

TEMA 4.2 : THREADS Y SEMÁFOROS

- Hasta ahora solo hemos visto procesos: [recordatorio página 1]

- ↳ Procesos cooperativos: Al tener un Sistema Operativo multiprogramado, va bien distribuir las tareas entre ellos, cada uno con sus tareas especializadas.

Necesariamente, tienen que mandarse información entre ellos, nos aprovechamos de recursos compartidos (como disco) → pipes?
Van gran problema de rendimiento (muchos overheads :(), y se reduce el paralelismo

- Cambiamos de filosofía: ahora los procesos no ejecutan, sino que son contenedores de recursos.

- ↳ Ahora un proceso es estático: contiene todos los punteros a los recursos.

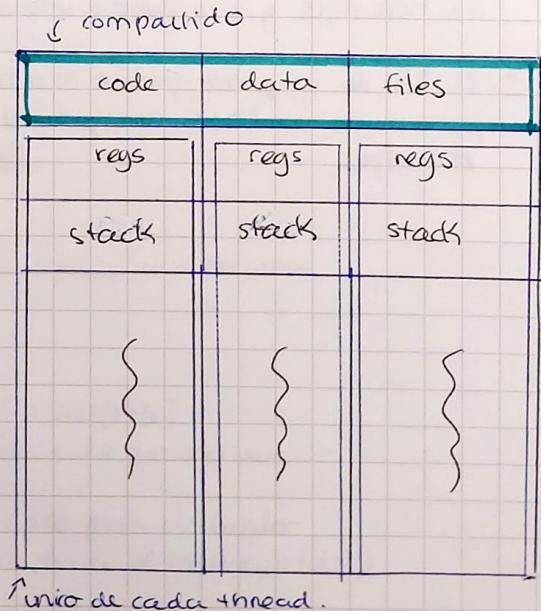
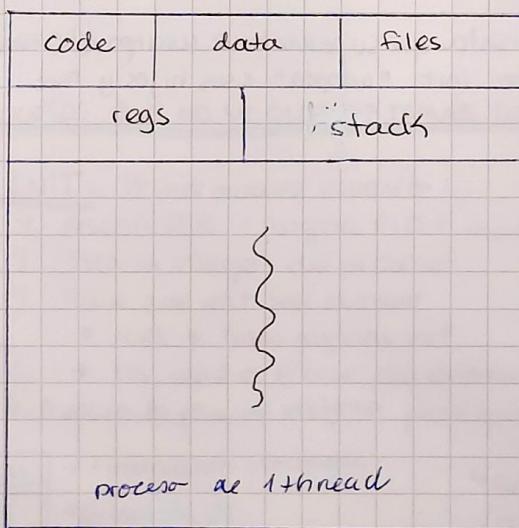
Los threads [hilos] ejecutan! → Piden recursos, y el Sistema Operativo se lo da al proceso que contiene el thread.

- * Cada thread es de 1 proceso, pero cada proceso puede tener varios threads (no se comparten!).

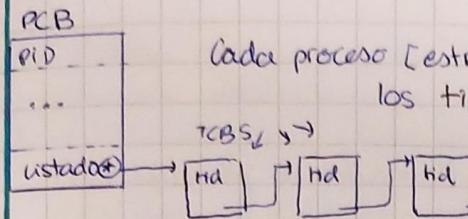
Threads son la unidad mínima de planificación de la CPU, ahora son estos los que pasan a RUN, READY, BLOCKED, procesos ESTÁTICOS!

Cada thread tiene su THREAD CONTROL BLOCK (TCB) ↴

- ↳ Con su thread ID (tid), cada uno tiene su propia pila de usuario/sistema
- ↳ Un TLS (thread local storage) para guardar variables a nivel de thread.



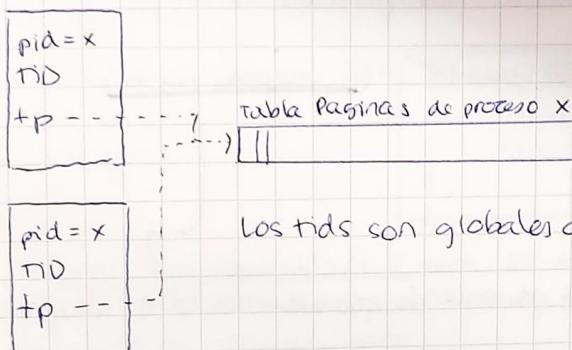
• procesos & threads en windows: Relación Explícita



Cada proceso [estructura que existe!] tiene una serie de threads, los tids son locales al proceso, y en el sistema global se identifican con el pid.
muy fácil de concentrarlos.

• procesos & threads en Linux : Relación Implicita

NO existe la estructura de un proceso, todo son threads sueltos .



A través de punteros se comparten información (apuntan a lo mismo), pero no hay ninguna estructura donde puedes encontrar todos los threads del proceso x concentrados.

Creación de threads en Linux:

pa void pointer? (*)

`sys_pthread_create (void * (*func) (void * param), void * param)`

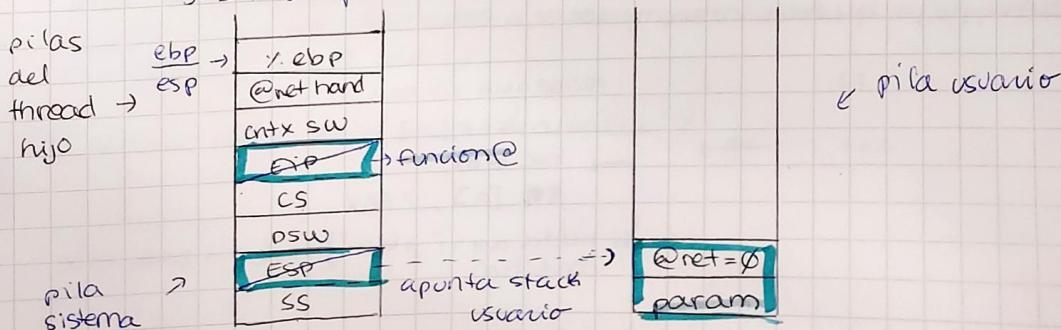
// crea nuevo hilo que ejecuta una función determinada con x parámetro

1. Allocata TCB y asigna tid
2. Inicializa campos TCB (copia datos del union task_union).

3. Inicializa contexto ejecución para el hijo : [ten en cuenta que por ahora tenemos : pila de sistema copiada del padre, pila de usuario vacía].

↳ 3.1 → En pila sistema machacamos en contexto hardware EIP y ESP para que retorne a func.

3.2 → En pila usuario empilamos : parámetro, $@ret = \emptyset$.



4. Dene el thread nro 0 en la ready queue.

5. Devuelve el tid asignado

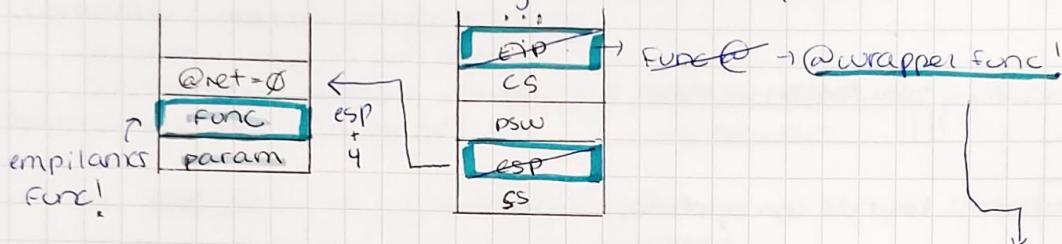
{

① Los threads únicamente se destruyen con la llamada a pthread-exit

Como no podemos suponer que la func que pasa el usuario llame a esta función.

↳ Creamos un "wrapper" que llame a func, retorne al wrapper, y ejecute pthread-exit.

Entonces la pila de usuario y sistema nos queda:



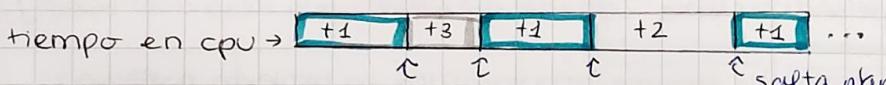
Hay un pequeño problema, estamos asumiendo que sabemos la @wrapper hay 2 opciones:

↳ obligar a wrapper(func) a estar en la misma @ lógica [hard coded]

↳ pedirle al SO la dirección de wrapper func.
Windows
Añade otro parámetro wrapper a la función sys_create_thread.

• El problema de las data race conditions.

Estos threads, no se ejecutan de inicio a fin in-interrumpidamente en la cpu, si no que, dependiendo de el quantum, en medio de ciertas instrucciones puede saltar a otros threads (multiprogramados).



Esto puede llevar a que variables no se accedan de manera correcta, o se apliquen operaciones varias veces, incluso cuando el código tiene una sola linea, como en realidad son varias líneas en ASM, este programa quizá queda interrumpido.

T1:

a++

T2:

a++

→

PROGRAMA EN ASM:

LD %EBX,[a]

INC %EBX

STP [a], %EBX

si salta aquí planificador → Data race condition.

• MECANISMO del sistema para evitar las data race conditions:

SEMAFOROS: marcar zonas del código para evitar que varios threads modifiquen datos a la vez → EXCLUSIÓN MÚTUA: Te pueden quitar de la CPU igualmente, pero ahora se deben ejecutar en cierto orden.

• Llamadas al sistema: semáforos: crear, iniciar, terminar

→ int sys_semCreate (sem_t *s, int value) {
// CREACIÓN: tmbn se llama: sem-init (s, value);

{ s->count = value; *
 init-list-head (&s->blocked); //cola de bloqueados vacía

* Se me ha ocurrido
poner la struct
de semáforo:

```
struct sem_t {  
    int count;  
    struct list-head blocked;  
};
```

→ int sys_semWait (sem_t *s) {

// BLOQUEA el thread que hace la llamada!

// inicio zona exclusión

s->count -;

if (s->count < 0) {

list-add (current(), &s->blocked);

sched-next(); // para que el planificador ejecute otros

}

→ int sys_semPost (sem_t *s) {

// DESBLOQUEA → final zona exclusión

// tmbn se llama: sem-signal (s);

s->count ++;

if (s->count <= 0) {

list-head (*l = list-first (&s->blocked));

list-add (l, &ready-queue);

}

*Queremos que sean llamadas a sistema para que no salte el planificador
en medio de este ↑ código.

*) ¿Cuál debe ser el valor de "value"? ↪

int a = \emptyset ; \downarrow lo ponemos a 1
 sem_t s;
 sem_init(&s, ①);

T1: T2:
 1) sem_wait(&s); 2) sem_wait(&s);
 a++;
 3) sem_post(&s); 4) sem_post(&s);

► [MUTEX]: Permitimos que 1 solo thread acceda de forma simultánea
 exclusión mutua

① T1 ↓
 1) s->count = \emptyset
 blocked = \sim .
 2) T2 ↓
 s->count = -1
 blocked = T2

② T1 ↓
 3) s->count = \emptyset
 blocked = $\cancel{T2}$ → T2 a la ready-queue.
 4) s->count = 1
 -

!! Dead locks !! → Cuando tenemos varios threads esperando que alguien los desbloquee (pero todos están bloqueados)

Como evitar: cumplir al menos 1 de estas condiciones:

- ↳ Tener recursos compartidos
- ↳ Poder quitarle un recurso a un thread
- ↳ Poder conseguir todos los recursos de forma atómica
- ↳ Ordenar las peticiones de recursos.

► SINCRONIZACIÓN: sem-init(s, \emptyset) Sirve para forzar orden de ejecución:

{ T1 { T2
 { A1 { A2
 { B1 { B2
 { C1 { C2
 { D1 { D2

→

{ T1 { T2
 { S-W(&s, \emptyset) { A2
 { A1 { B2
 { B1 { S-p(&s)
 { C1 { C2
 { D1 { D2

↑ instrucciones

↑ De esta manera, las instrucciones A2 & B2 se ejecutan antes que A1.

► RESTRICCION DE RECURSOS:

sem_init(&s, 1) // valor por defecto +1: [no se usa mucho]

No protege regiones, sino que funciona como semáforo de recursos

sem_t s;
 sem_init(&s, ②)

Permite
 que solo
 ② threads
 accedan
 a
 Data
 Unit

sem_w(&s);

++DU.access(); ←

sem_post(&s);

RESUMEN "VALUE"

1 → Proteger Regiones

\emptyset → Sincronización

>1 → Acceso a recursos

TEMA 5;1: SISTEMA DE FICHEROS

¿Cómo se organiza el disco?

- Prev [1970N]: No hay un disco duro, hay discos de 178KB que tenían de 8 a 40 entradas, se podían ver el nombre y los metadatos en el "trozo raíz". No había directorios
metadatos: lista de sectores $8\text{bytes} + 3\text{de tipo fichero}$

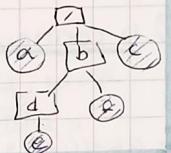
sector \rightarrow De 256 a 512B \rightarrow Unidad mínima transferencia disco.

- Primeros discos duros ($<100\text{MB}$)

→ Mejoran los tiempos de lectura/escritura

→ Mucho más almacenamiento: organización en árbol de directorios se crean los directorios! → ficheros especiales.

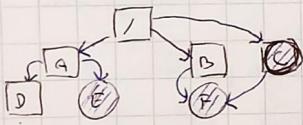
1) NOMBRE fichero	2) nro	3) METADATOS
listado	...	



ACTUALIDAD: mucha capacidad y miles de ficheros. \rightarrow El path absoluto empieza a ser estúpidamente largo.

Evolucionamos de árbol a grafo \rightarrow introducción a soft & hard links

→ El fichero /B/F \equiv /C (mismos datos)



¿Qué pasa cuando eliminamos alguno de los ficheros?

¿Cómo sabemos cuantos ficheros están referenciando los mismos datos?

INODO: Nueva estructura, que para cada "conjunto" de datos, mantiene cuantos ficheros lo referencian.

En Windows se llaman Records

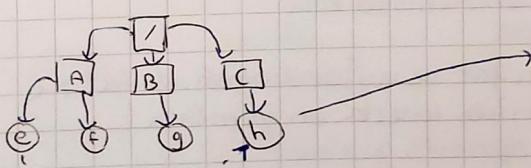
Ahora los directorios contienen un listado: \rightarrow nombre fichero | num inodo

→ ¿Qué contiene un inodo?:

- 1) #num referencias
- 2) #major y #minor
- 3) metadatos

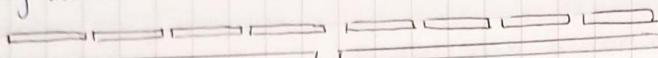
Cada una de estas entradas es un HARD LINK!

¿Qué es un soft link?: Cuando los datos de un fichero son la RUTA a otro fichero.



fichero @: /c/h \rightarrow Su major y minor
fichero #: datos ... \rightarrow lo declaran de tipo soft link

• ¿Qué metadatos? → los que permitan localizar los datos en disco
se trabaja con bloques de disco → unos cuantos sectores consecutivos.

↳ Hay varias maneras de tener bloques + discos pero se usa:
sectores → 
bloques → 

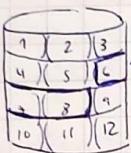
BLOQUES DEL MISMO
TAMAÑO + NO COMPARTEN
SECTORES.

→ Hay bastante fragmentación interna, pero solo en el último bloque de los datos, hoy en día representa 0,001% espacio, y es muy fácil de gestionar.

Ahora ya sabemos como se organizan los sectores, pero, ¿cómo se organizan los bloques en disco?

∅ ↳ contigua + estática

ISO 96-60 →



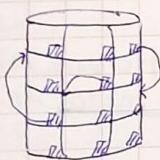
INODO:
size + primer bloque

Si que se usa en
DVD's, CD's :)

Muy fácil de acceder pero no es realista: los ficheros acceden; en un ordenador necesitamos que sea dinámico.

∅ ↳ contigua + dinámica → crea una fragmentación externa que te cagas.

↳ asignación encadenada: lista dinámica de bloques asociados al fichero (Acceso aleatorio)



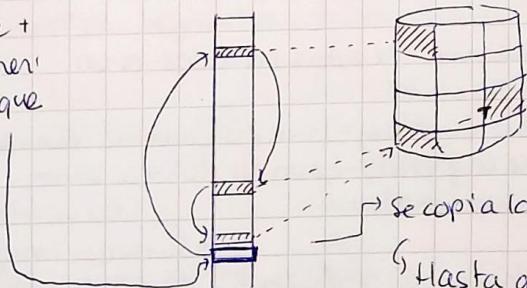
INODO: size + index primer bloque

→ soluciona fragmentación
→ los accesos se vuelven complicados.

↳ asignación encadenada en tabla [FAT: File Allocation Table]

INODO:
size +
primer'
bloque

Tabla FAT:



la FAT contiene tantas posiciones como bloques haya en disco.

Ya no se paga la latencia de acceso a disco para saber cuál es el siguiente.

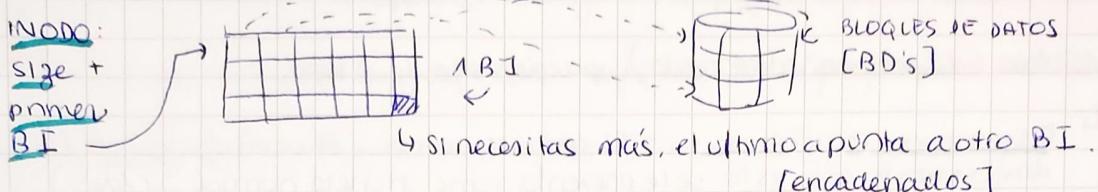
↳ Hasta que la fat se vuelve demasiado fat :-(

Tiene tantas entradas que ya no es factible tenerla en memoria.

* Ahora se tiene que apagar el ordenador de manera ordenada!

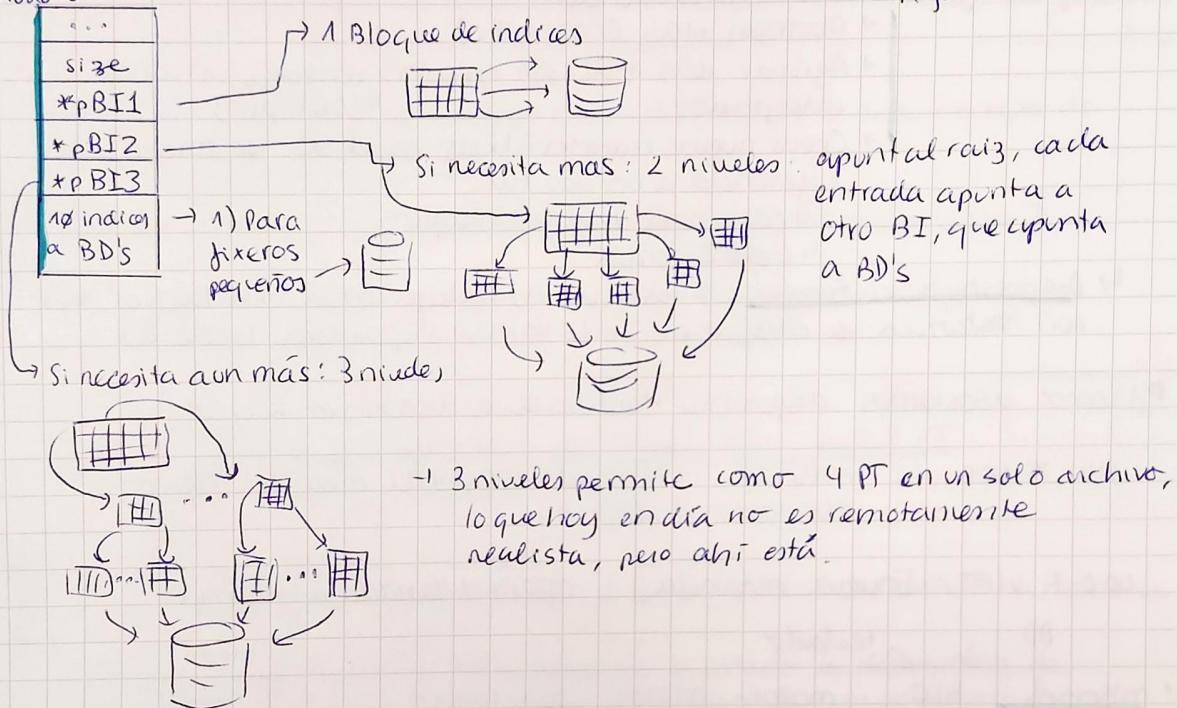
↳ Asignación indexada: [NTFS : windows]

• Nuevo tipo de bloques especializados → Bloques de Índices [BI]

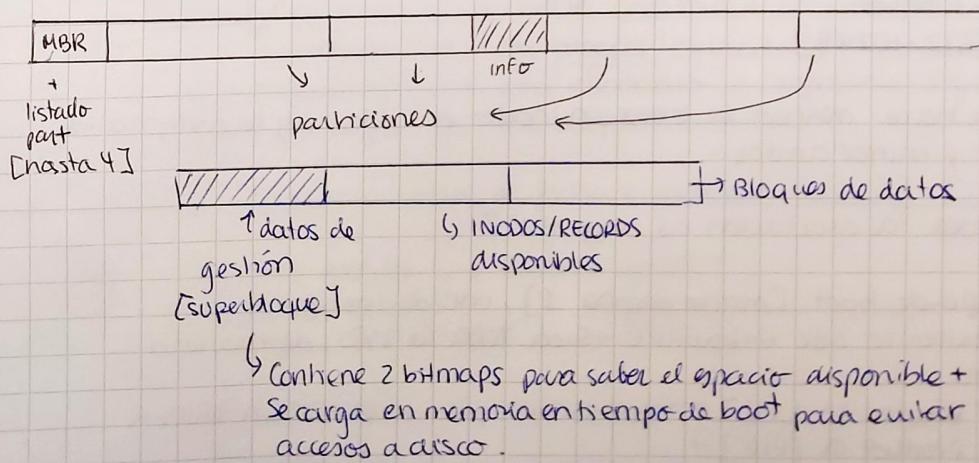


Cada entrada en el BI corresponde a un BD

6 Asignación indexada multí nivel [Ext 2 : Linux] : bloques de índices en forma de árbol



▶ CONO ESTA EL DISCO ORGANIZADO (de manera genérica)



TEMA 5.2 → E/S Y ACCESO A DISPOSITIVOS

todas las llamadas para interactuar con dispositivos son genéricas (open, read, ..) ¿Cómo hace el SO para saber qué se hace con cada dispositivo concreto?

Los dvs (dispositivos abreviados), pueden ser de 3 tipos:

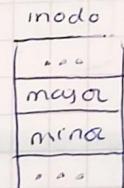
- 4)  Dispositivos hardware → El coche en sí: El usuario no va a trabajar directamente con esto.
 - 4)  Dispositivo lógicos → Lo que nos ofrece el sistema de ficheros ya sea un fichero de un dispositivo hw (`/dev/sda`) un fichero de datos de usuario, un directorio... es todo.
Hay 3 subclases según su función
 - Abstraen HW (`/dev/sda`)
 - Agrupar devs HW: por ejemplo consola, muchos integrados. 
 - Crear nuevas características en el SO: NO hacen referencia a un dev HW.
 - ↳ `/dev/null` → agujero negro
 - ↳ directorios
 - 4) Dispositivos virtuales: El usuario no trabaja con los devs lógicos, sino con instancias de ellos, a eso se le llaman dispositivos virtuales.

→ ¿Cómo asociamos dispositivo hardware a dispositivo lógico?

`+HW LOG` : `mknod` → llamada al sistema crea dev lógico asociado a dev hw.

`LOGIC VIRT: [nueva instancia]`: open → llanada a sistema.

DD
mKnod
Dispositivo que
funciona por bloques
de datos [512-1024B]
b/c major minor dev-lógico
Byte a byte ↑ depende
 "tipo de dispositivo"



• Lo que hace mknod es crear un inodo en /dev, y le asigna el mayor y menor dardos.

¿Cómo se crea la asociación al desp. HIW?

↳ En tiempo de boot [mirar página 1], uno de los pasos es RECONOCIMIENTO DEL HARDWARE → Saca TODA la info, de manera estandar te dan 2 datos

→ vendor ID [VID] → Nos sirven para encontrar un DRIVER
→ product ID [PID]

Existe una carpeta con todos los DRIVERS que tiene el SO, busca ahí si tiene alguno compatible; si no encuentra cosa.

¿Qué es un driver? → Fichero con código que solo funciona en modo sistema.

↳ Tienen una estructura muy definida: definida por el programador del SO

HEADER
↓ funciones
Una vez encontrado el DRIVER分配a un descriptor de dispositivo

HEADER
punteros
a funciones
1) copia el header del driver
2) copia las direcciones de memoria donde están las funciones que se usan: open, read, write...

En Linux el VID+PID se traduce a un solo número que dice qué tipo de dispositivo es → El MAJOR!

↑ Esto pasa con todos los dispositivos HW pinchados al computador]

↓ Nos queda un listado de descripciones de dispositivo inicializadas, apuntando a las funciones del driver correspondiente.

• int open (char *filename, int flags, [int mode]) {
 // Crea instancia de dispositivo lógico

- 1) Abre el dispositivo filename
2) trae el modo que corresponde (el que se ha hecho con mknod) a memoria

inode
...
MAJOR
...
3) Coje el MAJOR y busca en el listado de descriptores de dispositivos el que le corresponde (con el mismo major)
 ↳ Si no error

↳ nueva entrada \oplus^1 [TI]

↳ En el SO a veces existe una TABLA DE INODOS, donde cada entrada contiene pares (*inode), (*descriptor dispositivo)

Esto es lo que verdaderamente está asociando un dispositivo lógico con el driver que sabe trabajar con ese dispositivo HW.

2) añade entrada a la tabla de ficheros abiertos [global] [TFA]

↳ #ref * puntero lect/esc * tabla de inodos []
 ↳ cuantas TC referencian ↳ la que se acaba de crear esta entrada

3) se va a la tabla de canales del proceso: nueva entrada! [TC]

↳ [...] * entrada TFA, la que acaba de crear!

↳ RETORNA EL NUM DE ENTRADA DE LA TC.

④ Si hacemos un segundo open de el mismo dispositivo:

- ! → NO se hará nueva entrada en la TABLA DE INODOS
- ! → (S) se hará nueva entrada en TFA i en TC del proceso.

En el fork se copia la TC del proceso y se incrementa el numero de referencias en la TFA.

lo que permite tener acceso concurrente y compartido al mismo dispositivo con instancias distintas.

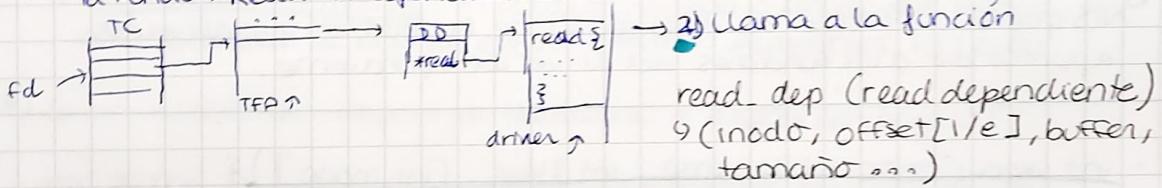
¿Cómo se lee el dispositivo? read (urappp) llama a:

int sys-read (int fd, char *buff, size_t size) {

// Para leer un dispositivo de manera genérica

// Fd es la posición en la tabla de canales (lo que retorna open)

- 1) En la TC del proceso, coje el puntero a la entrada en la TFA, y en la TFA → Tabla de Inodos → descriptor dispositivo → Driver con la función Read correspondiente



- 3) Una vez leido; incrementa el puntero de lectura/escritura en la TFA, y retorna el numero de bytes que ha leido (resultado en buffer)

?

El único problema que nos da esta implementación es que read-dep puede tener gran latencia, y no podemos permitir que se nos quede bloqueado tanto tiempo a que termine el read.

SOLUCIÓN: Crear el estado de BLOCKED!

- En vez de que el thread que ejecuta este read se coma toda la latencia, vamos a crear un estado para que el thread principal pueda continuar ejecutando

NUEVO PROCESO SISTEMA: EL GESTOR ○ Vamos a modificar el open

... para que cuando sea necesario...

Los procesos a partir de ahora no hacen E/S, sino que hacen peticiones de E/S

• E/S Sincrona con Gestor

El gestor () es quien acaba realizando la E/S [se crea en tiempo de boot]

Para que esto funcione, el puntero al device-descriptor que hay en la tabla de nodos no debe ser uno que apunte directamente al driver, sino que va a haber un segundo device-descriptor con los punteros a las funciones dependientes del gestor () y este ejecutara, buscando en el device-descriptor que apunta a driver el read-dependiente cuando corresponda

[VER ESQUEMA!] (p.14) *

Para evitar que el gestor esté constantemente preguntando si hay alguna nueva petición (Polling), hacemos que funcione con interrupciones, de esta manera nadie se come la latencia.

También se añaden un semáforo de exclusión mutua a cada cola para que no trabajen a la vez (no sale en el esquema)

Hoy en día los SOs no funcionan así, sino que funciona de manera asíncrona, para que todo pueda seguir ejecutando hasta que se necesiten los datos de la E/S

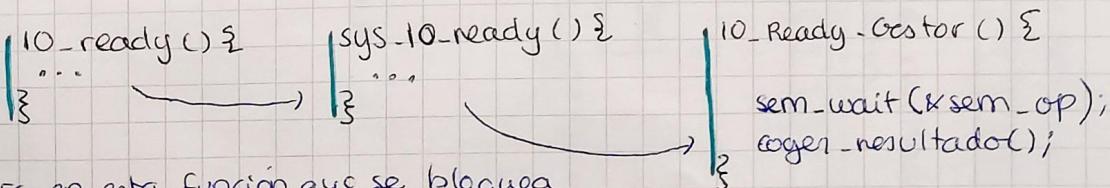
• E/S Asíncrona

la llamada a sistema ahora es int-read-async (fd, *buffer, size); y esta llamada, sys-read-async hace exactamente lo mismo pero llama a:

no bloquea el proceso

```
int read-dep-gestor-async (fd, ...){  
    encolar-petición();  
    sem-post (&sem-gestor);  
}
```

Implementamos una nueva llamada a sistema, que nos dice si los datos de la E/S están listos



```
int read (fd, *buffer, size){  
    read-async();  
    10-Ready();  
}
```

[El código del gestor es el mismo]

ESQUEMA : E/S Sincrona con Gestor

TC del proceso

```
int read (
    fd -->
    ...
)
```

TFA

PTI

TI

PDID

device-descrip

read*

read-dependiente-gestor () {

encolar-petición();

sem-post(&sem-gest);

sem-wait(&sem-op);

cojer-resultado();

Gestor

while(1) {

: sem-wait(&sem-gest);

cojer-petición();

do E/S;

encolar-resultado();

sem-post(&sem-op);

cola de peticiones q- IOBR

cola de resultados

q- iofin

device-descriptor

*read-driver

driver

read-dcp

④ sem-op → inicializado con value = \emptyset , hay un semáforo para cada petición IOBR

⑤ sem-gest → uno por gestor, inicializado a \emptyset .