

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.

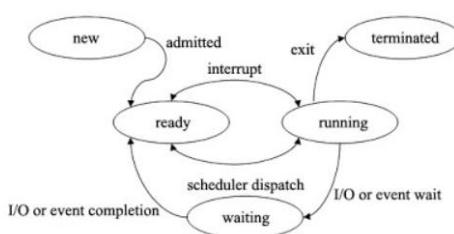
Si el **quantum** és molt gran, els processos acabaran els seus temps de CPU abans d'acabar el quantum. Si és molt petit, hi haurà molts canvis de processos i la CPU perdrà rendiment.

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)
// [3] --> Arguments que li passariem a la funció (pot no haver-n'hi)
// [4] --> 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);
    }
}
```

Accessos a Disc

Inode

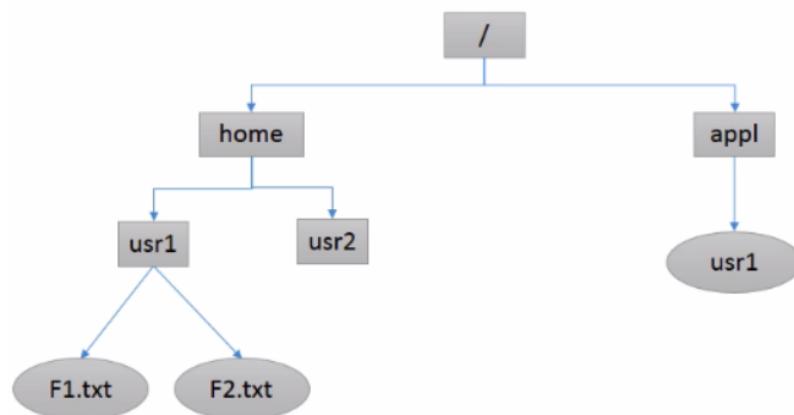
Un inode conté tota l'informació bàsica d'un fitxer (tamany, propietari, permisos, etc.). Per simplificar-ho, a SO només utilitzem els camps següents:

- Nombre de Referències
- Tipus (d, l, -) [D = directori, l = softlink, - = "normal"]
- Llista de blocs de dades de l'inode

INODE
Nombre de Referències (mínim 1)
d, l, -
Nº de Bloc (Si ocupa més d'un, es van posant consecutivament; simulem que ocupa 2 o més).
Nº de bloc
etc....

EXEMPLE D'ACCES A DISC

Primer, veurem una mica de detalls del **sistema de fitxers**, farem tots els passos, donat un sistema de fitxers: Generar els inodes i blocs de dades equivalents i calcularem els accessos a disc per a dos codis diferents. Suposarem el següent sistema d'arxius basat en inodes:



Podem deduir d'aquest sistema d'arxius (i perquè ens ho diuen en l'enunciat normalment):

- Els quadrats són directoris.
- Els rodons són altres tipus d'arxius.
 - /appl/usr1 és un softlink a /home/usr1
 - /home/usr1/F1.txt i /home/usr1/F2.txt són hardlinks.
- Un inode ocupa 1 bloc.
- F1.txt ocupa 5 blocs de dades.
- 1 bloc de dades són 256 bytes.
- Els fitxers "." i ".." no estan dibuixats per a simplificar.

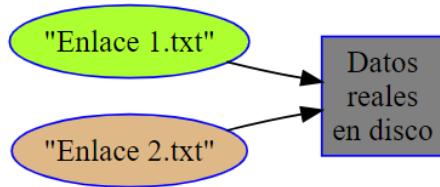
Hard links i softlinks

Hardlinks i softlinks són dos mètodes d'enllaçar noms de fitxers amb inode, dos tipus de vincles.

- **Hardlink:** Enllaça el nom amb un inode de tipus "normal" (fitxer de dades), en les dades d'aquest inode hi ha les dades a les quals volem accedir. Apunten a un arxiu en el sistema d'arxius; associen dos o més fitxer compartint el mateix inode, això fa que cada hardlink sigui una còpia exacta de la resta de fitxers associats, tant de dades com de permisos, propietari... Encara que els anomenem de diferents formes, tant els hardlinks com els arxius originals ofereixen la mateixa funcionalitat. Al modificar les dades apuntades per a qualsevol d'ells, es canvién les dades reals emmagatzemades al disc, quedant modificats els dos per igual.
 - Quan tenim N noms de fitxers que són hardlinks (entenem que al mateix inode), significava que tenen el mateix nombre de inode; són directament el mateix fitxer. F1.txt i F2.txt han de tindre el mateix nombre de inode.

Exemple de Hardlink

En l'imatge de sota (extreta de Wikipedia) hi ha dos HardLinks: "Enlace1.txt" i "Enlace2.txt". Els dos apunten a les mateixes dades físiques. Si el nom de fitxer "Enlace1.txt" s'obra en un editor de text, es modifica i es guarda, aquests canvis seran també visibles en el fitxer "Enlace2.txt" i viceversa. Això es degut a que apunten a les mateixes dades.



Tornant al esquema del sistema d'arxius que hem vist, anem a definir com queden la taula d'inodes (en el disc) i els blocs de dades:

- Primer hem d'assignar nombres d'inode als fitxers seqüencialment de 0 ... a N.
- F1.txt i F2.txt recordem que son hardlinks a un mateix arxiu, i per tant han de tenir el mateix inode.
- Els enumerem seguint l'ordre de creació (ens ho inventem), però podríem seguir altres mètodes d'enumeració, tot i que es recomana utilitzar una enumeració llògica i clara.

INFORMACIÓ EN EL DISC

- Metadades (taula de inodes).
 - Dades com tipus de fitxer, nombre d'enllaços (incloent . i ..) i llista de blocs de dades.
- Dades (blocs de dades).
 - Els blocs que són directoris, softlinks i blocs de fitxers de dades.

BLOCS DE DADES

Directori:	Softlink:	Dades "normals"
És una taula (nom, inode). Exemple: /home/usr1	Posem el path (camí) del fitxer al qual apunta: Exemple: /appl/usr1	Si tenim l'informació la podem posar, sinó indicada: Exemple: F1.txt conté a's.

Com quedaria? [INODES]

Nº Inode	0	1	2	3	4	5	6
Nº Refs	*(1)						
Tipus (d, l, -) *(2)	d	d	d	d	d	l	-
Bloc 1	0	1	2	3	4	5	6 *(3)
Bloc 2							7
Bloc 3							8
Bloc 4							9
Bloc 5							10

*(1) El nombre de referències l'emplenem al final.

*(2) Del bloc 0 al 4, són directoris. El 5 és un softlink. I el 6 és un fitxer de dades.

*(3) Els directoris ocupen 1 bloc, el soft-link també, el fitxer F1.txt i F2.txt comparteixen inode i ocupen 5 blocs.

Blocs de dades (En necessitem 11: 0...10)

[Imatge bloc de dades]

- Els directoris tenen dues columnes: Nom del fitxer i el nombre d'inode.
- Han de tindre sempre els fitxers "." i ".."
- El soft-link ha de contenir el nom del fitxer al que apunta, ja que és un inode diferent.
- Els noms F1.txt i F2.txt són simplement el mateix inode (ja que són hard-links)
- Se suposa que hi ha 256 bytes en cada bloc de dades

Resultat final (amb les referències).

Nº Inode	0	1	2	3	4	5	6
Nº Refs	4	4	2	2	2	1	2
Tipus (d, l, -)	d	d	d	d	d	l	-
Bloc 1	0	1	2	3	4	5	6
Bloc 2							7
Bloc 3							8
Bloc 4							9
Bloc 5							10

EXERCICI D'EXEMPLE

```
fd = open("/Home/Usr1/F1.txt", O_RDONLY);
ret = read(fd, &c, sizeof(char));
while(ret > 0){
    write(1, &c, sizeof(char));
    ret = read(fd, &c, sizeof(char));
}
close(fd);
```



- Com es tradueix?
 - Hem d'accedir a l'inode de F1.txt i copiarlo a la taula d'inodes a memòria: accés al disc.
 - Llegir tot el contingut de F1.txt: Accessos a disc (byte a byte).
 - Hem d'escriure el que hem llegit per la sortida std.

1. fd = open("/Home/Usr1/F1.txt", O_RDONLY);

- Amb buffer caché: Tot el que llegim es guarda a la buffer cache (els inodes ocupen 1 bloc de disc):
 - Llegim l'inode arrel (0).
 - Llegim les dades de l'inode arrel (bloc de dades 0); veiem que "home" és l'inode 1.
 - Llegim l'inode 1 (Home)
 - Llegim les dades de l'inode 1 (bloc de dades 1); veiem que Usr1 es l'inode 3.
 - Llegim l'inode 3.
 - Llegim les dades de l'inode 3 (bloc de dades 3); Veiem que F1.txt es l'inode 6.
 - Llegim l'inode 6; inode destí; ho copiem en la taula d'inodes a memòria.

2 i 5. ret = read(fd, &c, sizeof(char));

- S'executen 256*5 crides a sistema (hi ha 5 blocs de dades) per a llegir les dades.
 - Si hi ha buffer cache farem 5 accessos al disc.
 - D'altra manera farem 256*5 accessos a disc.
- Farem un últim read per a detectar que hem arribat al final (el read que retorna 0).

3.write(1, &c, sizeof(char));

- Si el canal 1 és la consola : no genera accessos a disc.

7. close(fd);

- Com no hem modificat l'inode no fa falta escriure'l a disc.

```

#include <stdlib.h>           // Miscellaneous (https://pubs.opengroup.org/onlinepubs/009604599/basedefs/stdlib.h.html)
#include <stdio.h>             // I/O and miscellaneous (https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/stdio.h.html)
#include <string.h>             // strcpy, strlen (https://pubs.opengroup.org/onlinepubs/7908799/xsh/string.h.html)
#include <unistd.h>             // fork, pipe, dup2, dup, close, open, execlp, read, write (and many others) (https://pubs.opengroup.org/onlinepubs/009696699/basedefs/sys/wait.h.html)
#include <sys/wait.h>            // waitpid, wait... (https://pubs.opengroup.org/onlinepubs/009696699/basedefs/sys/wait.h.html)
#include <sys/types.h>            // Data types (https://pubs.opengroup.org/onlinepubs/009695399/basedefs/sys/types.h.html#tag_13_67)
#include <sys/stat.h>             // S_IRUSR, S_IWUSR, S_IXUSR (https://pubs.opengroup.org/onlinepubs/009695399/basedefs/sys/stat.h.html)
#include <sys/un.h>               // Main socket functions (https://pubs.opengroup.org/onlinepubs/009696699/basedefs/sys/un.h.html)
#include <fcntl.h>                // O_TRUNC, O_CREATE, O_RDONLYM O_RDWR, O_WRONLY... (https://pubs.opengroup.org/onlinepubs/009695399/basedefs/sys/fcntl.h.html)
#include <time.h>                  // Clock instructions (https://pubs.opengroup.org/onlinepubs/009695399/basedefs/sys/time.h.html)
#include <errno.h>                 // errno, EEXIST (https://pubs.opengroup.org/onlinepubs/009696899/basedefs/errno.h.html)

///////////////
// Functions use //
///////////////

int fd[2];
int fd, fd_substituit, ret;
char buffer[128];

pipe(fd);                      // Crea una pipe sense nom i indica la sortida d'aquesta a df[0] i l'entrada a fd[1]

dub2(fd, fd_substituit);        // Sobreescriu fd a fd_substituit (mirar "Conectar pipe amb terminal o altres pipes" més avall)

dub(fd);                      // Ocupa el primer canal lliure amb una copia de fd

close(fd);                     // Tanca el canal fd, això es important sobretot per tancar els escriptors i així
                                // evitar que els lectors es quedin en un bucle infinit

ret = read(fd, buffer, sizeof(buffer)); // fd -> canal d'on volem llegir, buffer -> on volem guardar lo llegit,
                                         // sizeof(buffer) -> nº bytes a llegir, ret -> bits llegits
                                         // Si s'intenta llegir una pipe buida es quedrà bloquejat fins que hi hagin dades
                                         // En una pipe read només retornarà 0 quan no hi hagin escriptors d'aquesta (per això és important tancar-los)

ret = write(fd, buffer, sizeof(buffer)); // fd -> canal on volem escriure, buffer -> el que volem escriure,
                                         // sizeof(buffer) -> nº bytes a escriuer, ret -> bits escrits
                                         // Si s'intenta escriure en una pipe plena es quedrà bloquejat fins que es buida suficientment

fd = open("nom", acces_mode, permision_flags); // El "nom" pot ser un fitxer o una pipe (amb nom clar), fd -> canal al que s'assgina el fitxer
                                                // acces_mode: O_RDONLY, O_WRONLY, O_RDWR (un d'aquest és sempre necessari)
                                                // O_CREAT -> crea; O_TRUNC -> Sobreescriu; O_APPEND -> afageix al final
                                                // Podem conquerenar els modes d'accés amb una '|'
                                                // permision_flags: si renim que <rw-rw-rwx> (en grups de tres fan referencia a usuari, grup i others)
                                                // podem ficar un 1 als que volem activar i convertir-ho en octal
                                                // per exemple si volem read & write per l'usuari, fariem: 110000000 -> 600

int nova_posicio = lseek(fd, desplaçament, relatiu_a); // Només per fitxers, fd és el canal del fitxer
                                                       // Posició accedita (nova_posicio) = relatiu_a + desplaçament (pot ser negatiu)
                                                       // relatiu_a: SEEK_SET (com un .begin()), SEEK_CUR (posició actual), SEEK_END (com un .end())

mknod("nompipe", S_IFIFO | 600, 0); // "nompipe" (autodescriptiu); S_IFIFO (volem una pipe basicament, si vols +info ves al segon
                                         // link de <sys/stat.h> per veure les altres possibilitats)
                                         // 600 -> fa referència als permisos, obtinguts de la mateixa forma que "permision_flags"
                                         // El 0 del final no sé que vol dir pero no el tocaria, si tu ho saps pots canviar aquesta frase (
                                         // 0 -> fa referència a la mateixa forma que "permision_flags"

/////////////
// Creació pipe bàsica //
/////////////

{
    int fd[2];

    pipe (fd); //Retorna -1 si hi ha hagut algun error en crear la pipe

    write (fd[1], "Hola", 4); //Escriure a la pipe

    char buff[128];
    read (fd[0], buff, strlen(buff)); //Llegir de la pipe (buff = "Hola"). Retorna 0 si la pipe no té escriptors
}

```

```

///////////
// Creació pipe amb nom //
///////////

{

if (mknod("npipe", S_IFIFO | 600, 0) < 0 && errno != EEXIST)
{
    perror("Error creant la pipe");
    exit(1);
}

int fd = open("nom", O_RDONLY); //Guardem el canal de lectura de la pipe a fd;
//{...
close (fd);
}

///////////
// Pipe amb Forks //
///////////

{

int pid = fork(); //Retorna -1 si hi ha hagut error;

//Procés del fill
if (pid == 0) {
    close (fd[1]); //Tanquem el canal d'escriptura a la pipe per poder llegir
    char c;
    while (read (fd[0], &c, sizeof(c)) < 0) {
        write (1, &c, sizeof(c)); //Escrivim a la terminal el que anem llegint de la pipe
    }
    close (fd[0]);
    exit (0);
}
close (fd[1]); //Tanquem el canal d'escriptura a la pipe perquè el fill pugui llegir!!
close (fd[0]); //Tanquem el canal de lectura ja que no el necessitem
wait (NULL); //Esperem el fill
exit (0);
}

///////////
// Obrir un fitxer per llegir i/o escriure //
///////////

fd = open ("Nom_fitxer", O_RDONLY); //Canal de lectura al fitxer
fd = open ("Nom_fitxer", O_WRONLY); //Canal d'escriptura al fitxer
fd = open ("Nom_fitxer", O_RDWR); //Canal d'escriptura al fitxer


///////////
// Conectar pipe amb terminal o altres pipes //
///////////

dup2 (fd[0], 0); //El canal de lectura de la pipe passa a ser el canal de lectura de la terminal
close (fd[0]);

dup2 (fd[1], 1); //El canal d'escriptura de la pipe passa a ser el canal d'escriptura de la terminal
close (fd[1]);

dup2 (fd1[0], fd2[1]); //El canal de lectura de la pipe 1 passa a ser el canal d'escriptura de la pipe 2

int newfd = dup(fd); //Copia el canal fd al primer canal disponible i indica quin és aquest (ident. newfd)

```

```

/////////////// Accés directe a posicions de fitxers /////////////////
// Accés directe a posicions de fitxers //
/////////////// Accés directe a posicions de fitxers ///////////////


fd = open ("Nom_fitxer", O_RDWR);
lseek (fd, desplaçament, [*]);           // Mira a sota

//[*]:
    // SEEK_SET: punter = desplaçament
    // SEEK_CUR: punter = punter + desplaçament
    // SEEK_END: punter = file.size() + desplaçament
    // Si tenim:
        lseek (fd, -1, SEEK_END);
        read(fd, buffer, sizeof(char));
    // Llegirem l'últim char del fitxer

// El pare escriu a la pipe, el fill llegeix de la pipe i escriu per stdout
//-----


int main () {
    int pipefd[2] , pid;
    char c;
    char buff[]="Codificacio Cesar de 3.\n Text pla : hola !\nText xifrat : ";
    write(1, buff, strlen(buff));
    if (pipe(pipefd) < 0) {
        perror(" Error en crear la pipe");
        exit (1) ;
    }
    if (write(pipefd[1], "hola!", 5) < 0) {
        perror("Error en escriure a la pipe") ;
        exit(2);
    }

    pid = fork() ;
    if (pid < 0) {
        perror("Error de fork");
        exit(1);
    }

    // Un procés escriu per una pipe i llegeix el contingut d'aquesta
    //-----
    int main()
    {
        int pipefd[2], r;
        char c, buff[8];
        if (pipe(pipefd) < 0) {
            perror("Error en crear la pipe");
            exit(1);
        }
        if ((r = write(pipefd[1], "hola!\n", 6)) < 0) {
            perror("Error en escriure a la pipe");
            exit(2);
        }
        if ((r = read(pipefd[0], buff, r)) < 0) {
            perror("Error en llegir de la pipe");
            exit(3);
        }
        write (1, buff, r);
        exit (0) ;
    }

    //-----


    if (pid == 0) {
        close(pipefd[1]); /* important !!!! */
        while (read(pipefd[0], &c, 1) > 0) {
            c +=3;
            if (write (1, &c, 1) < 0) {
                perror("Error en escriure a la stdout");
                exit(2);
            }
        }
        close(pipefd[0]);
        write (1, "\n", 1) ;
        exit(0);
    }

    close(pipefd[1]); /* important !!!! */
    close(pipefd[0]);
    wait(NULL);
    exit(0);
}

```

Funcions

```
void func(void) {
    // ...
}
```

Parámetros

```
// Si ejecutamos ./program

int main(int argc, char *argv[]) {
    // argc -> 1
    // argv[0] -> "./program"
    return 0;
}

// Si ejecutamos ./program a b c

int main(int argc, char *argv[]) {
    // argc -> 4
    // argv[0] -> "./program"
    // argv[1] -> "a"
    // argv[2] -> "b"
    // argv[3] -> "c"
    return 0;
}
```

Variables

- Simples: `type variable_name;`
- Vectores: `type vector_name[vector_size];`

El tamaño de un vector puede ser un número, una constante definida anteriormente o se puede dejar en blanco así: `int vec[] = {1,2,3};`
Adoptará el tamaño de aquello a lo que se asigne.

Tipos

- `int` (entero)
- `char` (carácter)
- **punteros**
 - `int *`
 - `char *`
 - `void *`
 - ...
- No existen `bool`, usamos ints. `1 -> true, 0 -> false`
- Los `string` son un `char*` terminado con el carácter invisible '`\0`'

Strings

- Vectores de chars acabados en '`\0`'
- '`\n`' -> Salto de línea
- `strlen(s)` -> devuelve el tamaño de s (un string)
- `sprintf(s, "Hello World %d", 3)` -> Guarda el string "Hello world 3" en la variable s. Puede usar diversos formatos
 - `%d` -> int
 - `%c` -> char
 - `%s` -> string
 - ...
- `strcmp(s1, s2)` -> Compara las strings devuelve `0` si son iguales, `>0` si `s2` es mayor, `<0` si `s2` es menor.

Headers

```
#include <stdio.h>
#include <stdlib.h>

#include "lo_que_sea.h"
```

Constants

```
#define PI 3.14
#define CONSTANT value
```

Variables globales

Las podemos usar en cualquier función del programa

```
int a, b;
char c;
int vector[CONSTANT];
char buff[CONSTANT];
```

Asignación

```
int x = 4;
char a = 'A';
int *p = &x;
int v[10];
v[0] = 1;
v[1] = 2;
v[9] = 19;
```

Comparaciones

- Igual (`x == y`)
- No igual (`x != y`)
- Mayor (`x > y`)
- Mayor o igual (`x >= y`)

int -> string

```
int x = 1000;
char buffer[64];
sprintf(buffer, "%d", x);
write(1, buffer, strlen(buff));

// OUT: 1000
```

string -> int

```
char *s = "314";
int x = atoi(s);

// x = 314;
```


¿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)

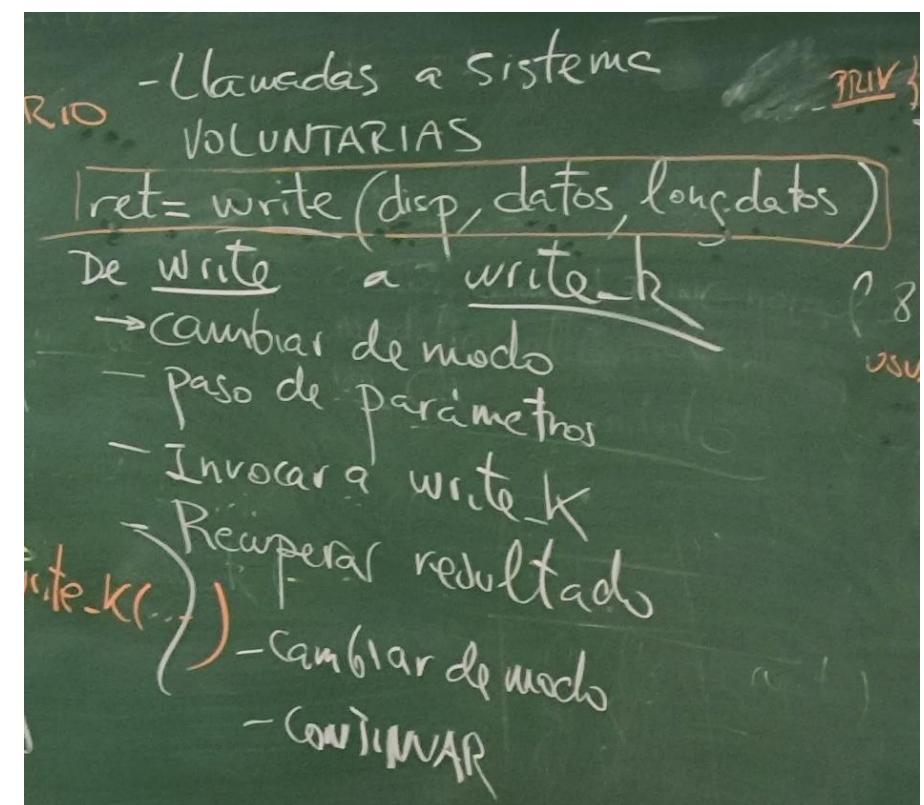
¿Cuando se ejecuta código de kernel?

- 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



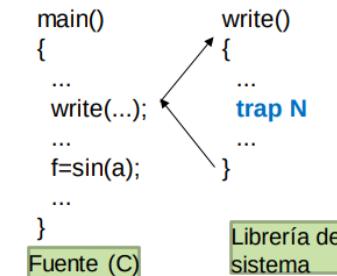
Requerimientos llamadas a sistema

- 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

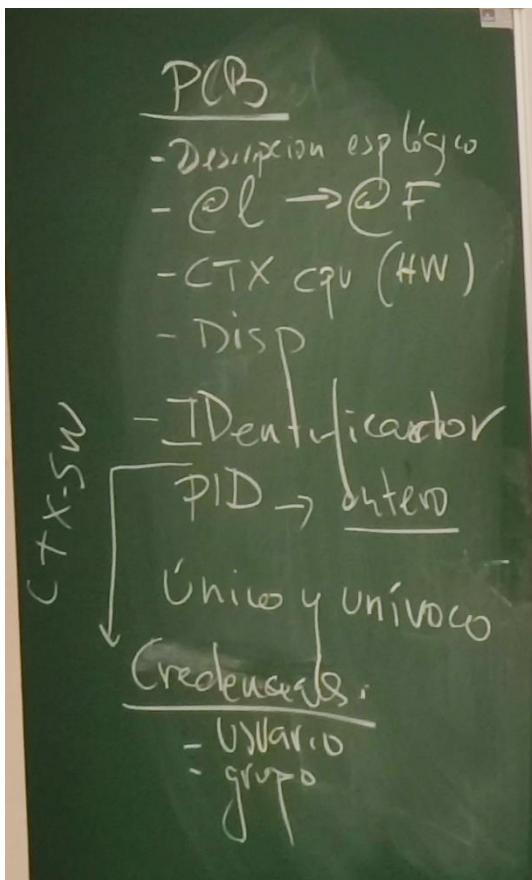
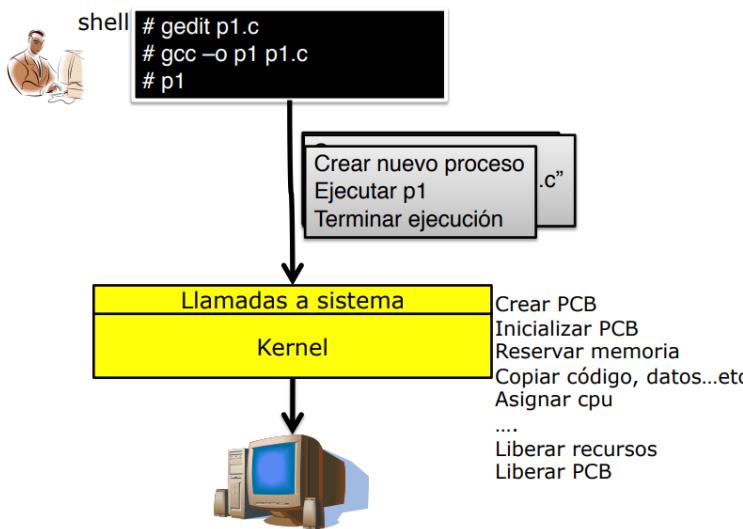
- La librería de sistema se encarga de traducir de la función que ve el usuario a la petición de servicio explícito al sistema
 - Pasa parámetros al kernel
 - Invoca al kernel → TRAP
 - Recoge resultados del kernel
 - Homogeneiza resultados (todas las llamadas a sistema en linux devuelven -1 en caso 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.

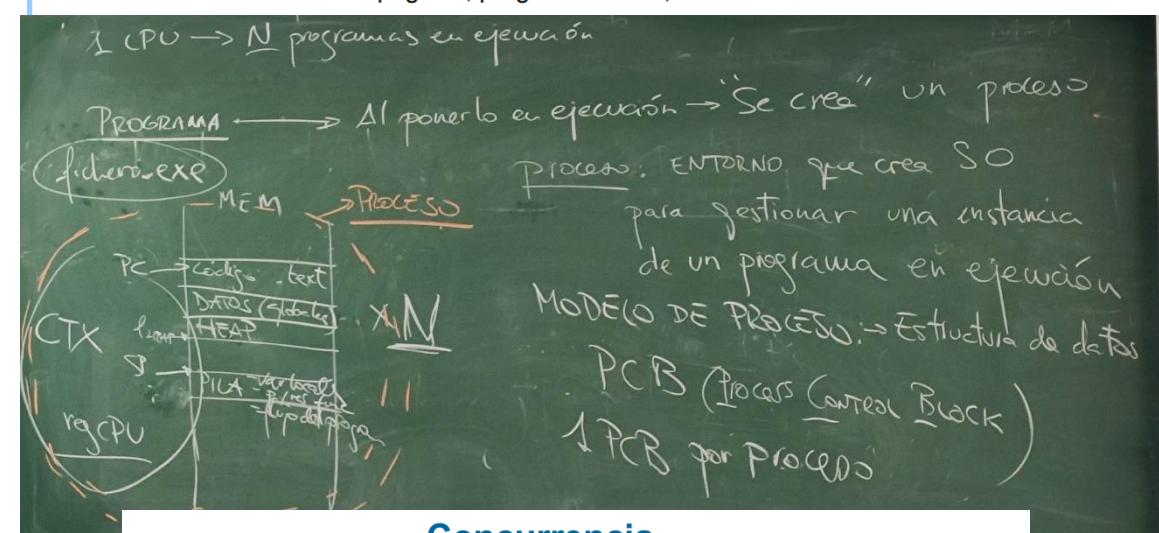
	Función “normal”	Función kernel
Pasamos los parámetros	Push parámetros	DEPENDE
Para invocarla	call	sysenter, int o similar
Al inicio	Salvar los registros que vamos a usar (push)	Salvamos todos los registros (push)
Acceso a parámetros	A través de la pila: Ej: mov 8(ebp).eax	DEPENDE
Antes de volver	Recuperar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

Procesos: ¿Como se hace?



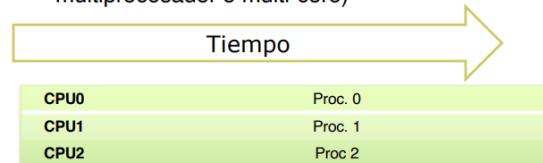
Process Control Block (PCB)

- 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

- 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)

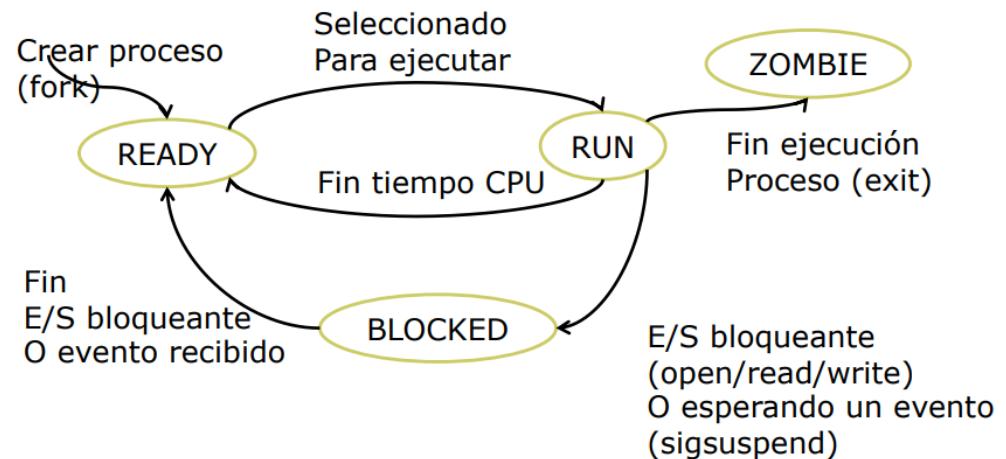
Estados de un proceso

- 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 (bloqueante)	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
```

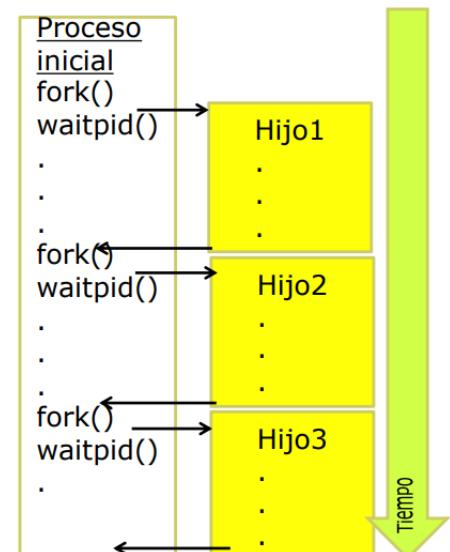
Caso 2: queremos que hagan lo mismo

```
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.

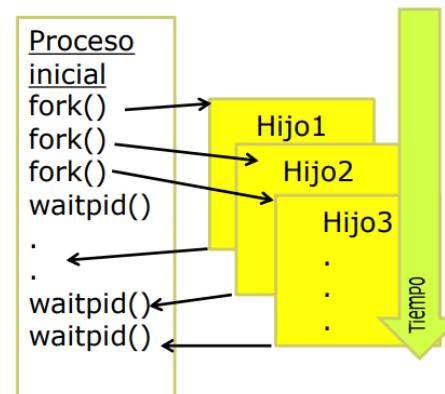
```
1. #define num_procs 2
2. int i,ret;
3. for(i=0;i<num_procs;i++){
4.   if ((ret=fork())<0) control_error();
5.   if (ret==0) {
6.     // estas líneas solo las ejecuta el
7.     // hijo
8.     codigohijo();
9.     exit(0); //
10.  }
11.  waitpid(-1,NULL,0);
12. }
```



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*)

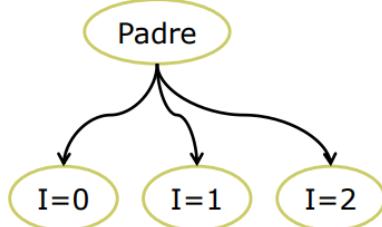
- Como se implementa esto?



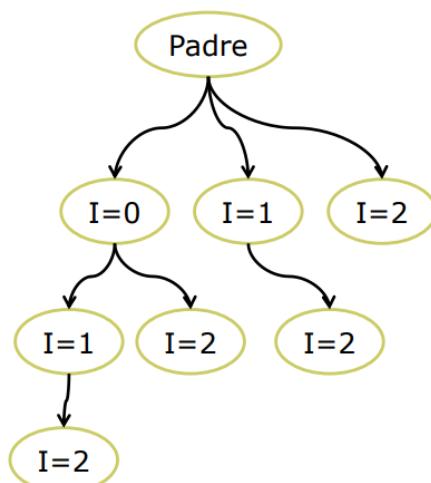
```
...
ret = fork();
if (ret == 0) {
    execvp("/bin/ls", "ls", "-l", (char *)NULL);
}
// A partir de aquí, este código sólo lo ejecutaría el padre
...
```

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

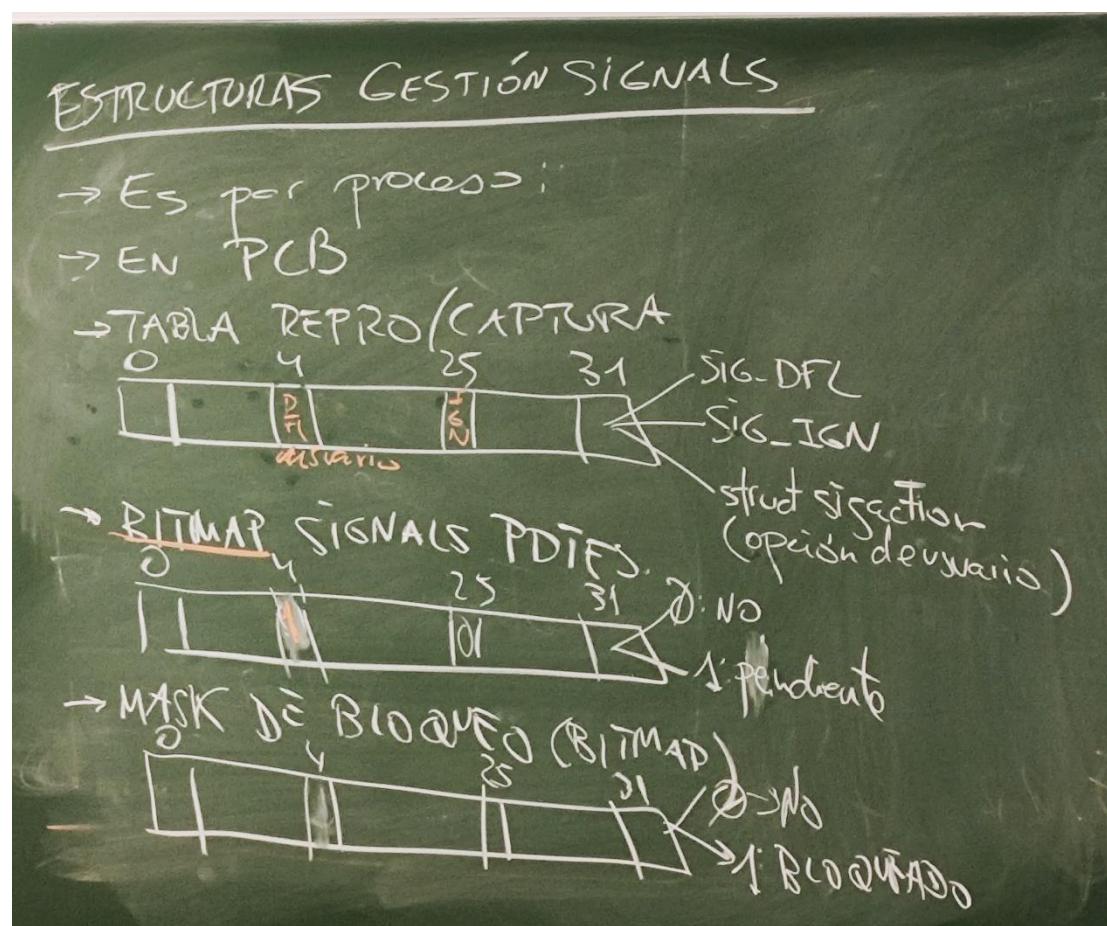
Estructuras de datos del kernel

- 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

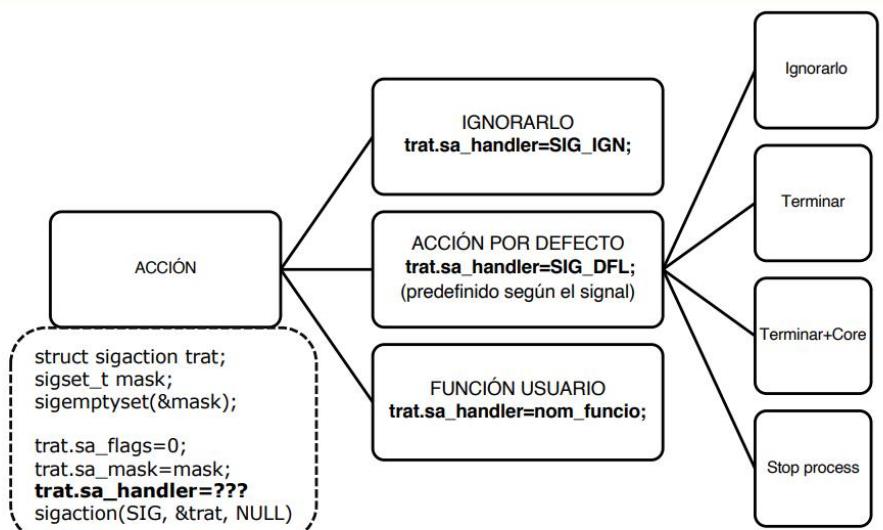
sa_handler → routine que queréis
RAS RASF

sa_flags →

sa_mask → bitmap

↳ Se añade (1) a (el
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

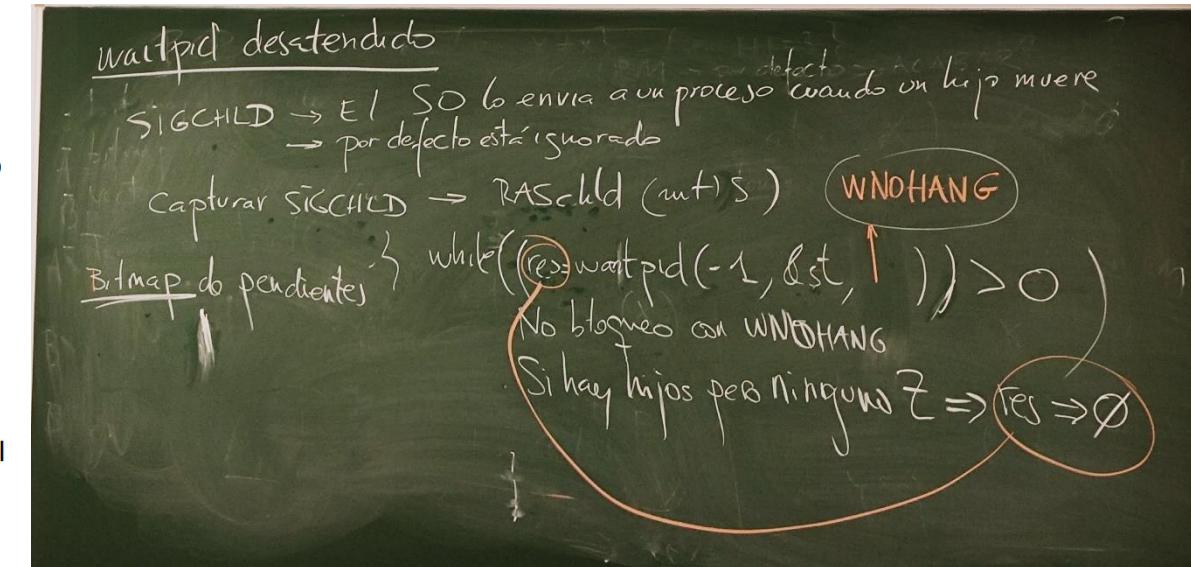
- El hijo hereda la tabla de acciones asociadas a los signals del proceso padre
- La máscara de signals bloqueados se hereda
- Los eventos son enviados a procesos concretos (PID's), el hijo es un proceso nuevo → La lista de eventos pendientes se borra (tampoco se heredan los temporizadores pendientes)

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.

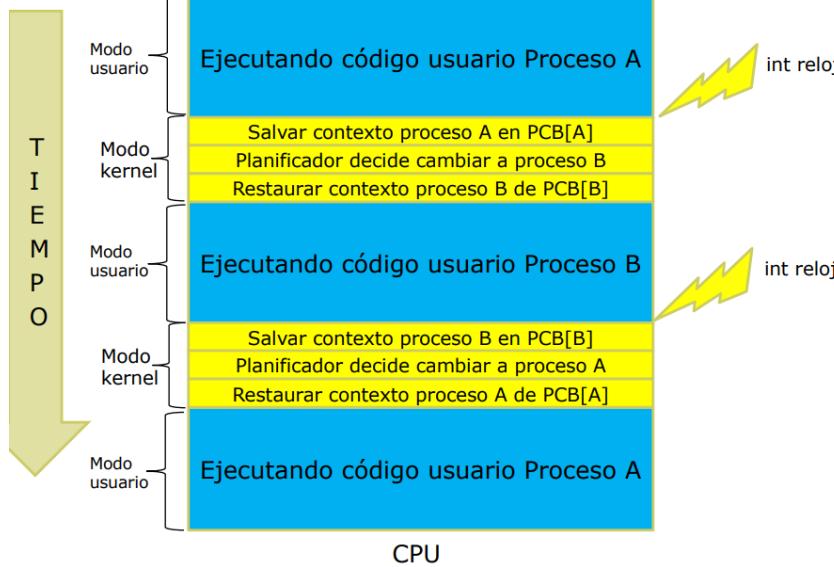
Planificación

- 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.

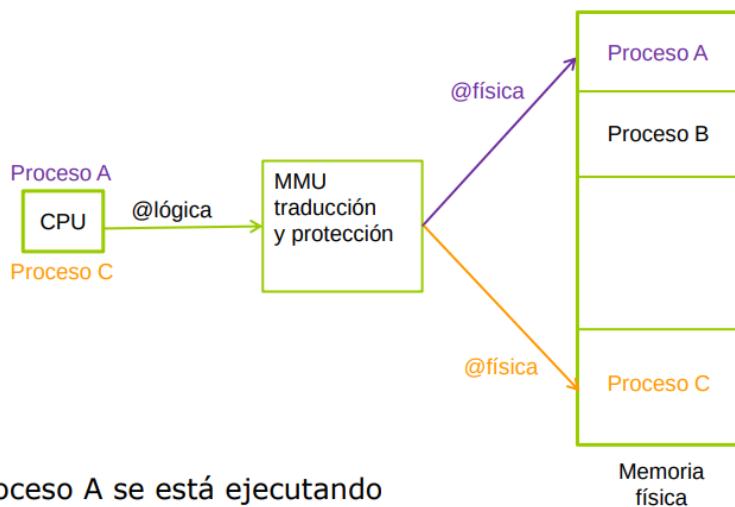
Completely Fair Scheduling

- 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

Espacio de @ de un proceso

- Espacio de direcciones del procesador
 - Conjunto de @ que el procesador puede emitir, (depende del bus de direcciones)
- Espacio de direcciones lógicas de un proceso
 - Conjunto de @ lógicas que un proceso puede referenciar (que el kernel decide que son válidas para ese proceso)
- Espacio de direcciones físicas de un proceso
 - Conjunto de @ físicas asociadas al espacio de direcciones lógicas del proceso (decidido también por el kernel)
- Correspondencia entre @ lógicas y @ físicas
 - Fija: Espacio de @ lógicas == Espacio de @ físicas
 - Traducción:
 - Al cargar el programa en memoria: el kernel decide donde poner el proceso y se traducen las direcciones al copiarlas a memoria
 - **Al ejecutar: se traduce cada dirección que se genera**
 - Colaboración entre HW y SO
 - » HW ofrece el mecanismo de traducción
 - » Memory Management Unit (MMU)
 - » El Kernel lo configura

Sistemas multiprogramados



Soporte HW: MMU

- MMU(Memory Management Unit). Componente HW que ofrece la traducción de direcciones y la protección del acceso a memoria. Como mínimo ofrece **soporte a la traducción y a la protección** pero puede ser necesario para otras tareas de gestión
- **SO es el responsable de configurar la MMU con los valores de la traducción de direcciones correspondientes al proceso en ejecución**
 - Qué @ lógicas son válidas y con qué @ físicas se corresponden
 - Asegura que cada proceso sólo tiene asociadas sus @ físicas
- **Soporte HW a la traducción y a la protección entre procesos**
 - MMU recibe @ lógica y usa sus estructuras de datos para traducirla a la @ física correspondiente
 - Si la @ lógica no está marcada como válida o no tiene una @ física asociada genera una excepción para avisar al SO
- **SO gestiona la excepción en función del caso**
 - Por ejemplo, si la @ lógica no es válida puede eliminar al proceso (SIGSEGV)

Tareas del SO en la gestión de memoria

- Carga de programas en memoria
- Reservar/Liberar memoria dinámicamente (mediante llamadas a sistema)
- Ofrecer compartición de memoria entre procesos
 - Con COW habrá compartición de forma transparente a los procesos en zonas de solo lectura
 - Existe compartición explícita de memoria (mediante llamadas a sistema) pero no lo trabajaremos este curso
- Servicios para la optimización del uso de memoria
 - COW
 - Memoria virtual
 - Prefetch

Servicios básicos: carga de programas

- El ejecutable debe estar en memoria para ser ejecutado, pero los ejecutables están en "disco"
- SO debe
 1. Leer e Interpretar el ejecutable (los ejecutables tienen un formato)
 2. Preparar el esquema del proceso en memoria lógica y **asignar memoria física**
 1. **Inicializar estructuras de datos** del proceso
 1. Descripción del espacio lógico
 1. Qué @ lógicas son válidas
 2. Qué tipo de acceso es válido
 2. Información necesaria para configurar la MMU cada vez que el proceso pasa a ocupar la CPU
 2. **Inicializar MMU**
 3. **Leer secciones del programa** del disco y escribir memoria
 4. Cargar registro **program counter** con la dirección de la instrucción definida en el ejecutable como **punto de entrada**

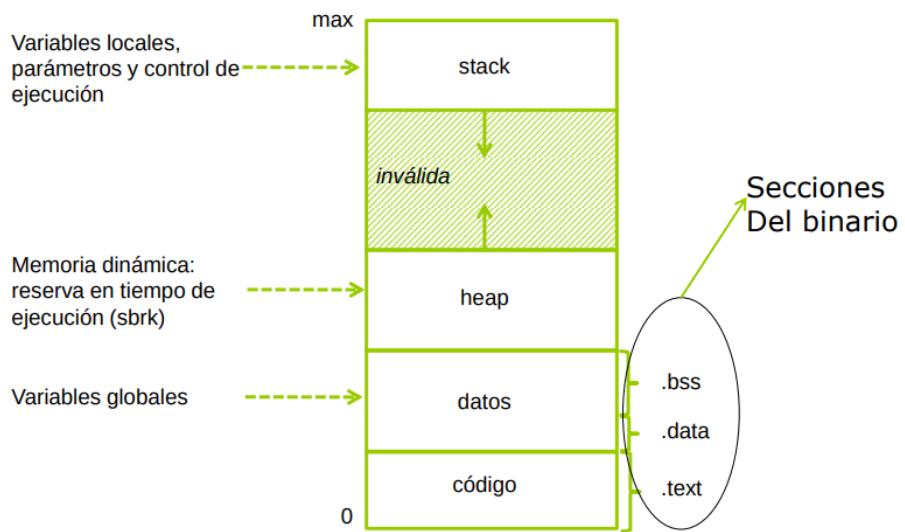
Carga: formato del ejecutable

- PASO 1: Interpretar el formato del ejecutable en disco
 - Si la traducción se hace en tiempo de ejecución... ¿que tipo de direcciones contienen los binarios? Lógicas o Físicas
 - Cabecera del ejecutable define las secciones: tipo, tamaño y posición dentro del binario (podéis probar `objdump -h programa`)
 - Existen diferentes formatos de ejecutable
 - ELF (*Executable and Linkable Format*): es el más extendido en sistemas POSIX

Algunas secciones por defecto de un ejecutable ELF	
.text	código
.data	Datos globales inicializados
.bss	datos globales sin valor inicial
.debug	información de debug
.comment	información de control
.dynamic	información para enlace dinámico
.init	código de inicialización del proceso (contiene la @ de la 1ª instrucción)

Carga: Esquema del proceso en memoria

- PASO 2: Preparar el esquema del proceso en memoria lógica
 - Esquema habitual



Carga: Optimización carga bajo demanda

- Optimizaciones: **carga bajo demanda**
 - **Una rutina no se carga hasta que se llama**
 - Se aprovecha mejor la memoria ya que no se cargan funciones que no se llaman nunca (por ejemplo, rutinas de gestión de errores)
 - Se acelera el proceso de carga (aunque se puede notar durante la ejecución)
 - Hace falta un mecanismo que detecte si las rutinas no están cargadas. Por ejemplo:
 - SO:
 - Registra en sus estructuras de datos que esa zona de memoria es válida y de dónde leer su contenido
 - En la MMU no le asocia una traducción
 - Cuando el proceso accede a la @, la MMU genera una excepción para avisar al SO de un acceso a una @ que no sabe traducir
 - SO comprueba en sus estructuras que el acceso es válido, provoca la carga y reanuda la ejecución de la instrucción que ha provocado la excepción

Servicio: Reservar/Liberar memoria dinámica

- Linux sobre Pentium
 - Interfaz tradicional de Unix poco amigable
 - brk y sbrk (usaremos esta)
 - Permiten modificar el límite del heap . El SO no tiene conciencia de que variables hay ubicadas en que zonas, simplemente aumenta o reduce el tamaño del heap
 - Programador es responsable de controlar posición de cada variable en el heap → La gestión es compleja

```
'limite_anterior_heap(int) sbrk(tamaño_variacion_heap);
```

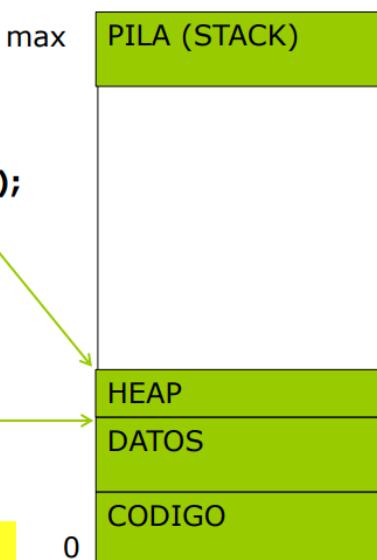
- >0 aumenta el heap
- <0 reduce el heap
- ==0 no se modifica

Sbrk: ejemplo

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


pids=sbrk(num_procs*sizeof(int));


for(i=0;i<10;i++){
    pids[i]=fork();
    if (pids[i]==0){
        ...
    }
}
sbrk(-1*num_procs*sizeof(int));
```



Sencillo si tenemos una variable, que
Pasaría si tenemos varias y queremos
"liberar" una del medio del heap??
NO SE PUEDE!

Servicio: Reservar/Liberar memoria dinámica

- La librería de C añade la gestión que vincula las direcciones con las variables.
 - Es una gestión transparente al kernel
- Librería de C. Pedir memoria: malloc(tamaño_en_bytes)
 - Si hay espacio **consecutivo** suficiente, lo marca como reservado y devuelve la dirección de inicio
 - Si no hay espacio **consecutivo** suficiente, aumenta el tamaño del heap
 - La librería de C gestiona el heap, sabe que zonas están libres y que zonas usadas. Intentar satisfacer peticiones sin recurrir al sistema
 - Al aumentar el heap, se reserva más de lo necesario con el objetivo de reducir el número de llamadas a sistema y ahorrar tiempo. La próxima petición del usuario encontrará espacio libre
- Librería de C. liberar memoria: free(zona_a_liberar)
 - Cuando el programador libera una zona se decide si simplemente pasa a formar parte de la lista de zonas libres o si es adecuado reducir el tamaño del heap
 - La librería ya sabe que tamaño tenía la zona ya que se supone que corresponde con una zona pedida anteriormente con malloc

Como sería con malloc/free

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


pids=malloc(num_procs*sizeof(int));


for(i=0;i<10;i++){
    pids[i]=fork();
    if (pids[i]==0){
        ...
    }
}
free(pids);
```

A la hora de pedir es igual, pero al Liberar hemos de pasar un puntero Concreto, no un tamaño

Memoria dinámica (IV): ejemplos

- Qué diferencias a nivel de *heap* observáis en los siguientes ejemplos?

- Ejemplo 1:

```
...  
new = sbrk(1000);  
...
```



- Ejemplo 2:

```
...  
new = malloc(1000);  
...
```



- Ejemplo 1:

```
...  
ptr = malloc(1000);  
...
```



- Ejemplo 2:

```
...  
for (i = 0; i < 10; i++)  
    ptr[i] = malloc(100);  
...
```



- Se reservan las mismas posiciones de memoria lógica?

- Ejemplo1: necesitamos 1000 bytes consecutivos
 - Ejemplo2: Necesitamos 10 regiones de 100 bytes

- Qué errores contienen los siguientes fragmentos de código?

- Código 1:

```
...  
for (i = 0; i < 10; i++)  
    ptr = malloc(SIZE);  
  
// uso de la memoria  
// ...  
  
for (i = 0; i < 10; i++)  
    free(ptr);  
...
```



```
int *x, *ptr;  
...  
ptr = malloc(SIZE);  
...  
x = ptr;  
...  
free(ptr);  
  
sprintf(buffer,"...%d",  
*x);
```



- Código 1: ¿Qué pasará en la segunda iteración del segundo bucle?
- Código 2: ¿Produce error siempre el acceso a “*x”?

Servicios básicos: asignación de memoria

- Cuando tenemos un problema de asignar una cantidad X (en este caso memoria) en una zona más grande, dependiendo de la solución aparecen problemas de **FRAGMENTACION**
 - También aparece en la gestión del disco
- Primera aproximación: **asignación contigua**
 - Espacio de @ físicas contiguo
 - ▶ Todo el proceso ocupa una partición que se selecciona en el momento de la carga
 - Poco flexible y dificulta aplicar optimizaciones (como carga bajo demanda)
- **Asignación no contigua**
 - Espacio de @ físicas no contiguo
 - Aumenta flexibilidad
 - Aumenta la granularidad de la gestión de memoria de un proceso
 - Aumenta complejidad del SO y de la MMU
- Basada en
 - **Paginación** (particiones fijas)
 - **Segmentación** (particiones variables)
 - Esquemas combinados
 - ▶ Por ejemplo, segmentación paginada

Asignación: Paginación

■ Esquema basado en paginación

- Espacio de @ lógicas dividido en particiones de tamaño fijo: **páginas**
- Memoria física dividida en particiones del mismo tamaño: **marcos**
- Asignación
 - ▶ Para cada página del proceso buscar un marco libre
 - Lista de marcos libres
 - ▶ Puede haber fragmentación interna
- Cuando un proceso acaba la ejecución devolver los marcos asignados a la lista de libres
- **Página: unidad de trabajo del SO**
 - ▶ Facilita la carga bajo demanda
 - ▶ Permite especificar protección a nivel de página
 - ▶ Facilita la compartición de memoria entre procesos
 - ▶ Normalmente, por temas de permisos, una página pertenece a una región de memoria (código/datos/heap/pila)

■ MMU

● Tabla de páginas

- Para mantener información a nivel de página: validez, permisos de acceso, marco asociado, etc....
- Una entrada para cada página
- Una tabla por proceso
- Suele guardarse en memoria y SO debe conocer la @ base de la tabla de cada proceso (por ejemplo, guardándola en el PCB)
- Procesadores actuales también disponen de TLB (*Translation Lookaside Buffer*)
 - Memoria asociativa (cache) de acceso **más rápido** en la que se almacena la información de traducción para las páginas activas
 - Hay que actualizar/invalidar la TLB cuando hay un cambio en la MMU
 - Gestión HW del TLB/Gestión Software (SO) del TLB
 - Muy dependiente de la arquitectura

Asignación: Paginación

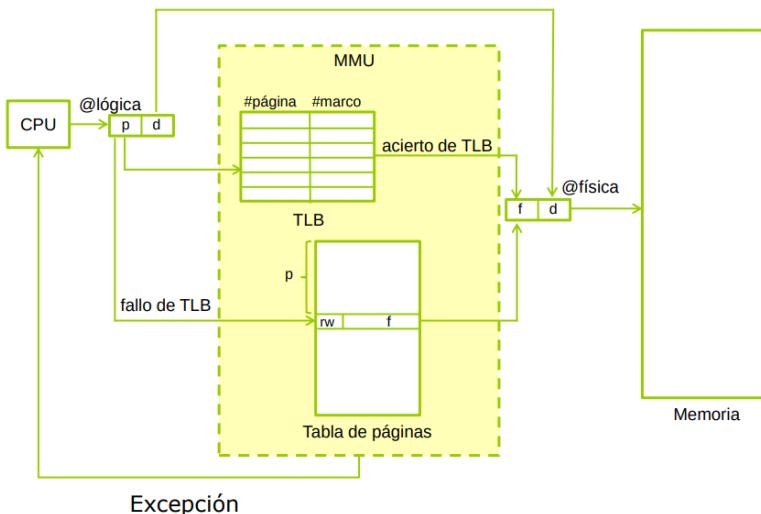
- PROBLEMA: Tamaño de las tablas de página (que están guardadas en memoria)
- Tamaño de página potencia de 2
 - Tamaño muy usado 4Kb (2^{12})
 - Influye en
 - Fragmentación interna y granularidad de gestión
 - Tamaño de la tabla de páginas
- Esquemas para reducir el espacio ocupado por las TP: TP multinivel
 - TP dividida en secciones y se añaden secciones a medida que crece el espacio lógico de direcciones

	Espacio lógico de procesador	Número de páginas	Tamaño TP
Bus de 32 bits	2^{32}	2^{20}	4MB
Bus de 64 bits	2^{64}	2^{52}	4PB

Asignación: Problema fragmentación

- Fragmentación de memoria: memoria que está libre pero no se puede usar para un proceso
 - **Fragmentación interna:** memoria asignada a un proceso aunque no la necesita. Esta reservada pero no ocupada.
 - **Fragmentación externa:** memoria libre y no asignada pero no se puede asignar por no estar contigua. No esta reservada pero no sirve.
 - Se puede evitar compactando la memoria libre si el sistema implementa asignación de @ en tiempo de ejecución
 - Costoso en tiempo

Asignación: Paginación



Asignación: Segmentación

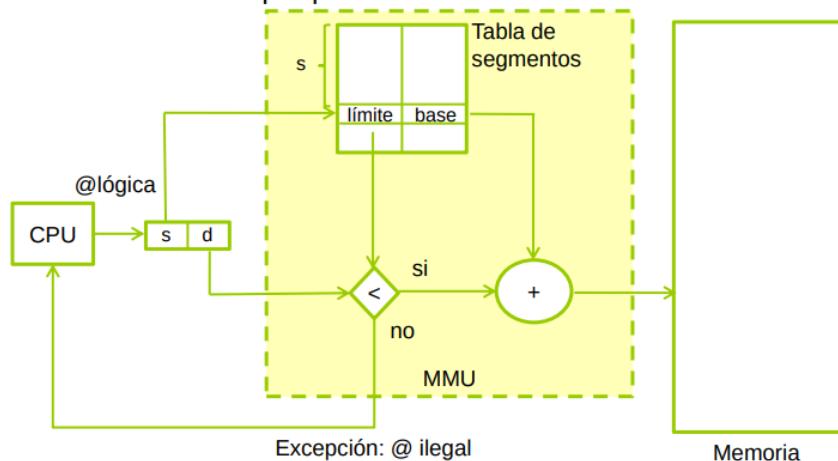
■ Esquema basado en segmentación

- Se divide el espacio lógico del proceso teniendo en cuenta el tipo de contenido (código, datos, etc)
 - Aproxima la gestión de memoria a la visión de usuario
- Espacio de @ lógicas dividido en particiones de tamaño variable (**segmentos**), **ajustado a lo que se necesita**
 - Como mínimo un segmento para el código y otro para la pila y los datos
 - Las referencias a memoria que hace el programa están formadas por un segmento y el desplazamiento dentro del segmento
- Memoria física libre contigua forma una partición disponible
- Asignación: Para cada segmento del proceso
 - Busca una partición en la que quepa el segmento
 - Posible políticas: first fit, best fit, worst fit
 - Selecciona la cantidad de memoria necesaria para el segmento y el resto continúa en la lista de particiones libres
 - Puede haber fragmentación externa
 - No todos los "trozos" libres son igual de buenos.

Asignación: Segmentación

■ MMU

- Tabla de segmentos
 - ▶ Para cada segmento: @ base y tamaño
 - ▶ Una tabla por proceso



Asignación: Esquema mixto

■ Esquemas combinados: segmentación paginada



- Espacio lógico del proceso dividido en segmentos
- Segmentos divididos en páginas
 - ▶ Tamaño de segmento múltiplo del tamaño de página
 - ▶ Unidad de trabajo del SO es la página

Optimizaciones: COW (Copy on Write)

- Objetivo: reducir la reserva/inicialización de memoria física hasta que sea necesario
 - Si no se accede a una zona nueva → no necesitamos reservarla realmente
 - Si no modificamos una zona que es una copia → no necesitamos duplicarla
 - Ahorra tiempo y espacio de memoria
- En el fork:
 - **Retrasar el momento de la copia de código, datos, etc mientras sólo se acceda en modo lectura**
 - Se puede evitar la copia física si los procesos sólo van a usar la región para leer, por ejemplo el código
 - Se suele gestionar a nivel de página lógica: se van reservando/copiando páginas a medida que se necesita
- Se puede aplicar
 - Dentro de un proceso : al pedir memoria dinámica
 - Entre procesos (por ejemplo, fork de Linux)
 - En general siempre que se aumenta/modifica el espacio de direcciones.

COW: Implementación

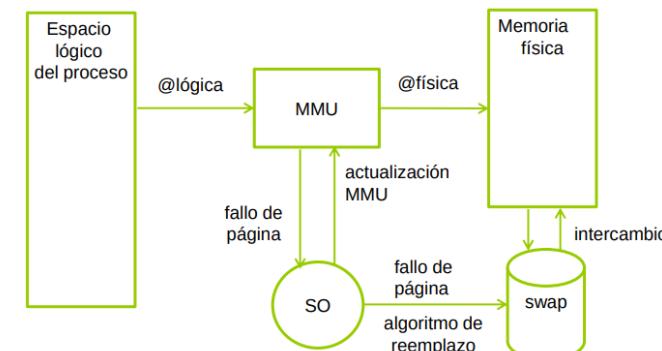
- La idea es: el kernel asume que se podrá ahorrar la reserva de la memoria física, pero necesita un mecanismo para detectar que no es así y realizar la reserva si realmente SI era necesaria
- En el momento que habría que hacer la asignación:
 - En la estructura de datos que describe el espacio lógico del proceso (en el PCB) el SO marca la región destino con los permisos de acceso reales
 - En la MMU el SO marca la región destino y la región fuente con permiso sólo de lectura
 - En la MMU el SO asocia a la región destino las direcciones físicas asociadas:
 - ▶ A las regiones del padre si era un fork (misma traducción, memoria compartida)
 - ▶ A una páginas que actúan de comodín en el caso de memoria dinámica
- Si un proceso intenta escribir en la zona nueva, la MMU genera excepción y SO la gestiona haciendo la reserva real y reiniciando el acceso

COW: ejemplo

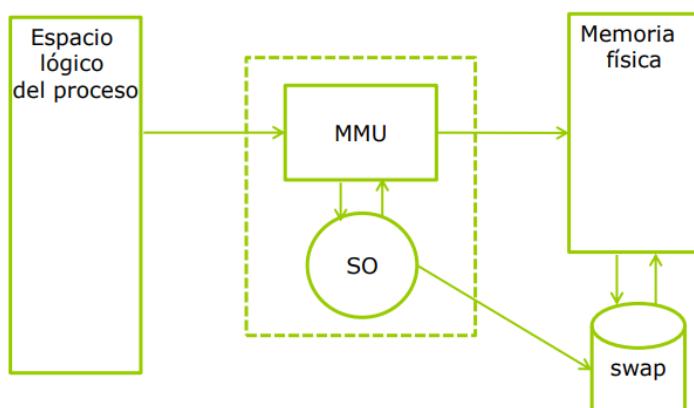
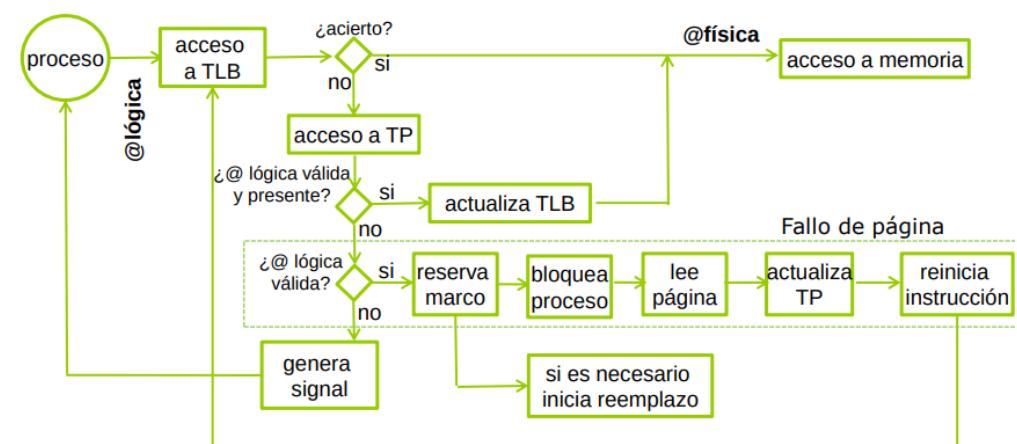
- Proceso A ocupa:
 - Código: 3 páginas, Datos 2 páginas, Pila: 1 página, Heap: 1 página
- Si proceso A ejecuta fork, justo después del fork:
 - Total memoria física:
 - Sin COW: proceso A= 7 páginas + hijo A= 7 páginas= 14 páginas
 - Con COW: proceso A= 7 páginas + hijo A=0 páginas = 7 páginas
- Al cabo de un rato... depende de lo que hagan los procesos, por ejemplo:
 - Si hijo A muta (y el nuevo espacio del hijo ocupa 10 páginas):
 - Sin COW: proceso A= 7 páginas + hijo A= 10 páginas= 17 páginas
 - Con COW: proceso A= 7 páginas + hijo A=10 páginas = 17 páginas
 - Si hijo A no muta, depende de lo que haga, pero el código al menos puede ser compartido, suponiendo que el resto no lo sea:
 - Sin COW: proceso A= 7 páginas + hijo A= 7 páginas= 14 páginas
 - Con COW: proceso A= 7 páginas + hijo A=4 páginas = 11 páginas
- En cualquier caso hay que ver que páginas se modifican (por lo tanto no se pueden compartir) y que páginas si se pueden compartir

Optimizaciones: Memoria Virtual (IV)

- **Reemplazo de memoria:** cuando SO necesita liberar marcos
 - Selecciona una página víctima y actualiza la MMU eliminando su traducción
 - Guarda su contenido en el área de swap para que se pueda recuperar
 - Asigna el marco ocupado a la página que se necesita en memoria
- Cuando se accede a una página guardada en el área de swap
 - MMU no puede hacer la traducción: genera excepción
 - **Fallo de página**
 - SO
 - Comprueba en las estructuras del proceso que el acceso es válido
 - Asigna un marco libre para la página (lanza el reemplazo de memoria si es necesario)
 - Localiza en el área de swap el contenido y lo escribe en el marco
 - Actualiza la MMU con la @física asignada



Pasos en el acceso a memoria



Optimizaciones: Memoria prefetch

- Objetivo: minimizar número de fallos de página
- Idea: anticipar qué páginas va a necesitar el proceso en el futuro inmediato y cargarlas con anticipación
- Parámetros a tener en cuenta:
 - Distancia de prefetch: con qué antelación hay que cargar las páginas
 - Número de páginas a cargar
- Algoritmos sencillos de predicción de páginas
 - Secuencial
 - Strided

Resumen: Linux sobre Pentium

- Llamada a sistema **exec**: provoca la **carga** de un nuevo programa
 - Inicialización del PCB con la descripción del nuevo espacio de direcciones, asignación de memoria, ...
- Creación de procesos (**fork**):
 - Inicialización del PCB con la descripción de su espacio de direcciones (copia del padre)
 - Se utiliza COW: hijo comparte marcos con padre hasta que algún proceso los modifica
 - Creación e inicialización de la TP del nuevo proceso
 - ▶ Se guarda en su PCB la @ base de su TP
- Planificación de procesos
 - En el **cambio de contexto** se actualiza en la MMU la @ base de la TP actual y se invalida la TLB
- Llamada a sistema **exit**:
 - Elimina la TP del proceso y libera los marcos que el proceso tenía asignados (si nadie más los estaba usando)

Resumen: Linux sobre Pentium

- Memoria virtual basada en segmentación paginada
 - Tabla de páginas multinivel (2 niveles)
 - ▶ Una por proceso
 - ▶ Guardadas en memoria
 - ▶ Registro de la cpu contiene la @ base de la TP del proceso actual
 - Algoritmo de reemplazo: aproximación de LRU
 - ▶ Se ejecuta cada cierto tiempo y cuando el número de marcos libres es menor que un umbral
- Implementa COW a nivel de página
- Carga bajo demanda
- Soporte para librerías compartidas
- Prefetch simple (secuencial)

Independència: principis de disseny

- L'objectiu és que els processos (codi principalment) sigui independent del dispositiu que s'està accedint
- Operacions d'E/S **uniformes**
 - Accés a tots els dispositius mitjançant les **mateixes crides al sistema**
 - Augmenta la simplicitat i la portabilitat dels processos d'usuari
- Utilitzant **dispositius虚拟s**
 - El procés no especifica concretament sobre quin dispositiu treballa, sinó que utilitza uns identificadors i hi ha una traducció posterior
- Amb possibilitat de **redirecccionament**
 - El sistema operatiu permet a un procés canviar l'assignació dels seus dispositius virtuals

```
% programa < disp1 > disp2
```



Dispositiu virtual

■ Nivell virtual: Aïlla l'usuari de la complexitat de gestió dels dispositius físics

- Estableix correspondència entre **nom simbòlic** (nom de fitxer) i l'aplicació d'usuari, a través d'un **dispositiu virtual**
 - Un nom simbòlic és la representació al sistema d'un dispositiu
 - /dev/dispX o bé .../dispX un nom de fitxer
 - Un dispositiu virtual representa un dispositiu en ús d'un procés
 - Dispositiu virtual = canal = descriptor de fitxer. És un **nombre enter**
 - Els processos tenen 3 canals estàndard:
 - » Entrada estàndard canal 0 (stdin)
 - » Sortida estàndard canal 1 (stdout)
 - » Sortida error estàndard canal 2 (stderr)



- Les crides a sistema de transferència de dades utilitzen com identificador el dispositiu virtual

Dispositiu lògic

■ Nivell lògic:

- Estableix correspondència entre dispositiu virtual i dispositiu (físic?)
- Gestiona dispositius que poden tenir, o no, representació física
 - P.ex. Disc virtual (sobre memòria), dispositiu nul
- Manipula blocs de dades de mida independent
- Proporciona una interfície uniforme al nivell físic
- Ofereix compartició (accés concurrent) als dispositius físics que representen (si hi ha)
- En aquest nivell es tenen en compte els permisos, etc
- A Linux s'identifiquen amb un nom de fitxer



Dispositiu físic

■ Nivell físic: Implementa a baix nivell les operacions de nivell lògic

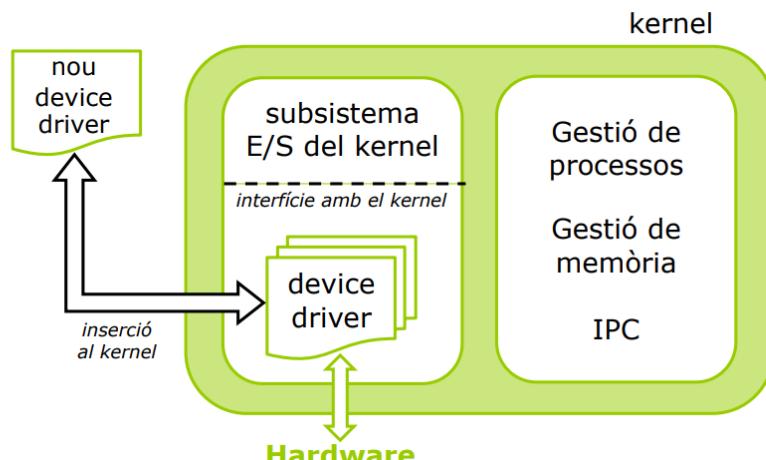
- Tradueix paràmetres del nivell lògic a paràmetres concrets
 - P.ex. En un disc, traduir posició L/E a cilindre, cara, pista i sector
- Inicialitza dispositiu. Comprova si lliure; altrament posa petició a cua
- Realitza la programació de l'operació demandada
 - Pot incloure examinar l'estat, posar motors en marxa (disc), ...
- Espera, o no, a la finalització de l'operació
- Retorna els resultats o informa sobre algun eventual error
- A Linux, un dispositiu físic s'identifica amb tres paràmetres:
 - Tipus: **Block/Character**
 - Amb dos nombres: major/minor
 - **Major**: Indica el tipus de dispositiu
 - **Minor**: instància concreta respecte al major



Device Driver

■ S'identifiquen unes operacions comunes (interfície) i les diferències específiques s'encapsulen en mòduls del SO: **device driver**

- Aïlla, la resta del kernel, de la complexitat de la gestió dels dispositius
- A més, protegeix el kernel en front d'un codi escrit per "altres"



Device Driver

■ **Controlador de dispositiu (en genèric):** Conjunt de rutines que gestionen un dispositiu, i que permeten a un programa interactuar amb aquest dispositiu

- Respecten la interfície definida pel SO (*obrir, llegir, escriure, ...*)
 - ▶ Cada SO té definida la seva pròpria interfície
- Implementen tasques dependents del dispositiu
 - ▶ Cada dispositiu realitza tasques específiques
- Contenen habitualment codi de baix nivell
 - ▶ Accés als ports d'E/S, gestió d'interrupcions, ...
- Queden encapsulades en un arxiu binari



Device Driver (DD) a linux: +detalle

■ Contingut d'un Device Driver (DD)

- **Informació general sobre el DD:** nom, autor, llicència, descripció, ...
- **Implementació de les funcions genèriques d'accés als dispositius**
 - ▶ *open, read, write, ...*
- **Implementació de les funcions específiques d'accés als dispositius**
 - ▶ Programació del dispositiu, accés als ports, gestió de les ints, ...
- **Estructura de dades amb llista d'apuntadors a les funcions específiques**
- **Funció d'inicialització**
 - ▶ S'executa en instal·lar el DD
 - ▶ Registra el DD en el sistema, associant-lo a un *major*
 - ▶ Associa les funcions genèriques al DD registrat
- **Funció de desinstal·lació**
 - ▶ Desregistra el DD del sistema i les funcions associades

■ Exemple de DD: veure `myDriver1.c` i `myDriver2.c` a la documentació del laboratori

Inserció dinàmica de codi : Mòduls de Linux

- Els kernels actuals ofereixen un mecanisme per afegir codi i dades al kernel, **sense necessitat de recompilar** el kernel
 - ▶ Una recompliació completa del kernel podria tardar hores...
- Actualment la inserció es fa dinàmicament (en temps d'execució)
 - *Dynamic Kernel Module Support* (linux) o *Plug&Play* (windows)
- Mecanisme de moduls a Linux
 - Fitxer(s) compilats de forma especial que contenen codi i/o dades per afegir al kernel
 - Conjunt de comandes per afegir/eliminar/veure mòduls
 - Ho veurem al laboratori

Mòduls d'E/S a linux: +details

■ Pasos per afegir i utilitzar un nou dispositiu:

- **Compilar** el DD, si escau, en un format determinat: .ko (kernel object)
 - ▶ El tipus (block/character) i el major/minor estan especificats al codi del DD
- **Instal·lar** (inserir) en temps d'execució les rutines del driver
 - ▶ `insmod fitxer_driver`
 - ▶ Recordar que el driver està associat a un major
- **Crear un dispositiu lògic** (nom d'arxiu) i lligar-lo amb el disp. Físic
 - ▶ Nom arxiu ↔ block/character + major i minor
 - ▶ **Comanda** `mknod`
 - `mknod /dev/mydisp c major minor`
- **Crear el dispositiu virtual (creda a sistema)**
 - `open("/dev/mydisp", ...);`

Físic

Lògic

Virtual

Exemples de dispositius (1)

- Funcionament d'alguns dispositius lògics: terminal, fitxer, pipe, socket

1. Terminal

- Objecte a nivell lògic que representa el conjunt teclat+pantalla
- “Habitualment” els processos el tenen a com a entrada i sortida de dades

2. Fitxer de dades

- Objecte a nivell lògic que representa informació emmagatzemada a “disc”. S'interpreta com una seqüència de bytes i el sistema gestiona la posició on ens trobem dintre aquesta seqüència.

■ Pipe

- Objecte a nivell lògic que implementa un buffer temporal amb funcionament FIFO. Les dades de la pipe s'esborren a mesura que es van llegint. Serveix per intercanviar informació entre processos
 - ▶ Pipe **sense nom**, connecta processos amb parentesc ja que només es accessible via herència
 - ▶ Pipe **amb nom**, permet connectar qualsevol procés que tingui permís per accedir al dispositiu

■ Socket

- Objecte a nivell lògic que implementa un buffer temporal amb funcionament FIFO. Serveix per intercanviar informació entre processos que es trobin en diferents computadores connectades per alguna xarxa
- Funcionament similar a les pipes, tot i que la implementació interna és molt més complexa ja que utilitzà la xarxa de comunicacions.

Tipus de fitxers a Linux



- A Linux, tots els dispositius s'identifiquen amb un fitxer (que pot ser de diferents tipus)
 - block device
 - character device
 - Directory
 - FIFO/pipe
 - Symlink
 - **regular file** → Son els fitxers de dades o “ordinary files” o fitxers “normals”
 - Socket
- Anomenarem fitxers “especials” a qualsevol fitxer que no sigui un fitxer de dades: pipes, links, etc

Creació de fitxers a Linux



- La majoria de tipus de fitxers es poden crear amb la crida a sistema/comanda mknod
 - Excepte directoris i soft links
 - Al laboratori utilitzarem la comanda, que **no serveix per crear fitxers de dades**
- mknod nom_fitxer TIPUS major minor
 - TIPUS= c → Dispositiu de caràcters
 - TIPUS=b → Dispositiu de blocs
 - TTIPUS=p → Pipe (no cal posar major/minor)

Tipus de Fitxers



dev name	type	major	minor	description
/dev/fd0	b	2	0	floppy disk
/dev/hda	b	3	0	first IDE disk
/dev/hda2	b	3	2	second primary partition of first IDE disk
/dev/hdb	b	3	64	second IDE disk
/dev/tty0	c	3	0	terminal
/dev/null	c	1	3	null device

Estructures de dades del kernel: Inodo



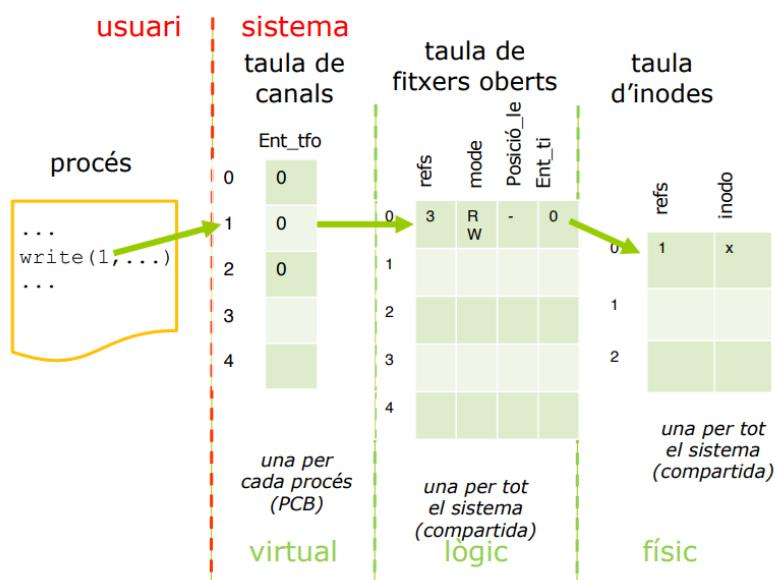
- Estructura de dades que conté tota la informació relativa a un fitxer
 - tamany
 - tipus
 - proteccions
 - propietari i grup
 - Data de l'últim accés, modificació i creació
 - Nombre d'enllaços al inodo (nombre de noms de fitxers que apunten a...)
 - índexos a les dades (indexación multinivel) → ho veurem al final del tema relacionat amb la gestió de la informació del disc
- Està tota la informació d'un fitxer excepte el nom, que està separat
- S'emmagatzema a disc, però hi ha una copia a memòria per optimitzar el temps d'accés

Estructures de dades de kernel



- Per procés
 - Taula de canals (TC): per cada procés (es guarda al PCB)
 - ▶ Indica a quins fitxers estem accedint.
 - ▶ S'accedeix al fitxer mitjançant el canal, que és un **índex a la TC**
 - ▶ El canal és el disp. Virtual
 - ▶ Un canal referencia a una entrada de la taula de fitxers oberts (TFO)
 - ▶ Camps que assumiren: `num_entrada_TFO`
- Global:
 - Taula de fitxers oberts (TFO):
 - ▶ Gestiona els fitxers en ús en tot el sistema
 - ▶ Poden haver entrades compartides entre més d'un procés i per varies entrades del mateix procés
 - ▶ Una entrada de la TFO referencia a una entrada de la TI.
 - ▶ Camps que assumirem: `num_referencies`, `mode`, `punter_le`, `num_entrada_ti`
 - Taula d'inodes (TI):
 - ▶ Conté informació sobre cada objecte físic obert, incloent les rutines del DD
 - ▶ És una copia en memòria de les dades que tenim al disc (es copia a memòria per eficiència)
 - ▶ Camps que assumirem: `num_referencies`, `dades_inode`

Estructures de dades de kernel



Operacions bàsiques d'E/S

Crida a sistema	Descripció
open	Donat un nom de fitxer i un mode d'accés, retorna el nombre de canal per poder accedir
read	Llegeix N bytes d'un dispositiu (identificat amb un nombre de canal) i ho guarda a memòria
write	Llegeix N bytes de memòria i els escriu al dispositiu indicat (amb un nombre de canal)
close	Allibera el canal indicat i el deixa lliure per ser reutilitzat.
dup/dup2	Duplica un canal. El nou canal fa referència a la mateixa entrada de la TFO que el canal duplicat.
pipe	Crea un dispositiu tipus pipe llesta per ser utilitzada per comunicar processos
lseek	Mou la posició actual de L/E (per fitxers de dades)

Les crides open, read i write poden bloquejar el procés

Open

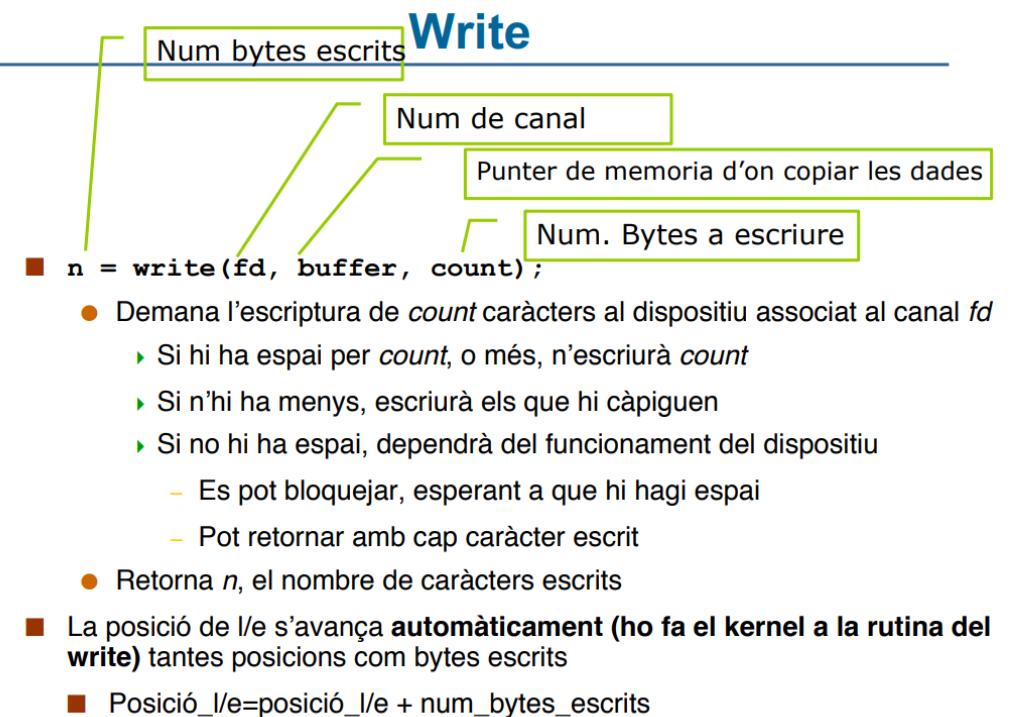
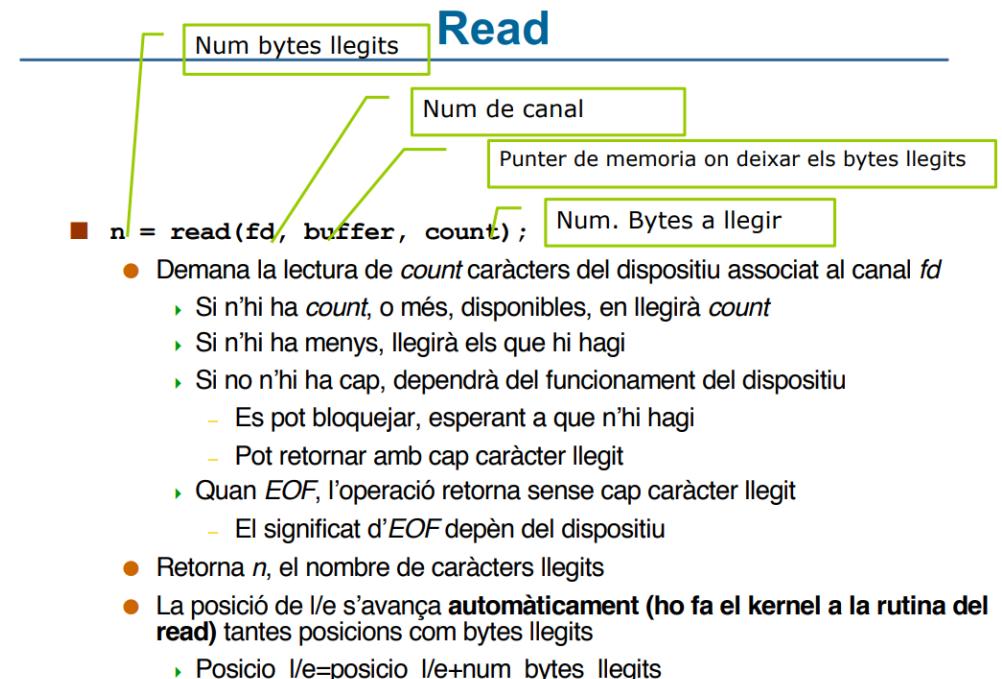
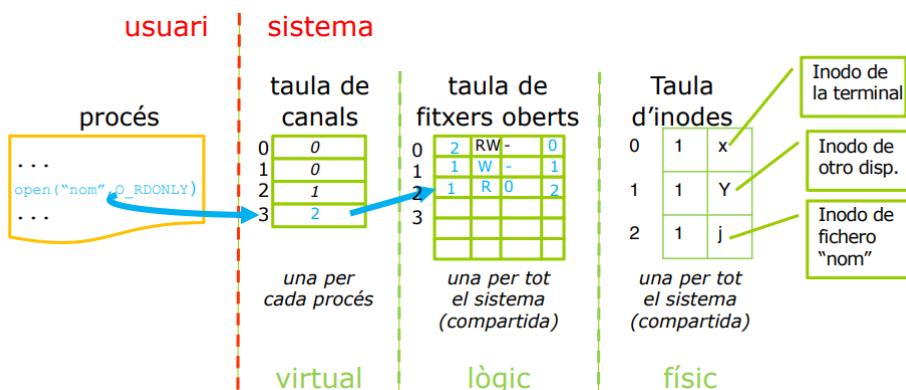
- Per tant, com s'associa un nom a un dispositiu virtual?
- `fd = open(nom, access_mode[, permission_flags]);`
 - Vincula nom de fitxer a dispositiu virtual (descriptor de fitxer o canal)
 - ▶ A més, permet fer un sol cop les comprovacions de proteccions. Un cop verificat, ja es pot executar `read/write` múltiples cops però ja no es torna a comprovar
 - Se li passa el **nom** del dispositiu i retorna el dispositiu virtual (`fd: file descriptor` o canal) a través del qual es podran realitzar les operacions de lectura/escriptura ,etc
 - El **access_mode** indica el tipus d'accés. Sempre ha d'anar un d'aquests tres com a mínim :
 - ▶ O_RDONLY (lectura)
 - ▶ O_WRONLY (escriptura)
 - ▶ O_RDWR (lectura i escriptura)

Open: Creació

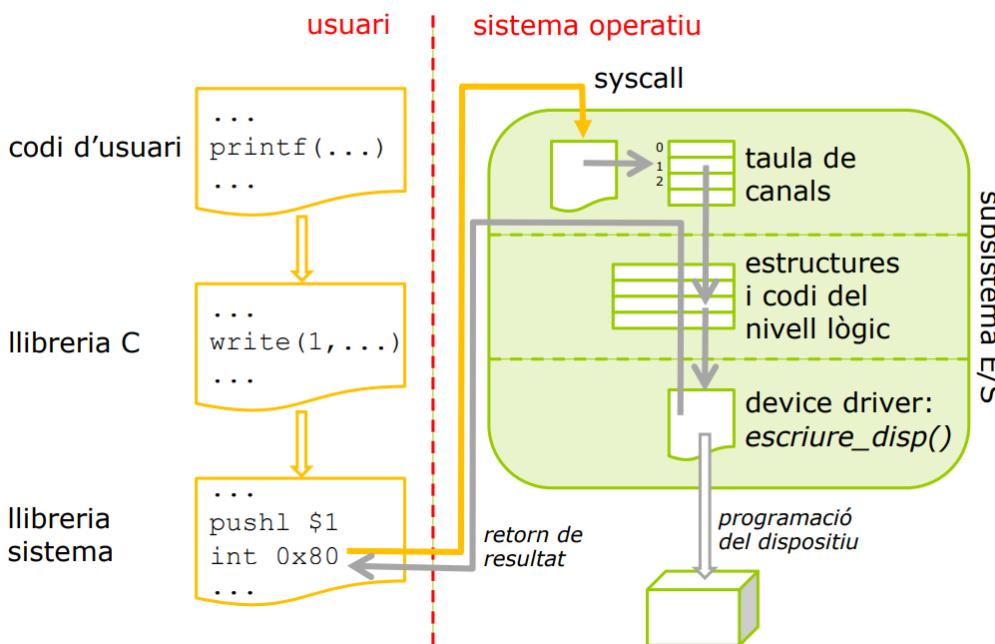
- Els fitxers especials han d'existir abans de poder accedir
- S'han d'especificar els `permission_flags`
 - És una OR (l) de: S_IRWXU, S_IRUSR, S_IWUSR, etc
- Els fitxers de dades es poden crear a la vegada que fem l'open:
 - En aquest cas hem d'afegir el flag O_CREAT (amb una OR de bits) a l'accés_mode
- No hi ha una crida a sistema per eliminar dades d'un fitxer parcialment, només les podem esborrar totes. Si volem truncar en contingut d'un fitxer afegirem el flag O_TRUNC a l'accés_mode
 - Ex1: `open("X", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR)` → Si el fitxer no existia el crea, si existia no té efecte
 - Ex2: `open("X", O_RDWR|O_CREAT|O_TRUNC, S_IRWXU)` → Si el fitxer no existia el crea, si existia s'alliberen les seves dades y es posa el tamany a 0 bytes.

Open: Estructures de dades

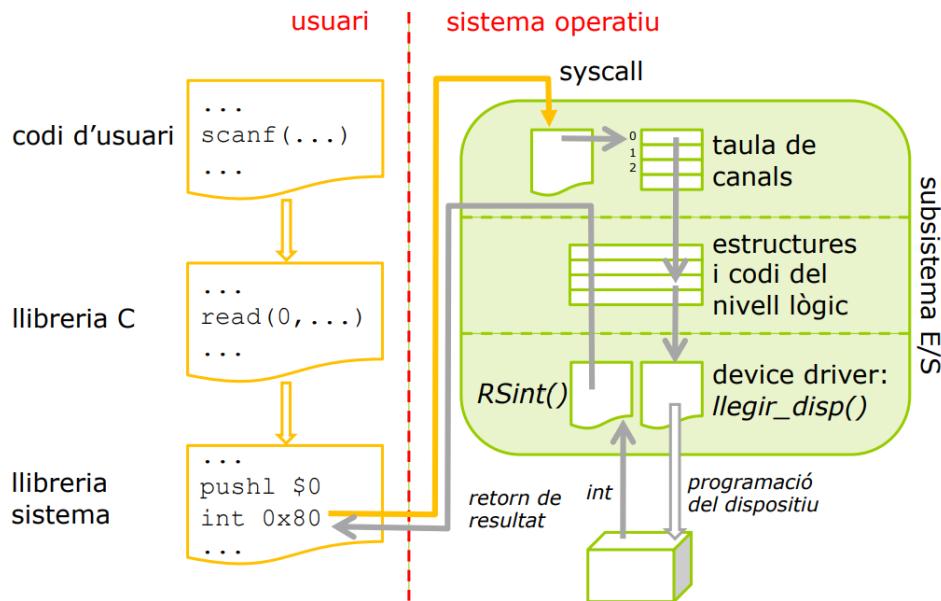
- Open (cont): Efecte sobre les estructures de dades internes del sistema
 - Ocupa un canal lliure de la TC. **Sempre serà el primer disponible**
 - Ocupa una nova entrada de la TFO: Posició L/E=0**
 - Associa aquestes estructures al DD corresponent (major del nom simbòlic). Pot passar que diferents entrades de la TFO apunten al mateix DD



Exemple: Escriptura a un dispositiu



Exemple: Lectura d'un dispositiu



Dup/dup2/close



- `newfd = dup(fd);`
 - Duplica un canal
 - Ocupa el primer canal lliure, que ara contindrà una còpia de la informació del canal original `fd`
 - Retorna `newfd`, l'identificador del nou canal

 - `newfd = dup2(fd, newfd);`
 - Igual que `dup`, però el canal duplicat és forçosament `newfd`
 - Si `newfd` era un canal vàlid, prèviament es tanca

 - `close(fd);`
 - Allibera el canal `fd` i les estructures associades dels nivells inferiors
 - Cal tenir en compte que diferents canals poden apuntar a la mateixa entrada de la TFO (p.ex. si `dup`), per tant, tancar un canal significaria decrementar -1 el comptador de referències a aquella entrada
 - El mateix passa amb els apuntadors a la llista d'inodes
- pipe**



- `pipe(fd_vector); // Dispositiu de comunicacions FIFO`
 - ▶ Crea una `pipe` **sense nom**. Retorna 2 canals, un de lectura i un altre d'escriptura
 - ▶ No utilitza cap nom del VFS (`pipe` **sense nom**) i, per tant, no executa la crida al sistema `open`
 - ▶ Només podrà ser usada per comunicar el procés que la crea i qualsevol descendent (directe o indirecte) d'aquest (pq heretaran els canals)
 - Per crear una `pipe` **amb nom**, cal fer: `mknod + open`
 - Internament: També crea 2 entrades a la taula de fitxers oberts (un de lectura i un d'escriptura) i una entrada temporal a la taula d'inodes.

- Utilitzacio
 - Dispositiu per comunicar processos
 - És bidireccional però, idealment cada procés l'utilitza en una única direcció, es aquest cas, el kernel gestiona la sincronització

- Dispositiu bloquejant:
 - ▶ Lectura: Es bloqueja el procés fins que hi hagi dades a la `pipe`, tot i que no necessàriament la quantitat de dades demandades.
 - Quan no hi ha cap procés que pugui escriure i `pipe` buida, provoca EOF (`return==0`) → *Si hi ha processos bloquejats, es desbloquejen*
 - ▶ Escriptura: El procés escriu, a no ser que la `pipe` estigui plena. En aquest cas es bloqueja fins que es buidi.
 - Quan no hi ha cap procés que pugui llegir el procés rep una excepció del tipus `SIGPIPE` → *Si hi ha processos bloquejats, es desbloquejen*
 - ▶ Cal tancar els canals que no s'hagin de fer servir! Altrament bloqueig





- La posició de l/e es modifica manualment per l'usuari. Permet fer accessos directes a posicions concretes de fitxers de dades (o fitxers de dispositius que ofereixin accés seqüencial)

- S'inicialitza a 0 a l' open
- S' incrementa automàticament al fer read/write
- El podem moure manualment amb la crida lseek

■ nova_posicio=lseek(fd, desplaçament *relatiu_a*)

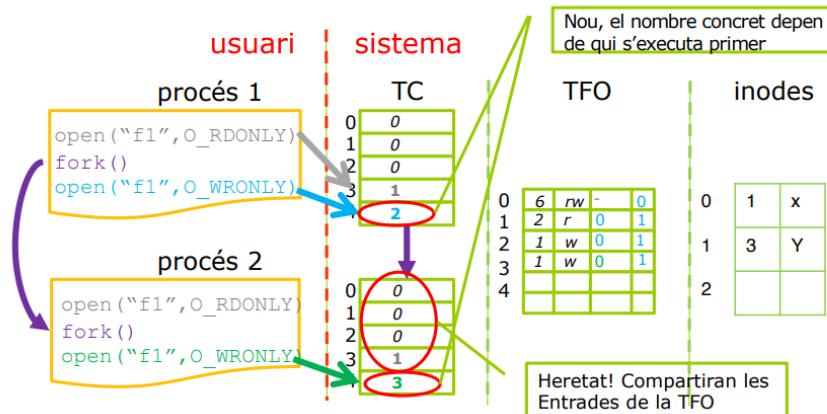
- SEEK_SET: posicio_l/e=desplaçament
- SEEK_CUR: posicio_l/e=posicio_l/e+desplaçament
- SEEK_END: posicio_l/e=file_size+desplaçament
- "desplaçament" pot ser negatiu

E/S i execució concurrent (1)



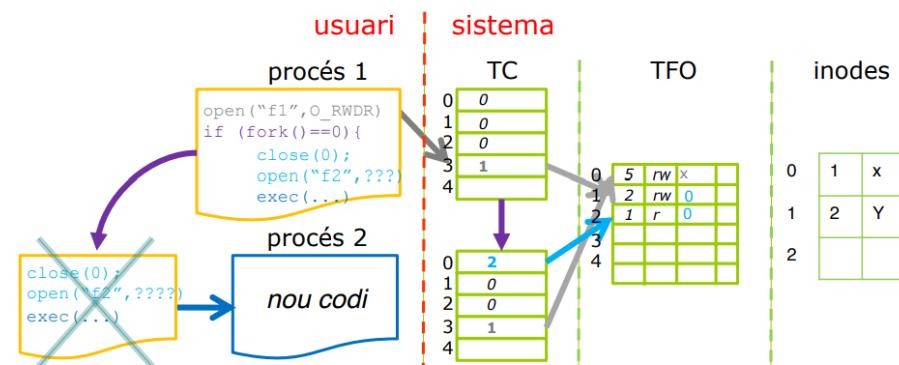
E/S i fork

- El procés fill hereta una **còpia** de la taula de canals del pare
 - ▶ Totes les entrades obertes apunten al mateix lloc de la TFO
- Permet **compartir** l'accés als dispositius oberts abans del fork
- Els següents *open* ja seran independents



E/S i exec

- La nova imatge **manté** les estructures internes d'E/S del procés
- El fet d'executar *fork+exec* permet realitzar redireccions abans de canviar la imatge



E/S i execució concurrent (3)



- Si un procés està bloquejat en una operació d'E/S i l'operació és interrompuda per un signal podem tenir dos comportaments diferents:
 - Després de tractar el signal, l'operació interrompuda es reinicia (el procés continuarà bloquejat a l'operació)
 - Després de tractar el signal, l'operació retorna error i la *errno* pren per valor EINTR
- El comportament depèn de:
 - La gestió associada al signal:
 - ▶ si flag SA_RESTART a la sigaction → es reinicia l'operació
 - ▶ en cas contrari → l'operació retorna error
 - L'operació que s'està fent (per exemple, mai es reinicen operacions sobre sockets, operacions per a esperar signals, etc.)
- Exemple de com protegir la crida al sistema segons el model de comportament:


```
while ( (n = read(...)) == -1 && errno == EINTR );
```

Característiques terminal



- Terminal: Funcionament (canonical ja que és el cas per defecte)
 - Lectura
 - El buffer guarda caràcters fins que es pitja el CR
 - Si hi ha procés bloquejat esperant caràcters, li dona els que pugui
 - Altrament ho guarda; quan un procés en demana li dona directament
 - ^D provoca que la lectura actual acabi amb els caràcters que hi hagi en aquell moment, encara que no n'hi hagi cap:

```
while ( (n=read(0, &car, 1)) > 0 )
      write(1, &car, 1);
```
 - Això, per convenció, s'interpreta com a final de fitxer (EOF)
 - Escriptura
 - Escriu un bloc de caràcters
 - Pot esperar que s'escrigui el CR per ser visualitzat per pantalla
 - El procés no es bloqueja
 - Aquest comportament bloquejant pot ser modificat mitjançant crides al sistema que canvien el comportament dels dispositius

Característiques pipes



Pipe

- Lectura
 - Si hi ha dades, agafa les que necessita (o les que hi hagi)
 - Si no hi ha dades, es queda bloquejat fins que n'hi hagi alguna
 - Si la pipe està buida i no hi ha cap possible escriptor (tots els canals d'escriptura de la pipe tancats), el procés lector rep *EOF*
 - Cal tancar sempre els canals que no es facin servir
- Escriptura
 - Si hi ha espai a la pipe, escriu les que té (o les que pugui)
 - Si la pipe està plena, el procés es bloqueja
 - Si no hi ha cap possible lector, rep *SIGPIPE*
- Aquest comportament bloquejant pot ser modificat mitjançant crides al sistema que canvien el comportament dels dispositius (fcntl)

Dispositius de xarxa



Socket

- Funciona similar a una *pipe*, excepte que per comptes de dos canals (fd[2]) utilitza un únic canal per la lectura i l'escriptura del *socket*
- Es crea un *socket*, que es connecta al *socket* d'un altre procés en una altra màquina, connectada a la xarxa
 - Es pot demanar connexió a un *socket* remot
 - Es pot detectar si una altra es vol connectar al *socket* local
 - Un cop connectats, es poden enviar i rebre missatges
 - *read /write* o, més específicament, *send /recv*
- Es disposa de crides addicionals que permeten implementar servidors concurrents o, en general, aplicacions distribuïdes
- Habitualment, s'executen aplicacions amb un model client-servidor

Espai de noms: Directori



- Directori: Estructura lògica que organitza els fitxers.
- És un fitxer especial (tipus directori) gestionat pel SO (els usuaris no el poden obrir i manipular)
- Permet enllaçar els noms dels fitxers amb els seus atributs
 - atributs
 - Tipus de fitxer directori(d)/block(b)/character(c)/pipe(p)/link(l),socket(s), de dades(-)
 - tamany
 - propietari
 - permisos
 - Dades de creació, modificació, ...
 - ...
 - Ubicació al dispositiu de les seves dades (en cas de fitxers de dades)

En Linux, tot això està a l'inode

Linux: Visió d'usuari

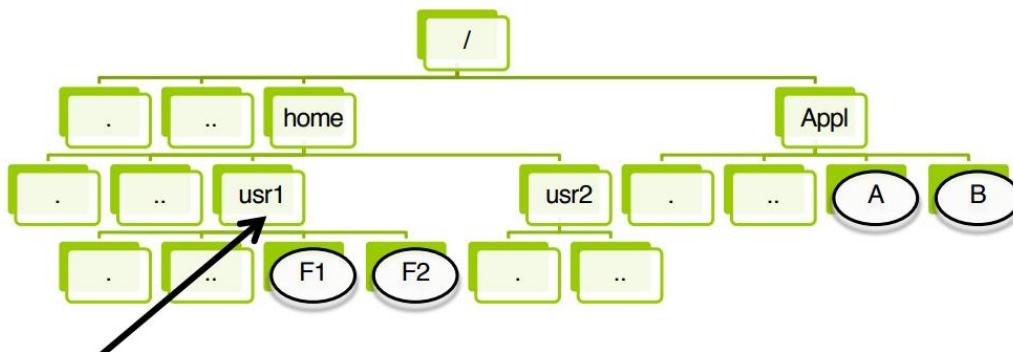


- Els directoris s'organitzen de forma jeràrquica (formen un graf)
- Permeten que l'usuari classifiqui les seves dades
- El Sistema de Fitxers organitza els dispositius d'emmagatzemament (cada un amb el seu esquema de directoris), en un espai de noms global amb un únic punt d'entrada
- El punt d'entrada és el directori "root" (Arrel), de nom "/"
- Un directori sempre té, com a mínim, 2 fitxers especials (de tipus directori)
 - . Referència al directori actual
 - .. Referència al directori pare

Linux: Visió d'usuari



- Cada fitxer es pot referenciar de dues formes:
 - Nom absolut (únic): Camí des de l'arrel + nom
 - Nom relatiu: Camí des del directori de treball + nom



Si estem aquí, podem referir-nos a f2 com:
Nom relatiu: f2
Nom absolut: /home/usr1/F2

Linux: Noms de fitxers

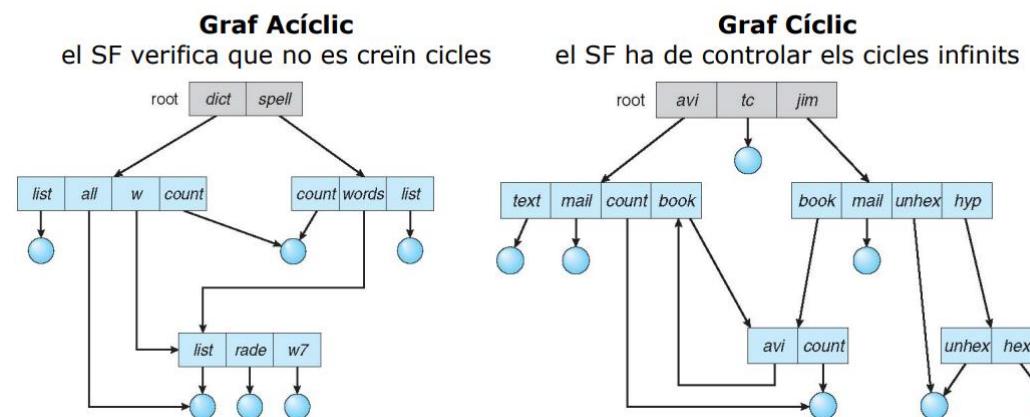


- Com el nom del fitxer està separat de la informació (inode), a Linux es permet que un inode sigui referenciat per més d'un nom de fitxer
- Hi han dos tipus de vincles entre nom de fitxer i inode
 - Hard-link:
 - ▶ El nom del fitxer referència directament al nombre d'inode on estan els atributs+info de dades
 - ▶ És el més habitual
 - ▶ A l'inode hi ha un nombre de referències que indica quants noms de fitxers apunten a aquest inode
 - És diferent al nombre de referències que hi ha a la taula d'inodes!!!
 - Soft-link
 - ▶ El nom del fitxer no apunta directament a la informació sinó que apunta a un inode que conté el nom del fitxer destí (path absolut o relatiu)
 - ▶ És un tipus de fitxer especial (l)
 - ▶ Es crea amb una comanda (ln) o crida a sistema (symlink)

Implementació jerarquia directoris



- La existència dels dos tipus de vincles influeix en la estructura del directori (que es un graf)
 - No es permeten cicles amb hard-links
 - Si se permeten cicles amb soft-links



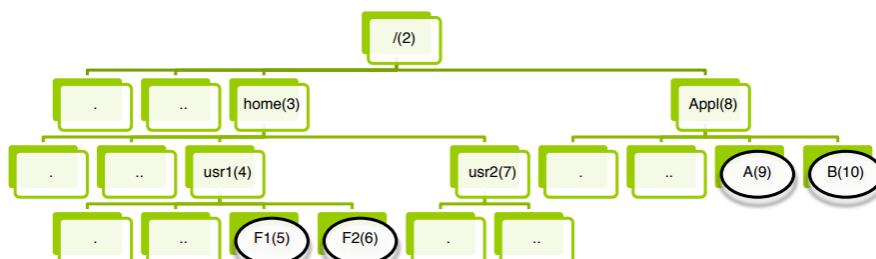
Problemes dels directoris en grafs

- Backups (copies de seguretat)
 - No fer copies del mateix fitxer dues vegades
- Eliminació d'un fitxer
 - Soft links
 - ▶ El sistema no comprova si hi han soft-links a un fitxer. A l'hora d'accendir es detectarà que no existeix.
 - Hard links
 - ▶ El fitxer només s'elimina quan el comptador de referències arriba a zero

Visió interna de directoris



- El contingut d'un directori permet relacionar el nom del fitxer amb el seu número d'inode
- Pex: A l'exemple anterior, si assignem nombres d'inodes



Nom	inode
.	2
..	2
home	3

Directori /

Cas especial

Directori /home/usr1

Nom	inode
.	4
..	3
F1	5

Nom	inode
F2	6

Permisos d'accés a fitxers



- El SF permet assignar diferents permisos als fitxers
 - Es defineixen nivells d'accés i operacions que es poden fer
- Linux:
 - Nivells d'accés: Propietari, Grup d'usuaris, Resta d'usuaris
 - Operacions: Lectura (r), Escriptura (w), Execució (x) → No confondre amb el mode d'accés!
 - Quan es defineix de forma numèrica, s'utilitza la base 8 (octal). Alguns valors típics

R	W	X	Valor numèric
1	1	1	7
1	1	0	6
1	0	0	4

Crides a sistema: espai de noms i permisos



Servicio	Llamada a sistema
Crear / eliminar enllaç a fitxer/soft-link	link / unlink/symlink
Cambiar permisos de un fitxer	chmod
Cambiar propietari/grup de un fitxer	chown / chgrp
Obtenir informació del Inodo	stat, Istat, fstat

- Existeixen més crides a sistema que ens permeten manipular:

- Permisos
- Vincles
- Característiques
- etc



- Per a cada fitxer, el SO ha de saber quins són els seus blocs.

- Taula d'índexos: blocs assignats a un fitxer

- Està a l'inode del fitxer: Quants indexos?

- ▶ Esquema multinivel: índexos directes i indirectes

Índexos a blocs de dades (1/4 Kb)

- 10 índexos directes

- ▶ (10 blocs = 10/40Kb)

- 1 índice indirecte

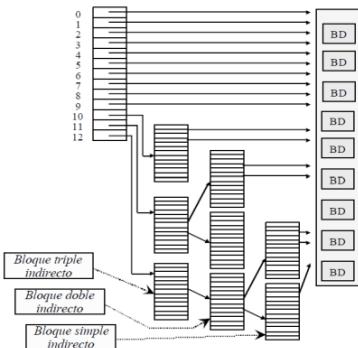
- ▶ (256/1024 blocs = 256Kb/4Mb)

- 1 índice indirecte doble

- ▶ (65K/1M blocs = 65Mb/4 Gb)

- 1 índice triple indirecte

- ▶ (16M/1G blocs = 16Gb/4 Tb)



Metadades



Metadades persistents: guardades a disc

- Inodes i llista de blocs d'un fitxer

- Directoris

- Llista de blocs lliures

- Llista d'inodes lliures

- ... i altra informació necessària per al sistema de fitxers (quin es l'inode del directori arrel, tamany de bloc...)

Superbloc: Bloc de la partició que conté les metadades del sistema de fitxers (redundància per a que sea tolerant a fallades)

Metadades i memòria



- Zona de memòria per a guardar els últims inodes llegits

- Zona de memòria per a guardar el últims directoris llegits

- Buffer cache: zona de memòria per a guardar els últims blocs llegits

- Superbloc: es guarda també a memoria



- Localitza l'inode del fitxer amb el que es vol treballar i el deixa a memòria

- Quin és l'inode? Cal llegir el directori del fitxer. Dos situacions:

- ▶ O el directori està a la cache de directoris i simplement s'accedeix

- ▶ O el directori està a disc i s'ha de llegir (i es deixa a memòria).

- ▶ Com sabem quins blocs llegir? → estan a l'inode

- ▶ Quin es l'inode? → està al directori (repetim el procés)

Si localitza l'inode

- Comprova si l'accés es correcte (els permisos d'accés són els adequats) → si no retorna erro

- Si és un soft-link, llegeix el path del fitxer al que apunta i localitza l'inode corresponent (repetim el procés)

- ▶ Si el path es curt, es troba al mateix inode

- ▶ Si el path es llarg, es guarda en un bloc de dades separat

- Modifica la taula de canals, la taula de fitxers oberts i la taula d'inodes

Si no localitza l'inode: error per accedir a un fitxer que no existeix

No accedeix a cap bloc de dades del fitxer objectiu



- Tenim a la taula d'inodes l'inode corresponent (i la informació sobre els blocs de dades)

- Tenim a la TFO el valor del punter de lectura/escritura (la posició del fitxer que toca accedir)

Cal calcular els blocs involucrats:

- Primer comprovem si estem al final del fitxer (valor del punter == mida del fitxer)

- Després calculem els blocs

- ▶ Dividint el valor del punter entre la mida de bloc obtenim l'index del primer bloc a llegir

- ▶ Amb el paràmetre mida de la syscall podem calcular quants blocs més cal llegir

- Si els blocs ja s'havien utilitzat abans els podem trobar a memòria, si no cal llegir-los de disc (i es deixen a memòria).



- La diferència amb el read es que si l'escriptura es fa al final del fitxer pot ser necessari afegir més blocs al fitxer. En aquest cas, caldrà modificar la llista de blocs lliures del superbloc i actualitzar l'inode del fitxer amb els nous blocs i la nova mida del fitxer