

## Estructura dels Sistemes Operatius

El sistema operatiu bàsicament proveeix un entorn on els programes poden ser executats. Tot i les petites diferències que pot haver-hi entre sistemes operatius, tots tenen uns trets bàsics comuns, els quals tractarem.

Un dels aspectes més importants dels sistemes operatius és la capacitat de treballar amb diferents programes a la vegada, coneguda com **multiprograma**. D'aquesta manera s'incrementa l'ús de la CPU amb el fet de gestionar l'organització de les tasques de manera que la CPU sempre tingui alguna cosa que executar.

La idea funciona de la següent manera: El sistema operatiu manté diferents tasques en memòria siultàniament. Com que, en general, la memòria és massa petita per a emmagatzemar totes les tasques, aquestes són guardades inicialment al disc en l'anomenada **job pool**. Esquema de la memòria:

Sistema operatiu
tasca1
tasca2
tasca3
tasca4

El conjunt de les tasques en memòria pot ser un subconjunt de les tasques de la *job pool*. El sistema operatiu escull i comença a executar una de les tasques de memòria. Eventualment, una tasca pot haver d'esperar alguna acció, com una operació d'Entrada/Sortida (E/S), per a completar-se. En un sistema no-multiprograma, la CPU es quedaria esperant aquesta acció; en canvi, en un sistema multiprograma, el SO simplement canvia a una **nova tasca**, i així de manera successiva. Quan la primera tasca aconsegueix acabar l'espera, la CPU torna a "donar-li atenció". Sempre que hi hagi alguna tasca per executar, la CPU no para mai.

Els sistemes multiprograma proveeixen un entorn on els diferents recursos del sistema són utilitzats de manera efectiva. El **time sharing** també conegut com a *multitasking* és una extensió llògica del multiprograma. Els sistemes *multitask* o de **temps compartit**, la CPU executa multiples tasques canviant entre elles, de manera tant ràpida que l'usuari ho pot utilitzar tot a la vegada.

El temps compartit necessita un sistema **interactiu**, en el qual l'usuari dóna instruccions al sistema operatiu o un programa directament, utilitzant dispositius d'entrada com pot ser un teclat, un ratolí... I espera els resultats immediats al dispositiu de sortida, demanera que el **temps de resposta** hauria de ser curt. Cada *usuari* (usuaris, programes) té la sensació que TOTA la CPU està sent utilitzada per ell, però en realitat és compartida amb molts *usuaris*.

Un programa carregat a memòria el qual s'està executant és anomenat un **procés**. Quan s'executa un procés, és durant un curt període de temps normalment, ja que o bé ha acabat o necessita esperar a una acció E/S, la qual pot ser *interactiva* com una sortida a una pantalla, una entrada des de teclat; les quals van a la "velocitat de les persones", ja que han d'esperar a l'usuari humà. Com que aquesta velocitat és molt lenta per un ordinador, i deixar la CPU en *standby* seria una pèrdua d'eficiència, el sistema canviarà ràpidament a una altra tasca d'un altre *usuari*.

Com que fer *multitasking* en un sistema multiprograma significa carregar diferents tasques a memòria simultàniament, si múltiples tasques estan preparades per a ser portades a memòria però no hi ha prou espai per totes, alors el sistema ha d'escollar entre elles; fer aquesta decisió implica la **distribució de tasques**. Al tenir la possibilitat de tenir diferents programes en la mateixa memòria requereix una forma de **gestió de memòria**; i amés si diferents tasques estan preparades per a ser portades a terme, el sistema ha d'escollar quina anirà primer, aquesta gestió l'anomenem **distribució de la CPU**. Per últim, el fet de executar diferents programes concurrent-ment, requereix que no es "molestin" entre ells, en tots els aspectes (gestió de memòria, de CPU, etc).

En un sistema amb *multitasking* ha d'assegurar un temps de resposta ràpid. Això s'aconsegueix gràcies a un procés d'**intercanvi**, on els processos són intercanviats dins i fora de la memòria principal al disc. Un mètode més comú és la **memòria virtual**, una tècnica que permet l'execució d'un procés que no està complet en memòria; això permet a l'usuari executar programes més grans que la pròpia **memòria física**.

## 2. Les operacions del Sistema Operatiu.

Els sistemes operatius moderns es basen en les **interrupcions**. Si no hi ha processos per a executar, ni accions E/S per atendre, i no hi ha usuaris als quals respondre, un sistema operatiu es quedarà esperant algun event. Els events venen normalment senyalats per alguna interrupció o **trap**. Una **trap** (o excepció) és una interrupció generada pel software causada o bé per un error (eg. una divisió per zero o un accés de memòria no vàlid) o per una sol·licitud específica d'un programa d'usuari. Per a cada tipus d'interrupció, diferents segments de codi en el SO determinen què s'hauria de fer, són les anomenades **routines de servei d'interrupcions**.

Com que els usuaris i el sistema operatiu comparteixen el hardware i el software de l'ordinador, hem d'assegurar que un error en el programa només pugui causar problemes al programa en particular, ja que compartint un sol *bug* en un programa podria causar problemes a molts processos. O fins i tot errors minoritaris com cambiar dades utilitzades per altres programes o el propi sistema operatiu.

Sense protecció cap a aquests tipus d'errors tenim dues opcions: O només executem un sol procés a la vegada, o totes les sortides són dubtoses de ser correctes. Un bon sistema operatiu ha d'assegurar que un programa incorrecte o maliciós no pugui causar problemes als altres programes.

## 2.1 Operacions de mode dual i multimode

Per tal d'assegurar la correcte execució del sistema operatiu, hem de distingir entre les execucions del codi del propi sistema operatiu i les del codi de l'usuari. Per això trobem dos modes separats d'execució: **mode usuari** i **mode privilegiat** (també sistema, kernel, root...). Un bit anomenat **mode bit** s'afegeix al hardware per indicar el mode actual *kernel* (0), *usuari* (1). Amb el bit de mode podem distingir entre tasques executades a petició del sistema operatiu o a petició de l'usuari. Quan s'està executant codi d'una aplicació d'usuari, el sistema està en *mode usuari*. De tota manera, quan una aplicació d'usuari o un usuari demana serveis del sistema operatiu (a través de *crides al sistema*), el sistema ha de canviar de mode usuari a mode kernel per a completar la petició.

Quan es provoca una interrupció el sistema canvia a mode kernel, sempre que el sistema operatiu guanya control del sistema, està en mode kernel i sempre torna a mode usuari abans de donar el control a una aplicació d'usuari de nou.

Aquesta tècnica ens permet assegurar el sistema operatiu d'executar operacions perilloses. Aquestes operacions perilloses estan designades com **instruccions privilegiades**. Només són permeses d'executar en mode kernel, si s'intenta fer en mode usuari el hardware no farà l'instrucció sinó donarà un *trap* al sistema operatiu. La instrucció per a canviar de mode, és una instrucció privilegiada per exemple.

## CONCEPTES

### Que és un procés?

Un procés és un programa en execució. Un programa és un algoritme escrit en un llenguatge de programació.

Es pot afirmar que cada procés té la seva CPU virtual en el sentit que sempre es té una còpia d'ella. A aquesta CPU virtual, se li diu **Bloc de Control de Procés** (PCB), el qual es una estructura de dades que conté informació referent al procés, informació que necessita el Sistema Operatiu per a poder planificar els processos (assignar CPU i desallotjar-los [veure Round Robin]), això significa que cada procés té el seu propi PCB.

- Cada vegada que un procés entra en estat d'execució (RUNNING) aquesta còpia s'utilitzarà per a carregar la CPU amb els valors que permeten continuar aquest procés exactament des d'on va acabar l'anterior execució.
- Cada vegada que un procés sigui parat per qualsevol cosa, s'han d'agafar els valors de la CPU per a actualitzar-los en el PCB.

Algunes informacions del PCB són:

- **Estat del procés:** Ready, Running o Blocked.
- **PC (Contador de Programa):** Conté la direcció de la següent instrucció a executar pel procés.
- **Informació de planificació:** Aquesta informació conté: prioritat del procés, apuntadors a cues de planificació...
- **Informació del sistema d'arxius:** Proteccions, identificacions d'usuari, grup...
- **Informació de l'estat d'E/S:** Conté sollicituds pendents d'E/S, dispositius d'E/S assignats al procés, etc...

Per a fer ús del PCB el Sistema Operatiu utilitza una Taula de Processos, de manera que cada entrada en aquesta taula fa referència a un PCB.

### Procés fill

És un procés creat a partir d'un `fork()`; des d'un altre procés. Aquest es crea en total semblança al procés pare. Quan un procés crea un altre, l'esquema de processos s'organitza en forma d'arbre. Cada procés té el seu PID propi, amés del PPID (PID del procés pare), els quals **són únics per a cada procés**.

Aquest procés fill hereda les següents característiques del pare:

- Codi, dades i pila
- Programació dels signals
- ID d'usuari i ID de grup
- Variables d'entorn
- Màscara de signals (sigmask)
- PC del pare (continuarà al mateix lloc que el pare).

I no herederà:

- Contadors interns
- Alarmes pendents
- Signals pendents

## Thread (Fils d'execució) ↗

Com hem dit, un procés és la unitat d'assignació de recursos d'un programa en execució. Entre aquests recursos trobem els **fils d'execució** (threads) d'un procés. Un **thread** és un mecanisme que permet a una aplicació realitzar diverses tasques a la vegada de manera concurrent. És la mateixa filosofia que utilitza el SO per a executar diferents processos a la vegada enfocat a executar subprocessos dins un mateix procés; però és una mica diferent ja que per definició els processos no comparteixen espai de memòria entre ells, en canvi els threads si.

Imaginem tres processos, **A**, **B** i **C** executant-se a la mateixa vegada. Què passaria si el procés A, necessitava mostrar una interfaç gràfica i a la vegada estar escribint un arxiu? Sense els threads això no seria possible. La idea del thread és permetre que el procés pugui executar una o més tasques a la vegada (o almenys que així ho vegi l'usuari ;)). De tal manera que cada vegada que a un procés li correspon un **quantum** d'execució del sistema, aquest alterni entre una de les seves tasques o una altre.

Això ens dona la necessitat de reestructurar el concepte de procés, ja que un procés ara no és la **unitat mínima d'execució**, ara un procés és un conjunt de threads. Quan un procés, aparentment, no utilitza threads; està executant un **únic thread**.

Hem de tenir en compte dues coses:

- La memòria de treball del procés, segueix sent assignada per procés, però els threads dins el procés comparteixen tots la mateixa regió de memòria, el mateix espai de direccions.
- El SO assigna Quants a cada thread creat, i ja no calendaritza processos sinó cada un dels threads en execució al sistema.
- Cada thread té el seu propi context (estat d'execució), així que cada vegada que es suspen un thread per a permetre l'execució d'un altre, el seu contexte es guarda i es re estableix novament només quan és el seu torn [veure Round Robin més avall].

El procés segueix sent una part important ja que és qui se li assigna la prioritat d'execució, la memòria i els recursos, privilegis i altres dades importants. El thread és només qui s'executa.

Imaginem un sistema amb **2 CPU** i una aplicació que executa més de dos threads: El què passarà és que només dos threads s'estaràn executant en paral·lel en un moment donat, i el sistema operatiu alternarà l'execució d'aquests threads de tal manera que tots tinguin Quants assingats, però només hi haurà **2 threads en paralel**.

---

## Estats d'un procés i Propietats ↗

---

### Algoritme Round Robin ↗

És un algoritme de gestió i repartiment equitatiu i senzill de la CPU entre els processos que evita la monopolització de la CPU molt utilitzat en entorns de **temps compartit** o **multitasking**.

Consisteix en definir una unitat de temps petita, anomenada **quantum** la qual s'assigna a cada process per a que estigui en mode **READY**. Si el procés esgota el seu quantum (Q) de temps, s'escull un nou procés per a ocupar la CPU. Si el procés es bloqueja o acaba abans d'esgotar el seu quantum també s'alterna l'ús de la CPU.

És per això que surgeix la necessitat d'un rellotge de sistema; un dispositiu que genera periòdicament interrupcions. El quantum d'un procés equival a un nombre fixe de cicles de rellotge. Al haver-hi una interrupció de rellotge, o (és el mateix) la fi d'un quantum es cedeix el control de la CPU al procés seleccionat per el planificador.

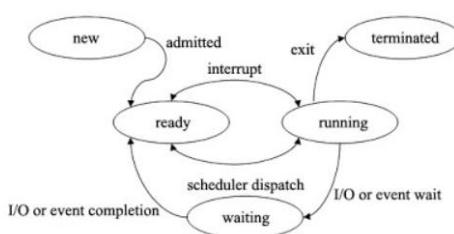
Si el **quantum** és molt gran, els processos acabaran els seus temps de CPU abans d'acabar el quantum. Si és molt petit, hi haurà molts canvis de processos i la CPU perdrà rendiment.

Processos	Temps d'execució
P1	53
P2	17
P3	68
P4	24

Anem a fer un exemple amb la taula que veiem a sobre:

Per aquest exemple imaginarem 4 processos, amb els seus respectius temps d'execució i un Quantum (Q) de 20.

Hem de tenir en compte que quan hi ha un *canvi de context* (canvi de procés a la CPU) s'ha de guardar l'estat del procés que s'estava executant al PCB i llavors cargar a la CPU l'estat del PCB del nou procés.



Imaginem que **P1** entra a la CPU amb un temps de 53, es carrega el seu PCB i el rellotge del sistema comença contar, quan arriba a 20 es llança una interrupció de rellotge. Es treu **P1** de la CPU, s'actualitza el seu PCB i el procés torna a la cua de READY. Ara entrerà **P2** a la CPU, que era el següent en la llista de ready, un cop passat el seu temps d'execució ja que  $TE(P2) < Q$  acabarà el procés, **P2** haurà acabat i farà exit, per tant anirà a estat ZOMBI o TERMINATED. Ara entrerà el procés **P3** que necessita un imput per a continuar, un cop passats 10q demana l'entrada, s'actualitzarà el seu PCB i passa a la cua de processos BLOCKED o WAITING, i llavors entra **P4** a la CPU i passa el mateix que amb **P1**. Es va seguir l'algoritme fins que tots els processos han acabat.

---

## Fi d'execució d'un procés (fill) ↗

`void exit(int status);` Causa un acabament normal del procés i es retorna el valor status al procés pare.

Un procés pot acabar la seva execució de manera **voluntària** (exit) o **involuntària** (signals).

Quan un procés vol acabar la seva execució (voluntàriament), alliberar els seus recursos i alliberar les estructures de kernel que té reservades per ell, s'executa la crida de sistema **exit**.

Si volem sincronitzar el pare amb la finalització del seu/seus fill/s, podem utilitzar la crida a sistema **waitpid** (veure apartat COMANDOS -> **waitpid**).

El fill pot enviar informació de finalització (*exit code*) al pare a través de la crida al sistema **exit** i el pare la recull a través del **waitpid**.

- El SO fa d'intermediari, guarda aquesta informació fins que el pare la consulta.
- Mentre el pare no consulta aquesta informació, el PCB del fill no serà alliberat i el procés es queda en estat ZOMBIE.
  - Convé fer **waitpid** dels processos que creem per tal d'alliberar els recursos ocupats del kernel.

Si un procés mor sense alliberar els PCBs dels fills, el procés **init()** del sistema ho farà.

---

## Signals ↗

Podem interpretar un signal com una notificació de que ha passat un event, i cada event té un signal associat.

Cada procés té un tractament associat a cada signal el qual pot ser modificat [veure *sigaction* a comandos].

Per exemple el signal **SIGCONT** fa que un procés continui si aquest estava parat; **SIGSTOP** fa parar el procés el qual el rep; **SIGCHLD** té un tractament predefinit de **IGNORE** i notifica que el procés fill ha mort o ha estat parat; **SIGSEGV** indica una referència invàlida a memòria, coneguda també com Segmentation Fault; **SIGALRM** indica que el temps d'alarma ha acabat [veure *alarm* a comandos] i el tractament és acabar el procés; **SIGKILL** fa acabar el procés també; **SIGINT** és un signal que té un tractament predefinit d'acabar i és donat a través de la combinació de tecles Ctrl+C en el shell.

---

## Màscares ↗

Són estructures de dades que permeten designar quins signals pot rebre un procés en un moment determinat de l'execució.

## COMANDOS ↗

### `fork();` ↗

Crea un procés fill

El pare i el fill s'executaràn concurrent-ment ("a la vegada"), es duplicarà el codi del pare juntament amb la pila i les seves dades i s'assignaran al fill.

El fill inicia la execució en el punt on estava el pare en el moment de la creació, per tant el PC del fill és el mateix del pare.

[Per saber més sobre processos fills veure apartat **Processos fills a CONCEPTES BÀSICS**]

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

Al fer un fork, l'espai de direccions és el mateix. Si volem executar un altre codi, aquest procés fill ha de **mutar** (canviar el binari del procés).

**execlp:** Fa canviar (mutar) l'executable d'un procés per un altre executable (però el procés segueix sent el mateix):

- Tot el **contingut** de l'espai de direccions canvia, codis, dades, pila...
  - Es reinicia el PC a la primera instrucció (main).
- Es manté tot el que està relacionat amb l'**identitat del procés**.
  - Contadors d'ús intern, signals pendents, etc...
- Es modifiquen aspectes relacionats amb l'executable o l'espai de direccions:
  - Es defineix per defecte la taula de signals (mask).

## Valors de retorn:

- Si el fork ha tingut èxit, es retorna el PID del fill al pare.
- Si el fork ha tingut èxit, es retorna 0 en el fill.
- Si el fork ha fallat es retorna -1 al pare, i el fill no serà creat; *errno* contindrà informació de l'error.

## SIGNALS

Els signals poden:

- Ignorar un event (Tractament designat com a IGNORE, o tractament per defecte és IGNORE).
- Realitzar accions per defecte.
- Realitzar accions definides amb *sigaction*.

Quan el sistema rep un signal, aquest passa a executar un tractament per aquest. Si el tractament és una funció dissenyada (no tractament per defecte), ha de rebre com a paràmetre el número de signal (ja que una mateixa funció podria tractar diferents signals).

### **sigaction**

Reprograma l'acció associada a un signal concret, és a dir, permet canviar el tractament d'un signal.

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

- *signum*: especifica el signal, ha de ser un vàlid. (sigkill i sigstop no són vàlids).
- *act*: Si és diferent de NULL, la nova acció per el signal *signum* és instalada des de *act*.
- *oldact*: Si és diferent de NULL, la acció prèvia a la nova es guarda a *oldact*.

L'struct *sigaction* conté les variables següents:

- **sa\_handler**
  - *SIG\_IGN* - Ignorarà el signal.
  - *SIG\_DFL* - Farà el tractament per defecte.
  - *foo\_whatever* - Funció que haurem creat per a tractar-lo.
- **sa\_mask**
  - Buida, ja que només s'afegeirà el signal que estem capturant.
  - Es restaurarà l'anterior al acabar.
- **sa\_flags**
  - 0 -> Configuració per defecte
  - *SA\_RESETHAND* -> després de tractar el signal es restaurarà el tractament per defecte.
  - *SA\_RESTART* -> Si estàs fent una crida al sistema i reps un signal, es reiniciarà la crida.

```
void foo(int x){  
    //codi que executarà la funció.  
}  
  
struct sigaction s;  
/* Inicialitzem l'estructura de dades s amb:  
   handler -> Ha de contenir el nom de la funció que tractarà el signal.  
   mask -> una màscara que crearem  
   flags -> SA_RESTART  
*/  
  
sigaction(SIGUSR1, &s, NULL);
```

## kill

Envia un signal a un procés.

```
kill [opcions] <pid> [...]
```

El signal per defecte de kill és TERM. Altres sinals poden ser especificats amb -<signal>; per exemple -SIGKILL, -9 o -KILL.

## sigsuspend

bloqueja el procés que l'executa fins que rebi un signal (amb tractament diferent de IGNORE).

*int sigsuspend(const sigset\_t \*mask);*

Sigsuspend reemplaça temporalment la mascara de signal del procés que fa la crida amb la màscara del paràmetre *mask*, llavors suspen el procés fins que rebi un signal amb l'acció de invocar una funció de tractament de signal (*signal handler*) o d'acabar el procés.

Quan arriba un signal, si el pid és positiu, el signal sig s'envia al procés amb l'ID especificat per pid.

Si pid és 0, llavors signal sig s'envia a tots els processos del grup del procés que ha fet la crida. Si pid és -1, sig s'envia a tots els processos pels quals el procés que ha fet sigsuspend té permís per a enviar signals, excepte el procés 1 (init). Si pid és < -1 llavors sig s'envia a tots els processos del grup de processos els quals el seu pid és -pid.

### Valors de retorn

- Si el signal acaba -> No hi ha retorn.
- Si s'ha fet restore de la signal mask a l'estat d'abans de la crida -> Sigsuspend sempre retorna -1 (amb errno).

---

## sigprocmask

Permet modificar la màscara de signals bloquejats del procés.

*int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset);*

La *màscara de signals* és el conjunt de signals els quals estàn bloquejats actualment per qui els crida.

el valor de *how* pot ser:

- **SIG\_BLOCK** -> Els signals bloquejats són la unió dels actualment bloquejats i els del paràmetre *set*.
- **SIG\_UNBLOCK** -> Els signals de *set* es borren dels signals bloquejats actualment.
- **SIG\_SETMASK** -> Els signals bloquejats i permesos passen a ser els de *set*.

Si *oldset* no és null, s'hi guardarà el valor previ de la mascara de signals. Si és NULL, la signal mask no canviará (*how* serà ignorat), però el valor de la mascara de signals es retorna a *oldset* (si aquest no és NULL).

**valor de retorn** 0 si ha estat un èxit, -1 si ha estat un error. Si hi ha un error, *errno* indicarà la causa d'aquest error. \*\*!\*\*No es poden bloquejar SIGKILL o SIGSTOP. Cada thread té la seva pròpia màscara de signals.

**exemple del que normalment es fa a l'assignatura:** mask = vector de signals activats i desactivats. *int sigprocmask(SIG\_BLOCK, mask, NULL);*

---

## alarm

Programa l'enviament d'un SIGALARM després d'*x* segons.

*unsigned int alarm(unsigned int x);*

Si *x* és zero, totes les alarmes pendents són cancel·lades, en tots els events. *alarm()* retorna el nombre de segons restants abans que qualsevol alarma anterior està prevista de disparar-se. Si no hi havia alarmes anteriors, retorna zero.

## sleep

Bloqueja un procés durant el temps que es passa per al paràmetre.

*sleep NUMBER[SUFFIX]... SUFFIX: s (seconds), m (minutes), h (hours), d (days)*

---

## kill

Envia un event concret a un procé.

---

## waitpid

*waitpid(pid\_t pid, int \*status, int options);*

Aquesta crida a sistema s'utilitza per a esperar a un canvi d'estat d'un procés fill des del procés el qual fa la crida i obtenir la informació del fill el qual l'estat ha canviat. Un *canvi d'estat* es considera: El fill ha acabat; el fill ha sigut parat per un signal; o el fill ha sigut reprès a través d'un signal. En el cas d'un fill que ha acabat, utilitzar el waitpid permet al sistema alliberar els recursos (PCB) associats amb aquest fill; si no es fa un wait/waitpid, el fill es queda en estat *zombie* i no s'alliberarà l'espai del PCB del fill mort (per això es diu *zombie*).

Executar aquesta comanda suspen l'execució del procés que fa la crida fins que un fill (especificat en el paràmetre *pid*) ha canviat d'estat. El paràmetre *pid* pot ser:

- \*\*< -1\*\* Espera qualsevol procés fill el qual tingui un ID de grup de processos igual que el valor absolut de *pid*.
- -1 Espera qualsevol procés fill.
- 0 Espera qualsevol procés fill que tingui un ID de grup de processos igual que al del procés que fa la crida *waitpid*.
- >0 Espera el fill amb el PID igual a *pid*.

En el paràmetre *options* podem posar:

- WNOHANG: retorna immediatament si cap fill ha fet *exit*.
- WCONTINUED: retorna si un fill parat a reprès l'execució amb un SIGCONT.
- Alguns més que podem veure al man del bash.

Els valors de retorn del *waitpid* són:

- Èxit: Retorna el **valor del PID** del fill que ha canviat d'estat. (Exemple: 3 fills en estat zombi; retorna el pid del primer fill en acabar i passar a ZOMBIE i en *status* el codi de finalització del fill.)
  - Si tenim la opció WNOHANG i existeixen un o més fills que es corresponen amb el paràmetre *pid* especificat, però no han canviat d'estat, **es retorna 0**.
    - Exemple: Un fill en estat BLOCKED i un altre fill en estat READY.
- Error: **Retorna -1**.

EXEMPLES MÉS COMUNS:

- *waitpid(-1, NULL, 0)* --> Esperar (amb bloqueig si és necessari) a un fill qualsevol.
- *waitpid(pid\_fill, NULL, 0)* --> Esperar (amb bloqueig si és necessari) a un fill amb pid = *pid\_fill*.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h> // Pels signals
#include <signal.h> // Pels signals

// ____ Funció Usage || Com escriure __
void Usage()
{
    char buff[120]; // Valors per cada
    sprintf(buff, "Usage: ./ejemplo n\n"); // Char Int String
    write(1,buff,strlen(buff)); // (%c) (%d) (%s)
}

// _____ Definicions _____
int main (int argc, char *argv[]) {} //argc = n° elements de argv (argc == argv.size())

int n;
char *string; // O bé string[n]
char **vector_string; // O bé *vector_string[n], vector_string[n][n]

// _____ Recordatoris _____
exit(0); // --> El programa acaba normalment (bé)
exit(1); // --> El programa acaba per algun error
atoi(char *s); // --> Converteix un string a numero
getpid(); // --> Retorna el pid del programa
alarm(0); // --> Elimina el temporitzador i retorna qual li quedava a l'anterior
alarm(n); // --> En n (>0) s'enviarà un SIGALRM al procés

// _____ Mutar procés _____
execvp ("ls", "-l"); // Que executeràs a la terminal directament
// [1] --> Path (com que usem execvp només cal repetir [1] sense ./ si procedeix)
// [2] --> Arguments que li passariem a la funció (pot no haver-n'hi)
// [3] --> Confia

// _____ Forks _____
/* Base */
int pid;
pid = fork(); // Recorda, el fill rep 0 i el pare el pid del fill
if(...)

/* Fills sequencial */
for (int i = 0; i < n_fills and fork() == 0; ++i) //Nota que si fiquessim fork != 0 fariem un recursivitat
{
    //El pare no executa això, només els seus fills, nets...
}
// Tots executen aquesta part

/* Fills recurrents */
for (int i = 0; i < n_fills; ++i)
{
    pid = fork();
    if (pid == 0)
    {
        // Això ho executa el fill
        exit(0); // "Matem" el fil aquí
    }
    else if (pid < 0)
    {
        // S'ha produït un error
        perror("Error en el fork\n");
        exit(1);
    }
}
```



# ¿Que ofrece el SO?



## Arranque del sistema

- El hardware carga el SO ( o una parte) al arrancar. Se conoce como *boot*
  - El sistema puede tener más de un SO instalado (en el disco) pero sólo uno ejecutándose**
  - El SO se copia en memoria (todo o parte de él)
  - Inicializa las estructuras de datos necesarias (hardware/software) para controlar la máquina i ofrecer los servicios
  - Las interrupciones hardware son capturadas por el SO
  - Al final el sistema pone en marcha un programa que permite acceder al sistema
    - Login (username/password)
    - Shell
    - Entorno gráfico

## Como ofrecer un entorno seguro: “Modos” de ejecución

- El SO necesita una forma de garantizar su seguridad, la del hardware y la del resto de procesos
- Necesitamos instrucciones de lenguaje máquina privilegiadas que sólo puede ejecutar el kernel
- El HW conoce cuando se está ejecutando el kernel y cuando una aplicación de usuario. Hay una instrucción de LM para pasar de un modo a otro.
- El SO se ejecuta en un modo de ejecución privilegiado.**
  - Mínimo 2 modos (pueden haber más)**
    - Modo de ejecución NO-privilegiado , *user mode*
    - Modo de ejecución privilegiado, *kernel mode*
  - Hay partes de la memoria sólo accesibles en modo privilegiado y determinadas instrucciones de lenguaje máquina sólo se pueden ejecutar en modo privilegiado**
- Objetivo: Entender que son los modos de ejecución y porqué los necesitamos

## Acceso al código del kernel

- El kernel es un código guiado por eventos
  - Interrupción del flujo actual de usuario para realizar una tarea del SO
- Tres formas de acceder al código del SO (Visto en EC):
  - Interrupciones** generadas por el hardware ( teclado, reloj, DMA, ... )
    - asíncronas (entre 2 dos instrucciones de lenguaje máquina)
  - Los errores de software generan **excepciones** (división por cero, fallo de página, ... )
    - síncronas
    - provocadas por la ejecución de una instrucción de lenguaje máquina
    - se resuelven (si se puede) dentro de la instrucción
  - Peticiones de servicio de programas: **Llamada a sistema**
    - síncronas
    - provocados por una instrucción explícitamente (de lenguaje máquina)
    - para pedir un servicio al SO (llamada al sistema)
- Desde el punto de vista hardware son muy parecidas (casi iguales)

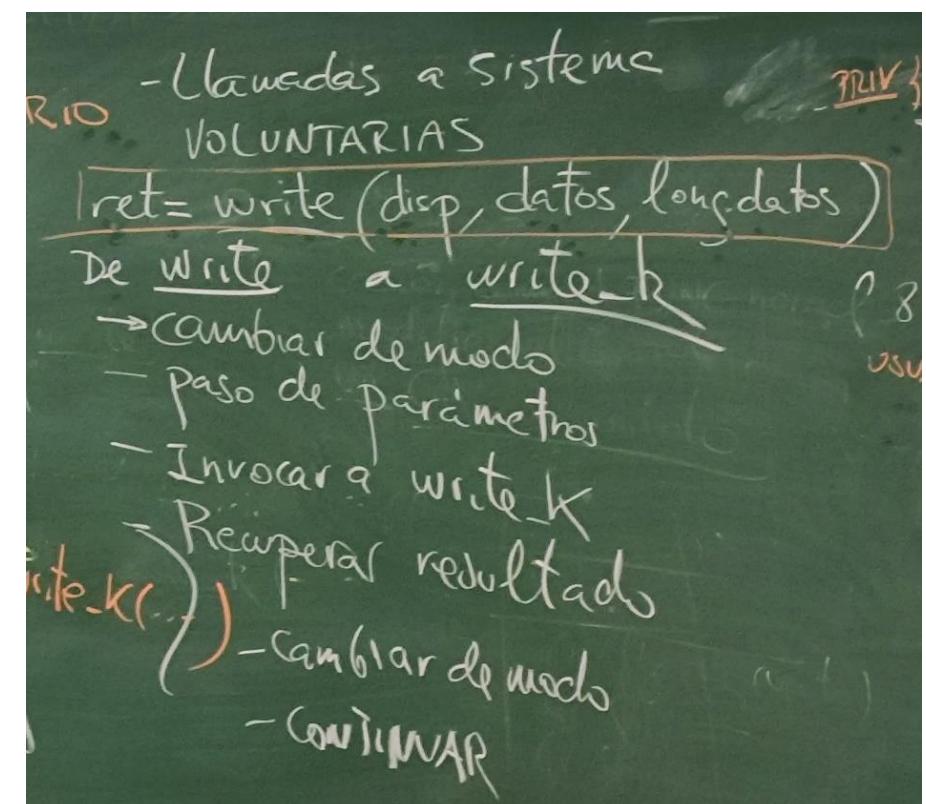
# ¿Cuando se ejecuta código de kernel?

- Cuando una aplicación ejecuta una llamada a sistema
- Cuando una aplicación provoca una excepción
- Cuando un dispositivo provoca una interrupción
  
- Estos eventos podrían no tener lugar, y el SO no se ejecutaría → El SO perdería el control del sistema.
- **El SO configura periódicamente la interrupción de reloj para evitar perder el control y que un usuario pueda acaparar todos los recursos**
  - Cada 10 ms por ejemplo
  - Típicamente se ejecuta la planificación del sistema



## Llamadas a Sistema

- Conjunto de **FUNCIONES** que ofrece el kernel para acceder a sus servicios
- Desde el punto de vista del programador es igual al interfaz de cualquier librería del lenguaje (C,C++, etc)
- Normalmente, los lenguajes ofrecen un API de más alto nivel que es más cómoda de utilizar y ofrece más funcionalidades
  - Ejemplo: Librería de C: printf en lugar de write
    - ▶ Nota: La librería se ejecuta en **modo usuario** y no puede acceder al dispositivo directamente
- Ejemplos
  - Win32 API para Windows
  - POSIX API para sistemas POSIX (UNIX, Linux, Max OS X)
  - Java API para la Java Virtual Machine



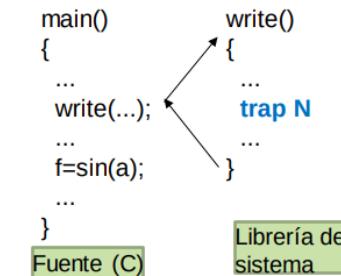
## Requerimientos llamadas a sistema

- Requerimientos
  - Desde el punto de vista del programador
    - Tiene que ser tan sencillo como una llamada a función
      - <tipo> nombre\_función(<tipo1> arg1, <tipo2> argc2..);
    - No se puede modificar su contexto (igual que en una llamada a función)
      - Se deben salvar/restaurar los registros modificados
  - Desde el punto de vista del kernel necesita:
    - Ejecución en modo privilegiado → soporte HW
    - Paso de parámetros y retorno de resultados entre modos de ejecución diferentes → depende HW
    - Las direcciones que ocupan las llamadas a sistema tienen que poder ser variables para soportar diferentes versiones de kernel y diferentes S.O. → por portabilidad

## Librerías “de sistema”

- El SO ofrece librerías del sistema para aislar a los programas de usuario de todos los pasos que hay que hacer para

1. Pasar los parámetros
2. Invocar el código del kernel
3. Recoger y procesar los resultados



- Se llaman librería de sistema pero se ejecuta en **modo usuario**, solamente sirven para facilitar la invocación de la llamada a sistema

## Solución: Librería “de sistema” con soporte HW

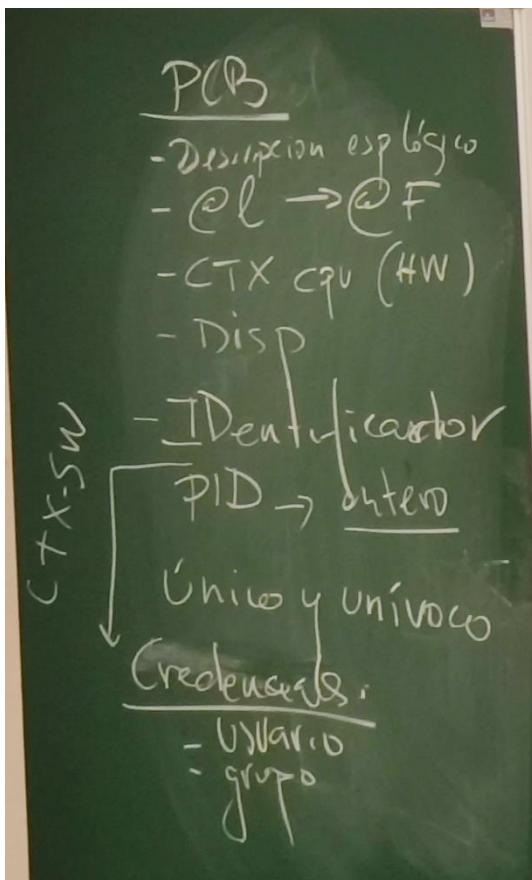
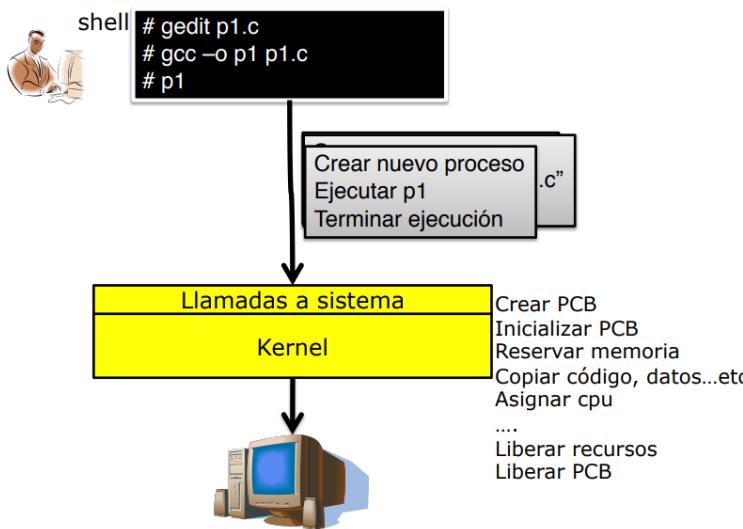
- La librería de sistema se encarga de traducir de la función que ve el usuario a la petición de servicio explícito al sistema
  - Pasa parámetros al kernel
  - Invoca al kernel → TRAP
  - Recoge resultados del kernel
  - Homogeneiza resultados (todas las llamadas a sistema en linux devuelven -1 en caso de error)
- ¿Cómo conseguimos la portabilidad entre diferentes versiones del SO?
  - La llamada a sistema no se identifica con una dirección, sino con un identificador (un número), que usamos para indexar una tabla de llamadas a sistema (que se debe conservar constante entre versiones)

## Código genérico al entrar/salir del kernel

- Hay pasos comunes a interrupciones, excepciones y llamadas a sistema
- En el caso de interrupciones y excepciones, no se invoca explícitamente ya que genera la invocación la realiza la CPU, el resto de pasos si se aplican.

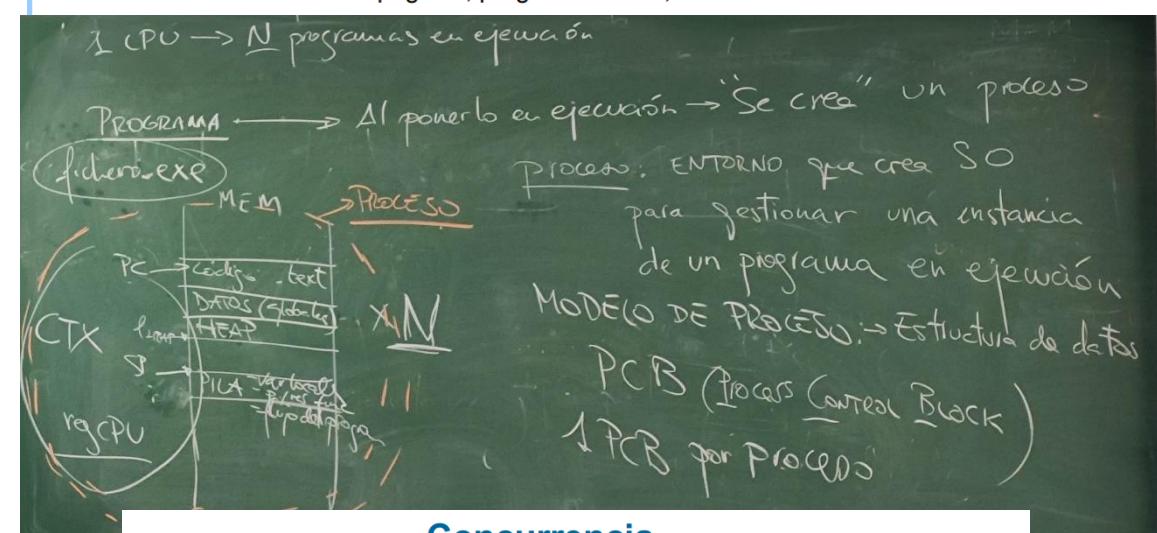
	Función “normal”	Función kernel
Pasamos los parámetros	Push parámetros	<b>DEPENDE</b>
Para invocarla	call	sysenter, int o similar
Al inicio	Salvar los registros que vamos a usar (push)	Salvamos todos los registros (push)
Acceso a parámetros	A través de la pila: Ej: mov 8(ebp).eax	<b>DEPENDE</b>
Antes de volver	Recuperar los registros salvados al entrar (pop)	Recuperar los registros salvados (todos) al entrar (pop)
Retorno resultados	eax ( o registro equivalente)	<b>DEPENDE</b>
Para volver al código que la invocó	ret	sysexit, iret o similar

## Procesos: ¿Como se hace?

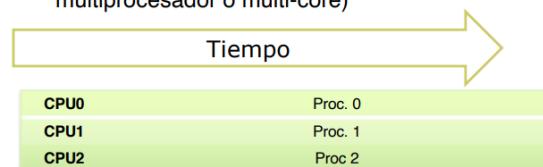


## Process Control Block (PCB)

- Contiene la información que el sistema necesita para gestionar un proceso. Esta información depende del sistema y de la máquina. Se puede clasificar en estos 3 grupos
  - Espacio de direcciones
    - ▶ descripción de las regiones del proceso: código, datos, pila, ...
  - Contexto de ejecución
    - ▶ SW: PID, información para la planificación, información sobre el uso de dispositivos, estadísticas,...
  - HW: tabla de páginas, program counter, ...



- Concurrencia es la **capacidad** de ejecutar varios procesos de forma simultánea
  - Si realmente hay varios a la vez es paralelismo (arquitectura multiprocesador o multi-core)



- Si es el SO el que genera un **paralelismo virtual** mediante compartición de recursos se habla de concurrencia



## Hilos de Ejecución (Threads) – Qué son?

- Entre los recursos está el/los **hilo/s de ejecución (thread)** de un proceso
  - Se trata de la instancia/flujo de ejecución de un proceso y es la mínima unidad de planificación del SO (asignación de tiempo de CPU)
    - Cada parte del código que se puede ejecutar de forma independiente se le puede asociar un thread
  - Tiene asociado el contexto necesario para seguir el flujo de ejecución de las instrucciones
    - Identificador (Thread ID: TID)
    - Puntero a Pila (Stack Pointer)
    - Puntero a siguiente instrucción a ejecutar (Program Counter),
    - Registros (Register File)
    - Variable errno
  - Los threads **comparten** los recursos del proceso (PCB, memoria, dispositivos E/S)

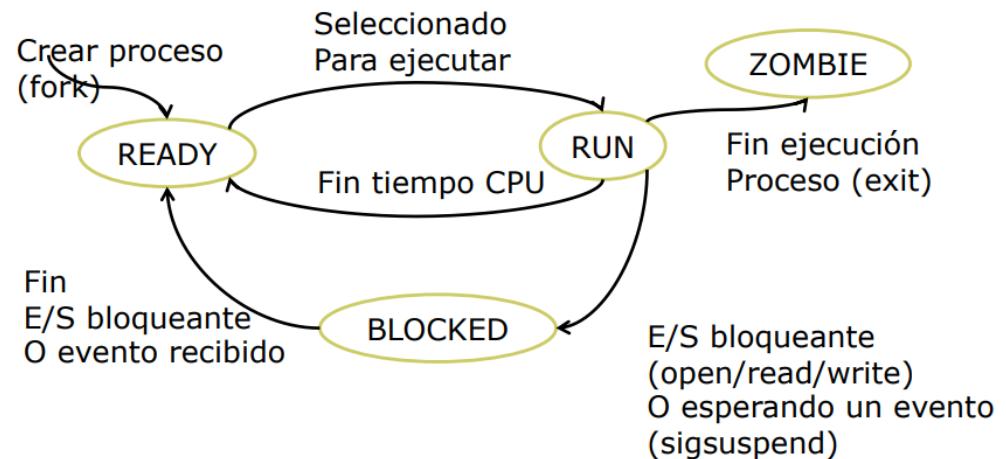
## Estados de un proceso

- Cada SO define un grafo de estados, indicando que eventos generan transiciones entre estados.
  - El grafo define qué transiciones son posibles y cómo se pasa de uno a otro
  - El grafo de estados, muy simplificado, podría ser:
    - run**: El proceso tiene asignada una CPU y está ejecutándose
    - ready**: El proceso está preparado para ejecutarse pero está esperando que se le asigne una CPU
    - blocked**: El proceso no tiene/consume CPU, está *bloqueado* esperando un que finalice una entrada/salida de datos o la llegada de un evento
    - zombie**: El proceso ha terminado su ejecución pero aún no ha desaparecido de las estructuras de datos del kernel
- Linux

## Hilos de Ejecución (Threads) – Para qué?

- Cuando y para qué se usan...
  - Explotar paralelismo (del código y de recursos hardware)
  - Encapsular tareas (programación modular)
  - Eficiencia en la E/S (Threads específicos para E/S)
  - Pipelining de solicitudes de servicio (para mantener QoS de servicios)
- Ventajas
  - Los threads tienen menor coste al crear/terminar y al cambiar de contexto (dentro del mismo proceso) que los procesos
  - Al compartir memoria entre threads de un mismo proceso, pueden intercambiar información sin llamadas al sistema
- Desventajas
  - Difícil de programar y *debuggar* debido a la memoria compartida
    - Problemas de sincronización y exclusión mutua
    - Ejecuciones incoherentes, resultados erróneos, bloqueos, etc.

## Ejemplo diagrama de estados



Los estados y las transiciones entre ellos dependen del sistema.  
Este diagrama es solo un ejemplo

## Ejemplo estados de un proceso

- Objetivo: Hay que entender la relación entre las características del SO y el diagrama de estados que tienen los procesos
  - Si el sistema es multiprogramado → READY, RUN
  - Si el sistema permite e/s bloqueante → BLOCKED
  - Etc
- SO con soporte para procesos con multiples threads
  - Estructuras para diferenciar threads del mismo proceso y gestionar estados de ejecución, entre otras cosas
    - ▶ P.Ej.: Light Weight Process (LWP) en SOs basados en Linux/UNIX

## Linux: Propiedades de un proceso

- Un proceso incluye, no sólo el programa que ejecuta, sino toda la información necesaria para diferenciar una ejecución del programa de otra.
  - **Toda esta información se almacena en el kernel, en el PCB.**
- En Linux, por ejemplo, las propiedades de un proceso se agrupan en tres: **la identidad, el entorno, y el contexto.**
- **Identidad**
  - Define quién es (identificador, propietario, grupo) y qué puede hacer el proceso (recursos a los que puede acceder)
- **Entorno**
  - Parámetros (argv en un programa en C) y variables de entorno (HOME, PATH, USERNAME, etc)
- **Contexto**
  - Toda la información que define el estado del proceso, todos sus recursos que usa y que ha usado durante su ejecución.

## Linux: Propiedades de un proceso (2)

- La **IDENTIDAD** del proceso define quien es y por lo tanto determina que puede hacer
  - **Process ID (PID)**
    - ▶ Es un identificador **ÚNICO** para el proceso. Se utiliza para identificar un proceso dentro del sistema. En llamadas a sistema identifica al proceso al cual queremos enviar un evento, modificar, etc
    - ▶ El kernel genera uno nuevo para cada proceso que se crea
- **Credenciales**
  - ▶ Cada proceso está asociado con un usuario (**userID**) y uno o más grupos (**groupID**). Estas credenciales determinan los derechos del proceso a acceder a los recursos del sistema y ficheros.

## Creación de procesos: opciones(Cont)

- Planificación
  - ▶ **El padre y el hijo se ejecutan concurrentemente (UNIX)**
    - El padre espera hasta que el hijo termina (se sincroniza)
- Espacio de direcciones (rango de memoria válido)
  - ▶ **El hijo es un duplicado del padre (UNIX), pero cada uno tiene su propia memoria física. Además, padre e hijo, en el momento de la creación, tienen el mismo contexto de ejecución (los registros de la CPU valen lo mismo)**
    - El hijo ejecuta un código diferente
- UNIX
  - **fork system call.** Crea un nuevo proceso. El hijo es un clon del padre
  - **exec system call.** Reemplaza (muta) el espacio de direcciones del proceso con un nuevo programa. El proceso es el mismo.

# Servicios básicos (UNIX)



Servicio	Llamada a sistema
Crear proceso	fork
Cambiar ejecutable=Mutar proceso	exec (execvp)
Terminar proceso	exit
Esperar a proceso hijo ( <b>bloqueante</b> )	wait/waitpid
Devuelve el PID del proceso	getpid
Devuelve el PID del padre del proceso	getppid

- Una llamada a sistema bloqueante es aquella que puede bloquear al proceso, es decir, forzar que deje el estado RUN (abandone la CPU) y pase a un estado en que no puede ejecutarse (WAITING, BLOCKED, ...., depende del sistema)

## Crear proceso: fork en UNIX



```
int fork();
```

- Un proceso crea un proceso nuevo. Se crea una relación jerárquica padre-hijo
- El padre y el hijo se ejecutan de forma **concurrente**
- La memoria del hijo se inicializa con una copia de la memoria del padre
  - Código/Datos/Pila
- El hijo inicia la ejecución en el punto en el que estaba el padre en el momento de la creación
  - Program Counter hijo= Program Counter padre
- Valor de retorno del fork es diferente (es la forma de diferenciarlos en el código):
  - ▶ Padre recibe el PID del hijo
  - ▶ Hijo recibe un 0.

## Creación de procesos y herencia



- El hijo HEREDA algunos aspectos del padre y otros no.
- **HEREDA** (recibe una copia privada de....)
  - El espacio de direcciones lógico (código, datos, pila, etc).
    - ▶ La memoria física es nueva, y contiene una copia de la del padre (en el tema 3 veremos optimizaciones en este punto)
  - La tabla de programación de signals
  - Los dispositivos virtuales
  - El usuario /grupo (credenciales)
  - Variables de entorno
- **NO HEREDA** (sino que se inicializa con los valores correspondientes)
  - PID, PPID (PID de su padre)
  - Contadores internos de utilización (Accounting)
  - Alarms y signals pendientes (son propias del proceso)

## Terminar ejecución/Esperar que termine



- Un proceso puede acabar su ejecución **voluntaria** (exit) o **involuntariamente** (signals)
- Cuando un proceso quiere finalizar su ejecución (**voluntariamente**), liberar sus recursos y liberar las estructuras de kernel reservadas para él, se ejecuta la llamada a sistema exit.
- Si queremos sincronizar el padre con la finalización del hijo, podemos usar waitpid: El proceso espera (si es necesario se bloquea el proceso) a que termine un hijo cualquiera o uno concreto
  - ▶ waitpid(-1,NULL,0) → Esperar (con bloqueo si es necesario) a un hijo cualquiera
  - ▶ waitpid(pid\_hijo,NULL,0)→ Esperar (con bloqueo si es necesario) a un hijo con pid=pid\_hijo
- El hijo puede enviar información de finalización (exit code) al padre mediante la llamada a sistema exit y el padre la recoge mediante wait o waitpid
  - ▶ El SO hace de intermediario, la almacena hasta que el padre la consulta
  - ▶ Mientras el padre no la consulta, el PCB no se libera y el proceso se queda en estado ZOMBIE (defunct)
    - Conviene hacer wait/waitpid de los procesos que creamos para liberar los recursos ocupados del kernel
  - Si un proceso muere sin liberar los PCB's de sus hijos el proceso init del sistema los libera

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



## pid\_t waitpid(pid\_t pid, int \*status, int options);

Parámetro pid==1 Estado hijos al hacer waitpid	options==0	options==WNOHANG
Algún hijo zombie	No se bloquea Trata la muerte de un zombie (no está especificado cual) Devuelve el pid del hijo tratado	Idem que options==0
Todos los hijos vivos	Se bloquea hasta que uno acaba (cualquiera) Trata la muerte del que haya acabado Devuelve el pid del hijo tratado	No se bloquea Devuelve 0
Parámetro pid==pidh Estado de pidh al hacer waitpid	options==0	options==WNOHANG
Zombie	No se bloquea Trata la muerte del hijo Devuelve pidh	Idem que flags=0
Vivo	Se bloquea hasta que acaba Trata la muerte del hijo Devuelve pidh	No se bloquea Devuelve 0

## Mutación de ejecutable: exec en UNIX



- Al hacer fork, el espacio de direcciones es el mismo. Si queremos ejecutar otro código, el proceso debe MUTAR (Cambiar el binario de un proceso)
- execvp: Un proceso cambia (muta) su propio ejecutable por otro ejecutable (pero el proceso es el mismo)
  - Todo el contenido del espacio de direcciones cambia, código, datos, pila, etc.
    - Se reinicia el contador de programa a la primera instrucción (main)
  - Se mantiene todo lo relacionado con la identidad del proceso
    - Contadores de uso internos, signals pendientes, etc
  - Se modifican aspectos relacionados con el ejecutable o el espacio de direcciones
    - Se define por defecto la tabla de programación de signals

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

## Creación procesos

### Caso 1: queremos que hagan líneas de código diferente

```
1. int ret=fork();
2. if (ret==0) {
3.   // estas líneas solo las ejecuta el hijo, tenemos 2 procesos
4. }else if (ret<0){
5.   // En este caso ha fallado el fork, solo hay 1 proceso
6. }else{
7.   // estas líneas solo las ejecuta el padre, tenemos 2 procesos
8. }
9. // estas líneas las ejecutan los dos
```

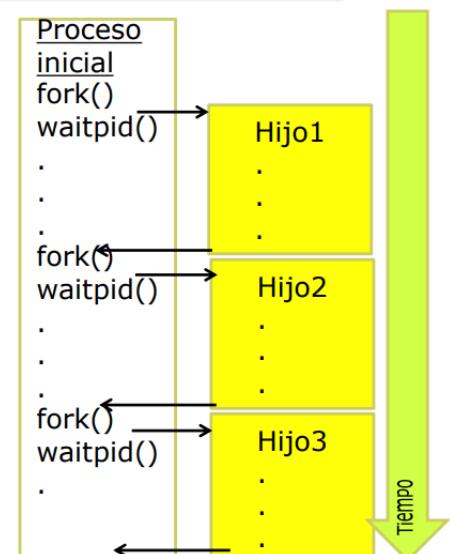
### Caso 2: queremos que hagan lo mismo

```
1. fork();
2. // Aquí, si no hay error, hay 2 procesos
```

## Esquema Secuencial

Secuencial: Forzamos que el padre espere a que termine un hijo antes de crear el siguiente.

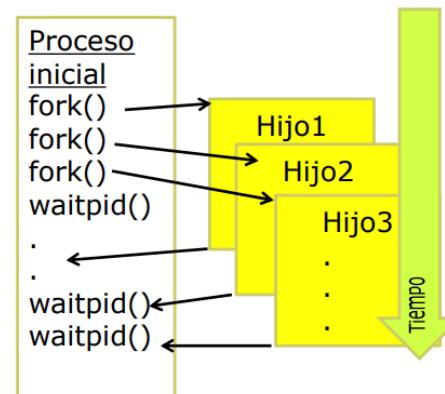
```
1. #define num_procs 2
2. int i,ret;
3. for(i=0;i<num_procs;i++){
4.   if ((ret=fork())<0) control_error();
5.   if (ret==0) {
6.     // estas líneas solo las ejecuta el
7.     // hijo
8.     codigohijo();
9.     exit(0); //
10.  }
11.  waitpid(-1,NULL,0);
12. }
```



## Esquema Concurrente

Concurrente; Primero creamos todos los procesos, que se ejecutan concurrentemente, y después esperamos que acaben..

```
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.         // estas líneas solo las ejecuta el
7.         // hijo
8.         codigohijo();
9.         exit(0); //
10.    }
11. }
12. while( waitpid(-1,NULL,0)>0);
```



## Ejemplo: exec

- Cuando en el *shell* ejecutamos el siguiente comando:

```
% ls -l
```

1. Se crea un nuevo proceso (*fork*)
2. El nuevo proceso cambia la imagen, y ejecuta el programa ls (*exec*)

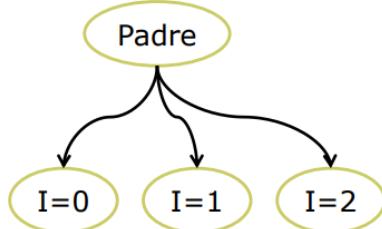
- Como se implementa esto?



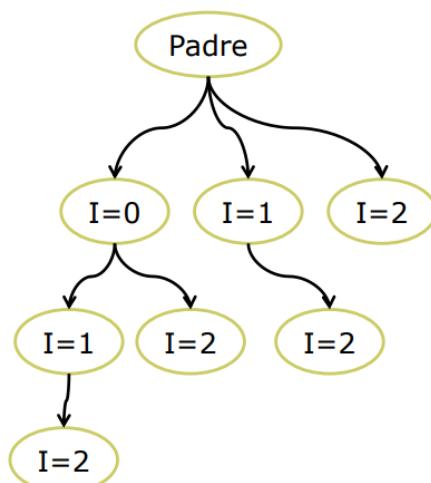
```
...
ret = fork();
if (ret == 0) {
    execvp("/bin/ls", "ls", "-l", (char *)NULL);
}
// A partir de aquí, este código sólo lo ejecutaría el padre
...
```

## Árbol de procesos (examen)

Con exit



Sin exit



## Terminación de procesos. exit

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

kernel

PCB (del proceso "A")
pid=...
exit\_code=   
...

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

Se consulta del PCB

Se guarda en el PCB

Sin embargo, exit\_code no vale 4!!! Hay que procesar el resultado

## Ejemplo: fork/exec/waitpid

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

void main(int argc, char *argv[])
{
    ...
    num_cmd = argc-1;
    for (i = 0; i < num_cmd; i++)
        lanzaCmd(argv[i+1]);
    // waitpid format
    // ret: pid del proceso que termina o -1
    // arg1 == -1 → espera a un proceso hijo cualquiera
    // arg2 exit_code → variable donde el kernel nos copiara el valor de
    // finalización
    // arg3 == 0 → BLOQUEANTE
    while ((pid = waitpid(-1, &exit_code, 0) > 0)
        trataExitCode(pid, exit_code);
    exit(0);
}

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

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



## trataExitCode

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

// PROGRAMADLA para los LABORATORIOS
void trataExitCode(int pid, int exit_code)
{
    int exit_code, statcode, signcode;
    char buffer[128];

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



- Para poder cooperar, los procesos necesitan comunicarse
  - Interprocess communication (IPC) = Comunicación entre procesos
- Para comunicar datos hay 2 modelos principalmente
  - Memoria compartida (Shared memory)
    - ▶ Los procesos utilizan variables que pueden leer/escribir
  - Paso de mensajes (Message passing)
    - ▶ Los procesos utilizan funciones para enviar/recibir datos

## Comunicación entre procesos en Linux

- Signals – Eventos enviados por otros procesos (del mismo usuario) o por el kernel para indicar determinadas condiciones (Tema 2)
- Pipes – Dispositivo que permite comunicar dos procesos que se ejecutan en la misma máquina. Los primeros datos que se envían son los primeros que se reciben. La idea principal es conectar la salida de un programa con la entrada de otro. Utilizado principalmente por la shell (Tema 4)
- FIFOs – Funciona con pipes que tienen un nombre el sistema de ficheros. Se ofrecen como pipes con nombre. (Tema 4)
- Sockets – Dispositivo que permite comunicar dos procesos a través de la red
- Message queues – Sistema de comunicación indirecta
- Semaphores - Contadores que permiten controlar el acceso a recursos compartidos. Se utilizan para prevenir el acceso de más de un proceso a un recurso compartido (por ejemplo memoria)
- Shared memory – Memoria accesible por más de un proceso a la vez (Tema 3)

## Tipos de signals y tratamientos(2)

### Algunos signals

Nombre	Acción Defecto	Evento
SIGCHLD	IGNORAR	Un proceso hijo ha terminado o ha sido parado
SIGCONT		Continua si estaba parado
SIGSTOP	STOP	Parar proceso
SIGINT	TERMINAR	Interrumpido desde el teclado (Ctrl-C)
SIGALRM	TERMINAR	El contador definido por la llamada alarm ha terminado
SIGKILL	TERMINAR	Terminar el proceso
SIGSEGV	CORE	Referencia inválida a memoria
SIGUSR1	TERMINAR	Definido por el usuario (proceso)
SIGUSR2	TERMINAR	Definido por el usuario (proceso)



## Tipos de signals y tratamientos(3)

- El tratamiento de un signal funciona como una interrupción provocada por software:
  - Al recibir un signal el proceso interrumpe la ejecución del código, pasa a ejecutar el tratamiento que ese tipo de signal tenga asociado y al acabar (si sobrevive) continúa donde estaba
  
- Los procesos pueden **bloquear/desbloquear** la recepción de **cada signal excepto SIGKILL y SIGSTOP** (tampoco se pueden bloquear los signals SIGFPE, SIGILL y SIGSEGV si son provocados por una excepción)
  - Cuando un proceso bloquea un signal, si se le envía ese signal el proceso no lo recibe y el sistema lo marca como pendiente de tratar
    - ▶ bitmap asociado al proceso, sólo recuerda un signal de cada tipo
  - Cuando un proceso desbloquea un signal recibirá y tratará el signal pendiente de ese tipo

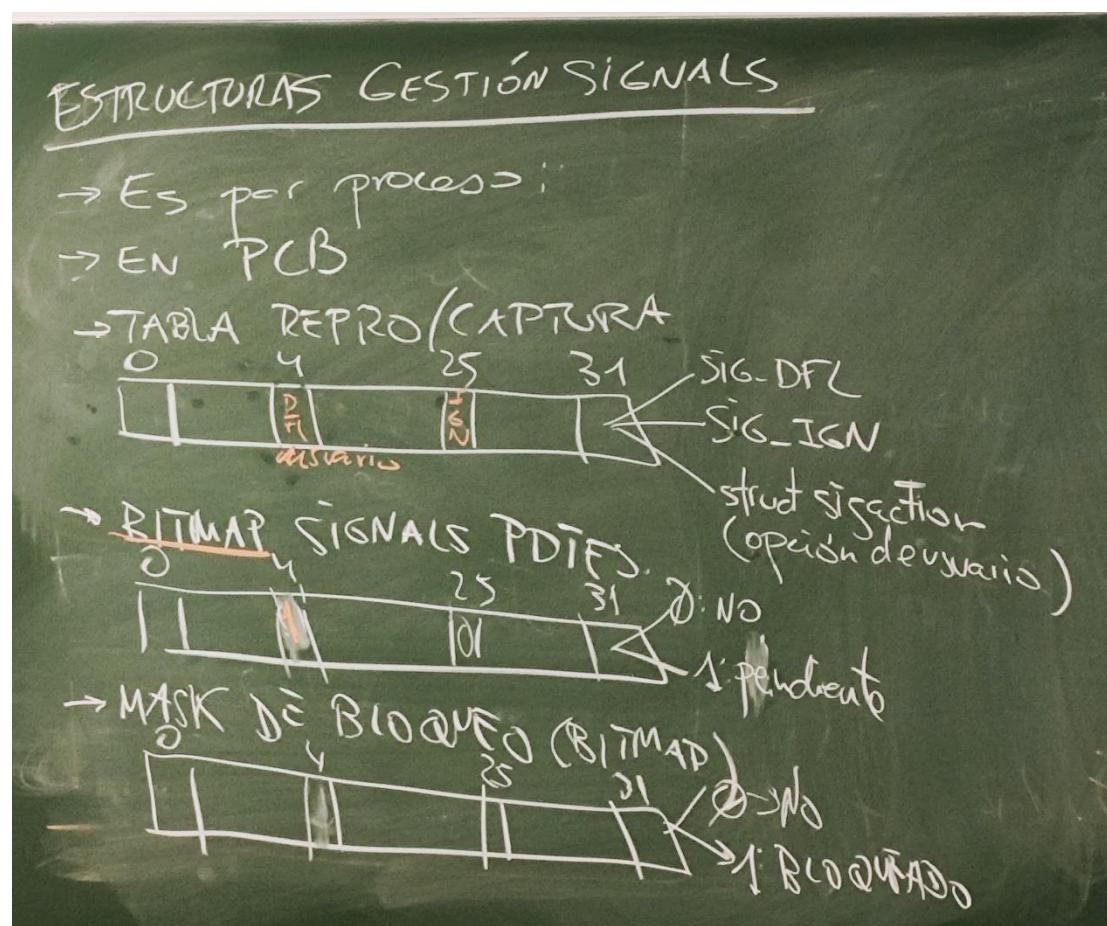
## Estructuras de datos del kernel

- La gestión de signals es por proceso, la información de gestión está en el PCB
  - Cada proceso tiene una **tabla de programación de signals** (1 entrada por signal),
    - ▶ Se indica que acción realizar cuando se reciba el evento
  - Un bitmap de **eventos pendientes** (1 bit por signal)
    - ▶ No es un contador, actúa como un booleano
  - Un único **temporizador** para la alarma
    - ▶ Si programamos 2 veces la alarma solo queda la última
  - Una **máscara de bits** para indicar qué signals hay que tratar

## Linux: Interfaz relacionada con signals

Servicio	Llamada sistema
Enviar un signal concreto	kill
Capturar/reprogramar un signal concreto	sigaction
Bloquear/desbloquear signals	sigprocmask
Esperar HASTA que llega un evento cualquiera (BLOQUEANTE)	sigsuspend
Programar el envío automático del signal SIGALRM (alarma)	alarm

- Fichero con signals: `/usr/include/bits/signum.h`
- Hay varios interfaces de gestión de signals incompatibles y con diferentes problemas, Linux implementa el interfaz POSIX



# Interfaz: Enviar / Capturar signals

## Para enviar:

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

- signum → SIGUSR1, SIGUSR2, etc

- Requerimiento: conocer el PID del proceso destino

## Para capturar un SIGNAL y ejecutar una función cuando llegue:

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

- signum → SIGUSR1, SIGUSR2, etc
- tratamiento → struct sigaction que describe qué hacer al recibir el signal
- tratamiento\_antiguo → struct sigaction que describe qué se hacía hasta ahora. Este parámetro puede ser NULL si no interesa obtener el tratamiento antiguo



## Definición de struct sigaction

### struct sigaction: varios campos. Nos fijaremos sólo en 3:

- sa\_handler: puede tomar 3 valores

- SIG\_IGN: ignorar el signal al recibirllo
- SIG\_DFL: usar el tratamiento por defecto
- función de usuario con una cabecera predefinida: void nombre\_funcion(int s);

- IMPORTANTE: la función la invoca el kernel. El parámetro se corresponde con el signal recibido (SIGUSR1, SIGUSR2, etc), así se puede asociar la misma función a varios signals y hacer un tratamiento diferenciado dentro de ella.

- sa\_mask: signals que se añaden a la máscara de signals que el proceso tiene bloqueados

- Si la máscara está vacía sólo se añade el signal que se está capturando
- Al salir del tratamiento se restaura la máscara que había antes de entrar

- sa\_flags: para configurar el comportamiento (si vale 0 se usa la configuración por defecto). Algunos flags:

- SA\_RESETHAND: después de tratar el signal se restaura el tratamiento por defecto del signal
- SA\_RESTART: si un proceso bloqueado en una llamada a sistema recibe el signal se reinicia la llamada que lo ha bloqueado

## CAPTURA

int sigaction(N SIG, &trat Nuevo,  
&trat Antiguo )

### struct sigaction

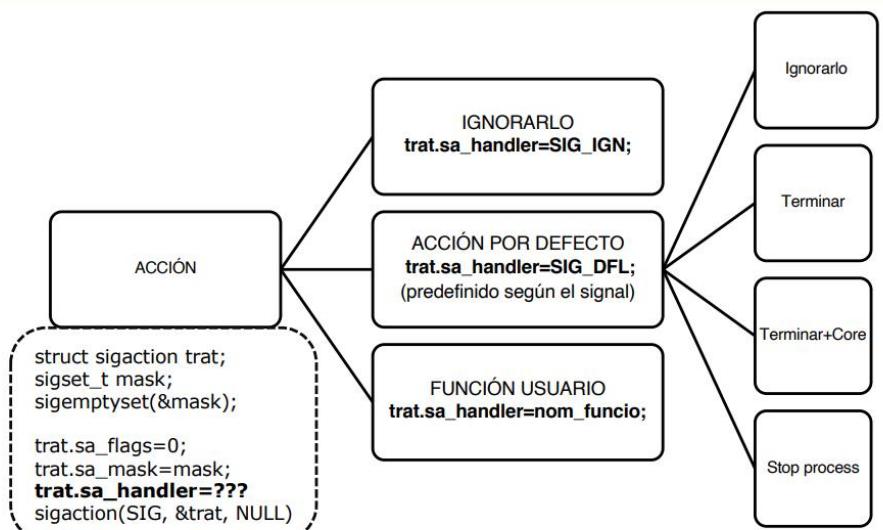
sa\_handler → routine que queréis  
RAS RASF

sa\_flags →

sa\_mask → bitmap

↳ Se añade (1) a la  
MASK del proceso durante  
la ejecución del handler

## Acciones posibles al recibir un signal



Donde SIG debe ser el nombre de un signal: SIGUSR1, SIGALRM, SIGUSR2, etc

# Manipulación de máscaras de signals

- `sigemptyset`: inicializa una máscara sin signals

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



- `sigfillset`: inicializa una máscara con todos los signals

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



- `sigaddset`: añade el signal a la máscara que se pasa como parámetro

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



- `sigdelset`: elimina el signal de la máscara que se pasa como parámetro

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



- `sigismember`: devuelve cierto si el signal está en la máscara

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



## Ejemplo: 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);
}
```

Podéis encontrar el código completo en: [signal\\_basico.c](#)

## Bloquear/desbloquear signals

- El proceso puede controlar en qué momento quiere recibir los signals

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



- Operación puede ser:

- `SIG_BLOCK`: **añadir** los signals que indica *mascara* a la máscara de signals bloqueados del proceso
- `SIG_UNBLOCK`: **quitar** los signals que indica *mascara* a la máscara de signals bloqueados del proceso
- `SIG_SETMASK`: hacer que la máscara de signals bloqueados del proceso pase a ser el parámetro *mascara*

## Esperar un evento



- Esperar (bloqueado) a que llegue un evento

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

- Bloquea al proceso hasta que llega un evento cuyo tratamiento no sea `SIG_IGN`
- **Mientras** el proceso está bloqueado en el `sigsuspend` *mascara* será los signals que no se recibirán (signals bloqueados),
  - Así se puede controlar qué signal saca al proceso del bloqueo
- Al salir de `sigsuspend` automáticamente se restaura la máscara que había y se tratarán los signals pendientes que se estén desbloqueando

## Sincronización: A envía un signal a B (2)

- El proceso A envía (en algún momento) un signal a B, B está esperando un evento y ejecuta una acción al recibirlo

Proceso A

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

- sigprocmask bloquea *evento*, así que si llega antes de que B llegue al sigsuspend no se le entrega
- Cuando B está en el sigsuspend el único evento que le puede desbloquear es el que se usa para la sincronización con 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);  
....  
}
```

## Control de tiempo: programar temporizado

- Programar un envío automático (lo envía 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;
```

## Alternativas en la sincronización de procesos

- Alternativas para “esperar” la recepción de un evento

1. **Espera activa:** El proceso consume cpu para comprobar si ha llegado o no el evento. Normalmente comprobando el valor de una variable
  - Ejemplo: while(!recibido);
2. **Bloqueo:** El proceso libera la cpu (se bloquea) y será el kernel quien le despierte a la recepción de un evento
  - Ejemplo: sigsuspend

- Si el tiempo de espera es corto se recomienda espera activa

- No compensa la sobrecarga necesaria para ejecutar el bloqueo del proceso y el cambio de contexto

- Para tiempos de espera largos se recomienda bloqueo

- Se aprovecha la CPU para que el resto de procesos (incluido el que estamos esperando) avancen con su ejecución

## Control de tiempo: Uso del temporizador

- El proceso programa un temporizador de 2 segundos y se bloquea hasta que pasa ese tiempo

```
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 proceso estará bloqueado en el sigsuspend, cuando pasen 2 segundos recibirá el SIGALRM, se ejecutará la función y luego continuará donde estaba

# Relación con fork y exec

## FORK: Proceso nuevo

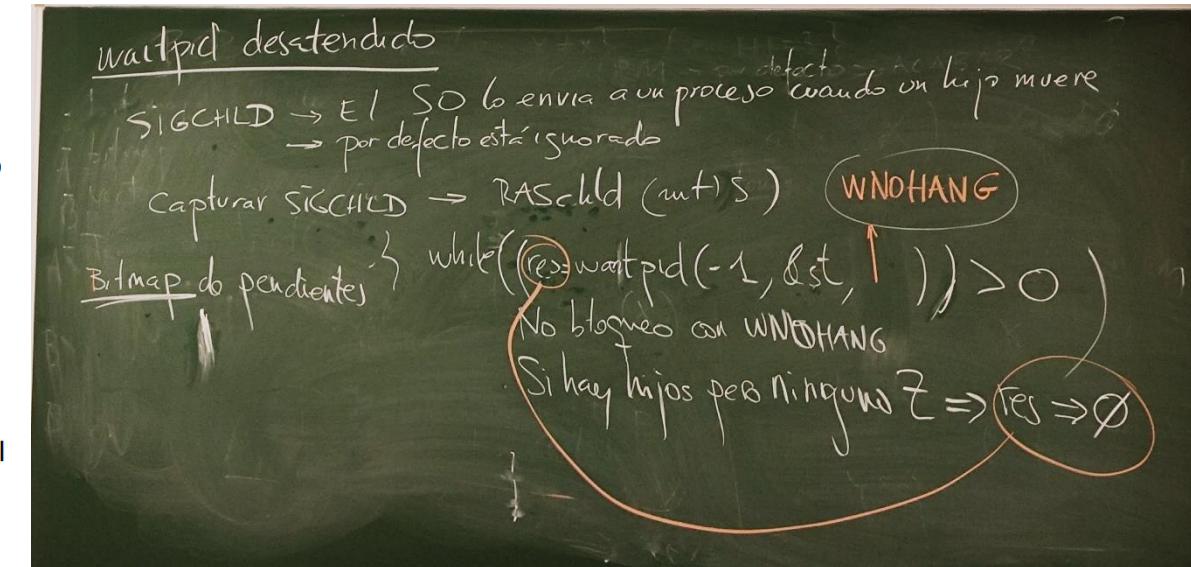
- El hijo hereda la tabla de acciones asociadas a los signals del proceso padre
- La máscara de signals bloqueados se hereda
- Los eventos son enviados a procesos concretos (PID's), el hijo es un proceso nuevo → La lista de eventos pendientes se borra (tampoco se heredan los temporizadores pendientes)

## EXECLP: Mismo proceso, cambio de ejecutable

- La tabla de acciones asociadas a signals se pone por defecto ya que el código es diferente
- Los eventos son enviados a procesos concretos (PID's), el proceso no cambia → La lista de eventos pendientes se conserva
- La máscara de signals bloqueados se conserva

## Ejemplo 2: espera activa vs bloqueo (1)

```
void main()
{
    configurar_esperar_alarma()
    trat.sa_flags = 0;
    trat.sa_handler=f_alarma;
    sigemptyset(&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);
}
```



## Ejemplo 2: espera activa vs bloqueo (2)



### Opción 1: espera activa

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



### Opción 2: bloqueo

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

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



## Datos: Process Control Block (PCB)

- Es la información asociada con cada proceso, depende del sistema, pero normalmente incluye, por cada proceso, aspectos como:
  - El identificador del proceso (PID)
  - Las credenciales: usuario, grupo
  - El estado : RUN, READY,...
  - Espacio para salvar los registros de la CPU
  - Datos para gestionar signals
  - Información sobre la planificación
  - Información de gestión de la memoria
  - Información sobre la gestión de la E/S
  - Información sobre los recursos consumidos (Accounting )

## Estructuras para organizar los procesos: Colas/listas de planificación

- El SO organiza los PCB's de los procesos en estructuras de gestión: vectores, listas, colas. Tablas de hash, árboles, en función de sus necesidades
- Los procesos en un mismo estado suelen organizarse en colas o listas que permiten mantener un orden
- Por ejemplo:
  - Cola de procesos – Incluye todos los procesos creados en el sistema
  - Cola de procesos listos para ejecutarse (ready) – Conjunto de procesos que están listos para ejecutarse y están esperando una CPU
    - ▶ En muchos sistemas, esto no es 1 cola sino varias ya que los procesos pueden estar agrupados por clases, por prioridades, etc
  - Colas de dispositivos– Conjunto de procesos que están esperando datos del algún dispositivo de E/S
  - El sistema mueve los procesos de una cola a otra según corresponda
    - ▶ Ej. Cuando termina una operación de E/S , el proceso se mueve de la cola del dispositivo a la cola de ready.

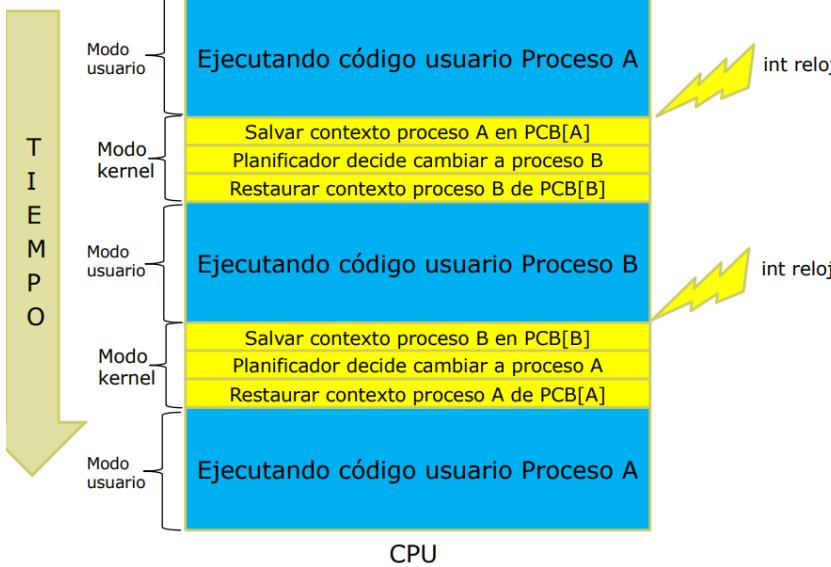
## Planificación

- Hay determinadas situaciones que provocan que se deba ejecutar la planificación del sistema
- Casos en los que el proceso que está RUN no puede continuar la ejecución → Hay que elegir otro → Eventos no preemptivos
  - Ejemplo: El proceso termina, El proceso se bloquea
- Casos en los que el proceso que está RUN podría continuar ejecutándose pero por criterios del sistema se decide pasarlo a estado READY y poner otro en estado RUN → La planificación elige otro pero es forzado → evento preemptivo
  - Estas situaciones dependen de la política, cada política considera algunos si y otros no)
  - Ejemplos: El proceso lleva X ms ejecutándose (RoundRobin), Creamos un proceso nuevo, se desbloquea un proceso,....
- Las políticas de planificación son preemptivas (apropiativas) o no preemptivas (no apropiativas)
  - No preemptiva: La política no le quita la cpu al proceso, él la "libera". Sólo soporta eventos no preemptivos. (eventos tipo 1 y 2)
  - Preemptiva: La política le quita la cpu al proceso. Soporta eventos preemptivos (eventos tipo 3) y no preemptivos.

## Mecanismos utilizados por el planificador

- Cuando un proceso deja la CPU y se pone otro proceso se ejecuta un cambio de contexto (de un contexto a otro)
- Cambios de contexto(Context Switch)
  - El sistema tiene que salvar el estado del proceso que deja la cpu y restaurar el estado del proceso que pasa a ejecutarse
    - ▶ El contexto del proceso se suele salvar en los datos de kernel que representan el proceso (PCB). Hay espacio para guardar esta información
  - **El cambio de contexto no es tiempo útil de la aplicación, así que ha de ser rápido.** A veces el hardware ofrece soporte para hacerlo más rápido
    - ▶ Por ejemplo para salvar todos los registros o restaurarlos de golpe
  - Diferencias entre cambio de contexto entre threads
    - ▶ Mismo proceso vs distintos procesos

## Mecanismo cambio contexto



## Round Robin (RR)

- El sistema tiene organizados los procesos en función de su estado
- Los procesos están encolados por orden de llegada
- Cada proceso recibe la CPU durante un periodo de tiempo (time quantum), típicamente 10 ó 100 miliseg.

  - El planificador utiliza la interrupción de reloj para asegurarse que ningún proceso monopoliza la CPU

- Eventos que activan la política Round Robin:
  1. Cuando el proceso se bloquea (no preemptivo)
  2. Cuando termina el proceso (no preemptivo)
  3. Cuando termina el quantum (preemptivo)
- Es una política **a apropiativa** o preemptiva
- Cuando se produce uno de estos eventos, el proceso que está run deja la cpu y se selecciona el siguiente de la cola de ready.
  - Si el evento es 1, el proceso se añade a la cola de bloqueados hasta que termina el acceso al dispositivo
  - Si el evento es el 2, el proceso pasaría a zombie en el caso de linux o simplemente terminaría
  - Si el evento es el 3, el proceso se añade al final de la cola de ready

## Rendimiento de la política

- Si hay N procesos en la cola de ready, y el quantum es de Q milisegundos, cada proceso recibe  $1/n$  partes del tiempo de CPU en bloques de Q milisegundos como máximo.
  - ▶ Ningún proceso espera más de  $(N-1)Q$  milisegundos.
- La política se comporta diferente en función del quantum
  - ▶  $q$  muy grande  $\Rightarrow$  se comporta como en orden secuencial. Los procesos recibirían la CPU hasta que se bloquearan
  - ▶  $q$  pequeño  $\Rightarrow$   $q$  tiene que ser grande comparado con el coste del cambio de contexto. De otra forma hay demasiado overhead.

## Completely Fair Scheduling

- Algoritmo usado en las versiones actuales de Linux
- Métrica objetivo: tiempo de uso de CPU de todos los procesos tiene que ser equivalente
  - Round Robin penaliza a los procesos intensivos en E/S
- Tiempo máximo de uso consecutivo de CPU (~quantum) es variable
  - Teóricamente, tiempo consumido de CPU para cada proceso debería ser el resultado de dividir el tiempo que lleva en ejecución entre el número de procesos que compiten por la CPU
  - A cada proceso se le asigna la CPU hasta que se bloquee, acabe o su tiempo de CPU alcance el teórico que debería tener
- Prioridad  $\rightarrow$  distancia al tiempo teórico de CPU (cuanto más lejos esté más prioritario)
- Crea grupos de procesos (criterio configurable por el administrador de la máquina) y permite contabilizar el uso de CPU por grupo
  - Objetivo: impedir que un usuario que ejecuta muchos procesos acapare la máquina

# Seguridad

- La seguridad ha de considerarse a cuatro niveles:
  - Físico
    - Las máquinas y los terminales de acceso deben encontrarse en un habitaciones/edificios seguros.
  - Humano
    - Es importante controlar a quien se concede el acceso a los sistemas y concienciar a los usuarios de no facilitar que otras personas puedan acceder a sus cuentas de usuario
  - Sistema Operativo
    - Evitar que un proceso(s) sature el sistema
    - Asegurar que determinados servicios están siempre funcionando
    - Asegurar que determinados puertos de acceso no están operativos
    - Controlar que los procesos no puedan acceder fuera de su propio espacio de direcciones
  - Red
    - La mayoría de datos hoy en día se mueven por la red. Este componente de los sistemas es normalmente el más atacado.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

void error_y_exit(char *s, int error) {
    perror(s);
    exit(error);
}

int main(int argc, char* argv[]) {
    for (int i = 1; i < argc; ++i) {
        int ret = fork();
        char s[50];
        switch (ret) {
            case 0:
                sprintf(s, "Soy el proceso HIJO: %d de %s\n", getpid(), argv[i]);
                write(1, s, strlen(s));
                exit(0);
            break;
            case -1:
                sprintf(s, "Ha fallado el fork del proceso: %d\n", getpid());
                error_y_exit(s, 1);
            break;
            default:
                waitpid(-1, NULL, 0);
            break;
        }
    }
}
```

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

void error_y_exit(char *s, int error) {
    perror(s);
    exit(error);
}

/* Ejecuta el comando ps -u username mediante la llamada al sistema execvp */
/* Devuelve: el código de error en el caso de que no se haya podido mutar */
void muta_a_PS(char *username) {
    execvp("ps", "ps", "-u", username, (char*)NULL);
    error_y_exit("Ha fallado la mutación al ps", 1);
}

int main(int argc, char* argv[]) {
    if (argc == 2) {
        int ret = fork();
        char s[50];
        switch (ret) {
            case 0:
                sprintf(s, "Soy el proceso HIJO: %d de %s\n", getpid(), argv[1]);
                write(1, s, strlen(s));
                muta_a_PS(argv[1]);
                while(1);
            break;
            case -1:
                sprintf(s,"Ha fallado el fork del proceso: %d\n", getpid());
                error_y_exit(s, 1);
            break;
            default:
                sprintf(s, "Soy el proceso PADRE: %d\n", getpid());
                write(1, s, strlen(s));
                while(1);
            break;
        }
    }
}

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

void error_y_exit(char *s, int error) {
    perror(s);
    exit(error);
}

int main(int argc, char* argv[]) {
    for (int i = 0; i < 4; ++i) {
        int pid = fork();
        switch (pid) {
            case 0:
                if (i == 0) execvp("./listaParametros", "a", "b", (char*)0);
                if (i == 1) execvp("./listaParametros", (char*)0);
                if (i == 2) execvp("./listaParametros", "25", "4", (char*)0);
                if (i == 3) execvp("./listaParametros", "1024", "hola", "adios", (char*)0);
                break;
            case -1:
                sprintf(s,"Ha fallado el fork del proceso: %d\n", getpid());
                error_y_exit(s, 1);
            break;
            default:
                break;
        }
    }
    while (wait(NULL) > 0);
}
```

# Constantes

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

void error_y_exit(char *s, int error) {
    perror(s);
    exit(error);
}

int main(int argc, char* argv[]) {
    for (int i = 1; i < argc; ++i) {
        int ret = fork();
        char s[50];
        switch (ret) {
            case 0:
                sprintf(s, "Soy el proceso HIJO: %d de %s\n", getpid(), argv[i]);
                write(1, s, strlen(s));
                exit(0);
            break;
            case -1:
                sprintf(s, "Ha fallado el fork del proceso: %d\n", getpid());
                error_y_exit(s, 1);
            break;
            default:
                sprintf(s, "Soy el proceso PADRE: %d\n", getpid());
                write(1, s, strlen(s));
            break;
        }
    }
    while(waitpid(-1, NULL, 0) > 0);
    char c;
    read(1, &c, sizeof(char));
}
```

## Primer programa: suma

```
#include <iostream>
using namespace std;
// This program reads two numbers and
// writes their sum
int main() {
    int x, y;
    cin >> x >> y;
    int s = x + y;
    cout << s << endl;
}
```

```
#include <stdio.h>
#include <string.h>
#define STDOUT 1
//This program receives two numbers //and writes their sum
int main(int argc,char *argv[])
{
    int x,y;
    char buff[128];
    x=atoi(argv[1]);
    y=atoi(argv[2]);
    int s=x+y;
    sprintf(buff,"%d\n",s);
    write(STDOUT,buff,strlen(buff));
}
```

- const tipo\_dato  
nombre\_variable=valor;
- #define nombre\_variable  
valor

## Decompose\_time

```
...include <iostream>
using namespace std;
// This program reads a natural number
// that represents an amount
// of time in seconds and writes the
// decomposition in hours,
// minutes and seconds
int main()
{
    int N;
    cin >> N;
    int h = N / 3600;
    int m = (N % 3600) / 60;
    int s = N % 60;
    cout << h << " hours, " << m << " minutes
    and "
    << s << " seconds" << endl;
}
```

```
#include <stdio.h>
#include <string.h>
#define STDOUT 1
// This program receives a natural number that
represents an amount
// of time in seconds and writes the
decomposition in hours,
// minutes and seconds
int main(int argc, char *argv[])
{
    int N;
    N=atoi(argv[1]);
    int h = N / 3600;
    int m = (N % 3600) / 60;
    int s = N % 60;
    char buff[128];
    sprintf(buff,"%d hours, %d minutes and %d
seconds\n",h,m,s);
    write(STDOUT,buff,strlen(buff));
}
```

// Pre: A is a non-empty vector  
// Post: returns the min value of  
the vector  
int minimum(const vector<int>&  
A) {  
 int n = A.size();  
 int m = A[0]; // visits A[0]  
// loop to visit A[1..n-1]  
for (int i = 1; i < n; ++i) {  
 if (A[i] < m) m = A[i];
}
return m;

// Pre: A is a non-empty vector  
// Post: returns the min value of  
the vector  
int minimum(int \*A, int size\_A) {  
 int n = size\_A;  
 int m = A[0]; // visits A[0]  
// loop to visit A[1..n-1]  
for (int i = 1; i < n; ++i) {  
 if (A[i] < m) m = A[i];
}
return m;
}

- int x,i,j;
  - Arithmetic operators: +, -, \*, /, %
  - char a,b,c;
  - bool A;
  - string
- No se pueden aplicar operadores básicos de string en C, hay que usar:  
- strlen :para calcular la longitud “usada” de un string,  
(es diferente del tamaño)  
- strcmp: para comparar dos strings

- char(i), int('a')ç
  - Visibilidad
  - Vectores
  - vector<type> name(n);
  - vector<int> S(n);
  - int x=S[0];
- (char) i, (int)'a'  
Visibilidad (igual)  
Vectores  
tipo name[n];  
int S[n];  
int x=S[0];
- No existe vector.h en C, sólo hay operaciones básicas.  
Para conocer el tamaño en BYTES de cualquier variable tenemos la función sizeof

```

void error_y_exit(char *msg,int exit_status)
{
    perror(msg);
    exit(exit_status);
}

/* ESTA VARIABLE SE ACCEDE DESDE LA FUNCION DE ATENCION AL SIGNAL Y DESDE EL MAIN */
int segundos=0;
/* FUNCION DE ATENCION AL SIGNAL SIGALRM */
void funcion_alarma(int s)
{
    if (s == SIGALRM) segundos=segundos+10;
    else {
        char buff[256];
        sprintf(buff, "ALARMA pid=%d: %d segundos\n",getpid(),segundos);
        write(1, buff, strlen(buff));
    }
}
int main (int argc,char * argv[])
{
    struct sigaction sa;
    sigset(SIG_BLOCK,&mask);

    sigemptyset(&mask);
    sigaddset(&mask, SIGALRM);
    sigaddset(&mask, SIGUSR1);
    sigprocmask(SIG_BLOCK,&mask, NULL);

    /* REPROGRAMAMOS EL SIGNAL SIGALRM */
    sa.sa_handler = &funcion_alarma;
    sa.sa_flags = SA_RESTART;
    sigfillset(&sa.sa_mask);

    if (sigaction(SIGALRM, &sa, NULL) < 0 || sigaction(SIGUSR1, &sa, NULL) < 0) error_y_exit("sigaction", 1);

    while (segundos<100)
    {
        alarm(10); /* Programamos la alarma para dentro de 10 segundos */
        /* Nos bloqueamos a esperar que nos llegue un evento */
        sigfillset(&mask);
        sigdelset(&mask, SIGALRM);
        sigdelset(&mask, SIGINT);
        sigdelset(&mask, SIGUSR1);
        sigsuspend(&mask);
    }
    exit(1);
}

all: ejemplo_alarm2 bucleInfinito ejemplo_alarm3 ejemplo_signal ejemplo_signal2 eventos eventos2

ejemplo_alarm2: ejemplo_alarm2.c
>> gcc -o ejemplo_alarm2 ejemplo_alarm2.c
bucleInfinito: bucleInfinito.c
>> gcc -o bucleInfinito bucleInfinito.c
ejemplo_alarm3: ejemplo_alarm3.c
>> gcc -o ejemplo_alarm3 ejemplo_alarm3.c
ejemplo_signal: ejemplo_signal.c
>> gcc -o ejemplo_signal ejemplo_signal.c
ejemplo_signal2: ejemplo_signal2.c
>> gcc -o ejemplo_signal2 ejemplo_signal2.c
eventos: eventos.c
>> gcc -o eventos eventos.c
eventos2: eventos2.c
>> gcc -o eventos2 eventos2.c
signal_perdido2: signal_perdido2.c
>> gcc -o signal_perdido2 signal_perdido2.c

clean:
>> rm ejemplo_alarm2
>> rm bucleInfinito
>> rm ejemplo_alarm3
>> rm ejemplo_signal
>> rm ejemplo_signal2
>> rm eventos
>> rm eventos2
>> rm signal_perdido2

```

```

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>

void error_y_exit(char *msg,int exit_status)
{
    perror(msg);
    exit(exit_status);
}

int segundos=0;
void funcion_alarma(int signal)
{
    char buff[256];
    segundos=segundos+10;
    sprintf(buff, "ALARMA pid=%d: %d segundos\n",getpid(),segundos);
    write(1, buff, strlen(buff));
}

int main (int argc,char * argv[])
{
    struct sigaction sa;
    sigset(SIG_BLOCK,&mask);

    /* EVITAMOS QUE NOS LLEGUE EL SIGALRM FUERA DEL SIGSUSPEND */
    sigemptyset(&mask);
    sigaddset(&mask, SIGALRM);
    sigprocmask(SIG_BLOCK,&mask, NULL);

    int pid;
    while (segundos<100) {
        alarm(10);
        pid = fork();
        if (pid < 0) error_y_exit("fork", 1);
        else if (pid == 0) {
            /*
             * //REPROGRAMAMOS EL SIGNAL SIGALRM
            sa.sa_handler = &funcion_alarma;
            sa.sa_flags = SA_RESTART;
            sigfillset(&sa.sa_mask);
            if (sigaction(SIGALRM, &sa, NULL) < 0) error_y_exit("sigaction", 1);
            */
            execvp("./bucleInfinito", "./bucleInfinito", NULL);
        }

        sigfillset(&mask);
        sigdelset(&mask, SIGALRM);
        sigdelset(&mask, SIGINT);
        sigsuspend(&mask);
    }
    exit(1);
}

```

```

int contador = 0;
int hijos = 0;

void error_y_exit(char* msg, int exit_status)
{
    perror(msg);
    exit(exit_status);
}

void trata_hijo(int s) {
    int pid, exit_code;
    char buff[256];
    while ((pid = waitpid(-1, &exit_code, WNOHANG)) > 0) {
        if (WIFEXITED(exit_code)) {
            int statcode = WEXITSTATUS(exit_code);
            sprintf(buff, "Termina el proceso %d con exit code %d\n", pid, statcode);
        }
        else {
            int signcode = WTERMSIG(exit_code);
            sprintf(buff, "Han matado al proceso %d antes de acabar alarm por el signal %d\n", pid, signcode);
        }
        write(1, buff, strlen(buff));
        hijos--;
        ++contador;
    }
}

void trata_alarma(int s)
{
    int main(int argc, char* argv[])
    {
        int pid, res;
        char buff[256];
        struct sigaction sa;
        sigset(SIG_BLOCK, &mask);
        int pid_vec[10];

        /* Evitamos recibir el SIGALRM fuera del sigsuspend */
        sigemptyset(&mask);
        sigadd(SIGALRM);
        sigprocmask(SIG_BLOCK, &mask, NULL);

        for (hijos = 0; hijos < 10; hijos++) {
            sprintf(buff, "Creando el hijo numero %d\n", hijos);
            write(1, buff, strlen(buff));

            pid = fork();
            if (pid == 0) /* Esta linea la ejecutan tanto el padre como el hijo */
            {
                sa.sa_handler = &trata_alarma;
                sa.sa_flags = SA_RESTART;
                sigfillset(&sa.sa_mask);
                if (sigaction(SIGALRM, &sa, NULL) < 0)
                    error_y_exit("sigaction", 1);

                /* Escribe aqui el codigo del proceso hijo */
                sprintf(buff, "Hola, soy %d\n", getpid());
                write(1, buff, strlen(buff));

                alarm(2);
                sigfillset(&mask);
                sigdelset(&mask, SIGALRM);
                sigdelset(&mask, SIGINT);
                sigsuspend(&mask);

                /* Termina su ejecucion */
                exit(0);
            }
            else if (pid < 0) {
                /* Se ha producido un error */
                error_y_exit("Error en el fork", 1);
            }
            else pid_vec[hijos] = pid;
        }
        /* Esperamos que acaben los hijos */
        sa.sa_handler = &trata_hijo;
        sa.sa_flags = SA_RESTART;
        sigaction(SIGCHLD, &sa, NULL);
        for (int i = 0; i < 10; ++i) kill(pid_vec[i], SIGUSR1);
        while (hijos > 0);
        sprintf(buff, "Valor del contador %d\n", contador);
        write(1, buff, strlen(buff));
        return 0;
    }
}

```

```

int contador = 0;

void trata(int s)
{
    if (s == SIGALRM) contador += 1;
    if (s == SIGUSR1) contador = 0;
    else if (s == SIGUSR2) {
        char buf[80];
        sprintf(buf, "Valor contador: %d\n", contador);
        write(1, buf, strlen(buf));
    }
}

int main(int argc, char* argv[])
{
    sigset(SIG_BLOCK, &mask);

    sigemptyset(&mask);
    sigadd(SIGALRM);
    sigadd(SIGUSR1);
    sigadd(SIGUSR2);
    sigprocmask(SIG_BLOCK, &mask, NULL);

    struct sigaction sa;
    sa.sa_handler = &trata;
    sa.sa_flags = SA_RESTART | SA_RESETHAND;
    sigfillset(&sa.sa_mask);

    sigaction(SIGALRM, &sa, NULL);
    sigaction(SIGUSR1, &sa, NULL);
    sigaction(SIGUSR2, &sa, NULL);
    while (1) {
        alarm(1);
        sigfillset(&mask);
        sigdelset(&mask, SIGALRM);
        sigdelset(&mask, SIGUSR1);
        sigdelset(&mask, SIGUSR2);
        sigdelset(&mask, SIGINT);
        sigsuspend(&mask);
    }
    return 0;
}

```

```

void error_y_exit(char *msg,int exit_status)
{
    perror(msg);
    exit(exit_status);
}

void trat_sigusr1(int s) {
    char buf[80];

    sprintf(buf, "Hijo: SIGUSR1 recibido \n");
    write(1, buf,strlen(buf));
}

void trat_sigalrm(int s) {
    char buf[80];

    sprintf(buf, "Padre: voy a mandar SIGUSR1 \n");
    write(1, buf,strlen(buf));
}

void main(int argc, char *argv[]) {
    int i,pid_h;
    char buf [80];
    int delay;
    struct sigaction sa, sa2;
    sigset(SIG_BLOCK, &mask);

    if (argc !=2) {
        sprintf(buf, "Usage: %s delayParent \n delayParent: 0|1\n", argv[0]);
        write(2,buf,strlen(buf));
        exit(1);
    }

    delay = atoi(argv[1]);
    //signal (SIGUSR1, trat_sigusr1);
    sa.sa_handler = &trat_sigusr1;
    sa.sa_flags = SA_RESTART;
    sigfillset(&sa.sa_mask);
    if (sigaction(SIGUSR1, &sa, NULL) < 0) error_y_exit("sigaction", 1);

    //signal (SIGALRM, trat_sigalrm);
    sa2.sa_handler = &trat_sigalrm;
    sa2.sa_flags = SA_RESTART;
    sigfillset(&sa2.sa_mask);
    if (sigaction(SIGALRM, &sa2, NULL) < 0) error_y_exit("sigaction", 1);

    pid_h = fork ();

    if (pid_h == 0) {
        sigset(SIG_BLOCK, &delay_mask);
        sigemptyset(&delay_mask);
        sigaddset(&delay_mask, SIGUSR1);
        sigprocmask(SIG_BLOCK, &delay_mask, NULL);

        sprintf(buf, "Hijo entrando al pause\n");
        write(1,buf,strlen(buf));
        //pause();

        sigfillset(&mask);
        sigdelset(&mask, SIGUSR1);
        sigdelset(&mask, SIGINT);
        sigsuspend(&mask);

        sigprocmask(SIG_UNBLOCK, &delay_mask, NULL);
        sprintf(buf, "Hijo sale del pause\n");
        write(1,buf,strlen(buf));
    } else {
        if (delay) {
            alarm(5);
            //pause();
            sigfillset(&mask);
            sigdelset(&mask, SIGALRM);
            sigdelset(&mask, SIGINT);
            sigsuspend(&mask);
        }

        sprintf(buf, "Padre manda signal SIGUSR1\n");
        write(1,buf,strlen(buf));
        if (kill (pid_h, SIGUSR1) < 0) error_y_exit("kill", 1);
        waitpid(-1, NULL, 0);
        sprintf(buf, "Padre sale del waitpid\n");
        write(1,buf,strlen(buf));
    }
}

```

## SIGNALS

sigset(SIGSET) → MÀSCARA de signals

sigfullset(SIGSET) → Afegix totes los signals a set

sigemptyset(SIGSET) → Tieu totes los signals de set

sigaddset(SIGSET, SIGUSR1) → Afegix SIGUSR1 a set

sigdelset(SIGSET, SIGUSR1) → Tieu SIGUSR1 de set

sigprocmask(SIG\_BLOCK, &SIGSET, NULL);

SIG\_BLOCK → Bloqueja les senyals de set

SIG\_UNBLOCK → Desbloqueja "

SIG\_SETMASK → Bloqueja només les senyals de set

sigaction(SIGSET, &SA, NULL);

Struct sigaction SA;

• SA\_HANDLER → fins que s'executa el febre el signal

• SA\_MASK → màscara de senyals bloquejades

• SA\_FLAGS → opcións extra (SA\_RESTART)

Sigsuspend(SIGSET);

↳ SUSPEN el procés fins que arribi un signal que NO estigui a set

execLP

Perdem:

- Signal handlers (sigaction)

Mantenim,

- Signals bloquejades (sigprocmask)

- Alarms pendents (alarm)

sa\_handler

Sigsuspen

↳ SUSPEN  
que

execLP

Per

Mar

A continuació mostrem el contingut parcial del fitxer /proc/21999/status en un moment donat:

Name: prog3  
State: Z (zombie)  
Tgid: 21999  
Ngid: 0  
Pid: 21999  
PPid: 21998  
TracerPid: 0  
Uid: 10014 10014 10014 10014  
Gid: 10000 10000 10000 10000  
FDSize: 0  
(...)

Què podem afirmar del procés 21999 respecte al moment en què hem consultat aquest fitxer?

Trieu-ne una:

- a. Cap ✓
- b. Crea un procés que executa el programa ls
- c. Crea dos processos, i cada procés executa una instància del programa ls

Trieu-ne una:

- a. En aquell moment, el procés 21999 no està consumint cpu, però està consumint memòria
- b. El procés 21999 en aquell moment no està consumint ni memòria ni cpu ✓
- c. Amb la informació mostrada, no podem saber si el procés 21999 està consumint memòria o cpu
- d. En aquell moment, el procés 21999 està consumint cpu però no està consumint memòria

A continuació mostrem el contingut parcial del fitxer /proc/21926/status en un moment donat:

Name: prog2  
State: S (sleeping)  
Tgid: 21926  
Ngid: 0  
Pid: 21926  
PPid: 21662  
TracerPid: 0  
Uid: 10014 10014 10014 10014  
Gid: 10000 10000 10000 10000  
FDSize: 256  
(...)

Què podem afirmar del procés 21926 respecte al moment en què hem consultat aquest fitxer?

Trieu-ne una:

- a. En aquell moment, el procés 21926 no estava consumint cpu ✓
- b. No és possible saber si en aquell moment el procés 21926 estava consumint cpu
- c. En aquell moment, podem afirmar que el procés 21926 no estava consumint memòria
- d. Amb la informació mostrada, no podem saber si el procés 21926 estava consumint cpu

Donat el codi següent, i suposant que cap crida a sistema dóna error, quants missatges veurem a la pantalla?

```
printf("Anem!");  
  
for(i=0;i<3;i++)  
    fork();  
  
printf("Bon dia!");
```

Considerant el següent codi:

```
execvp("cat", "cat", "test.txt", (char *)NULL);  
exit(1);
```

En quins casos s'executarà la crida exit(1)?

- a. 4 ✗
- b. 16
- c. 9
- d. 2

La teva resposta és incorrecta.

La resposta correcta és: 9

Donat el següent codi:

```
fork();  
  
execvp("ls","ls",NULL);  
execvp("ls","ls",NULL);
```

Suposant que cap crida a sistema retorna error, quants processos crea aquest codi? Què executarà cada procés?

Trieu-ne una:

- a. Es crea un procés, el fill executa un ls i el pare acaba l'execució sense fer res més
- b. Es crea un procés, tant el pare com el fill executaran 2 ls cadascun
- c. Es crea un procés, el pare executarà un ls y el fill un altre ls ✓
- d. Es creen 3 processos, tots els processos (el pare i cada fill) executaran un ls cadascun

Quin dels següents bucles que creen processos fills seria correcte si volem **crear un fill per cada argument de forma seqüencial, esperant a que acabi un fill per crear el següent** (com a propietari1).

- a. 

```
for (int i = 0; i < argc - 1; ++i) {  
    int pid = fork();  
    if (pid == 0) {  
        // Codi del fill  
        exit(0);  
    } else {  
        waitpid(pid, NULL, 0);  
    }  
}
```
- b. 

```
for (int i = 0; i < argc - 1; ++i) {  
    int pid = fork();  
    waitpid(pid, NULL, 0);  
}
```
- c. 

```
for (int i = 0; i < argc - 1; ++i) {  
    int pid = fork();  
    if (pid == 0) {  
        // Codi del fill  
        exit(0);  
    }  
}  
  
while (waitpid(-1, NULL, 0) > 0);
```

Quin dels següents bucles que creen processos fills seria correcte si volem **crear un fill per cada argument de forma concurrent, esperant a tots els fills al final** (com propietari2).

- a. 

```
for (int i = 0; i < argc - 1; ++i) {  
    int pid = fork();  
    waitpid(pid, NULL, 0);  
}
```

la siguiente llamada con parámetros correctos:  
`ret = sigprocmask(SIG_BLOCK,&mask, NULL);`
- b. 

```
for (int i = 0; i < argc - 1; ++i) {  
    int pid = fork();  
    if (pid == 0) {  
        // Codi del fill  
        exit(0);  
    }  
}  
  
while (waitpid(-1, NULL, 0) > 0);
```

Trieu-ne una:

  - a. Actualiza el parámetro `&mask` poniendo todos sus bits a 1
  - b. Bloqueará en la máscara del proceso los bits a 1 de `&mask` y desbloqueará en la máscara del proceso los bits a 0 de `&mask`
  - c. Sustituye la máscara del proceso por la indicada en `&mask` ✗
  - d. El conjunto de signals bloqueadas en el proceso será la unión de la máscara actual del proceso y el contenido de `&mask`
- c. 

```
for (int i = 0; i < argc - 1; ++i) {  
    int pid = fork();  
    if (pid == 0) {  
        // Codi del fill  
        exit(0);  
    } else {  
        waitpid(pid, NULL, 0);  
    }  
}
```

La teva resposta és incorrecta.

La respuesta correcta es: El conjunto de signals bloqueadas en el proceso será la unión de la máscara actual del proceso y el contenido de `&mask`

Dado el siguiente código y suponiendo que no hay ningún signal capturado. ¿Cuando se escribirá el mensaje "Adiós"?

```
void main(){  
sigset(SIG_BLOCK, mask);  
  
sigemptyset(&mask);  
alarm(5);  
sigsuspend(&mask);  
write(1,"Adiós\n",5);  
}
```

Trieu-ne una:

- a. Cuando se reciba cualquier signal capturado
- b. Ninguna de las otras respuestas
- c. Al cabo de 5 segundos ✗
- d. Nunca, pues la máscara del sigsuspend bloquea todos los signals
- e. Cuando llegue un SIGKILL, ya que no se puede bloquear ni ignorar.

La resposta correcta és: Ninguna de las otras respuestas

Dado el siguiente código (asumiendo que no hay errores en las llamadas a sistema)

```
int pids[10], i;  
for (i=0;i<3;i++){  
    fork();  
    pids[i] = getpid();  
    if (pids[i] == 0){  
        execvp("ls","ls", NULL);  
    }  
}  
while(waitpid(-1, NULL, 0) > 0){  
    sprintf(buffer, "Termina un proceso hijo\n");  
    write(1,buffer,strlen(buffer));  
}
```

¿Cuántas veces veremos el mensaje "Termina un proceso hijo"?

- a. 7 veces
- b. 3 veces ✗
- c. 8 veces

La resposta és incorrecta.

La respuesta correcta es:  
7 veces

Tenemos un programa llamado prog con el siguiente código:

```
main() {  
    execp("./prog", "prog", (char *)0);  
}
```

Suponiendo que lo ponemos en ejecución desde el directorio donde se encuentra y que tenemos permiso de ejecución sobre el ejecutable. Indica cuál de las siguientes afirmaciones es cierta.

Trieu-ne una:

- a. Fallará cuando el sistema se quede sin memoria para crear PCB's nuevos necesarios para las mutaciones ✗
- b. El proceso no acabará nunca, irá cambiando de estado y liberando la CPU cuando lo decida el planificador, y ejecutará para siempre mutaciones al mismo programa
- c. El proceso no acabará nunca, se quedará para siempre en estado RUN, ejecutando mutaciones al mismo programa
- d. Fallará al ejecutar execp porque un proceso no puede mutar al mismo programa que está ejecutando

La teva resposta és incorrecta.

La resposta correcta és: El proceso no acabará nunca, irá cambiando de estado y liberando la CPU cuando lo decida el planificador, y ejecutará para siempre mutaciones al mismo programa

¿Cuál de las siguientes características de signals se pierden al ejecutar un execp?

Trieu-ne una:

- a. Los signals pendientes y las alarmas ✗
- b. Ninguna característica de signals se pierden: al mutar se conservan tal y como estaban todas las características de signals
- c. La máscara de signals bloqueados
- d. El tratamiento asociado a cada signal

La teva resposta és incorrecta.

La respuesta correcta és: El tratamiento asociado a cada signal

Dado un sistema linux ¿cuál de las siguientes características **SÍ** son heredadas por los hijos?

Trieu-ne una:

- a. Los temporizadores activos
- b. El identificador de proceso padre
- c. El valor del program counter ✓
- d. Los signals pendientes de tratar

Un proceso en estado BLOCKED:

Trieu-ne una:

- a. Nunca consume CPU ✓
- b. Nunca consume CPU ni memoria
- c. Puede ser que consuma CPU pero nunca consume memoria
- d. Nunca consume memoria

Tenemos un código que utiliza la llamada "sigsuspend(&mask);", donde la variable "mask" tiene seleccionados la mitad de los signals. Obtendremos el mismo comportamiento del código, independientemente del signal que el proceso reciba, sustituyendo únicamente esta línea de código por "kill(getpid(), SIGSTOP);".

Trieu-ne una:

- a. Verdader
- b. Fals ✓

Cuando un proceso ejecuta la llamada a sistema exit....

Trieu-ne una:

- a. Finaliza su ejecución y nunca más recibirá la CPU
- b. Finaliza su ejecución pero pasa a estado READY hasta que su padre ejecuta un waitpid, entonces copia el parámetro del exit y finaliza
- c. Para su ejecución pero si recibe un SIGCONT antes de que su padre haga un waitpid continua
- d. Pasa a estado ZOMBIE y libera todos sus recursos, incluido el PCB. ✗

La teva resposta és incorrecta.

La respuesta correcta és: Finaliza su ejecución y nunca más recibirá la CPU

En fer fork, la reprogramació de les funcions d'atenció d'un signal:

```
int ret;  
ret = fork();  
if (ret == getpid()){  
    /* CODIGO */  
}
```

- a. Solo para el proceso hijo ✗
- b. nunca

Trieu-ne una:

- a. es perden totes excepte les de SIGKILL i SIGTERM ✗
- b. es perden
- c. només s'hereten les de SIGKILL i SIGTERM
- d. s'hereten

La resposta correcta és: s'hereten

¿Es posible que un proceso después de llamar a exit se quede en estado zombie para siempre?

Trieu-ne una:

- a. Sí, si el padre se queda vivo para siempre y no libera los PCBs de sus hijos muertos
- b. No, el sistema siempre libera los PCBs de los procesos zombies si su padre no lo hace ✗
- c. No, eso sólo sería posible si el proceso hubiera muerto por signal
- d. No, el estado zombie es algo temporal mientras no se recoge la causa de su muerte

La teva resposta és incorrecta.

La resposta correcta és: Sí, si el padre se queda vivo para siempre y no libera los PCBs de sus hijos muertos

Assumint que abans del bucle hem reprogramat el tractament del signal SIGINT perquè executi la funció func, quants missatges apareixeran a la pantalla?

```
void func(int s){  
    printf("Signal %d\n", s);  
}  
  
int main(int argc, char **argv){  
    ...  
    for(i=0;i<3;i++){  
        if (fork()==0){  
            kill(getppid(), SIGINT);  
            exit(0);  
        }  
    }  
}
```

Trieu-ne una o més:

- a. No ho podem saber amb certesa, perquè es poden solapar diversos SIGINT abans del tractament
- b. Podem assegurar que com a mínim es mostrarà un cop el missatge ✗
- c. Sempre 3 missatges perquè el procés rep 3 vegades el SIGINT ✗
- d. Pot passar que el procés pare no rebi cap vegada el signal

Les respostes correctes són: No ho podem saber amb certesa, perquè es poden solapar diversos SIGINT abans del tractament, Pot passar que el procés pare no rebi cap vegada el signal

Suposadament un procés no té capturat el SIGUSR1 i executa la següent llamada a sistema:  
kill(getpid(),SIGUSR1);

Indica cuál de las siguientes afirmaciones **ES FALSA**:

Trieu-ne una:

- a. Puede ser que el proceso muera como consecuencia de esta llamada a sistema ✗
- b. kill fallará porque el pid especificado no es válido
- c. Puede ser que el proceso continue la ejecución sin problemas

La teva resposta és incorrecta.

La resposta correcta és: kill fallará porque el pid especificado no es válido

Al codi següent:

```
int main()  
{  
    sigset(SIGINT, func);  
    alarm(5);  
    if (fork()==0) alarm(3);  
    sigfill(SIGINT);  
    sigsuspend(SIGINT);  
    exit(0);  
}
```

Per quin motiu, dels que poden succeir en executar aquest codi, es desbloquejarà el pare del sigsuspend?

Trieu-ne una:

- a. Sempre per SIGALRM
- b. Sempre per SIGCHLD
- c. Per algun signal diferent a SIGCHLD o SIGALRM
- d. Per cap, es queda bloquejat
- e. Sempre per SIGCHLD o per SIGALRM però no es pot saber per quin dels dos ✗

La resposta correcta és: Per cap, es queda bloquejat