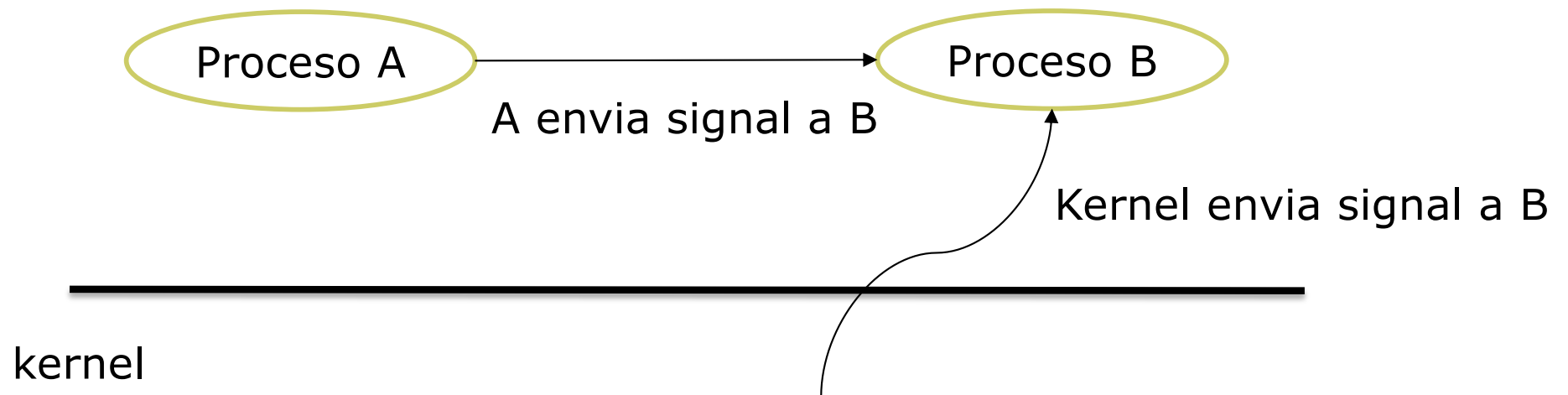
---



- **Signals – Esdeveniments enviats per altres processos (del mateix usuari) o pel kernel per indicar determinades condicions (Tema 2)**
- **Pipes – Dispositiu que permet comunicar dos processos que s'executen a la mateixa màquina. Les primeres dades que s'envien són les primeres que es reben. La idea principal és connectar la sortida d'un programa amb l'entrada d'un altre. Utilitzat principalment per la shell (Tema 4)**
- **FIFOS – Funciona amb pipes que tenen un nom el sistema de fitxers. S'ofereixen com a pipes amb nom. (Tema 4)**
- **Sockets – Dispositiu que permet comunicar dos processos a través de la xarxa**
- **Message queues – Sistema de comunicació indirecta**
- **Semaphores - Comptadors que permeten controlar l'accés a recursos compartits. S'utilitzen per prevenir l'accés de més d'un procés a un recurs compartit (per exemple memòria)**
- **Shared memory – Memòria accessible per més d'un procés alhora (Tema 3)**

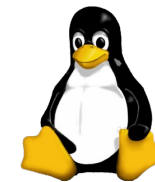
# Signals: idea

- Signals: notificaciones que pot rebre un procés per informar-lo que ha succeït un esdeveniment
- Els pot enviar el kernel o altres processos del mateix usuari





# Tipus de signals i tractaments (I)



- Cada possible esdeveniment té un signal associat
  - Els esdeveniments i els signals associats estan predefinitos pel kernel
    - ▶ El signal és un número, però hi ha constants definides per usar-les en els programes o en línia de comandes
- Hi ha dos signals que no estan associats a cap esdeveniment perquè el programador els faci servir com vulgui SIGUSR1 i SIGUSR2
- Cada procés té un tractament associat a cada signal
  - Tractaments per defecte
  - El procés pot capturar (modificar el tractament associat) tots els tipus de signals excepte SIGKILL i SIGSTOP

# Tipos de signals y tratamientos(2)

## ■ Algunos signals

| Nombre         | Acción Defecto | Evento  |
|----------------|----------------|---|
| <b>SIGCHLD</b> | Ign            | Un procés fill ha acabat o ha estat aturat          |
| <b>SIGCONT</b> |                | Continua si estava aturat                           |
| <b>SIGSTOP</b> | Stop           | Aturar procés                                       |
| <b>SIGINT</b>  | Term           | Interromput des del teclat (Ctrl+C)                 |
| <b>SIGALRM</b> | Term           | El comptador definit per la trucada alarm ha acabat |
| <b>SIGKILL</b> | Term           | Acabar el procés                                    |
| <b>SIGSEGV</b> | Core           | Referencia invàlida a memòria                       |
| <b>SIGUSR1</b> | Term           | Definit per l'usuari (procés)                       |
| <b>SIGUSR2</b> | Term           | Definido por el usuario (proceso)                   |

## ■ Usos que els donarem principalment

- Sincronització de processos
- Control del temps (alarmes)

# Tipus de signals i tractaments( 3)

---

- El tractament d' un signal funciona com una interrupció provocada per programari:
  - En rebre un signal el procés interromp l'execució del codi, passa a executar el tractament que aquest tipus de signal tingui associat i en acabar (si sobreviu) continua on estava
- Els processos poden bloquejar/desbloquejar la recepció de cada signal excepte SIGKILL i SIGSTOP (tampoc es poden bloquejar els signals SIGFPE, SIGILL i SIGSEGV si són provocats per una excepció)
  - Quan un procés bloqueja un signal, si se li envia aquest signal el procés no el rep i el sistema el marca com a pendent de tractar
    - ▶ bitmap associat al procés, només recorda un signal de cada tipus
  - Quan un procés desbloqueja un signal rebrà i tractarà el signal pendent d'aquest tipus

# Linux: Interfície relacionada amb signals

| Servicio   | Llamada sistema |
|--|-----------------|
| Trametre un signal concret                                     | kill            |
| Capturar/reprogramar un signal concret                         | sigaction       |
| Bloquear/desbloquear signals                                   | sigprocmask     |
| Esperar FINS que arriba un esdeveniment qualsevol (BLOQUEJANT) | sigsuspend      |
| Programar l'enviament automàtic del signal SIGALRM (alarma)    | alarm           |

- **Fitxer amb signals: `/usr/include/bits/signum.h`**
- Hi ha diverses interfícies de gestió de signals incompatibles i amb diferents problemes, Linux implementa l'interfície POSIX

# Interfície: Enviar / Capturar signals

## ■ Per a enviar:

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



- signum → SIGUSR1, SIGUSR2, etc

- Requeriment: conèixer el PID del procés destí

## ■ Per capturar un SIGNAL i executar una funció quan arribi:

```
int sigaction(int signum, struct sigaction *tratamiento,  
struct sigaction *tratamiento_antiguo)
```



- Signum → SIGUSR1, SIGUSR2, etc

- tractament → struct sigaction que descriu què fer en rebre el signal

- tractament antic → struct sigaction que descriu què es feia fins ara. Aquest paràmetre pot ser NULL si no interessa obtenir el tractament antic

# A envia un signal a B

---

- El procés A envia (en algun moment) un signal a B i B executa una acció en rebre'l

## Procés A

```
.....  
Kill( pid, evento);  
.....
```

## Procés B

```
int main()  
{  
    struct sigaction trat,viejo_trat;  
    /* código para inicializar trat */  
    sigaction(evento, &trat, &viejo_trat);  
    ....  
}
```

# Definició de struct sigaction

---

- struct sigaction: diversos camps. Ens fixarem només en 3:
  - sa\_handler: pot prendre 3 valors
    - ▶ SIG\_IGN: ignorar el signal en rebre' l
    - ▶ SIG\_DFL: usar el tractament per defecte
    - ▶ funció d'usuari amb una capçalera predefinida: void nombre\_funcion(int s);
      - IMPORTANT: la funció la invoca el kernel. El paràmetre es correspon amb el signal rebut (SIGUSR1, SIGUSR2, etc), així es pot associar la mateixa funció a diversos signals i fer un tractament diferenciat dins d'ella.
  - sa\_mask: signals que s'afegeixen a la màscara de signals que el procés té bloquejats
    - ▶ Si la màscara està buida només s'afegeix el signal que s'està capturant.
    - ▶ En sortir del tractament es restaura la màscara que hi havia abans d'entrar.
  - sa\_flags: per configurar el comportament (si val 0 es fa servir la configuració per defecte). Alguns flagells:
    - ▶ SA\_RESETHAND: després de tractar el signal es restaura el tractament per defecte del signal
    - ▶ SA\_RESTART: si un procés bloquejat en una crida a sistema rep el signal es reinicia la crida que l'ha bloquejat

# Estructures de dades del kernel

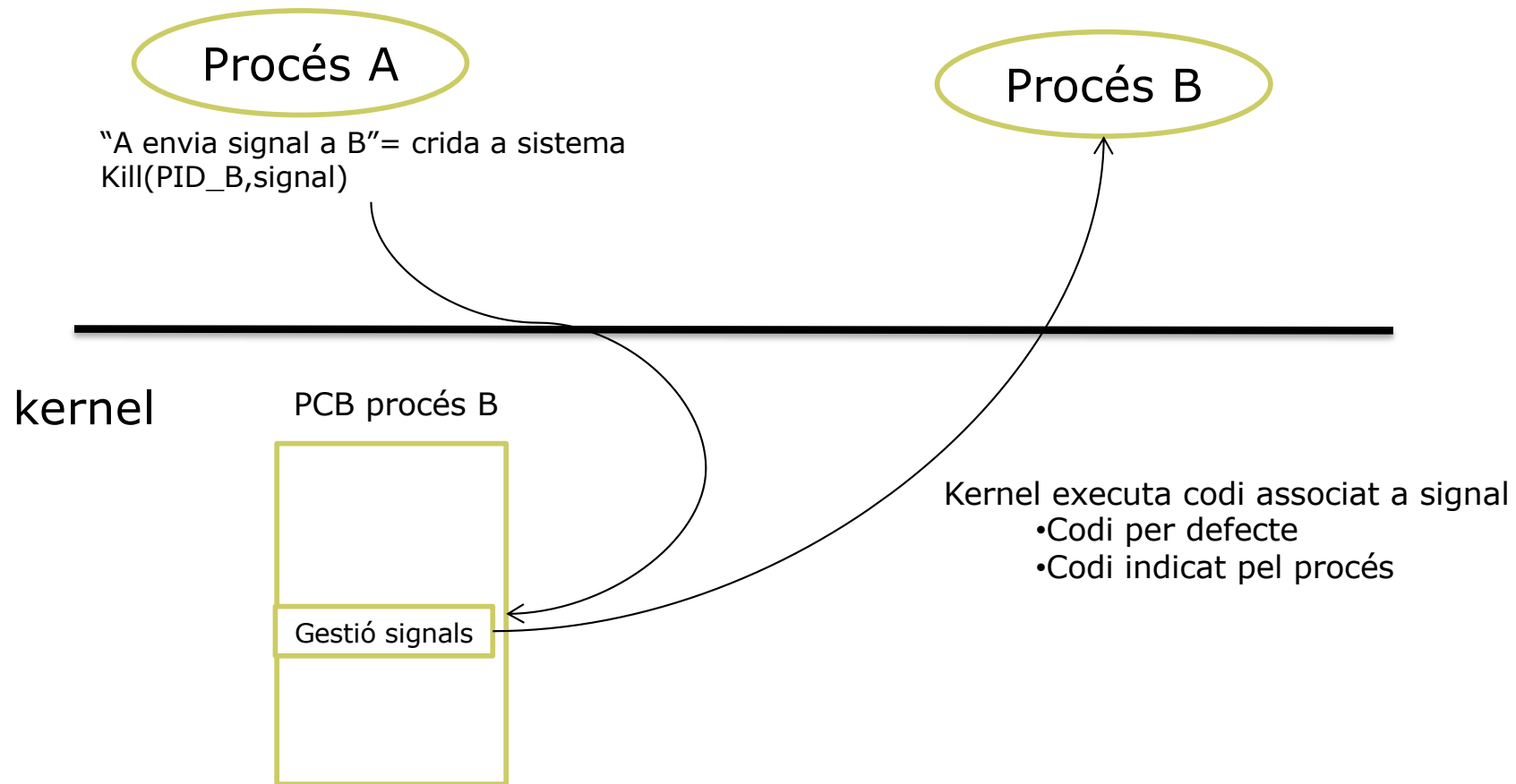


- La gestió de signals és per procés, la informació de gestió està en el PCB
  - Cada procés té una taula de programació d'signals (1 entrada per signal),
    - ▶ S'indica que acció realitzar quan es rebí l'esdeveniment
  - Un bitmap d'**esdeveniments pendents** (1 bit per signal)
    - ▶ No és un comptador, actua com un booleà
  - Un únic **temporitzador** per a l'alarma
    - ▶ Si programem 2 vegades l'alarma només queda l'última
  - Una **màscara de bits** per indicar quines signals cal tractar

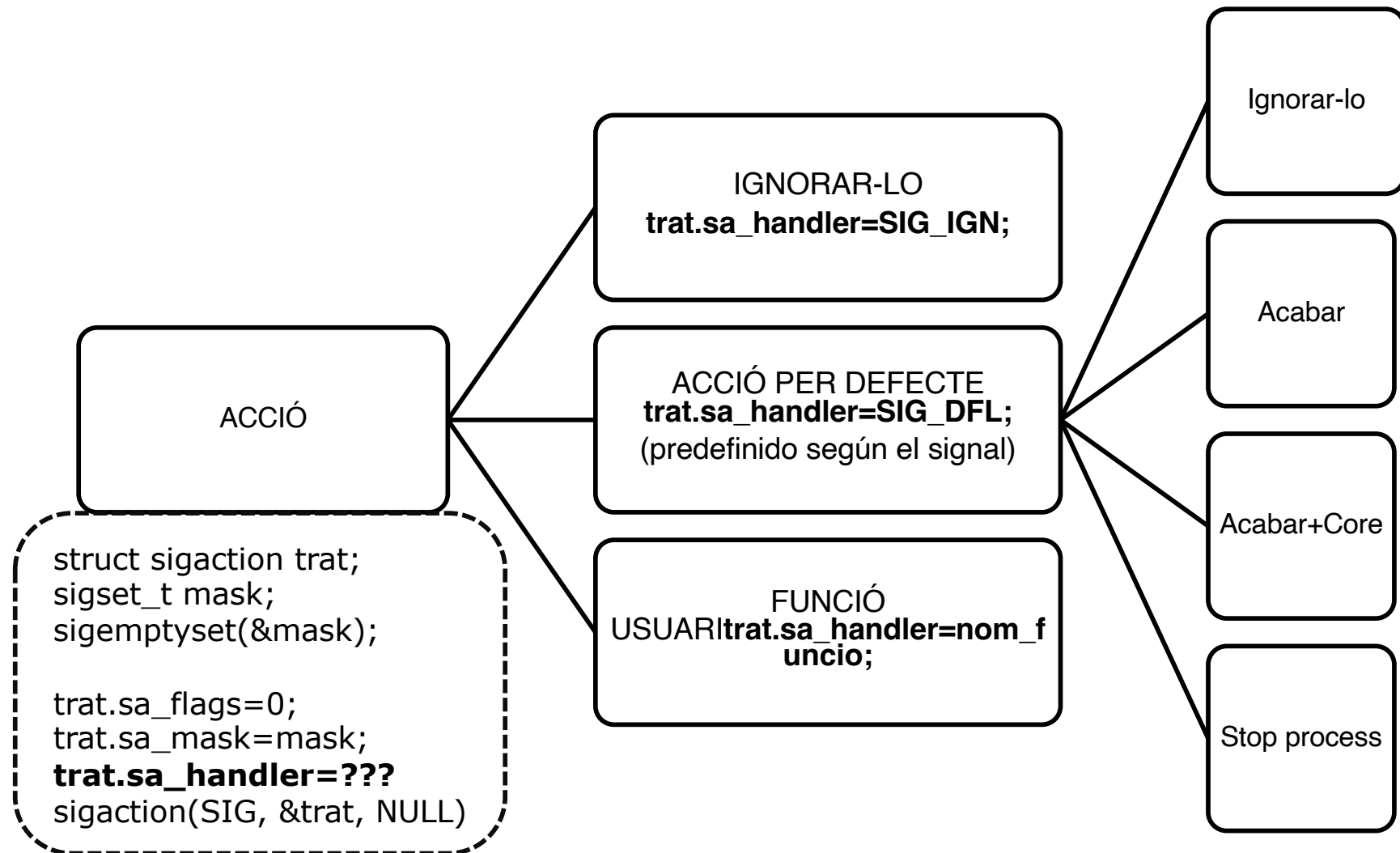


# Signals: Tramesa i recepció

¿Què succeeix realment?, el kernel ofereix el servei de passar la informació.



# Accions possibles en rebre un signal



On SIG ha de ser el nom d'un signal: SIGUSR1, SIGALRM, SIGUSR2, etc

# Manipulació de màscares de signals

- sigemptyset: inicialitza una màscara sense signals

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



- sigfillset: inicialitza una màscara amb tots els signals

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



- sigaddset: afegeix el signal a la màscara que es passa com a paràmetre

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



- sigdelset: elimina el signal de la màscara que es passa com a paràmetre

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



- sigismember: retorna cert si el signal està a la màscara

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



# Exemple: capturar 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); // Cuando llegue SIGINT se ejecutará
                                   // f_sigint

    while(1) {
        sprintf(buffer, "Estoy haciendo cierta tarea\n");
        write(1, buffer, strlen(buffer));
    }
}

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

*Podeu trobar el codi complet en: [signal\\_basico.c](#)*

# Bloquejar/desbloquejar signals

- El procés pot controlar en quin moment vol rebre els signals

```
int sigprocmask(int operacion, sigset_t *mascara, sigset_t  
*vieja_mascara)
```



- Operació pot ser:
  - ▶ SIG\_BLOCK: **afegir** els signals que indica mascara a la màscara de signals bloquejats del procés
  - ▶ SIG\_UNBLOCK: **treure** els signals que indica la màscara de signals bloquejats del procés
  - ▶ SIG\_SETMASK: **fer** que la màscara de signals bloquejats del procés passi a ser el paràmetre `mascara`

# Esperar un esdeveniment

## ■ Esperar (bloquejat) que arribi un esdeveniment

```
int sigsuspend(sigset_t *mascara)
```



- Bloqueja el procés fins que arriba un esdeveniment el tractament del qual no sigui SIG\_IGN
- **Mentre** el procés està bloquejat en el sigsuspend serà els signals que no es rebran (signals bloquejats),
  - ▶ Així es pot controlar quin signal treu al procés del bloqueig
- En sortir de sigsuspend automàticament es restaura la mascara que hi havia i es tractaran els signals pendents que s'estiguin desbloquejant

# Sincronització: A envia un signal a B(1)

- El procés A envia (en algun moment) un signal a B, B està esperant un esdeveniment i executa una acció en rebre'l

## Proceso A

```
.....  
Kill( pid, evento);  
.....
```

## Proceso B

```
void funcion(int s)  
{  
  ...  
}  
int main()  
{  
  sigaction(evento, &trat, NULL);  
  ....  
  sigemptyset(&mask);  
  sigsuspend(&mask);  
  ....  
}
```

¿Què passa si A envia l'esdeveniment abans que B arribi al sigsuspend?  
Què passa si B rep un altre esdeveniment mentre és al sigsuspend?

## Sincronització: A envia un signal a B (2)

- El procés A envia (en algun moment) un signal a B, B està esperant un esdeveniment i executa una acció en rebre'l

### Proceso A

```
.....  
Kill( pid, evento);  
....
```

- sigprocmask bloqueja esdeveniment, així que si arriba abans que B arribi al sigsuspend no se li entrega
- Quan B està en el sigsuspend l'únic esdeveniment que li pot desbloquejar és el que es fa servir per a la sincronització amb A

### Proceso B

```
void funcion(int s)  
{  
...  
}  
  
int main()  
{  
sigemptyset(&mask);  
sigaddset(&mask,evento);  
sigprocmask(SIG_SETMASK,&mask,NULL);  
sigaction(evento, &trat,NULL);  
....  
sigfillset(&mask);  
sigdelset(&mask,evento)  
sigsuspend(&mask);  
....  
}
```



# Alternatives en la sincronització de processos

---

## ■ Alternatives per "esperar" la recepció d'un esdeveniment

1. **Espera activa:** El procés consumeix cpu per comprovar si ha arribat o no l'esdeveniment. Normalment comprovant el valor d'una variable
  - ▶ Exemple: `while(!rebut);`
2. **Bloqueig:** El procés allibera l'actuació i serà el kernel qui el desperti a la recepció d'un esdeveniment
  - Exemple: `sigsuspend`

## ■ Si el temps d'espera és curt es recomana espera activa

- No compensa la sobrecàrrega necessària per executar el bloqueig del procés i el canvi de context

## ■ Per a temps d'espera llargs es recomana bloqueig

- S'aprofita la CPU perquè la resta de processos (inclòs el que estem esperant) avancin amb la seva execució

# Control de temps: programar temporitzat

---

- Programar un enviament automàtic (l'envia el kernel) de signal SIGALRM
  - `int alarm(num_secs);`

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

# Control de temps: Ús del temporitzador

- El procés programa un temporitzador de 2 segons i es bloqueja fins que passa aquest temps

```
void funcion(int s)
{
    ...
}
int main()
{
    sigemptyset(&mask);
    sigaddset(&mask, SIGALRM);
    sigprocmask(SIG_SETMASK,&mask, NULL);
    sigaction(SIGALRM, &trat, NULL);
    ....
    sigfillset(&mask);
    sigdelset(&mask,SIGALRM);
    alarm(2);
    sigsuspend(&mask);
    ....
}
```

El procés estarà bloquejat en el sigsuspend, quan passin 2 segons rebrà el SIGALRM, s'executarà la funció i després continuarà on estava

# Relació amb fork i exec

---

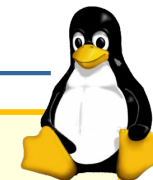
## ■ FORK: Procés nou

- El fill hereta la taula d' accions associades als signals del procés pare
- La màscara de signals bloquejats s'hereta
- Els esdeveniments són enviats a processos concrets (PID's), el fill és un procés nou → La llista d'esdeveniments pendants s'esborra (tampoc s'hereten els temporitzadors pendants)

## ■ EXECLP: **Mateix procés**, canvi d' executable

- La taula d' accions associades a signals es posa per defecte ja que el codi és diferent
- Els esdeveniments són enviats a processos concrets (PID's), el procés no canvia → La llista d'esdeveniments pendants es conserva
- La màscara de signals bloquejats es conserva

# Exemple 1: gestió de 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);
}
```

*Podeu trobar el codi complet a: cada\_segundo.c*

## Exemple 2: espera activa vs bloqueig(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(); // ¿Qué opciones tenemos?
        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);
}
```

## Exemple 2: espera activa vs bloqueig(2)

### Opció 1: espera activa

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



### Opció 2: bloqueig

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



- 
- Dades
  - Estructures de gestió
  - Polítiques de planificació
  - Mecanismes

# GESTIÓ INTERNA DE PROCESSOS



# Gestió interna

---

- Per gestionar els processos necessitem:
  - **Estructures de dades**
    - ▶ per representar les seves propietats i recursos → PCB
    - ▶ per representar i gestionar threads → **depende del SO**
  - **Estructures de gestió, que organitzin els PCB's en funció del seu estat o de necessitats d'organització del sistema**
    - ▶ Generalment són llistes o cues, però poden incloure estructures més complexes com taules de hash, arbres, etc.
    - ▶ Cal tenir en compte l'eficiència
      - Són ràpides les insercions/eliminacions?
      - Són ràpides les recerques?
      - Quin/quins seran els índexs de recerca? PID? Usuari?
    - ▶ Cal tenir en compte l'escalabilitat
      - Quants processos podem tenir actius en el sistema?
      - Quanta memòria necessitem per a les estructures de gestió?
  - **Algorisme/s de planificació**, que ens indiqui com gestionar aquestes estructures
  - **Mecanismes** que apliquin les decisions preses pel planificador

# Dades: Process Control Block (PCB)

---

- És la informació associada amb cada procés, depèn del sistema, però normalment inclou, per cada procés, aspectes com:
  - L'identificador del procés(PID)
  - Les credencials: usuari, grup
  - L'estat: RUN, READY,...
  - Espai per salvar els registres de laCPU
  - Dades per gestionar signals
  - Informació sobre la planificació
  - Informació de gestió de la memòria
  - Informació sobre la gestió de l'E/S
  - Informació sobre els recursos consumits (Accounting )

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

# Estructures per organitzar els processos: Cues/l·listes de planificació

---

- El SO organitza els PCB's dels processos en estructures de gestió: vectors, l·listes, cues. Taules de hash, arbres, en funció de les seves necessitats
- Els processos en un mateix estat solen organitzar-se en cues o l·listes que permeten mantenir un ordre
- Per exemple:
  - Cua de processos – Inclou tots els processos creats en el sistema
  - Cua de processos preparats per executar-se (ready) – Conjunt de processos que estan preparats per executar-se i estan esperant una CPU
    - ▶ En molts sistemes, això no és 1 cua sinó diverses ja que els processos poden estar agrupats per classes, per prioritats, etc
  - Cues de dispositius– Conjunt de processos que estan esperant dades de l'algun dispositiu d'E/S
  - El sistema mou els processos d'una cua a una altra segons correspongui
    - ▶ Ex. Quan acaba una operació d'E/S, el procés es mou de la cua del dispositiu a la cua de ready.

# Planificació

---

- L' algoritme que decideix quan un procés ha de deixar la CPU, qui entra i durant quant temps, és el que es coneix com a **Política de planificació** (o scheduler)
- La planificació s'executa moltes vegades (cada 10 ms, per exemple)→ ha de ser molt ràpida
- El criteri que decideix quan s'avalua si cal canviar el procés que està a la cpu (o no), que posem, etc, es coneix com a **Política de planificació**
- Periòdicament (10 ms.) en la interrupció de rellotge, per assegurar que cap procés monopolitza la CPU

# Planificació

---

- Hi ha determinades situacions que provoquen que s'hagi d'executar la planificació del sistema
- Casos en que el procés que està RUN no pot continuar l'execució → Cal triar un altre → Esdeveniments no apropiatius (*non preemptive*)
  - Exemple: El procés acaba, El procés es bloqueja
- Casos en què el procés que està RUN podria continuar executant-se però per criteris del sistema es decideix passar-lo a estat READY i posar-ne un altre en estat RUN → La planificació tria un altre però és forçat → esdeveniment apropiatiu (*preemptive*)
  - Aquestes situacions depenen de la política, cada política considera alguns si i d'altres no)
  - Exemples: El procés porta X ms (RoundRobin), Creem un procés nou, es desbloqueja un procés,...

# Planificador

---

- Les polítiques de planificació són apropiatives (*preemptive*) o no apropiatives (*non-preemptive*)
  - Non-preemptive cheduing: La política no li treu la crida al procés, ell “l'allibera”. Només suporta esdeveniments no preemptius. (esdeveniments tipus 1 i 2)
  - Preemptive: La política li treu la cpu al procés. Suporta esdeveniments preemptius (esdeveniments tipus 3) i no preemptius.
- Si el SO aplica una política preemptiva el SO és preemptiu

# Caracterització de processos

---

- Els processos solen presentar ràfegues de computació i ràfegues d'accés a dispositius (E/S) que poden bloquejar al procés
- En funció d'aquestes ràfegues, els processos es consideren:
  - Processos de càlcul: Consumeixen més temps fent càlcul que E/S
  - Processos d'E/S: Consumeixen més temps fent entrada/sortida de dades que càlcul

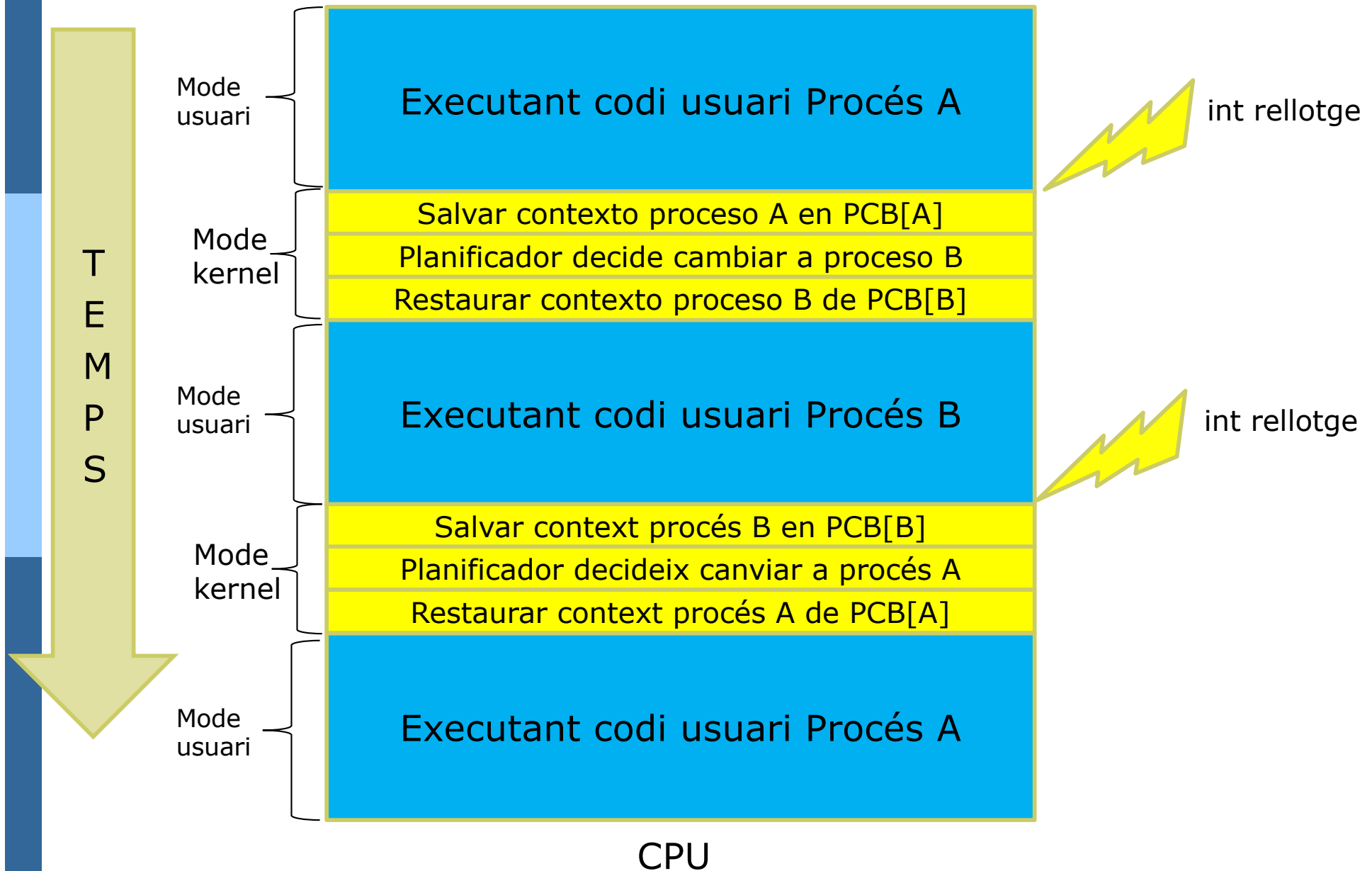
# Mecanismes utilitzats pel planificador

---

- Quan un procés deixa la CPU i es posa un altre procés s'executa un canvi de context (d'un context a un altre)
- Canvis de context(Context Switch)
  - El sistema ha de salvar l'estat del procés que deixa la cpu i restaurar l'estat del procés que passa a executar-se
    - ▶ El context del procés se sol salvar en les dades de kernel que representen el procés (PCB). Hi ha espai per guardar aquesta informació
    - ▶ **El canvi de context no és temps útil de l'aplicació, així que ha de ser ràpid.** De vegades el maquinari ofereix suport per fer-lo més ràpid
    - ▶ Per exemple per salvar tots els registres o restaurar-los de cop
  - Diferències entre canvi de context entre threads
    - ▶ Mateix procés vs diferents processos



# Mecanisme canvi context



# Objectius/Mètriques de la planificació

---

- Les polítiques de planificació poden tenir objectius diferents segons el sistema per al qual estiguin dissenyats, el tipus d'usuaris que vagin a usar-los, el tipus d'aplicació, etc. No obstant això, aquests són els criteris que solen considerar (poden haver-n'hi d'altres) que determinen el comportament d'una política
  - Temps total d'execució d'un procés (Turnaround time )- Temps total des que el procés arriba al sistema fins que acaba
    - ▶ Inclou el temps que està en tots els estats
    - ▶ Depèn del propi procés i de la quantitat de processos que hi hagi a la màquina
  - Temps d'espera d' un procés- Temps que el procés passa en estat ready

# Round Robin (RR)

---

- El sistema té organitzats els processos en funció del seu estat
- Els processos estan en cua per ordre d'arribada
- Cada procés rep la CPU durant un període de temps (time quantum), típicament 10 o 100 miliseg.
  - El planificador utilitza la interrupció de rellotge per assegurar-se que cap procés monopolitza la CPU

# Round Robin (RR)

---

- Esdeveniments que activen la política Round Robin:
  1. Quan el procés es bloqueja (no preemptiu)
  2. Quan acaba el procés (no preemptiu)
  3. Quan acaba el quantum (preemptiu)
- És una política **apropiativa** o preemptiva
- Quan es produeix un d'aquests esdeveniments, el procés que està run deixa la cpu i se selecciona el següent de la cua de ready.
  - Si l'esdeveniment és 1, el procés s'afegeix a la cua de bloquejats fins que acaba l'accés al dispositiu
  - Si l'esdeveniment és el 2, el procés passaria a zombie en el cas de linux o simplement acabaria
  - Si l'esdeveniment és el 3, el procés s'afegeix al final de la cua de ready

# Round Robin (RR)

---

## ■ Rendiment de la política

- Si hi ha  $N$  processos a la cua de ready, i el quantum és de  $Q$  mil·lisegons, cada procés rep  $1/n$  parts del temps de CPU en blocs de  $Q$  mil·lisegons com a màxim.
  - ▶ Cap procés espera més de  $(N-1)Q$  mil·lisegons.
- La política es comporta diferent en funció del quantum
  - ▶ *q molt gran*  $\Rightarrow$  es comporta com en ordre seqüencial.
    - Els processos rebran la CPU fins que es bloquegin
  - ▶ *q petit*  $\Rightarrow$   $q$  ha de ser gran comparat amb el cost del canvi de context. Altrament, hi ha massa *overhead*.

# Completely Fair Scheduling

---

- Algorisme usat en les versions actuals de Linux
- Mètrica objectiu: temps d'ús de CPU de tots els processos ha de ser equivalent
  - Round Robin penalitza els processos intensius a E/S
- Temps màxim d'ús consecutiu de CPU (*~quantum*) és variable
  - Teòricament, temps consumit de CPU per a cada procés hauria de ser el resultat de dividir el temps que porta en execució entre el nombre de processos que competeixen per la CPU
  - A cada procés se li assigna la CPU fins que es bloquegi, acabi o el seu temps de CPU assoleixi el teòric que hauria de tenir
- Prioritat → distància al temps teòric de CPU (com més lluny estigui més prioritari)
- Crea grups de processos (criteri configurable per l'administrador de la màquina) i permet comptabilitzar l'ús de CPU per grup
  - Objectiu: impedir que un usari que executa molts processos acapari la màquina



---

## **RELACIÓ ENTRE LES CRIDES A SISTEMA DE GESTIÓ DE PROCESSOS I LA GESTIÓ INTERNA DEL S.O. (DADES, ALGORISMES, ETC).**

# Què fa el kernel quan s'executa un...?

---

## ■ fork

- Es busca un PCB lliure i es reserva
- S'inicialitzen les dades noves (PID, etc)
- S'aplica la política de gestió de memòria (Tema 3)
- P.ex.: reservar memòria i copiar contingut de l' espai d' adreces del pare al fill
- S'actualitzen les estructures de gestió d'E/S (Tema 4 i 5)
- En el cas de Round Robin: S'afegeix el procés a la cua de ready

## ■ exec

- Se substitueix l'espai d'adreces pel codi/dades/pila del nou executable
- S'inicialitzen les dades del PCB corresponents: taula de signals, context, etc
- S' actualitzen el context actual del procés: variables d'entorn, argv, registres, etc.



# Què fa el kernel quan s'executa un...?

---

## ■ exit

- S'alliberen tots els recursos del procés: memòria, dispositius "en ús", etc
- A Unix: es guarda l'estat de finalització al PCB i s'elimina de la cua de ready (de manera que no podrà executar més)
- S'aplica la política de planificació

## ■ waitpid

- Es busca el procés a la llista de PCB's per aconseguir el seu estat de finalització
- Si el procés que busquem estava zombie, el PCB s'allibera i es retorna l'estat de finalització al seu pare.
- Si no estava zombie, el procés pare s'elimina passa d'estat run a bloquejats fins que el procés fill acabi.
  - ▶ S'aplicaria la política de planificació



---

# PROTECCIÓ I SEGURETAT

# Protecció i Seguretat

---

- La protecció es considera un problema Intern al sistema i la Seguretat es refereix principalment a atacs externs

# Protecció UNIX

---

- Els usuaris s'identifiquen mitjançant username i password (userID)
- Els usuaris pertanyen a grups (groupID)
  - Per a fitxers
    - ▶ Protecció associada a: Lectura/Escriptura/Execució(rwx)
      - Comando ls per consultar, chmod per modificar
      - S'associen als nivells de: Propietari, Grup, Resta d' usuaris
    - A nivell procés: Els processos tenen un usuari que determina els drets
- L'excepció és ROOT. Pot accedir a qualsevol objecte i pot executar operacions privilegiades
- També s'ofereix un mecanisme perquè un usuari pugui executar un programa amb els privilegis d'un altre usuari (mecanisme de *setuid*)
  - Permet, per exemple, que un usuari pugui modificar-se el seu password tot i que el fitxer pertany a root.

# Seguretat

---

- La seguretat s'ha de considerar a quatre nivells:
- Físic
  - Les màquines i els terminals d'accés s'han de trobar en habitacions/edificis segurs.
- Humà
  - És important controlar a qui es concedeix l'accés als sistemes i conscienciar els usuaris de no facilitar que altres persones puguin accedir als seus comptes d'usuari
- Sistema Operatiu
  - Evitar que un procés(s) saturi el sistema
  - Assegurar que determinats serveis estan sempre funcionant
  - Assegurar que determinats ports d'accés no estan operatius
  - Controlar que els processos no puguin accedir fora del seu propi espai d'adreces
- Xarxa
  - La majoria de dades avui dia es mouen per la xarxa. Aquest component dels sistemes és normalment el més atacat.