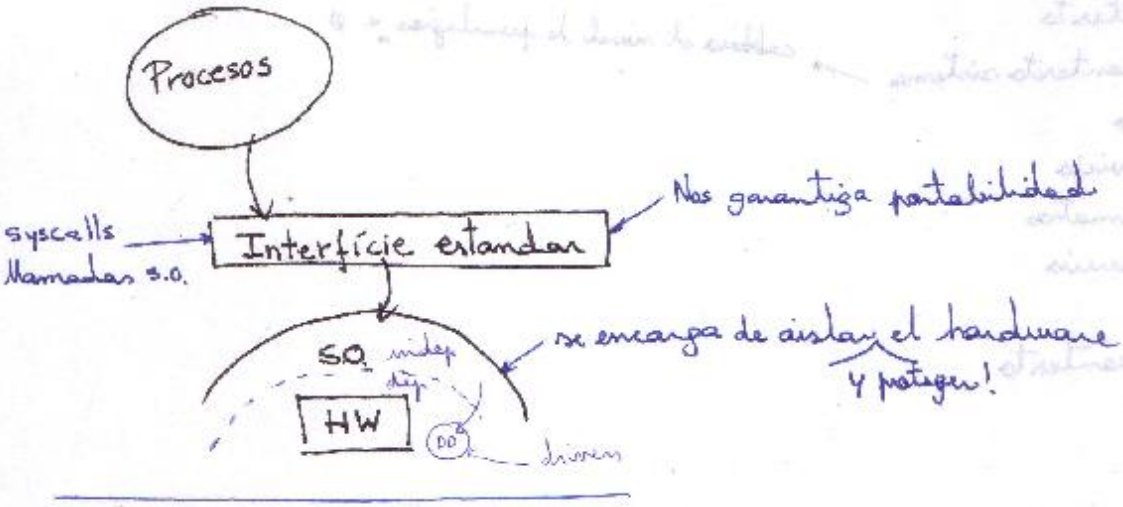
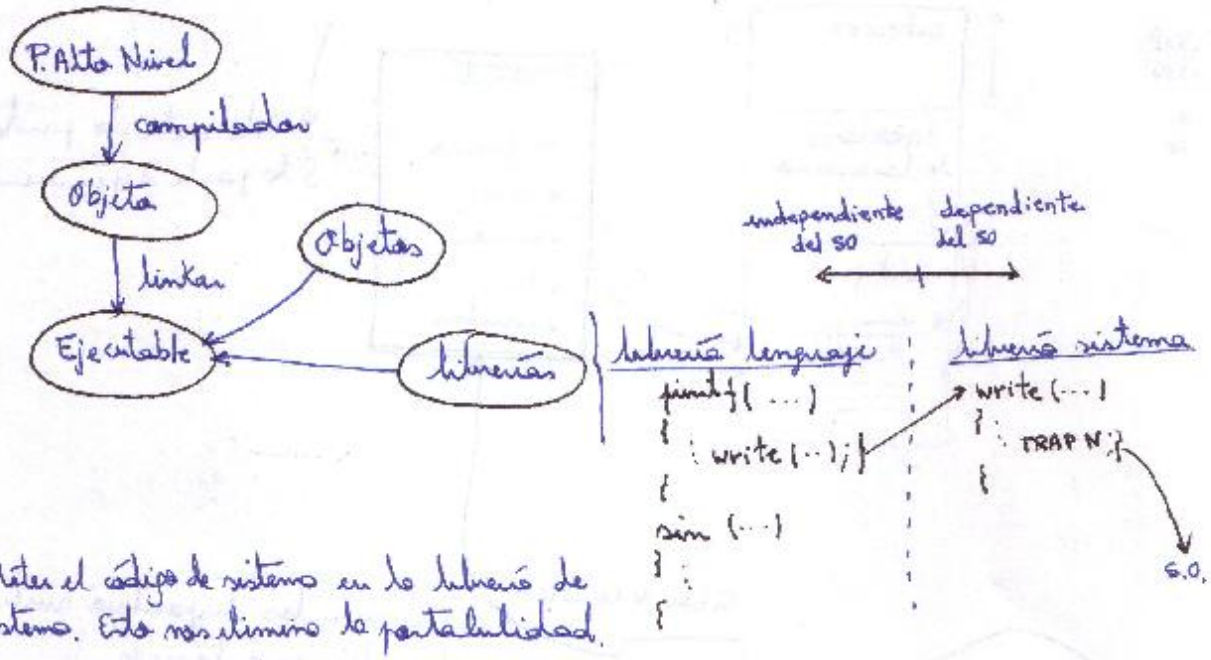
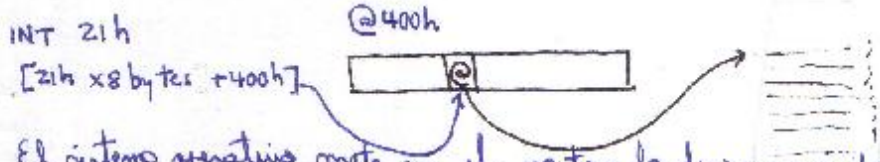




- inlining Meter el código de sistema en la librería de sistema. Esto nos elimina la portabilidad.

- TRAP / INT / Syscalls : Son interrupciones software,



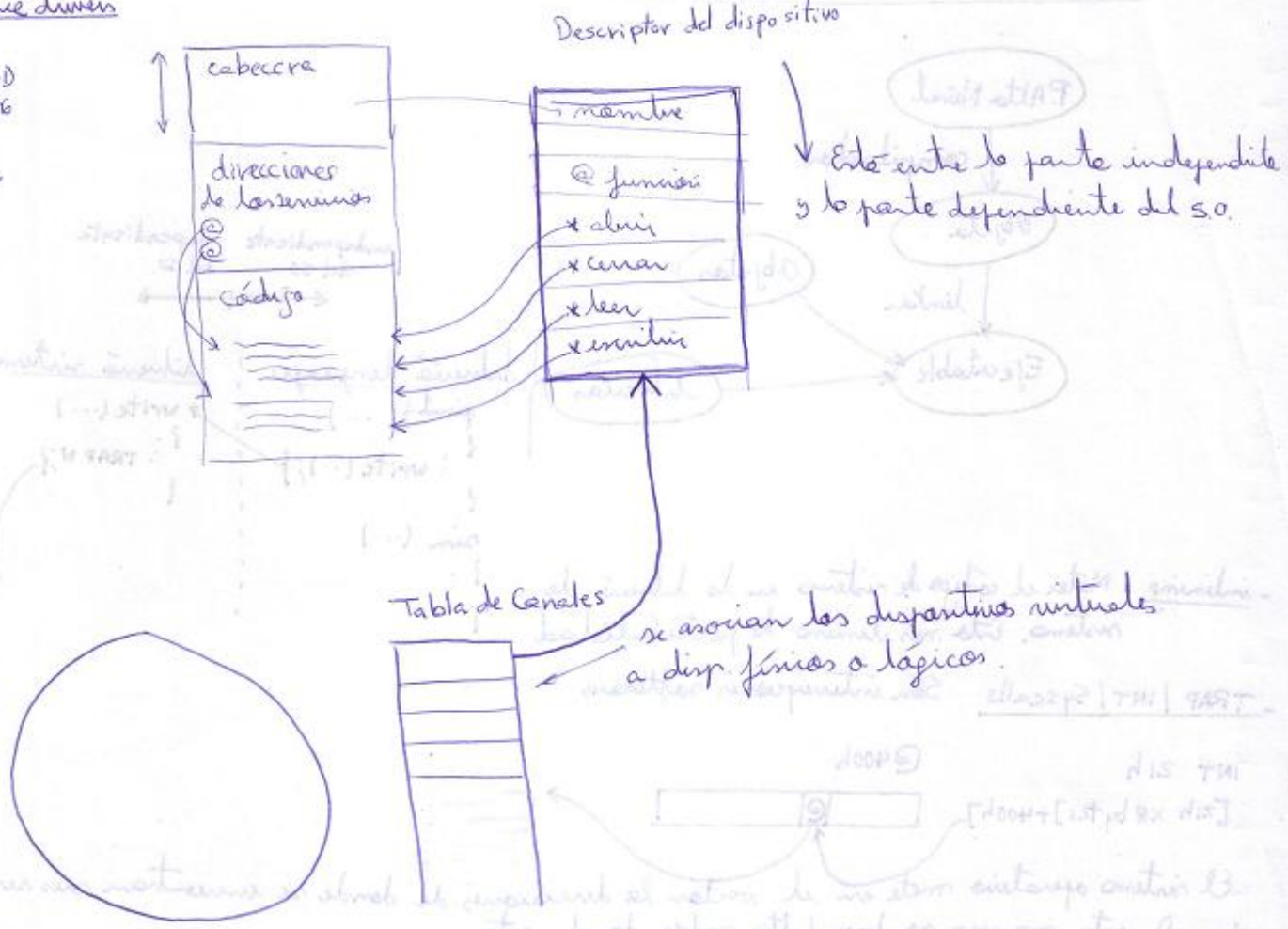
El sistema operativo mete en el vector las direcciones de donde se encuentran sus servicios. De esta manera no hace falta saber donde están.

TRAP/INT

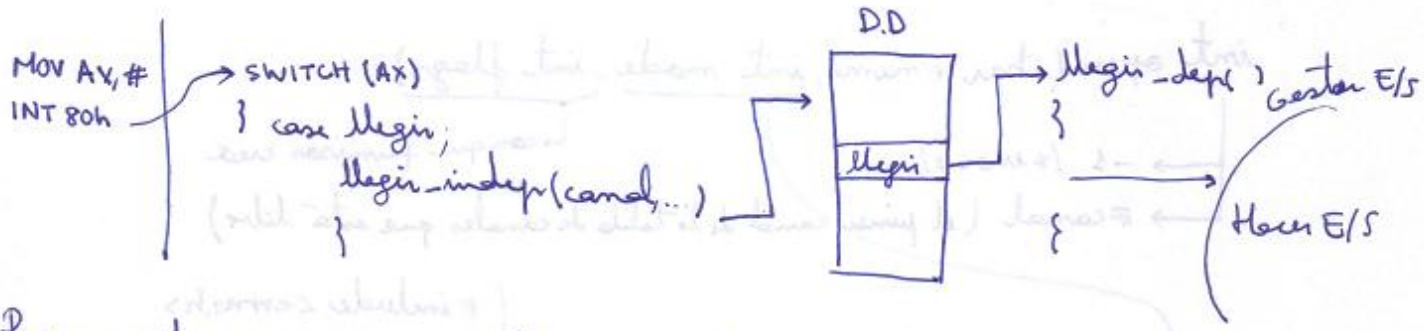
Salvar contexto
Restaurar contexto sistema → cambio el nivel de privilegios a 0
Id. servicio
Invoca servicio
Recup. parametros
Ejecutar servicio
Return
Restaurar contexto

Device drivers

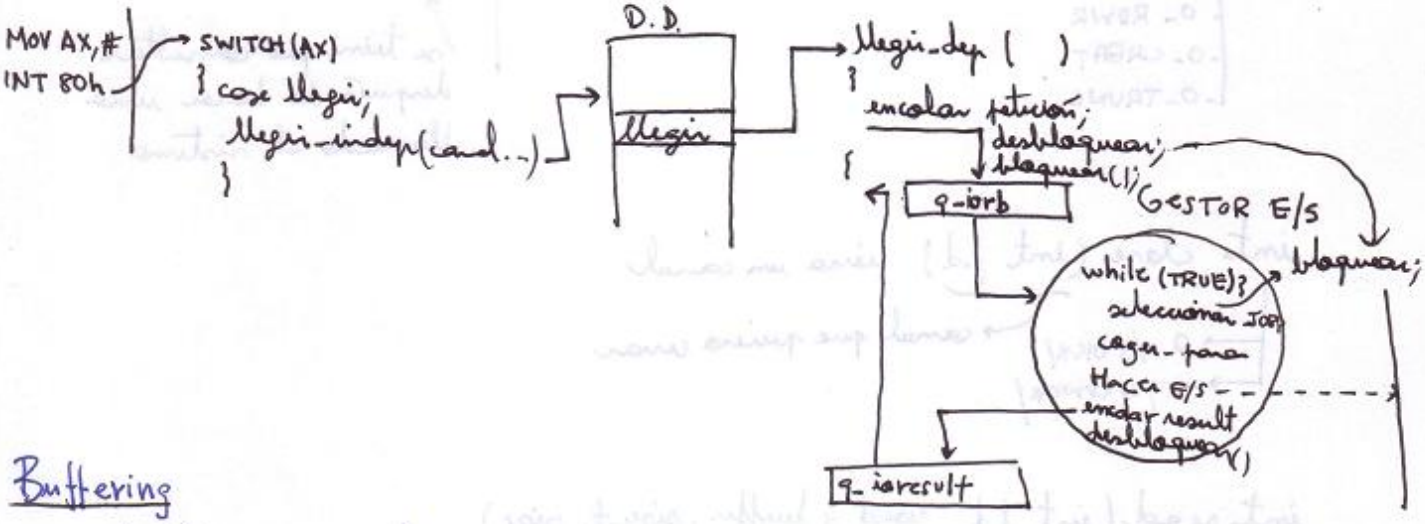
- .VxD
- .386
- .O
- .KO



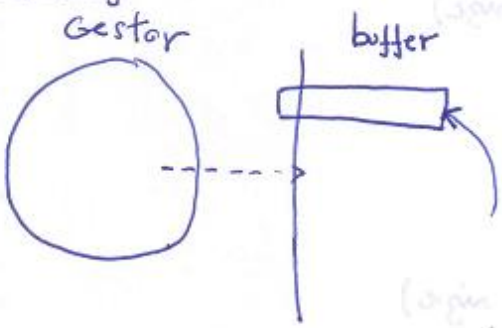
Debemos permitir que mientras hacemos E/S otros procesos se puedan ejecutar, por tanto.



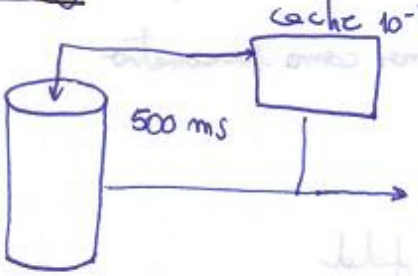
Pero con este esquema, un segundo proceso perdería su petición si accediera al mismo gestor. Portanto.



Buffering



Cacheing



Llamadas al sistema

int open (char * name, int mode, int flags)

- → -1 /* error */
- → #canal (el primer canal de la tabla de canales que esté libre)

mode → con que permisos crea

flags:
- O_RDONLY
- O_WRONLY
- O_RDWR
- O_CREAT
- O_TRUNC

```
#include <errno.h>
int errno
char *strerror(int errno)
```

↑ se tiene que consultar después de hacer una llamada al sistema

int close (int fd) cierra un canal

fd → canal que quiero cerrar

- → 0 /* OK */
- → -1 /* error */

int read (int fd, void * buffer, size_t size)

- → #bytes leidos
- → -1 /* error */

int write (int fd, void * buffer, size_t size)

int dup (int fd) duplica el canal que le pasamos como parametro

→ #canal

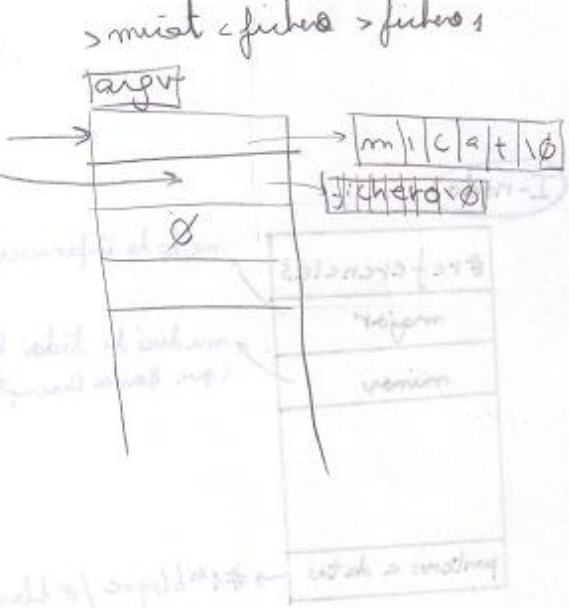
int dup2 (int fds, int fdd) copia fds en fdd.

→ fdd

2.2 *
```
main()
} char c;
  while (read (0,&c,1)>0)
    write (1,&c,1)
```

2.3 >micat ficheo

```
main (int argc, char **argv)
} char c; int fd;
 fd = open (argv[1], O_RDONLY)
 while(read(fd,&c,1)>0)
    write (1,&c,1);
 close(fd);
 {
```



>micat ficheo ficheo1

```
main (int argc, char **argv)
} char c; int fd, fd1;
 fd = open (argv[1], O_RDONLY)
 fd1 = open (argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0666)
 while (read (fd,&c,1)>0)
    write (fd1, &c, 1);
 close (fd);
 close (fd1);
 {
```

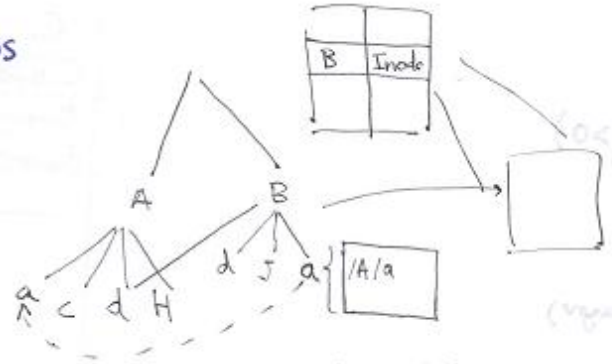
```
* main()
 } char c[256]
  while ((b=read (0, c, 256*sizeof(char)))>0)
     write(1,c,b);
 {
```

antes de escribir la memoria en formato interno de la maquina hay que hacer sprintf(buffer, "%d", int)
Al revés se hace ~~atoi~~ int atoi(char *str)

Sistema de ficheros

<nombre, I-nodo>

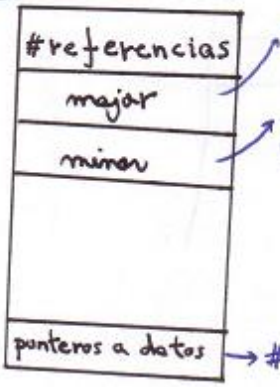
hard links



Softlinks: fichero que contiene la dirección de otro fichero o directorio (no asegura consistencia)

hardlinks: solo en un mismo dispositivo físico, y mismo partición. (pareja nombre i-nodo)

I-nodo



→ medio la información sobre el tipo de dispositivo

→ medio de todos los dispositivos de la clase "major" que voy a atacar. (que device descriptor voy a usar)

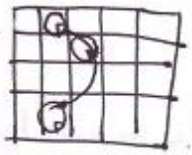
→ #1er bloque / #bloque de índices

*bloque: unidad mínima de transferencia del SO, agrupación de sectores.

As contiguo

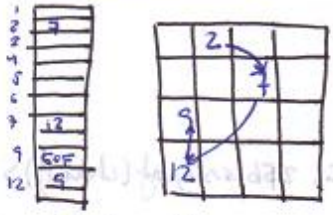
→ costoso si el archivo crece

As. encadenado



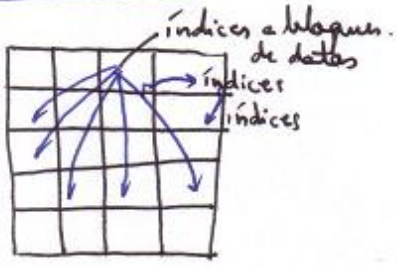
→ acceso aleatorio pésimo

As. encad tabla

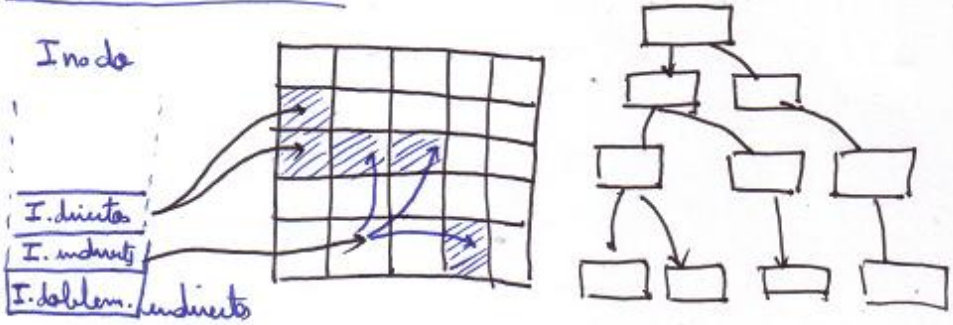


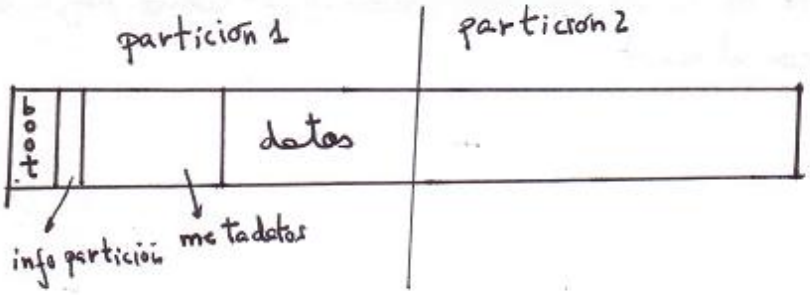
↑ esta tabla se carga en memoria, mejora los accesos aleatorios.

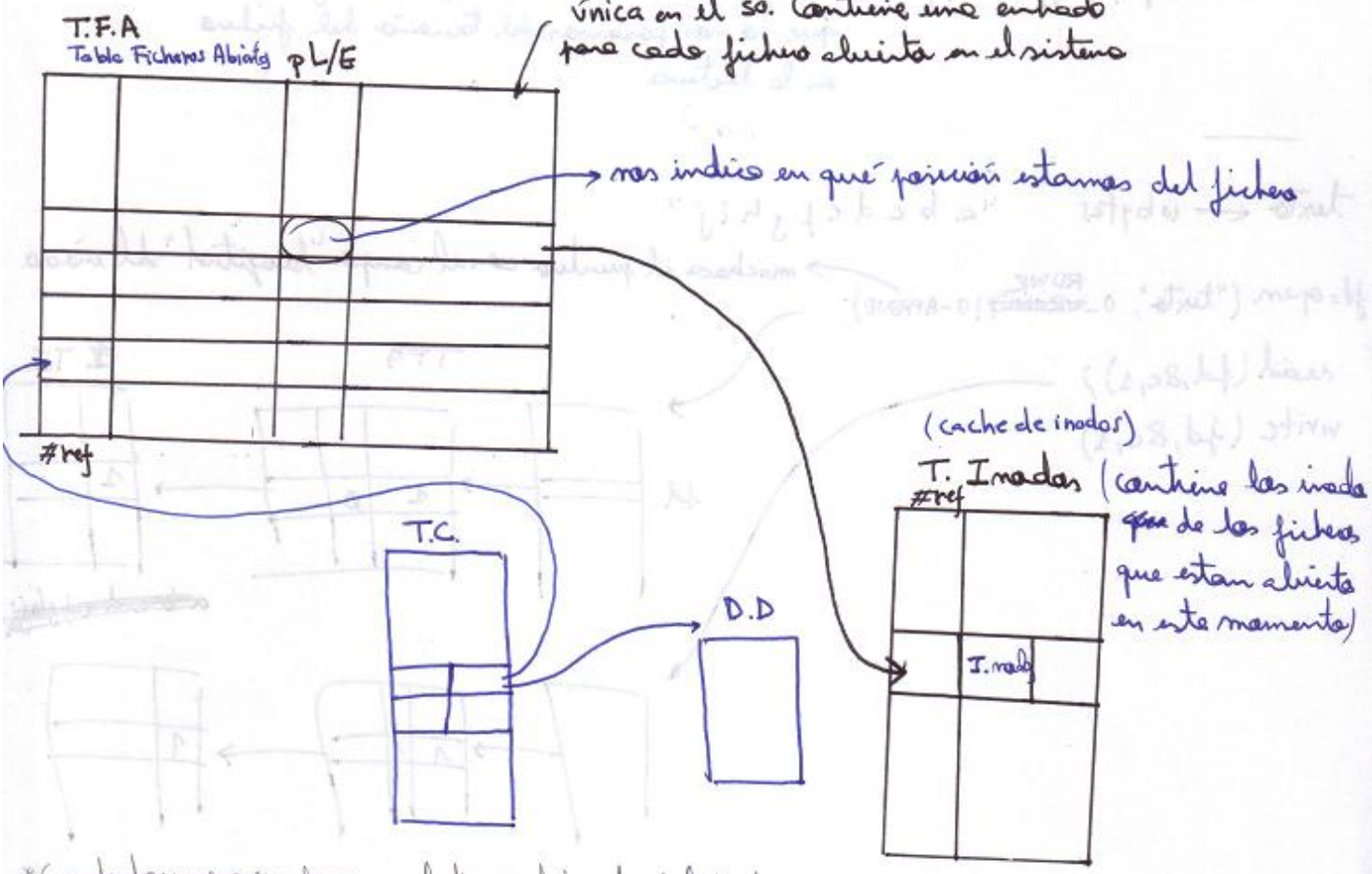
As. Indexado



As. Indexado multinivel







*Cuando hacemos open, lleva en el disco el inodo del fichero (mayor ; menor), y lo mete en la tabla de inodos, apunta al DD. Ahora mira en la tabla de canales i TFA, para ver si tenga un canal vacío, y una entrada vacía en TFA.

Coge el primer canal libre apunta al DD, y la meta en TFA, aumentando el #ref, y pone el pUE a 0. Finalmente ponemos la referencia a la tabla de Inodos en TFA, al Inodo que hace referencia al fichero abierto, incrementa el #ref.

El open siempre coge una nueva entrada en la tabla de Ficheros abiertos, pero no de Inodos.

El dup2 sí que aumentamos el nº de referencias de la TFA.

El write incrementa el p1/E tanto ~~vez~~ ~~posición cont~~ byte como haya conseguido escribir, la misma pauta con el read.

Ej.

texto ← 100 bytes

---

fd = open ("texto", O_RDONLY);

?. ← read (fd, buffer, 200);

el gestor de E/S se encarga de asegurar que no nos pasemos del tamaño del fichero en la lectura.

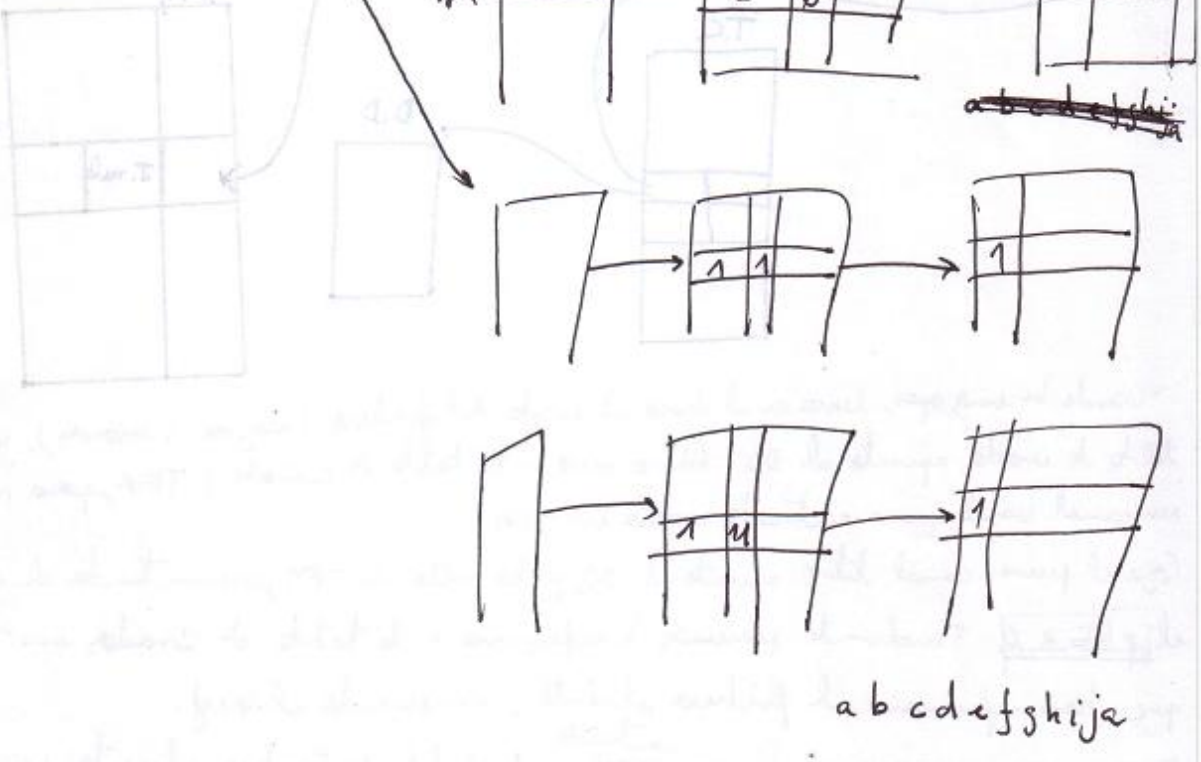
---

texto ← 10 bytes "a b c d e f g h i j"

fd = open ("texto", O_~~RDONLY~~ RDWR | O_APPEND); → mueve el puntero con el campo "longitud" del inodo

read (fd, &c, 1);

write (fd, &c, 1)



Llamada al sistema (cont)

```
int lseek(int fd, int whence, size_t offset)
```

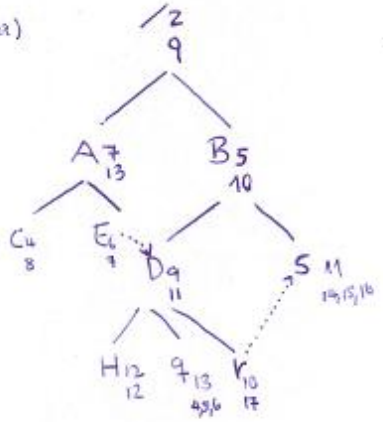
- SEEK_SET pL/E = offset
- SEEK_CUR pL/E += offset
- SEEK_END pL/E = length + offset

offset > fichero, hace más grande el fichero.
-offset > fichero, devuelve error, y la deja donde está.

lseek devuelve la nueva posición del pL/E si sale bien.
Dependiendo del SO, en caso de error devuelve -1 o la posición actual -1.

Ejercicio

14. a)



Referencias: Para el inodo 2, raíz, se mira en los bloques de datos cuantas veces aparece el inodo en las parejas nombre inodo.

2-4	9-3
4-2	10-1
5-3	11-1
6-1	12-2
7-3	13-1

b) /A/E/r /B/D/r

I2
B9 ← bloque de datos 9
I7
B13
I6 } softlink, substituye /A/E por /B/D, y vuelve a empezar
B7
I2
B9
I5
B10
I9
B11 ← otro link! Tenga que acceder a /B/S
I10
B17
~~I2~~
B9
I5
B10
I11

Tenemos I2 en memoria, por tanto no la contamos como acceso, por lo tanto 16 accesos.

c) Primero hay que mirar el número de entradas de mi fat, miramos a que bloques accedemos de 4 – 17.

después del 4 voy al 5, después 6, y el 6 no tiene siguiente

4	5	6	7	8	9	10	11	12	13	14	15	16	17
5	6	eof	eof	eof	eof	eof	eof	eof	eof	15	16	eof	eof

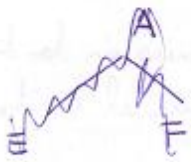
B8	B9	B10	B11	B12	B13
• 8	• 9	• 10	• 11	• 12	• 13
•• 13	•• 9	•• 9	•• 10	•• 11	•• 9
	A 13	D 11	H 12		< 8
	B 10	S 14	9 4		E 7
			r 17		

Por tanto los accesos (b) serían en FAT:

B9 B10 B16
B13 B11
B7 B17
B9 B9

↓ /B10 ✓ /B15

6.



12.

I-nodes		2	4	5	6	7	9
	Tipo	dir	link	dir	dat	link	dat
	Índices	9	10	11	4 12 6 5	8	13 7

Bloques

8: /B

9:

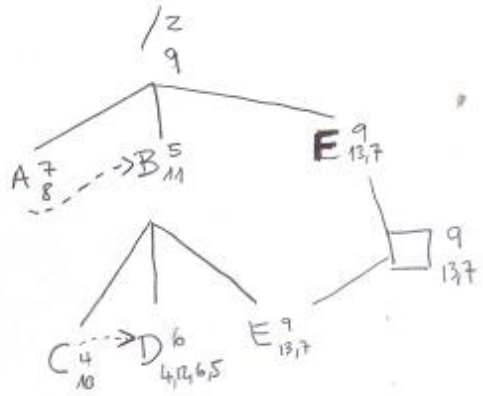
.	2
..	2
A	7
B	5
E	9

10: /B/D

11:

.	5
..	2
C	4
D	6
E	9

1) a)



b) /A/C

I2 - 1
B9 2
I7 } soft link, substituye /A/C por /A/B
B8 3
I2
B9 4
I5
B11 5
I4 } soft link /A/B/C /A/B/D
B10 6
I2 ? Hay permiso?
B9 7
I5
B11 8
I6
B6 9

buffer cache

I2	I3	I4	I5	I6	I7
0	1	2	3	4	5

9 accesos?

2) c)

4	5	6	7	8	9	10	11	12	13
12	~~raf~~	5	raf	raf	raf	raf	raf	6	7

B8	B9	B10	B11
B11	. 9	D4	. 11
	.. 9		.. 9
	A 8		C 10
	B 11		D 4
	F 13		E 13

~~B11~~	~~B9~~
B9	1
B8	
B11	2
B10	3
B4	4

4 accesos?

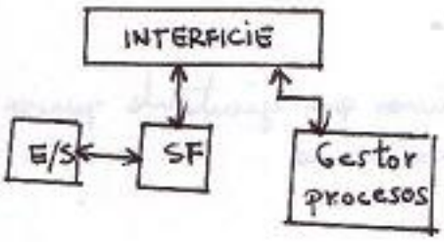
1 2 3
4 5 6
7 8 9
10 11 12
13 14 15

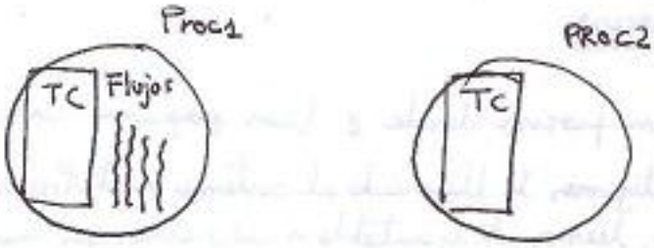
~~B9 B10 B11 B6 B4~~

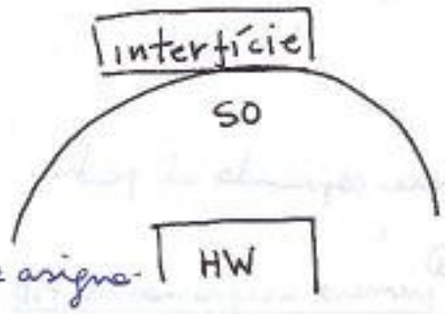
A	8	10

Cache

Gestor de procesos



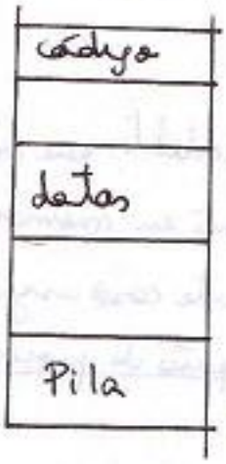




* Un proceso es la unidad mínima de asignación de recursos.

* Los flujos son la unidad mínima de planificación. (Threads)

Los flujos son quienes hacen las llamadas al sistema. Piden el recurso, pero se lo asignan al proceso.



IMPORTANTE: Los procesos no comparten flujos
- No comparten canales
- No COMPARTEN MEMORIA

Para un sistema op. un proceso no es sino que es un PCB, (estructura que identifica y caracteriza un proceso, Process Control bloc).

El campo más importante del PCB es el PID → PROCES IDENTIFIER

Hay otros campos como: Usuario propietario, Datos memoria, Puntero TC, Estado del proceso, Contexto del proceso (PC, registros, puntero pila, ...), Estadísticas sobre el proceso, datos de planificación. También tenemos el PPID, parent PID, el PID del padre.

El gestor de procesos se encarga de crear un proceso, hacerlo crecer, reproducirlo, matarlo y hacerlo desaparecer.

- Crear un proceso

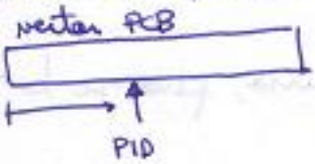
- Crear un proceso desde ø (con programa cargador)

Utilizamos la llamada al sistema CreateProcess. Le decimos que ejecutable queremos ejecutar, leemos el ejecutable y cargamos en memoria lo necesario.

- Copiar padre

En Unix se hace copiando al padre.

Para crear un proceso, ① primero asignamos el PID. El SO tiene un vector de PCB, que recorremos buscando la primera posición libre, para que el PID corresponda a esa posición. El PID es la posición dentro del vector de PCB, del PCB de mi proceso.



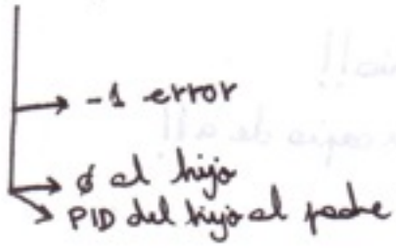
Ahora se dispone de una variable next_PID, que apunta al siguiente libre.
En Windows el PID es una dirección de memoria.

Lo siguiente es ② asignar memoria, en Windows leemos la cabecera del ejecutable que nos dirá cuanto espacio necesita para código, datos y pila y los pondrá en memoria. En linux, copiando al padre, tengo una jerarquía de procesos. En este caso asignar memoria es coger al padre y copiarla tal cual. Creamos una jerarquía de procesos. Dado un proceso padre, ya se cuales son sus hijas y viceversa.

Cuando un proceso copie al padre, copio memoria, pero también replico la tabla de canales, lo cual lleva a modificar la TFA, TI...

Lo siguiente es ③ asignar el PCB mediante el PID, lo inicializamos, y lo encolamos.

int fork() // crea un proceso

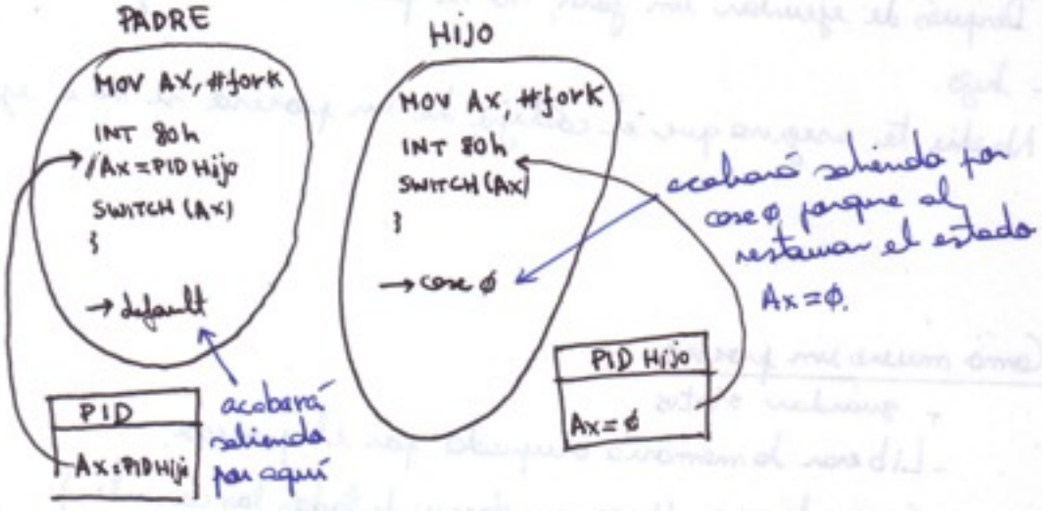


Después de llamar fork(), tenemos dos procesos. El padre, que es quien ha llamado a fork(), y el hijo, que es el nuevo proceso.

Ej: switch (fork())
} case -1 /* error */
case 0 /* hijo */
default: /* padre */
{

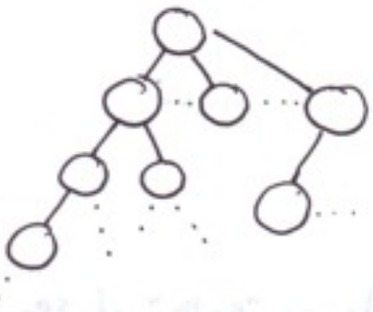
código propio del hijo

código propio de padre



Ej: while(fork())

while(TRUE) fork();



Ej: SWITCH (fork())
} case 0
printf("Soy el hijo\n");
break;
default
printf("Soy el padre\n");
}

Soy el hijo
Soy el padre

importante el break

Ej:
```
int a=0;
switch (forkU)
{
    case 0: a++
            printf("%d\n", a)
            break
    case 1: a++;
            printf("%d\n", a);
}
```

1 No comparten memoria!!
1 Cada proceso tiene su copia de a!!

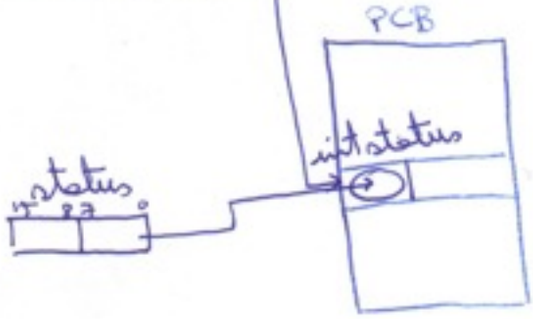
Después de ejecutar un fork, no se quien se va a ejecutar primero, si el padre o el hijo.
Nadie te asegura que el código de un proceso se va a ejecutar ininterrumpidamente.

- Como muere un proceso
  - guardar status
  - Liberar la memoria ocupada por el proceso.
  - Cerrar la T.C. (hacer un close(); de todas las canales)
  - Poner el estado de finalización en el PCB → Si tengo hijos → hijos PID:1

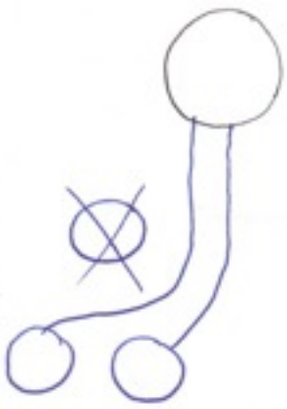
* mi PCB sigue asignado !!!

int exit (int status) ← se guarda en el PCB, para que el padre sepa porque he muerto (valor entre 0 y 255)

NO DEVUELVE NADA



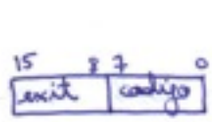
Todos los procesos tienen padre, ~~el~~ cuando se "Bota" el so, se inicia el proceso INIT con PID=1, que se encarga de que todos los hijos tengan padre.



Un padre tiene que asegurarse de que sus hijos no se quedan huérfanos, por tanto cuando un padre muere, el INIT adopta a sus hijos

Cuando el padre lee el estado, es cuando se libera el PCB

Como se libera el PCB?



status = status >> 8

int wait (int * status) recoge el estado de finalización de los hijos

- → -1 si no tiene hijos
- → si tiene hijos devuelve el pid del hijo muerto
  - → si ninguno muerto → se bloquea el proceso hasta que alguno hijo muere.

Si tiene más de un hijo, recoge el primero que encuentre. (el wait solo sirve para un hijo)

int wait pid (int pid, int * status, 0) espero a que muera el hijo con el pid en concreto.

- → -1 si ese hijo no existe
- → el pid

Un proceso puede mutar, ejecuta código diferente del padre.

int exec_ _ ← mutar un proceso.
l p
m -

- Tiran código
  datos
  pila
- cargan ejecutable

mantengo el PCB, por tanto mantengo el PID.
la tabla de canales
la programación de signals.

- → -1 error
- → no devuelve nada si va bien

Ejemplo:

> ls

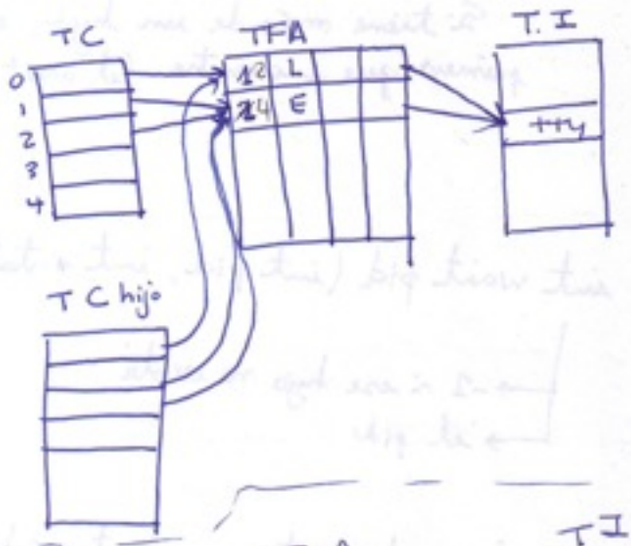
```
switch (fork())
{ case 0
    execlp ("ls", "ls", (char*)0)
    → /* si ejecuto esto ha habido un error */
      exit(0); ——→ lo ponemos siempre por si a caso.
  default: wait(&status)
```

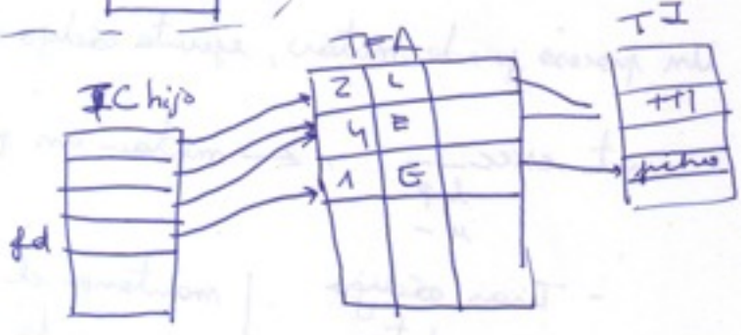
vayamos a usar la variable PATH para buscar "ls"
nombre de ejecutable
el argv[0] que tiene que recibir el ejecutable.
no quiero poner nada más

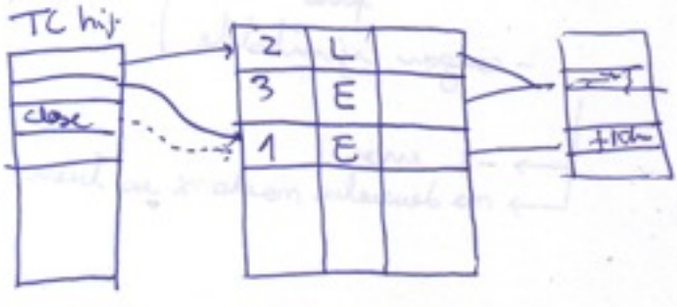
↑ si ejecutamos ls &, no se pone esa

> ls > fichero

```
switch (fork())
{ case 0
    fd = open("fichero", O_WRONLY | O_CREAT | O_TRUNC, 0600);
    dup2 (fd, 1);
    close (fd)
    execlp ("ls", "ls", (char*)0)
```





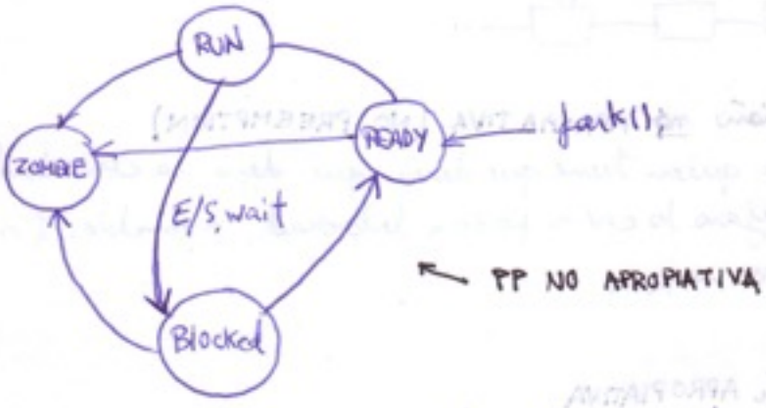


> cat > fichero < fich

```
switch (fork())
{   case0
        fd= open ("fichero", O_WRONLY | O_CREAT | O_TRUNC, 0666)
        dup2(fd,1);
        close (fd).
        fd= open ("fich", O_RDONLY);
        dup2(fd,0);
        close (fd);
        execlp ("cat","cat", 0);
        exit (0);
    default
        wait (&status);
```

Ciclo de vida

Un proceso que está en ready está preparado para seguir ejecutándose o para ejecutarse.
Los procesos pasan del estado ready a run, cuando se ejecutan.
En el momento de hacer un exit cuando el PCB sigue existiendo, está en el estado de zombie. Se puede pasar de cualquier estado a zombie.



Cuando estamos en un run y hacemos una llamada al sistema, pasamos al estado Blocked. Una vez bloqueada, necesitamos pasar por el estado de ready para que "alguien" me seleccione y pueda seguir ejecutándome.

Pasar un proceso de RUN a blocked ó zombie se llama cambio de contexto
y otro de ready a RUN

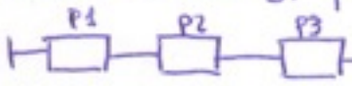
Para hacer un <u>cambio de contexto</u>:

- Guardar estado (los registros de la que el PCB)
- Cambiar estado RUN → BLOCKED
- Encolar PCB
- seleccionar otro proceso (en ready)
- cambio el estado → RUN
- preparo estructuras
- restauro el contexto del nuevo proceso

11

<u>Planificador</u> → proceso especial del SO que se encarga de multiplexar los procesos en el tiempo

- a largo plazo
- a medio plazo (se encarga de ver como va la memoria)
- <u>a corto plazo</u> → es el que define el ciclo de vida, y decide cuando se va a realizar un cambio de contexto (Política de planificación) y tiene el algoritmo de planificación, que decide quien va a ser el siguiente READY que va a pasar a RUN.

Dispone de una cola de ready, que es donde se guardan los PCB's en estado de READY. 

Política de planificación <u>no</u> APROPIATIVA (NO PREEMPTION)
Es el mismo proceso quien tiene que decir que deja la CPU, ~~nadie~~ nadie le puede apropiar la CPU solo dejara la CPU si pasa a bloqued, o zombie. (si se suicida o la mata el mismo SO) READY → RUN

Política de planificación APROPIATIVA

- DIFERIDA READY ↔ RUN

- INMEDIATA

READY ↔ RUN
RUN ↔ BLOCKED

ALGORITMO DE PLANIFICACIÓN

- FIFO → FCFS (NO APROPIATIVA)
- PRIORIDADES (ordenadas de mayor a menor prioridad)

starvation / inanición } para solucionar se usan las prioridades dinámicas.
Aging

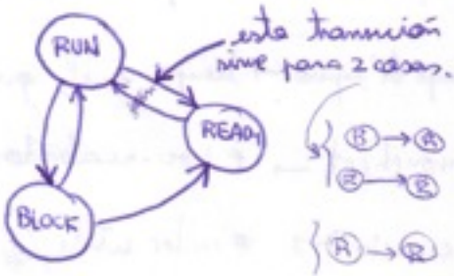
- ROUND ROBIN

·apropiación diferido
·a cada proceso se le asigna un quantum (tiempo seguido que puede estar un proceso en run)

· En caso de empate se coge el que más tiempo lleve en la cola.

- ROUND ROBIN + PRIORIDADES

Cola de ready ordenada por prioridades y un quantum para cada proceso.
Solo se puede usar con PP inmediatas.



- Colas multinivel

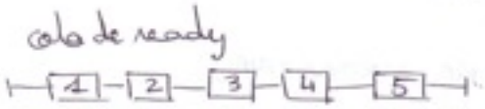
. Tenemos grupos de colas de ready. Dentro tenemos diferentes colas de ready que utilizan algoritmos de planificación diferentes. (cada grupo tiene una política diferente)
Los procesos no saltan de una cola a otra.

- Colas multinivel realimentadas

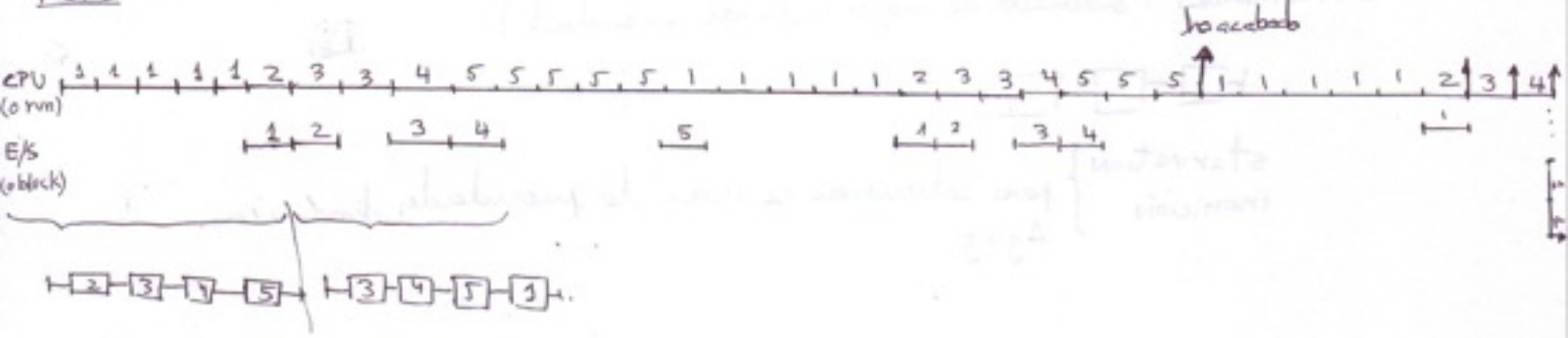
. Los procesos pueden saltar entre colas.

Ejercicios

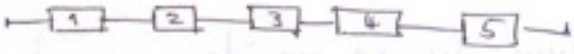
1.

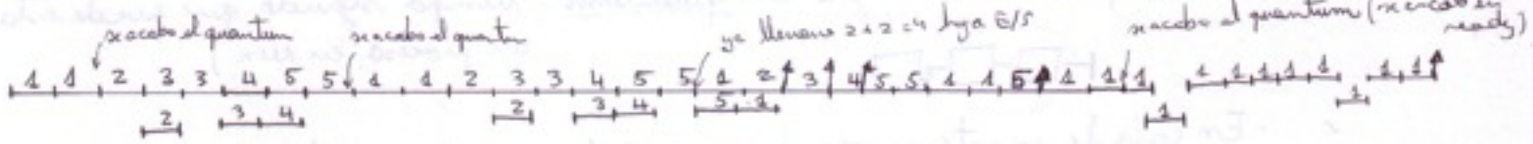


FCFS

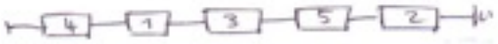


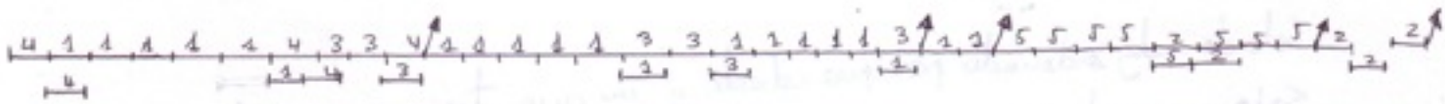
ROUND ROBIN





PRIORIDADES NO APROPIATIVAS





b) Tiempo de retorno → Tiempo total de un proceso en el sistema (el 4 está 10)

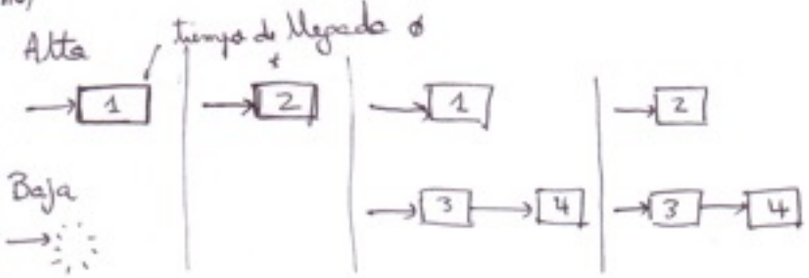
c) Tiempo de espera → tiempo en el que está encolado en la cola de ready. (El 4 está 5 (primer caso))

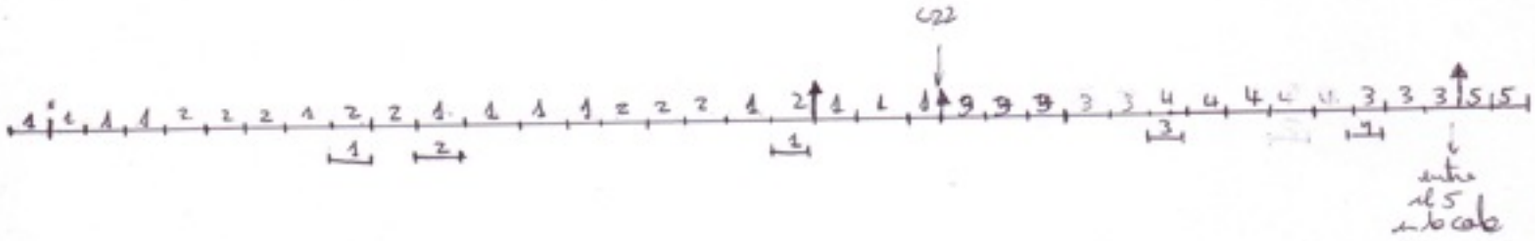
d) Throughput → # proc. acabados / total de ciclos

Eficiencia → # ciclos utils / # ciclos (ciclo útil = ciclo en los que hay un proceso ejecutándose) (que no sea el nulo)

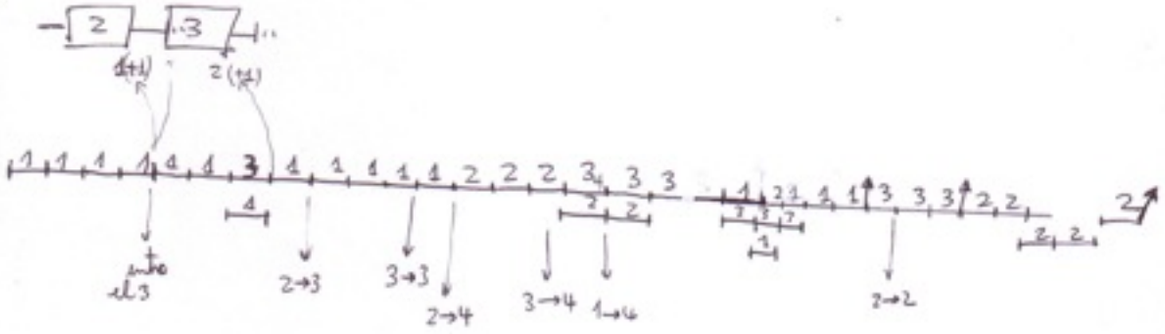
Tiempo de respuesta → Tiempo hasta la primera E/S.

10)


Alta
tiempo de llegada ∅
Baja


entre el 5 y el 6

12)

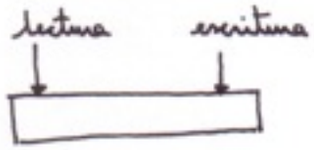

entro el 3
2→3
3→3
2→4
3→4
1→4
2→2

Comunicación entre procesos

> ls -la | wc -l
↑
pipe

Los pipes son buffers. Existe un tipo de pipe llamado named pipe, que en vez de ser un buffer en memoria, es un fichero.

Pipe (para procesos emparentados) Named pipe (para todos)



CREACIÓN

named pipe

"mi-pipe"
↓
- crear (mknod(char * name, S_IFIFO | 0666);)
- abrir open ("mi-pipe", O_RDONLY) ← el open de una named pipe es bloqueante hasta que otro proceso lo abra de escritura.

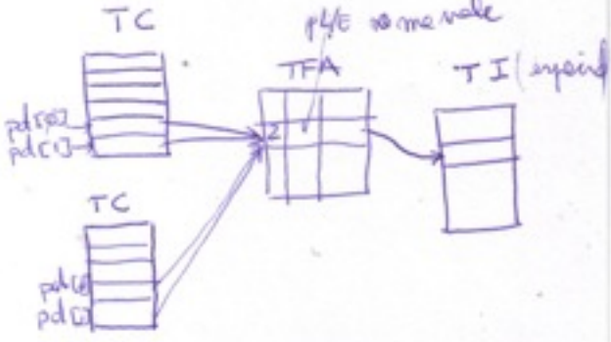
Ejemplo de sincronización entre los procesos:

```
P1                                  P3
mknod("mi_pipe...)                  mknod("mi_pipe...)
open("mi_pipe", O_RDONLY);          open("mi-pipe", O_WRONLY|
```

pipe sin nombre

- crear int pipe(int pd[2]) Devuelve 2 canales de la tabla de canales.
  ↳ pd[0] ← lectura
  ↳ pd[1] ← escritura

Para que otro proceso pueda leer de este pipe tengo que hacer que sea capaz de leer de los mismos canales. Por tanto deberemos hacer un fork.



lectura de pipes read read

El read de una pipe vacía con escritores es bloqueante

" " " " sin escritores → ∅

escritura de pipes write

El write de una pipe llena con lectores es bloqueante

" " " sin lectores → -1

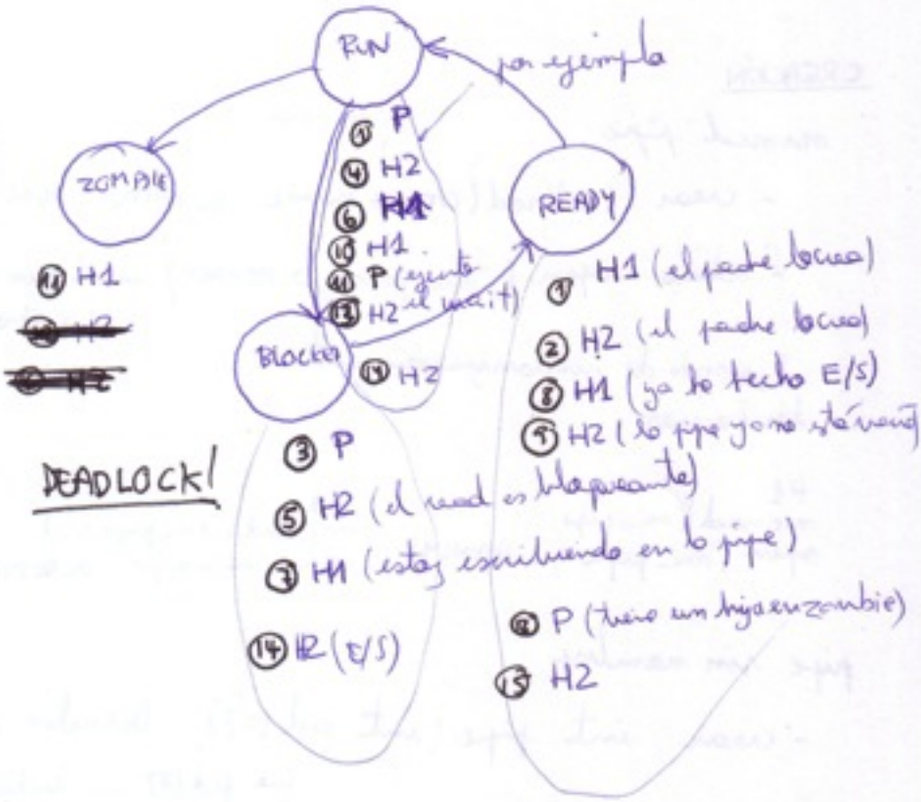
errno = EPIPE

sig PIPE

No podemos utilizar lseek

> ls -la | wc -l

```
int pd[2];
int s;
pipe(pd)
switch (fork())
{ case ∅  dup2(dp[1],1);
          execlp("ls","ls","-la",∅)
          exit(∅);
{
switch (fork())
{ case ∅  dup2(pd[0],∅);
          execlp("wc","wc","-l",∅);
          exit(∅);
}
wait(&s);
wait(&s);
exit(∅);
```

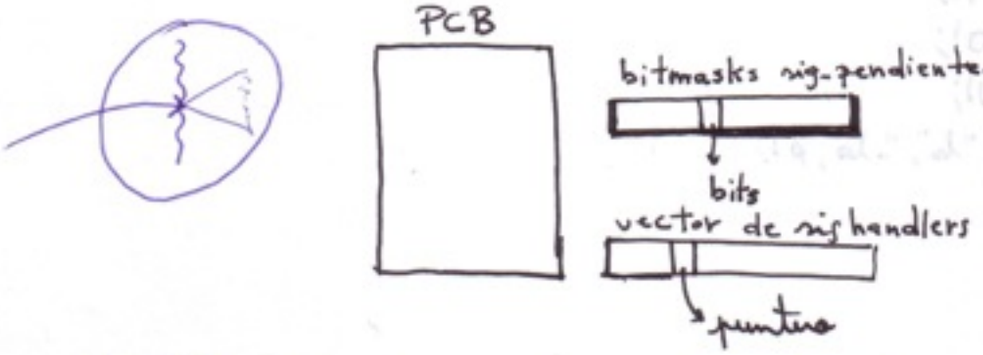


> ls -la | wc -l

```
int main()
{ int pd[2];
  pipe(pd);
  switch (fork())
  { case 0 : dup2(pd[1],1);
             close(pd[0]);
             close(pd[1]);
             execlp("ls","ls","-la",0);
            exit(0);
  {
  close(pd[1]);
  switch(fork())
  { case 0: dup2(pd[0], 0)
          close(pd[0]);
          execlp("wc","wc","-l",0);
          exit(0);
  {
   close pd[0]
   wait(&s);
   wait(&s);
  exit(0);
{
```

Signals

Es un evento asíncrono que se envía a un proceso. Este reacciona como si fuese una interrupción. En el punto en el que nos llega el signal saltamos a ejecutar la rutina de atención al signal.



La máscara de bits nos sirve para saber si de una clase de signal quedo alguno por servir.
El segundo vector me sirve para indicar que tengo que ejecutar cuando me llega un signal en concreto.

int kill (int pid, int signum)

- SIGKILL → > kill -9 pid
- SIGUSER1
- SIGUSER2

void (*signal) (int signum, void (*sighandler) (int))

- ↓ → -1
  - 0XFFFF..F
  - SIG_ERR
- puntero a función
- SIG_IGN
- SIG_DFT

Cuando un proceso pasa de ready a run es cuando mira los signals pendientes. Básicamente cuando pasamos el proceso a RUN, es cuando recuperamos la mascara de bits. Este es el tratamiento diferido. Si se recibe un signal que fuerza la finalización de un proceso (SIGKILL, SIGTERM, SIGSEGV), se hará un tratamiento inmediato y esté en el estado que sea, pasará a ZOMBIE.
En estado de READY si recibo un signal me lo apunto, si estoy en run, lo ejecuta inmediatamente, y si estoy en BLOCKED hago fallar la llamada al sistema, devuelva -1, ponga EINTR en ERRNO, y pase a ejecutar el proceso de nuevo.
Si esté en ZOMBIE, se pueden enviar SIGNALS pero no se tratan.

Cuando haga fork(), no hereda las signals pendientes pero sí la reprogramación.
Con exec..., las signals pendientes se mantienen pero la reprogramación no.
La reprogramación de un signal solo vale para una vez

int pause() bloquea el proceso hasta que reciba cualquier signal
→ -1 (siempre, clara!)

int alarm(int segs)
→ SIGALRM
→ -1
→ el numero de segundos que faltaban para que saltase la reprogramación anterior del temporizador

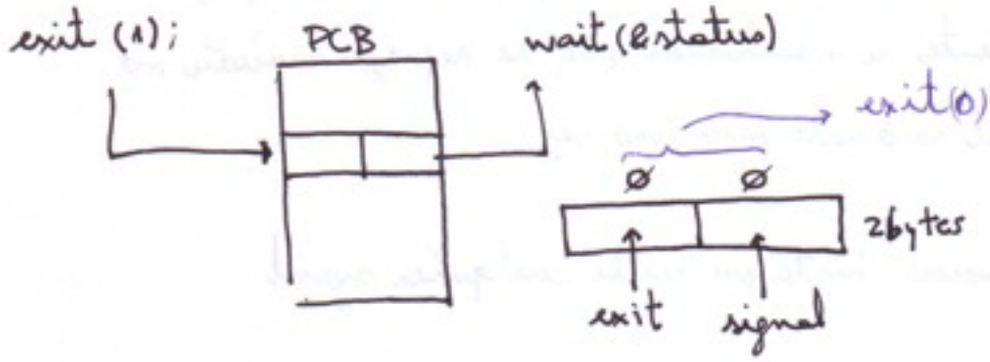
- Cada vez que reciba el signal SIGUSER1, escribe "hola", ~~si no llega en los siguientes no llega~~,

```
int main()
{
  signal(SIGUSR1, hola);
  while(1)
  { pause();
  }
}

void hola(int s)   → SIGUSER1
{ signal(SIGUSR1, hola);
  sprintf(b,"Hola\n");
  while(write(1,b,strlen(b))!=-1) && (errno==ENTR));
}
```

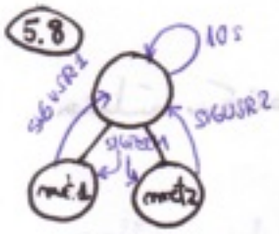
- Si en 10 segundos no llega,

```
int s1=0;   int hola()
            { signal(SIGUSR, hola);
              sprintf(b,"Hola\n");
              s1=0;
              while((write(1,b,strlen(b)) == -1) && (errno == ENTR));

void hola(int s)
{ signal(SIGALRM, temp);
  alarm(1);
  s1++;
  if (s1==10) exit(0);
}
```

```
int main()
{ signal(SIGUSR1
  signal(...
  alarm(1)
  while
```



SIGCHLD → el sistema operativo se lo manda al padre de un proceso cuando muere un hijo suyo. (el padre tiene redefinido el servicio a ese señal para hacer un wait (& status)




```
int pid1, pid2;

int main (int argc, char * argv)
{
  switch ( pid1 = fork() )
  { case 0 → { signal (SIGUSR1, SIG_DFT)
               signal (SIGUSR2, SIG_DFT)
    fd = open (argv[1], O_RDONLY);
    dup2 (fd, 0);
    close (fd);
    execlp ("metodo 1", "metodo 1", 0);
    exit(1);
  {
  switch ( pid2 = fork() )
  { case 0 → {
      fd = open (argv[1], O_RDONLY);
      dup2 ( fd, 0);
      close ( fd);
      execlp ("metodo 2", "metodo 2", 0);
      exit (1);
  }
```

```
signal (SIGUSR1, susr1);
signal (SIGUSR2, susr2);

void susr1(int s)
{ kill (pid2, SIGTERM);
  wait (& status);
  wait (& status);
  sprintf (buffer, "metodo 1 \n");
  write (1, buffer, strlen (buffer));
  exit (0);
{
void sal (int s)
{ kill (pid1, SIGTERM);
  kill (pid2, SIGTERM);
  wait (& status);
  wait (& status);
  sprintf ( buffer, "____");
  write (1, buffer; strlen ( buffer));
  exit(0);
}
```

5.9

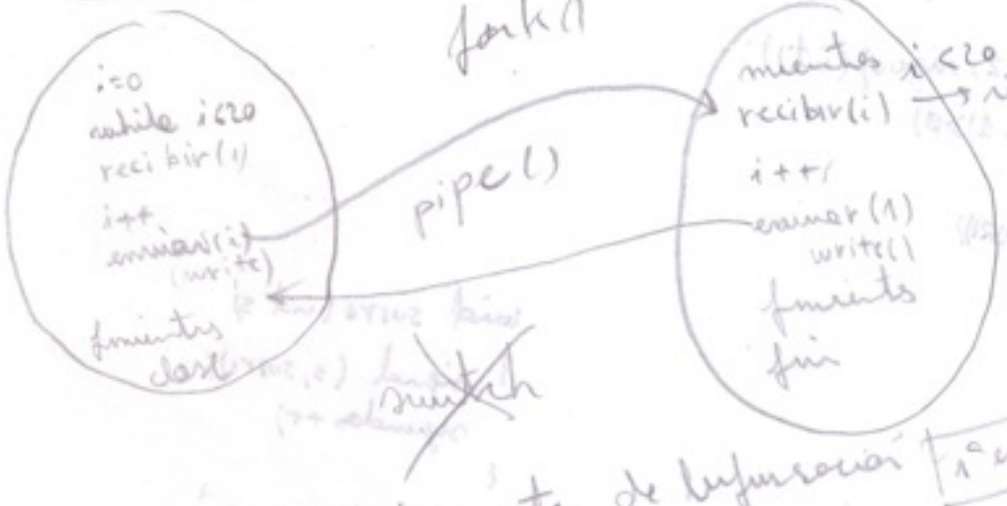
```
int pid1, pid2;
int segundos=1;
int main()
{ int pd[2];
  pipe(pd);
  switch(pids= fork())
  { case 0: signal(SIGUSR1, SIG-IGN);
            close(pd[1])
            read(pd[0], &pid2, sizeof(int));
            while(read(0, &c, 1)>0)
            { if (c=='a')
                kill(pid2,SIGUSR1);
            }
            exit(0);

    {
  switch(pid2= fork())
  { case 0: close(pd[0]);
            pid2= getpid();
            write(pd[1], &pid2, sizeof(int));
            signal(SIGUSR1, susr1);
            signal (SIGALARM, sal);
            alarm(segundos);
            while(1) pause();
            close
    {
    Ctrl+C
    close(pd[0]);
    close(pd[1]);
    wait(&status)
    kill(pid2,SIGKILL)
    wait(&status);
    exit(0);

    {
```

```
void susr1(int s)
{ signal(s, susr1);
  segundos++;
{

void sal(int s)
{ signal(SIGALARM, sal);
  kill(pid1, SIGUSR1);
  alarm(segundos);
{
```

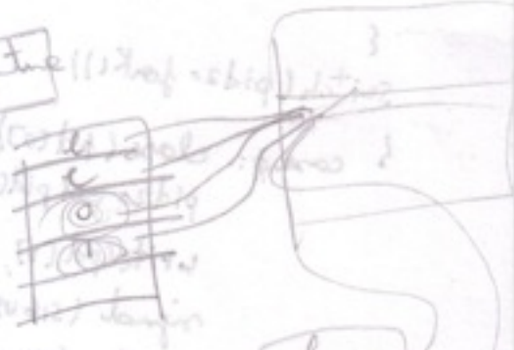
5.10

↑1
ls
↑2
cat
cat

e7870680
e6146575

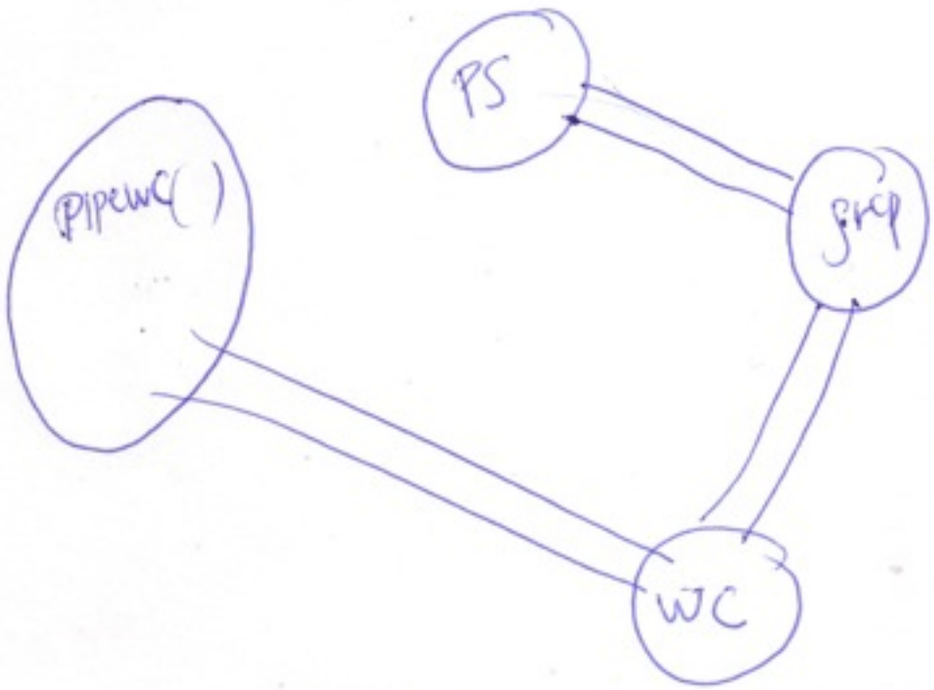


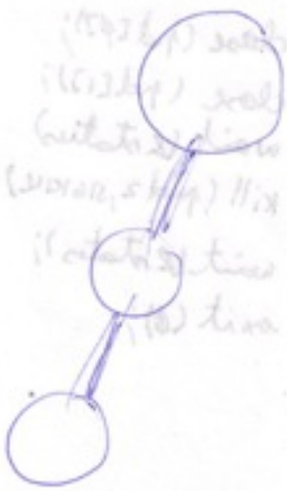
Única puta de bifurcación



Named pips!

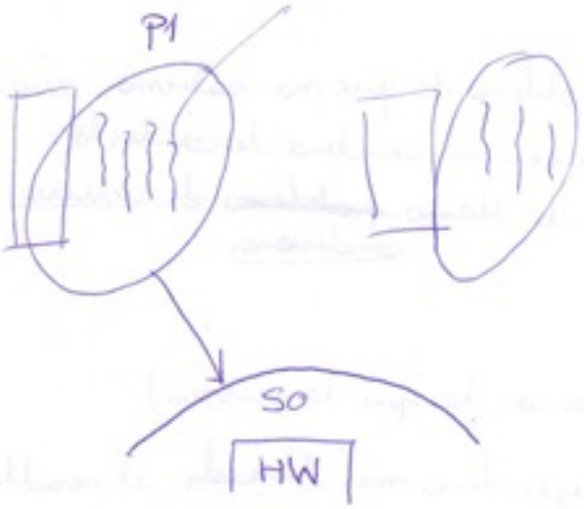
chats (padre anuncia, hija anuncia)





Threads

~~Memoria~~ THREAD (unidad mínima de planificación)



El planificador selecciona el proceso de la cola de ready, y otro planificador elige un thread dentro del proceso.

Para identificar los threads tenemos el TCB

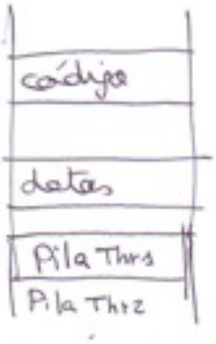
TCB

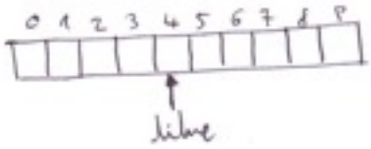
- **TID** Threads de diferentes procesos pueden tener el misma TID
  Para identificar un thread necesitamos PID.TID
- **PC** (contador)
- **SP** (puntero pila)
- registros procesador

Crear un thread es simplemente crear esa estructura, y es mucho más rápido que crear un proceso. Hacer el cambio de contexto también es muy sencillo y matar un thread es eliminar esa estructura.

LOS PROCESOS NO COMPARTEN MEMORIA, LOS THREAD SÍ. (Y TODOS LOS RECURSOS)

COMPARTEN	NO COMPARTEN
- Memoria	- Pila
- TC	- TID
- PCB	- PC + regs
- Código	- errno: cada uno tiene que saber los errores de sus llamadas al sistema.
- Datos	






```
int vector[N];
int libre = 0;
```

```
th1()
{ int pos;
  pos = libre;
  vector[pos] = 1;
  libre++;
}
```

```
th2()
{ int pos;
  pos = libre;
  vector[pos] = 2;
  libre++;
}
```

Este código tiene el problema de que no sabemos cuando el planificador va a hacer un cambio de contexto.
A este tipo de problema se le llama ~~problema~~ condiciones de carrera

Condición de carrera (variable accedidas y modificadas son las que la generan)

Tengo que garantizar que se ejecuten como se ejecuten mis threads, el resultado sea el deseado.

Sección Crítica: Trozo de código donde se accede y modifica una variable compartido

```
Th1()
{
  pos = libre;
  vector[pos] = 1;   } Sección crítica.
  libre++;
}
```

Inicio de sección crítica

Final de sección crítica

Tenemos que garantizar que todas las secciones críticas se ejecuten en exclusión mútua.

Test & Set

Es una función que se ejecuta de forma atómica

```
int t_&_s(int &a)
{
  int tmp = &a;
  *a = 1;
  return tmp;
{
```

ATÓMICA

Ejemplo.

```
int hay_alguien = 0;

th1()
{
  while( t_&_s (& hay_alguien));
  a++;
  hay_alguien = 0;
{
```

```
th2()
{
  while(t_&_s(& hay_alguien));
  a++;
  hay_alguien =0;
{
```

// es uno encuente, por tanto consume muchos recursos.

Semáforos

```
typedef sem_t struct
{ int cont;
  queue_t queue;
{
```

cola de TCB

Bloquean a los threads que pretenden entrar en una zona crítica.

sem_init (sem, n) inicializa un semáforo

sem_wait (sem) marca el inicio de la zona de exclusión mutua

sem_signal (sem) " " fin " " " " " "

Los threads que bloquea los pone en la cola.

```
sem_init
{ sem -> count = n;
{
sem_wait (sem)
{ sem -> count--;
 if (sem -> count < 0)
      bloquear (sem -> queue);{
sem_signal
{ sem -> count++;
 if (sem -> count <=0)
      desbloquear (sem -> queue)
```

Ejemplo

```
sem_t s;
sem_init (s, 1);
```

```
th1()
{ sem_wait (s);
 a++;
 sem_signal (s);
{
```

```
th2()
{ sem_wait (s);
 a++;
 sem_signal (s);
{
```

Valores de inicialización de los semáforos

1 → Exclusión mutua

0 → Sincronizar

>1 → Recursos

~~>1~~

sem_t (s); sem_init (0);

Necesito que T3 nunca se ejecute antes de T1.

th1	th2
	sem_wait (s);
T1	T3 ← espera signal
sem_signal (s);	
T2	T4

---

sem_t (s); sem_init (2);

th1	th2	th3	reserva lo imprescindible para mí
sem_wait(s)			} coger (ptr);
utilizar_ptr			
sem_signal (s);			} soltar (ptr);

Deadlocks con recursos

th1	th2
1 coger (disco)	3 coger (ptr) ← se bloquea hasta que th1 suelte lo imprescindible
2 coger (ptr)	4 coger (disco)
←	
soltar (ptr)	soltar
soltar (disco)	

Si ejecutamos ① → ③ → ④ ... se produce un abrazo mortal

{

Reserva circular

coger (disco) coger (ptr) ↔ coger (ptr) coger (disco) ← solución

---

Hold & Wait

coger (disco) → solución

coger (ptr)

bucle

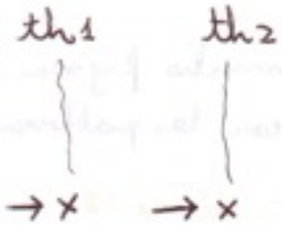
coger (disco) ← coger los recursos de forma atómica

if (!coger (ptr))

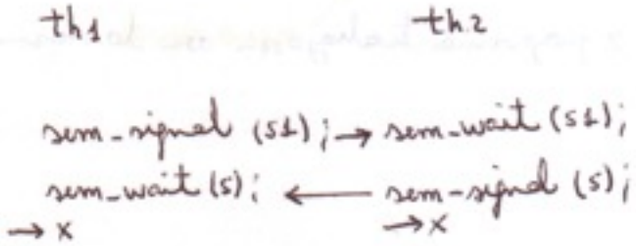
soltar (disco)

Exercici 6.3

Rendez-vous



La tengo que sincronizar en el punto x.



Exercici 6.4

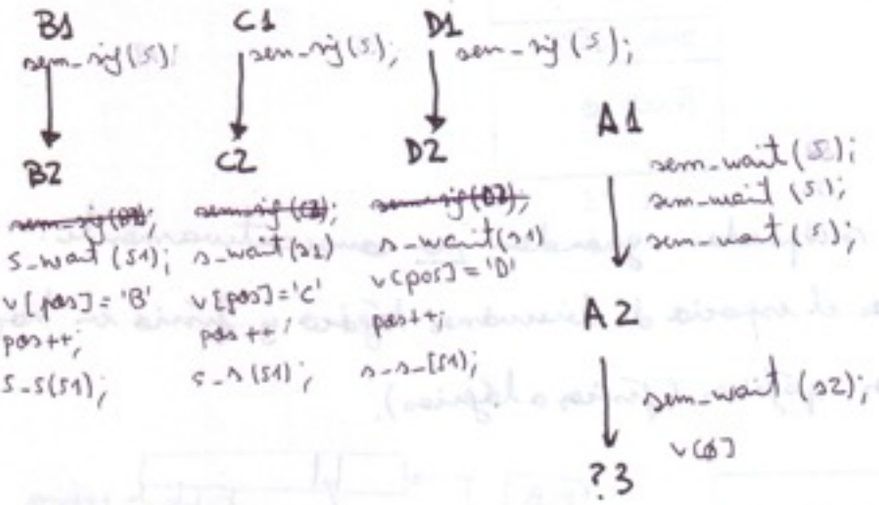
Planificació familiar

```
sem_t s;
sem_init(s,0);
```

```
char v[3]
int pos = 0
```

```
sem_t s1;
sem_init (s1,1);
```

```
sem_t s2;
sem_init(s2, 0);
```



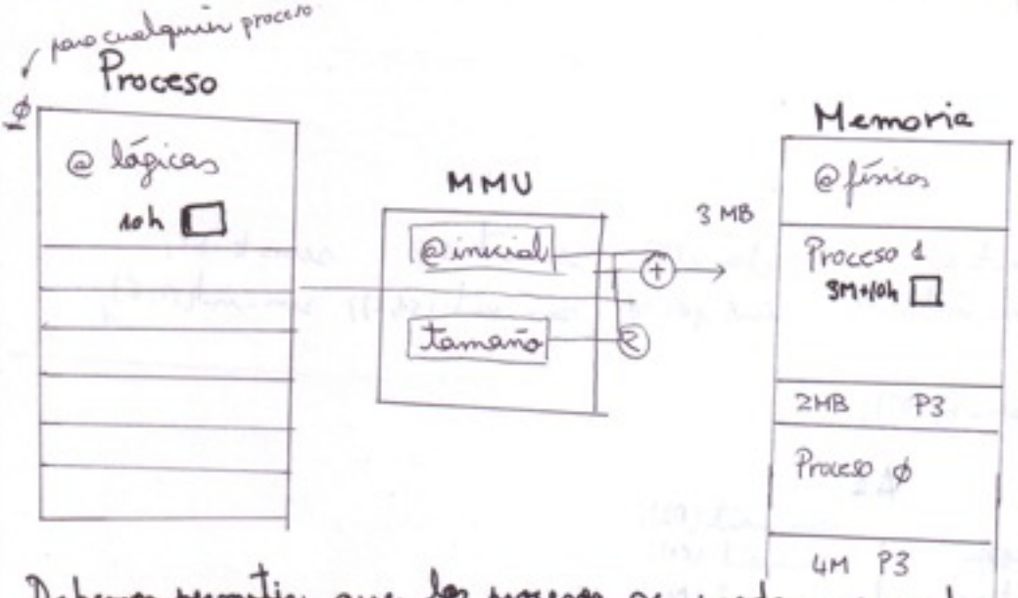
Memoria

Reubicación dinámico de código: Se incorpora un gestor, y cuando detectaba mucha fragmentación, movía un programa en la memoria, con los problemas que conllevaba.

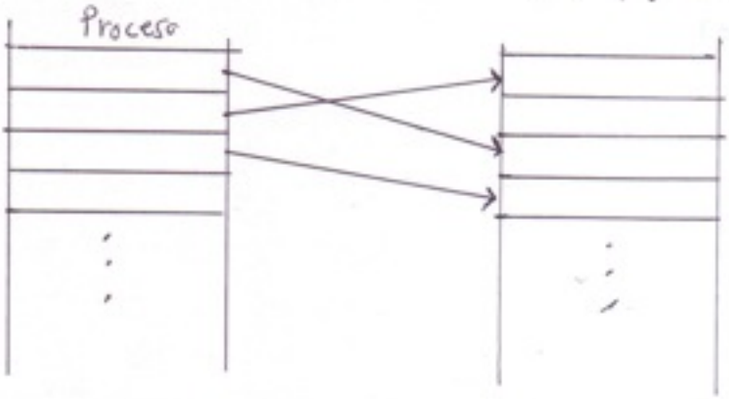
- Era muy lento
- Había direcciones que no se podían desadifian, si estaban en variables.

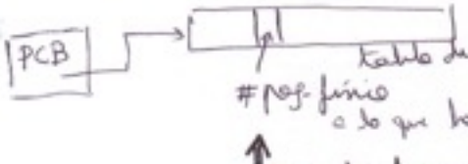
Solapamiento entre programas: Podía pasar que 2 programas trabajasen con la misma zona de memoria.

⇓ Solución



Debemos permitir que los procesos se puedan guardar no consecutivamente. Usaremos las Páginas. Dividiremos el espacio de direcciones lógico y físico en trozos del mismo tamaño que llamaremos páginas (físicas o lógicas).





Tamaño de página lógica = física = 4K = 2^{12}

Las traducciones se tienen que hacer por hardware. Usaremos para ello el TLB.

Si la página lógica no está en el TLB, provoca una interrupción (FALLO DE TLB). El SO lo atiende cambia el modo de privilegios y pasa a ejecutar un proceso llamado paginador. Es el planificador a medio plazo!

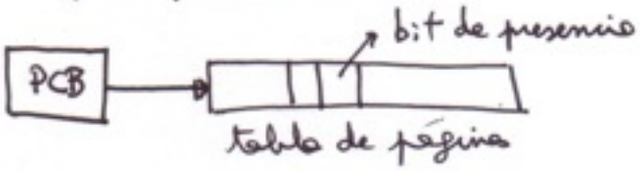
y encuentra la traducción de página lógica a página física

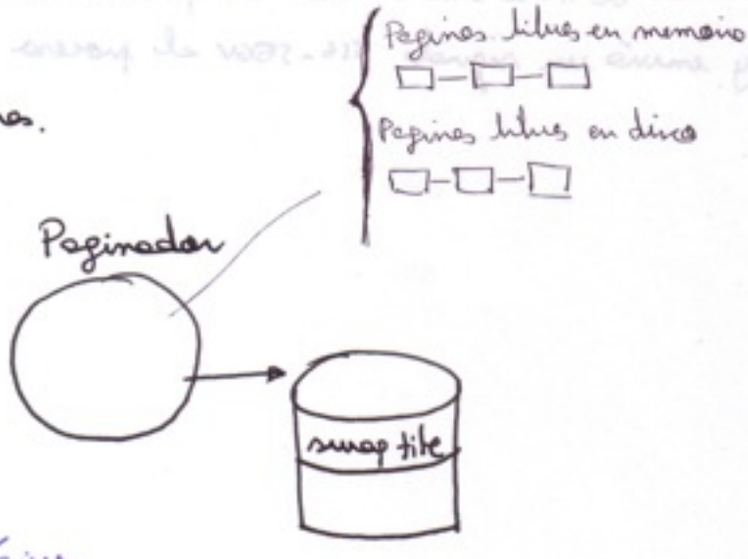
El paginador accede al PCB del proceso para acceder a la tabla de páginas y introduce la nueva entrada en el TLB, y el procesador reejecuta la instrucción que ha provocado el fallo de página.

Memoria Virtual (swap file)

El paginador diferencia tres tipos de páginas.

- Página lógica
- Página física en memoria
- Página física en disco





¿Qué pasa si la página está en disco? FALLO DE PÁGINA

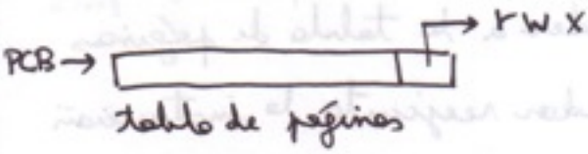
El paginador ve que la página está en disco (bit de presencia)

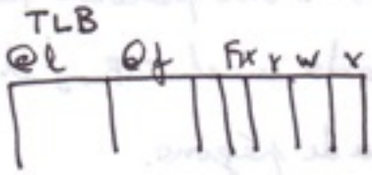
- Busca una página física en memoria que esté libre (el paginador tiene una lista de libres)
- Vuelca la página de disco y la pone en la página libre en memoria.
- Modifica el bit de presencia en la tabla de páginas del proceso, y el identificador de P.F.
- Modifica el TLB
- Reejecuta la instrucción

Podría ser que el mismo ~~fp~~ paginador se causase un fallo de TLB. Dependiendo del SO se hace una cosa o otra.
Se hace que el paginador salte dos veces. Es una opción
Lo que se hace actualmente es que el TLB contenga las entradas necesarias para que el paginador no cause ~~una~~ fallos de TLB.
Para que se pueda hacer esto, se necesita soporte hardware. Se pone en ~~los~~ ~~fp~~ el TLB un nuevo campo, llamado FIX, que nos marca qué entradas podemos modificar, y cuales no (porque son del paginador).

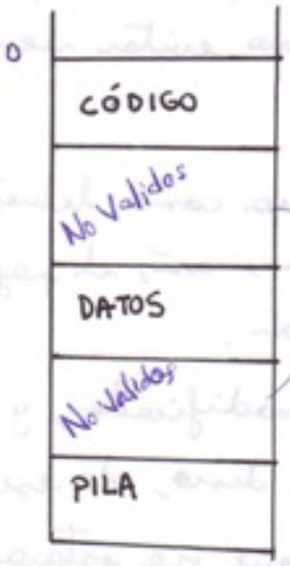
—

Se diferencian 3 tipos de página, que se marcan tanto en el TLB como en la tabla de páginas.

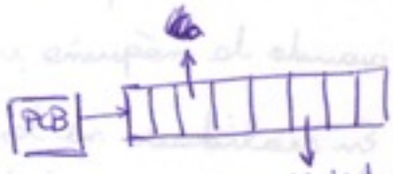




Cuando se hace un acceso "no permitido" a una página, el paginador ~~no~~ salta y envía un signal SIG-SEGV al proceso



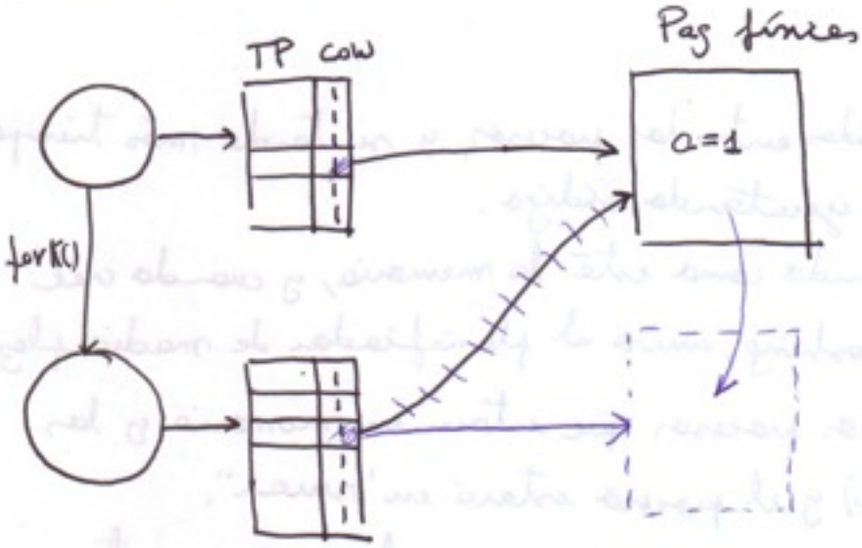
No tienen página física asociada, segmentation fault



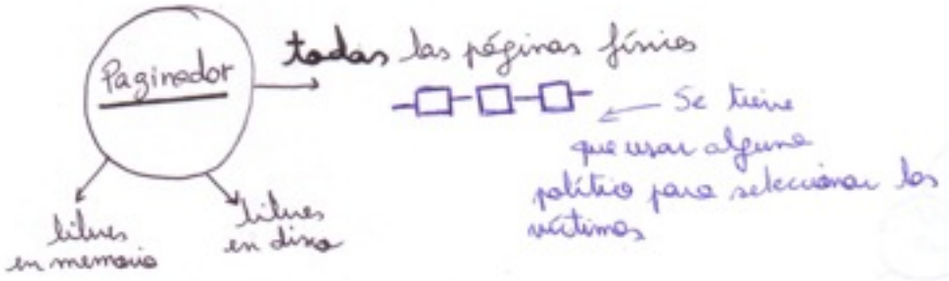
Validez: Este bit nos dice si la página lógica tiene traducción a física

Si se produce lo que sería un SIG_SEGV o la pila, se le da una página física.

Copy on Write



Si un proceso modificase una variable, entonces se duplica la página física y se marca el bit de copy on write a ϕ en ambas TP.



El proceso Page out Daemon mira como están las páginas y las que se usan para los va volcando en disco para tener memoria ~~lucha~~ libre. Para evitar reemplazos de páginas cuando la máquina está muy cargada.

~~También~~ En realidad no las libera, si no que las marca como libres, clasificándolas en libres modificadas, libres sin modificar, así, el paginador tirará siempre que pueda de las libres sin modificar.

De vez en cuando el Daemon va mirando su cola de modificadas, y cuando el sistema no esta muy cargada, las va volcando en disco, de ese modo se iran convirtiendo en libres sin modificar, porque ya estaran en disco las modificaciones.

Trashing

Las páginas estan muy repartidas entre los procesos, y se tarda más tiempo resolviendo fallos de página que ejecutando código.

Solución: El ~~paginador~~ daemon va mirando como está la memoria, y cuando cree que se acerca una situación de trashing, avisa al planificador de media plazo. El planificador entonces coge los procesos que estan en memoria y los vuelca en el disco (swap out) y el proceso estará en "swap".

Cuando el paginador vea que ya no está tan cargado, avisará otra vez al planificador a medio plazo, que irá haciendo swap in de los procesos que estaban en "swap"