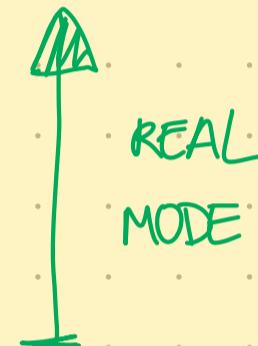
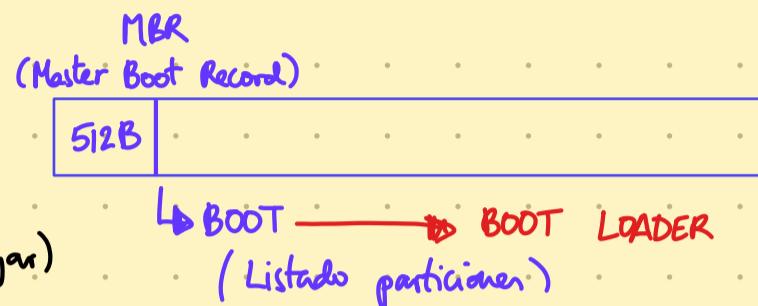


TEMA 1: BOOT

(mirar boot linux)

Contiene el primer código a ejecutar

- 1 Power on → HW reset (entramos componentes activos previamente)
- 2 Reconocimiento HW → Buscamos recursos (los menores posibles: MEM, CPU, BIOS)
- 3 LOAD BIOS (Basic I/O subsystem) → Cargamos la BIOS en MEM
- 4 POST (Power On Self Test) → Segundo reconocimiento, para ver detalles de los dispositivos conectados
 - ↳ Busca el SO (Bootable Device) como objetivo (normalmente disco)
- 5 MBR: Contiene código para cargar el BOOT
- 6 BOOT: Antes contenía SO entero, ahora una parte
- 7 BOOT LOADER → Kernel
- 8 Kernel (iniciamos para que la CPU sepa como trabajar)
 - * Init struct SW
 - * Init struct HW (viene dados por los fabricantes)
 - IDT, GDT, TSS, TP...
 - * HW check (tema 5)
 - * INIT LD (Define como trabaja la MEM y primer proc de usr)
 - * IDLE (proceso de sistema)
 - * PROTECTED MODE → Entramos a modo protegido (Activamos trad. de @Fis → @Lis. y también modos de priv.)
 - * USER MODE



LLAMADAS A FUNCIÓN (Reparo)

```
push 4
push 3
call foo
esp & esp + 4 * #param (int)
```

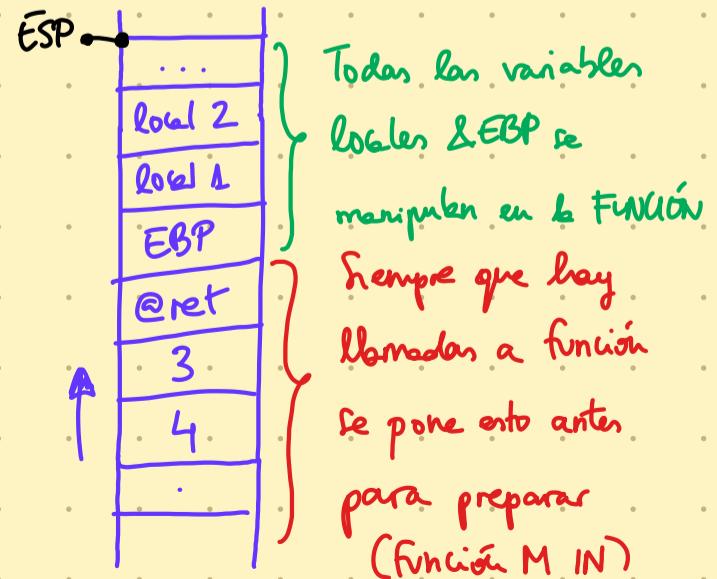
Antes del foo
(enlace dinámico)

```
int foo(int a, int b) {
    return a + b;
}
```

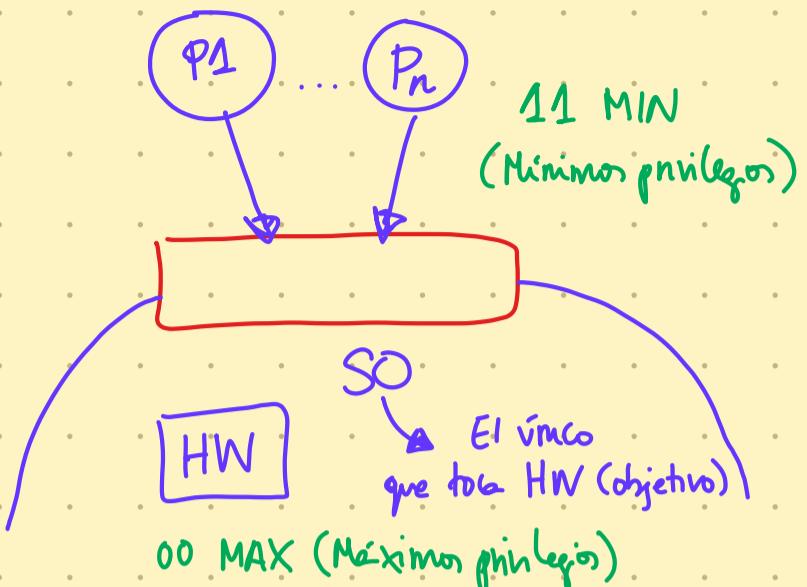
Función de ejemplo

```
push EBP
EBP & EBP
ESP & ESP - 4 * #var.local
EAX &
ESP & EBP
POP EBP
RET
```

Después de foo
(deshacer enlace dinámico)



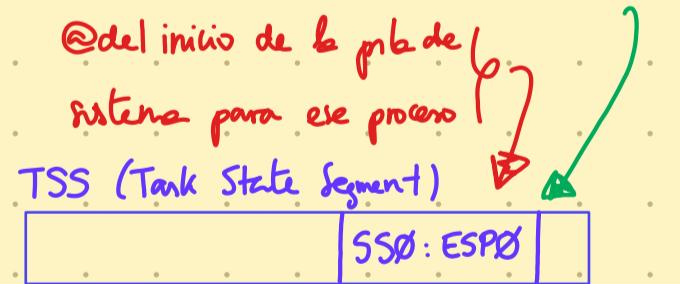
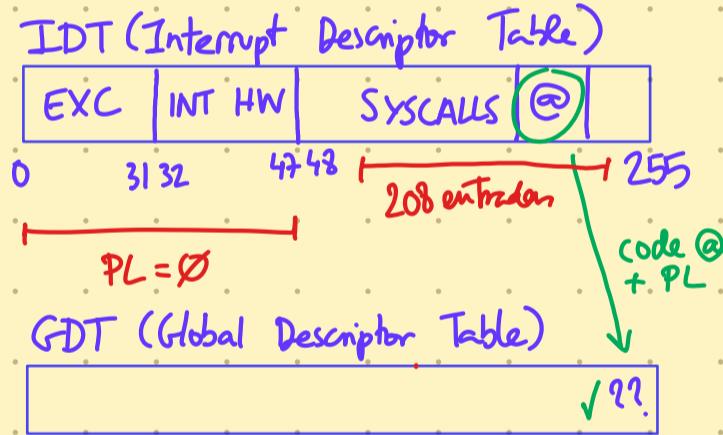
TEMA 2: SYSCALLS



- * Los niveles de privilegios definen quienes acceden no deseados a recursos

¿Cómo establecer privilegios?

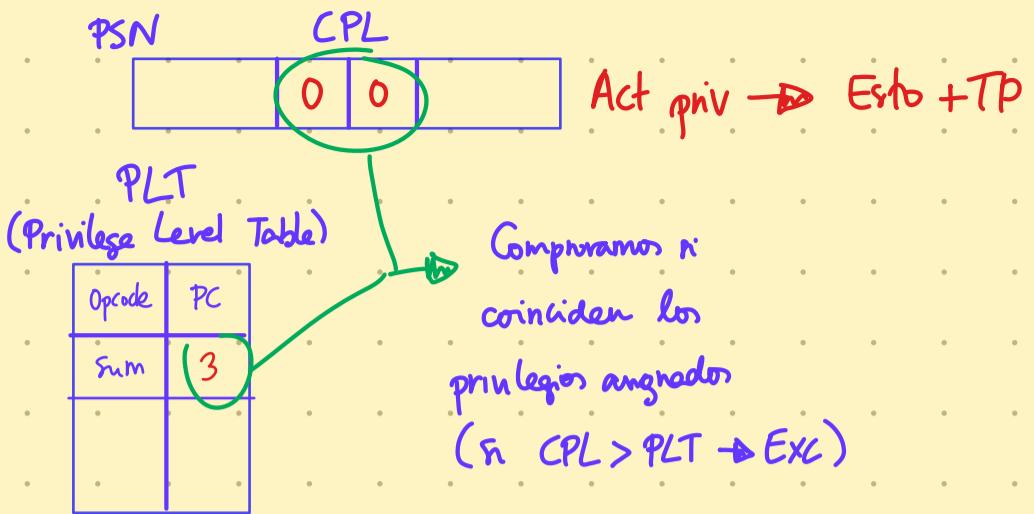
- * EXCEPCIONES: Se activan solo en modo vínculo cuando no se sabe hacer algo (INT)
- * INTERRUPCIONES HW: Acciones HW que fuerzan ejecución del SO (\rightarrow priv) (SYSENTER)
- * INTERRUPCIONES SW o SYSCALLS: Instrucciones que fuerzan un \rightarrow priv (SYSCALL)



- * Cargamos en la pila de sistema todos los valores previos del HW para cuando salga de la interrupción (Contexto HW)

- * La GDT describe cómo se puede usar un segmento de memoria (inicio, límite, permisos & nivel de privilegios)

- * La TSS guarda el estado de una tarea

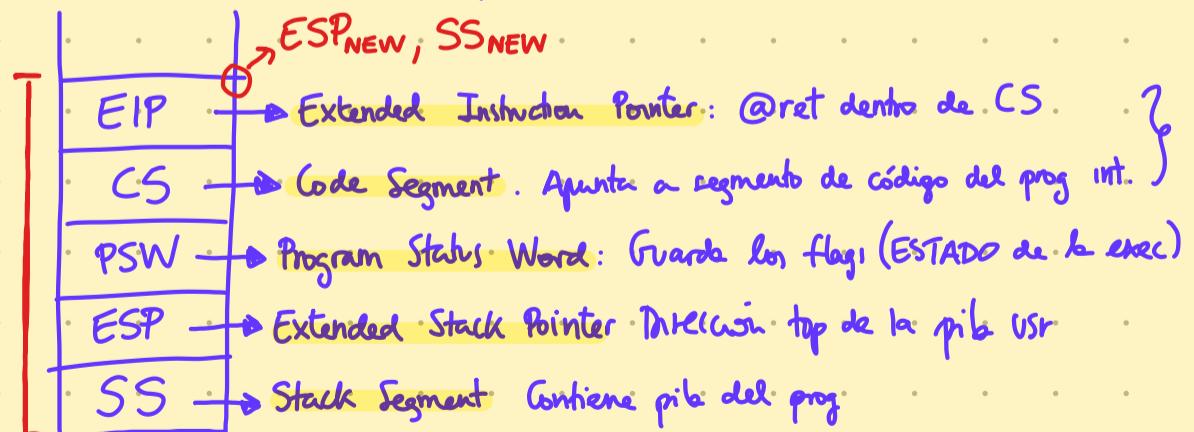


- * Cuando se ejecuta una instrucción con privilegio el HW comprobará que sea en CÓDIGO DE SISTEMA (Si no salta EXC)
- * Las instrucciones con privilegios solo las ejecutan el SO

¿Qué hace la CPU para hacer el cambio de privilegios?

- * Contiene las direcciones a SO donde se encuentra el Código para tratar int/syscall/sysenter y el nivel de privilegios asignados en un sector
- * Chequea que es válida la dirección de la IDT en la GDT
- * Va a la TSS a por la dirección del inicio de la pila de sistema

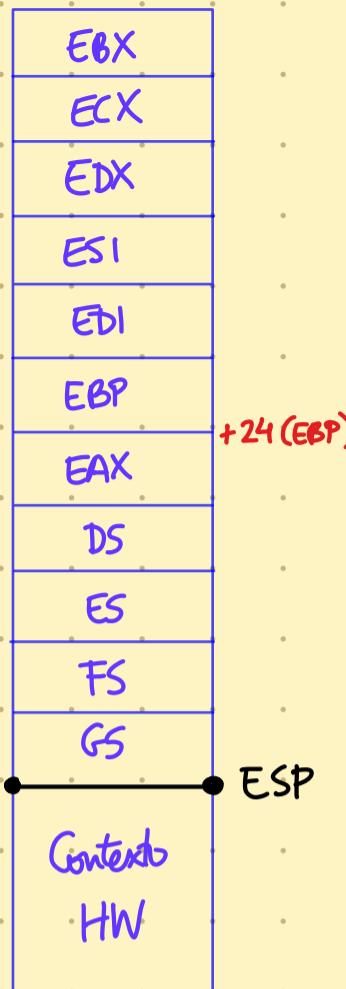
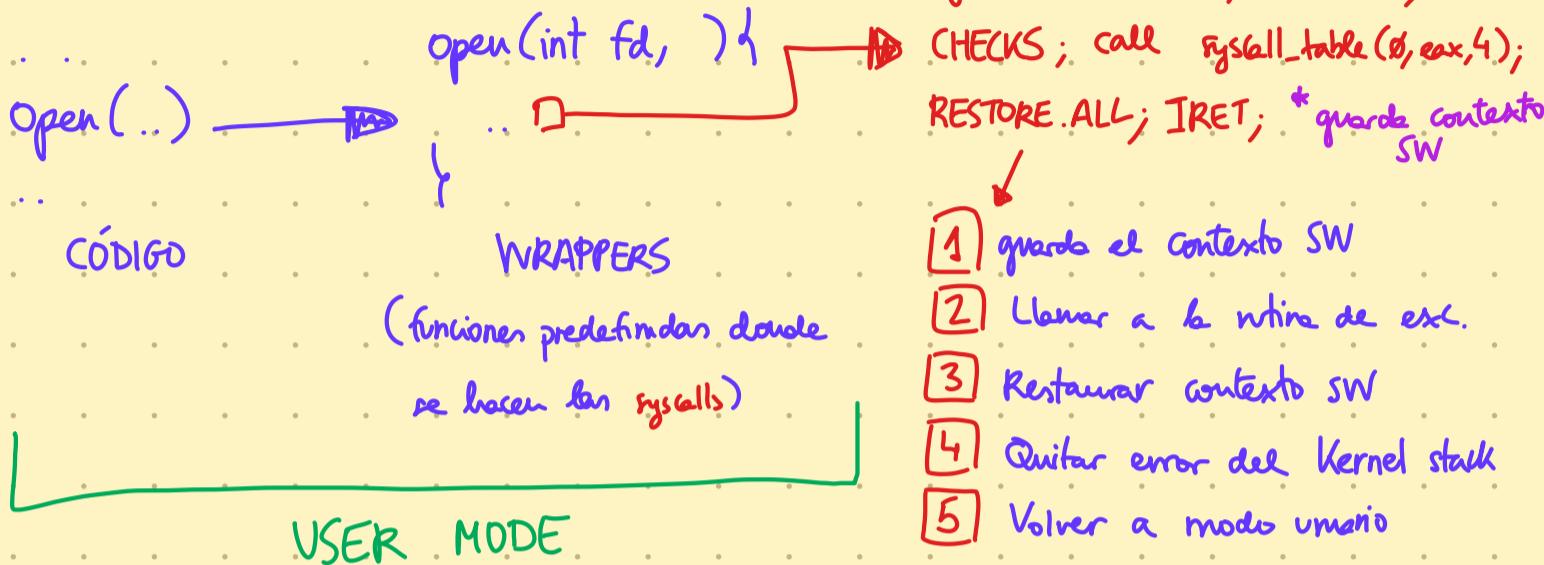
SYSTEM STACK \rightarrow Pila especial que usa la CPU en modo privilegiado



PROCEDIMIENTO DE ENTRAR AL SISTEMA

¿Cómo monta el SO en base a estos cambios en el procesador?

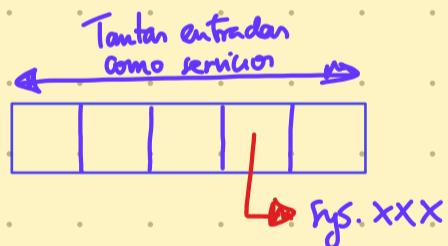
- * Excepciones → HW error en Kernel stack (solo guardamos contexto HW)
- * Interrupciones → No HW error en Kernel stack, notificamos entrada de la INT (inicio & fin)
- * SystemCalls → Instr. en ensamblador que provocan una int. SW (Guardamos contexto HW&SW)



En los sistemas Linux modernos, hay más de 206 servicios (canillas de syscalls en la IDT). Por tanto se usa \$EAX como selector de servicio (hay variación por canilla) dentro de 0x80 (en caso de Linux)

→ Los SO actualmente tiene solo **1 punto de entrada al sistema** (por seguridad)

¿Cómo selecciona EAX el handler del servicio adecuado?



- * Estructura SW que contiene las direcciones de los handlers de cada servicio
- * Rellenada en el BOOT del sistema operativo (no en la compilación, para garantizar la aleatoriedad en memoria)

¿Cómo voy a pasar parámetros al SO? → **POR REGISTRO**

1 Establezco el orden de los regs (ejemplo):

EBX, ECX, EDX, ESI, EDI, EBP

2 Lo primero que llamará un servicio será hacer push y redireccionar ESP

1 PUSH.

EBP ← fd

ECX ← mode

EAX ← #servicio

2 PUSH EBP

ESP ← EBP

...

Nos podemos dar cuenta de que la pila NO SÓLO guarda registros, sino que también constituye un frame de activación para los handlers a los servicios

IMPORTANTE SABER CONSTRUIR LA PILA DE SISTEMA



¿Qué devuelve el kernel al ejecutar la syscall?

- * handler ≥ 0 si todo ha ido bien
 - * si handler < 0, devuelve -CÓDIGO_ERROR
- } errno en variable de USUARIO, así que en el WRAPPER haremos la conversión devolviendo -1 y errno = CÓDIGO_ERROR

¿FUNCIONA?

Actualmente en el restore.all estamos poniendo en eax el num de servicio, pero ¿y si necesitamos saber el error? Tenemos que modificar EAX en el contexto SW para que se restaure bien:

1 call syscall_table(0,%eax,4) → 2 24(\$EBP) ← %EAX → 3 RESTORE.ALL

¿Cómo checkeamos?

Excepciones (las ejecuta el SO, NO la CPU)

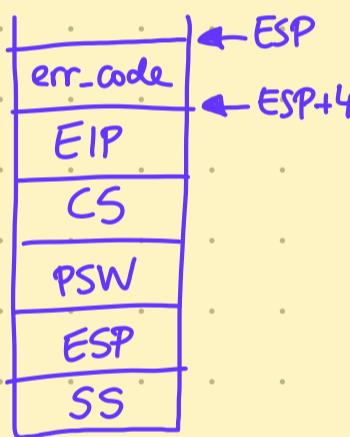
* PARAM CHECKS → Miramos que los parámetros sean correctos

En este caso tenemos 1 handler/excepción, a diferencia de las syscalls (1 handler igual para TODAS)

* RESOURCE CHECKS → Miramos que tengamos los recursos

* LET'S DO IT

Interrupciones (Igual que las excepciones, 1 handler/interrupción)



Contexto HW

clock_handler.

SAVE ALL;

call clock.routine(),

EOI (en reloj va ANTES del call)

RESTORE.ALL;

IRET;

```
excep handler( ) {  
    SAVE ALL;  
    call excepc-routine(void);  
    RESTORE.ALL,  
    ESP ← ESP + 4,  
    IRET;
```

- * Para volver a entregar la pila de sistema desplazamos ESP a ESP+4 ya que eso se ha añadido en el handler de la excepción.

End Of Interrupt (EOI) sirve para indicar el fin del tratamiento de la INT.

SOLO SE PUEDEN TRATAR DE FORMA SECUENCIAL

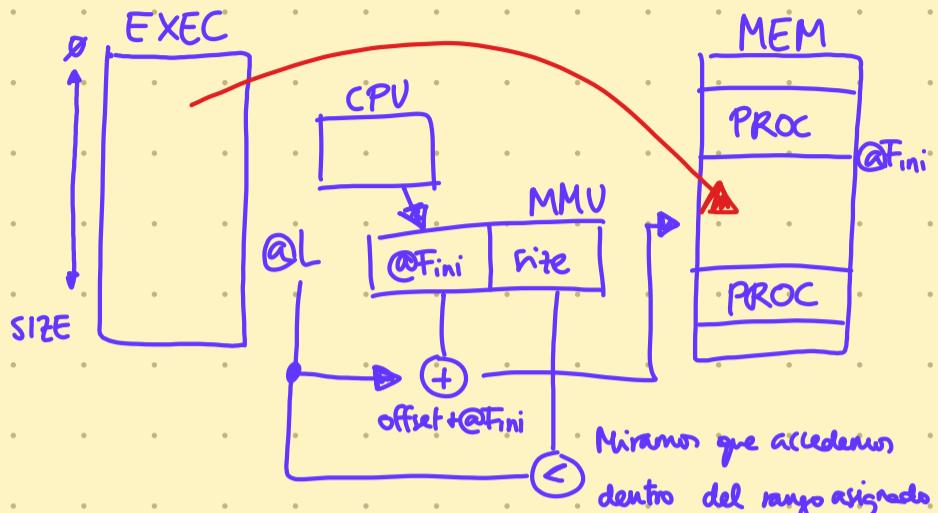
- * SYSENTER_CS_MSR → Contiene el selector CS del Kernel
- * SYSENTER_EIP_MSR → Contiene el punto de inicio del Kernel
- * SYSENTER_ESP_MSR → Apunta a la @ del inicio del TSS

Para interrupciones rápidas evitamos mecanismo típico & check de permisos. Estos registros se usan para pasar de user mode → sys mode.

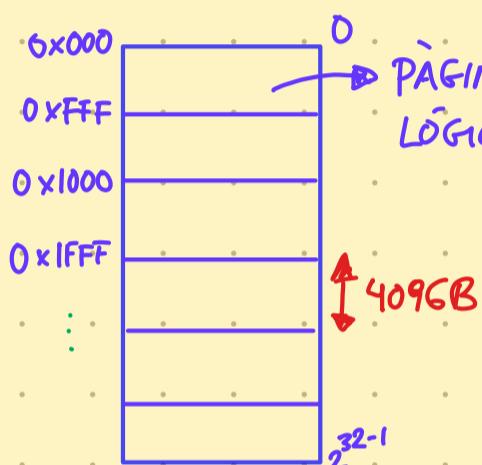
TEMA 3: MEMORIA

FRAGMENTACIÓN (el prog no cabe en dirs. contiguas)

¿Qué es un TLB?

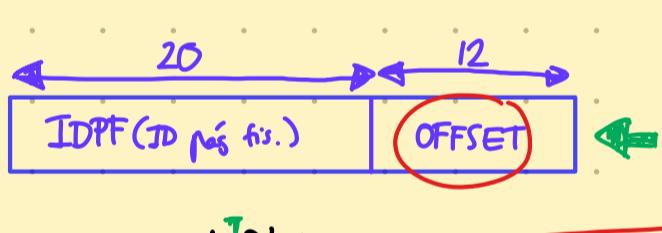
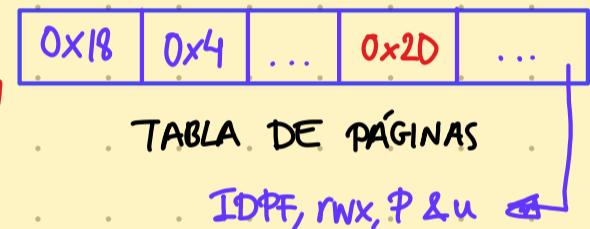


- * Todas las páginas lógicas son del mismo tamaño
- * La tabla de páginas guarda las traducciones de págs. físicas a págs. lógicas (1 por proceso)
- * El **OFFSET** se mantiene intacto en la conversión de página lógica a física

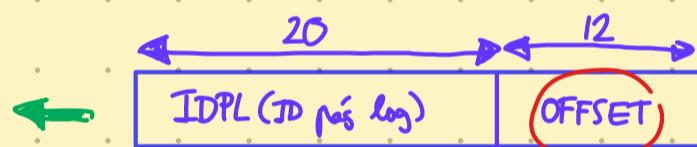


$$* \frac{2^{32} (\text{dir. lógicas})}{2^{12} (\text{tamaño pág})} = \boxed{2^{20} \text{ entradas}} \text{ en la tabla de páginas}$$

* Dos procesos normalmente no comparten TP



IDPL	IDPF	rwx	u
0x0	0x18		
0x1	0x4		
..	..		
0x7	0x20		
...	...		



0x1040

(*CR3. Registro que apunta a @Finí de la tabla de págs.)

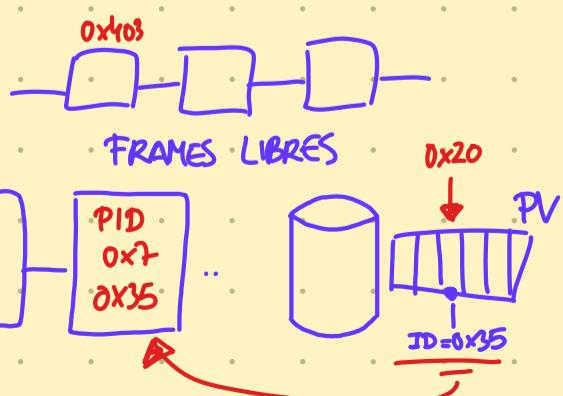
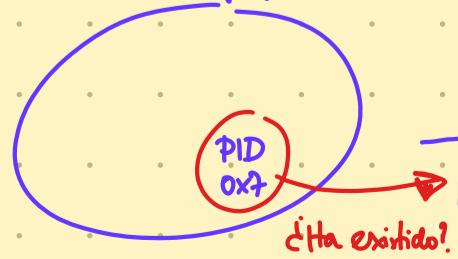
* El hit de presencia (P) define si una página lógica TIENE traducción física VÁLIDA

↳ Si hay fallo de TLB y P=1 en TP: Modificamos TLB

↳ Si hay fallo de TLB y P=0 en TP: EXC. PAGE FAULT

* Todos los procesos tienen mapeado en su tabla de páginas el Kernel (con el hit u podemos ver si ese rango de direcciones lógicas son accesibles)

PAGER (paginador)



* Cuando hay un MISS de TLB se carga la página física en una página virtual

* Se mira si ha existido previamente, y en caso de que no salta un SEGMENTATION FAULT

* Se le asigna un nuevo frame físico & modifica la TLB

REUBICACIÓN DINÁMICA

¿Cómo hacerlo la dinámica?

PAGINACIÓN

* La tabla de páginas tiene tantas entradas como páginas lógicas tiene el programa

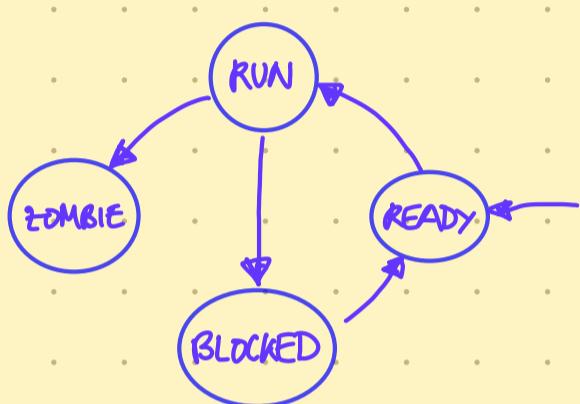
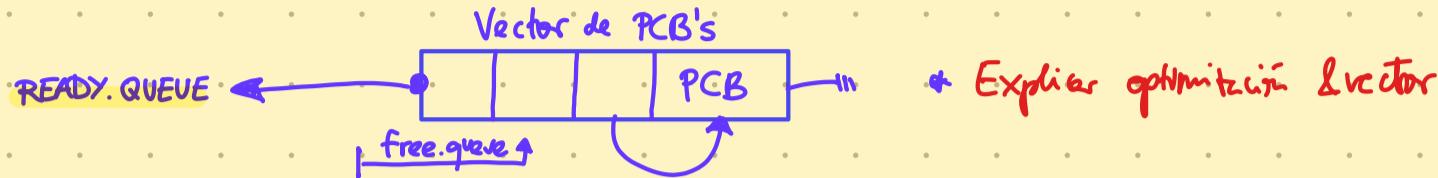
* Dividimos el programa en rangos de direcciones lógicas

* Dos procesos normalmente no comparten TP

TEMA 4. PROCESS MANAGEMENT

A partir de ahora los procesos serán PCB's (Process Control Block): PID, TP, Signals, TC

- EPROCESS
- task_struct



¿Cuáles son los estados de un PCB?

- * RUN → Está el proceso actual (detailed below)
- * READY → Un proceso está en READY cuando está en la cola ready.queue (está listo para ejecutarse, implementación).
- * BLOCKED → Un proceso pasa a BLOCKED cuando hace una operación de larga latencia (no gasta CPU)
- * ZOMBIE → El proceso sigue existiendo (no se libera el PCB) pero se han liberado los recursos de usuario.

ESTADOS DE UN PROCESO

¿Cómo accedemos al PCB del proceso actual (RUN)? → Registro GS en WIN (por ejemplo, en [0x3E8] tiene PID = 0x3E8)

¿Y en Linux?

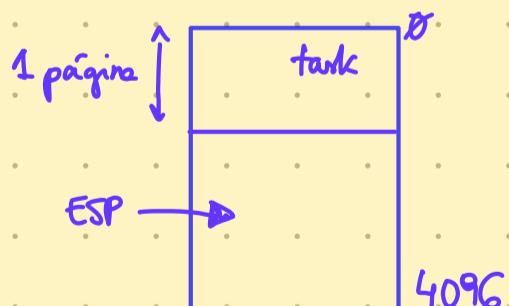
```
struct task_struct  
{  
    int PID;  
    *TP, *TC  
};
```

¿Qué es union? → Definición
union task_union
{
 struct task_struct task;
 unsigned long stack[1024];
};

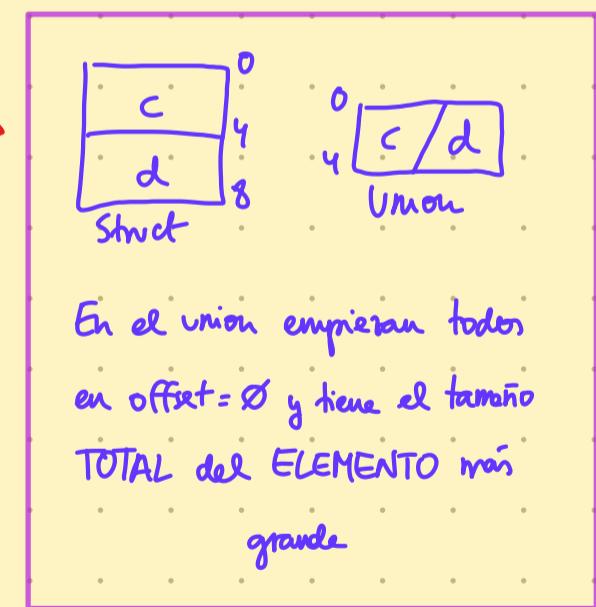
```
task_struct current()  
{  
    eax<-esp  
    eax<-eax & 0xFFFFF000  
};
```

FUNCTION CURRENT

(fija a ESP dentro del contexto del union)



Hay que saber SIEMPRE cuál en el tamaño de la pág

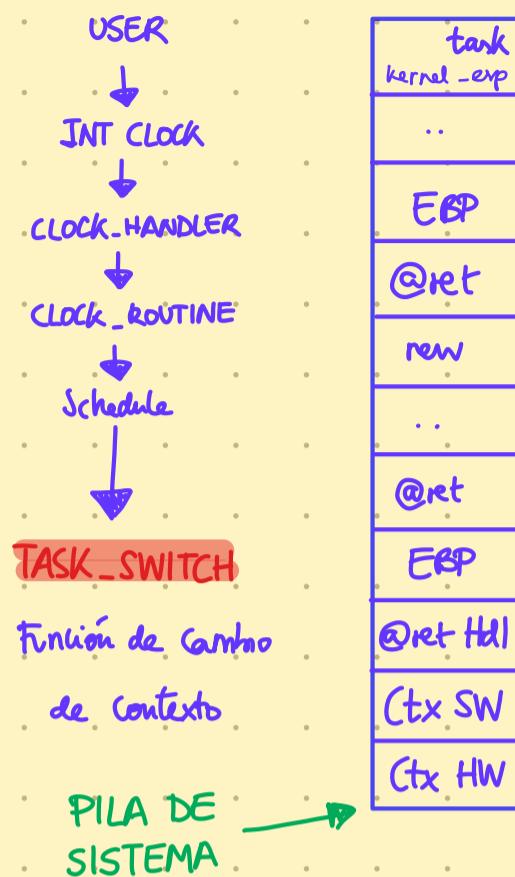


En el union empiezan todos en offset=0 y tiene el tamaño TOTAL del ELEMENTO más grande

* El tamaño HA DE SER MÚLTIPLA de tamaño de página (0x xxxxx000)

* Hay que asegurarse de que la pág de sistema no rehiente el task_struct. En ese caso modificaremos el tamaño de la stack añadiendo tamaño (múltiplo de páj)

¿Cómo funciona un cambio de contexto (cambio de proceso)?



```
void task.switch (union task union *new)
```

```
{ push EBP
```

```
    movl ESP, EBP
```

```
    set Q3 (new->task.TP),
```

```
tss.exp0 = &new->stack[1024];
```

```
current() = task.kernel-exp = ebp;
```

```
exp = new->task.kernel-exp, → Nuevo EBP al nuevo proceso
```

```
popl EBP ] → IMPORTANTES !!
```

```
ret
```

Borramos todo el TLB y ponemos el nuevo espacio de @Loop.

Cambiamos de proceso, ahora CURRENT() = NEW()

* OBSERVACIÓN: En modo usuario la pila de sistema está VACÍA

* Para que un proceso pueda entrar al task-switch debe tener encima de la pila un EBP y una dirección de retorno

¿Cómo se crea un proceso?

WIN No hay jerarquía de procesos (recomitamos un sistema de ficheros)

Linux sys-fork() → Devuelve -1 en caso de error, PID del hijo al padre y 0 al hijo

↳ No tenemos parámetros, por tanto no los comprobamos

↳ COMPROVACIÓN DE RECURSOS. Asignar task-union, TP, mem_física & otras estructuras

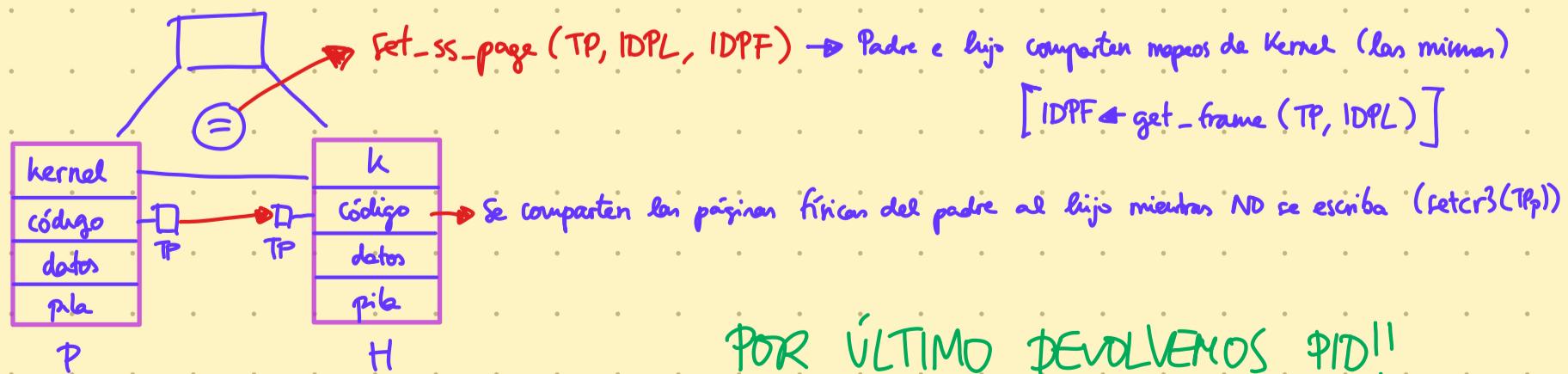
1 Inicializamos Contexto de ejecución
(task-union padre = task-union hijo)

2 Asignamos un PID al proceso hijo

3 Copiar memoria

4 Preparar ejecución del hijo

5 Encolar ready-paquete



¿Cómo lo gestiono desde el task switch?

list - &D → list

PADRE → syscall_handler → sysfork() → ready_queue → syscall_handler → user → clock_handler → clk_routine →
→ Scheduler → task_switch → syscall_handler → user

PROBLEMA 1: Interrupciones HW mientras estamos en modo sistema (hay en día ya solucionado)

PROBLEMA 2: Gestión de eax en pilas de sistema padre-hijo (quiero copiar el contexto SW igual pero eax con valores diferentes por el fork) → INT RET-FROM-FORK() { return 0; }

- * Necesito que la PRIMERA ret que el proc entre a RAM ejecute el ret-from-fork() antes de saltar a modo usuario
- * He de poner eax a 0 en el fork del hijo
- * Para ello, pushemos @ret-from-fork y añadimos un valor falso de EBP porque ya no lo necesitamos



Primer proceso: INIT

Es el único proceso que se crea en tiempo de Boot. Le tenemos de asignar su task-union, tabla de páginas e inicializarla.



- * C3-TP. Apuntar a TP de init
 - * tss.esp0. system_stack
 - * MSR[0x175]: system_stack
- { Hacerlo current()

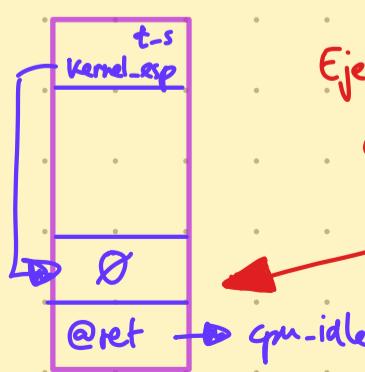


Proceso IDLE En estos rato hace un while(1), pero en SO's actuales es un proceso MUY complejo

- * NUNCA saltará a modo usuario
- * Preparar contexto de ejecución
- * NUNCA va encolado en la col de ready
- * Task Union & Tabla de Páginas

```

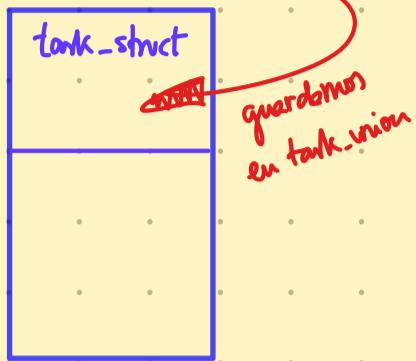
void cpu-idle()
{
    sti;
    while(1);
}
  
```



DESTRUCCIÓN DE PROCESOS

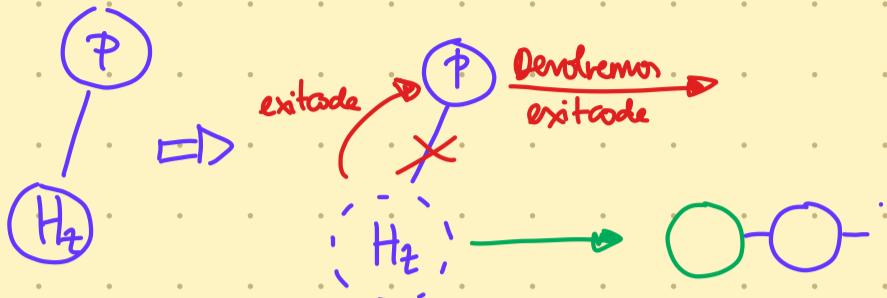
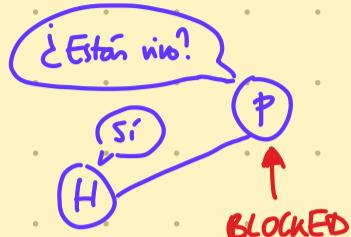
IMPORTANTE SABER IMPLEMENTAR!!

`int exit(int exitcode);`



`int wait(int *status);` → Para implementar esto faltaba una

COLA DE ZOMBIES



* Si algun hijo ha terminado → Debebe exitcode del hijo y encota el tank_union en la frecuencia

* Si tiene hijos pero ninguno está en estado de zombie → Bloqueamos proceso padre (BLOCKED)

- * Todos los procs son hijos de alguien (no liberamos el tank-switch)

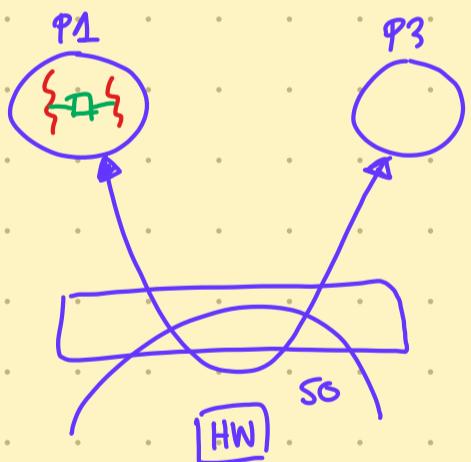
THREADS

(desde diapo 89 tema 4) → LEERSE R.R & POLÍTICAS DE PLANIFICACIÓN

* NO COOPERATIVOS → \$ ls

* COOPERATIVOS → \$ ls | less

Los procesos cooperativos tienen que estar en comunicación constante (sincronizados) para funcionar correctamente. Lo hacen a través del SO



* Los diferentes trozos de código que son independientes y del mismo proceso se llaman THREADS

* Un thread será mi UNIDAD MÍNIMA DE PLANIFICACIÓN

* He de decidir QUÉ RECURSOS necesita el thread y QUÉ COMPARTIR con el thread

Recursos NO compartidos

- * TCB (Thread Control Block)
- * TID (Thread ID)
- * TLS (Thread Local Storage)
 - errores

Recursos compartidos

- * EIP
- * Pila de usuario
- * Registros
- * Pila de sistema

- * Memoria
- * Sistemas de ficheros

¿Cómo creamos threads? (diapo 99)

`pthread_create(void *(*func)(void *), void *param);`

↳ [1] Param checking

↳ [2] Recursos → TCB → WINDOWS ETHREADS

LINUX UNION_TASK_UNION

→ USER STACK

→ SYSTEM STACK (En Linux NO pq esta en task-union)

→ TLS

↳ [3] Inicializar los campos

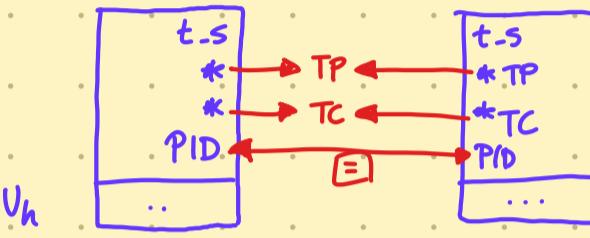
▪ $\text{t_s} \rightarrow \text{TP}$

▪ $\text{t_s} \rightarrow \text{TC}$

▪ Copiar $T_{U_p} \rightarrow T_{U_h}$

▪ INIT USER_STACK

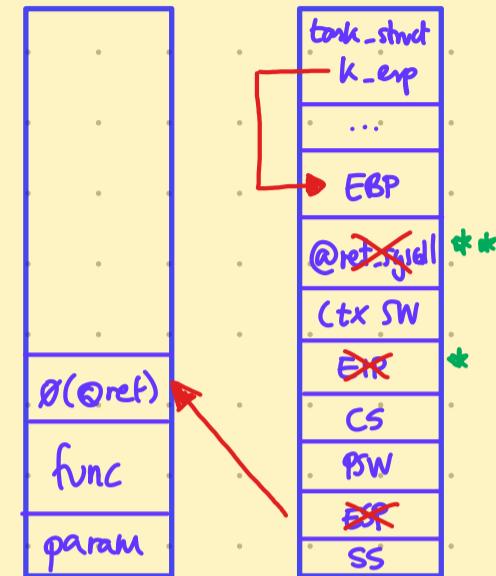
▪ Preparar EXEC



READY_QUEUE

(Relación EXPLÍCITA entre threads y procesos en Windows)

USER_STACK SYSTEM_STACK



* pthread_wrapper
** @ret_clock_muhre

int pthread_wrapper(void *(*func)(void *), void *param)

}

func(param);

pthread_exit();

}

La última instr para antes de salir

pthread_exit();

Para OBLIGAR a usarlo tienen de implementarlo

¿Qué malo puede pasar? → DEADLOCK

¿CÓMO EVITARLOS?

NO ENTRA

EN EL

PARCIAL

* Mínimo de 2 recursos no compartidos (Mutual Exclusion)

* Un thread consigue un recurso y espera por otro (Hold & Want)

* Solo el thread puede liberar sus recursos (No preempción)

* Hay de haber 2 o más flujos donde dependen recursos entre ellos (Circular wait)

* Tener recursos compartidos

* Poder quitarle un recurso a un thread

* Poder conseguir todos los recursos necesarios de forma anómala

* Ordenar peticiones de recursos (tener que conseguirlos en el mismo orden)

ROUND ROBIN

* Cada proceso tiene un TEMPORIZADOR (quantum) asignado por el SO

* El proceso a planificar se elige según FCFS → El sistema asigna un nuevo quantum

* La elección del quantum es muy importante → PEQUEÑO · muchos cambios de contexto, GRANDE · se approxima a FCFS

Constante o dinámico

También puede abandonar por E/S

El proceso abandona la CPU cuando acaba

OPTIMIZACIÓN TASK SWITCH

```
t_s(new) {  
    if (current() -> PID != new -> PID) setcr3(new -> TP); // IDEA: Fortar t_s entre threads  
    setcr3(new -> TP); // Problema de rendimiento. Podemos agrupar los threads de forma contigua en la coh de ready  
    tss esp0;  
    current() -> k-esp = esp;  
    esp -> (new -> k-esp);  
    pop esp  
    ret  
}
```

DUDAS CLASE

- * ¿Qué ocurre con la TP en una PF? ¿Qué hace el paginador? → Hemos de matar el proceso
(recienito saber la @ del error)
- * Hacer sys-exit
* Poner código de error
* Liberar PCB
* Fin mem virtual
- * Buscar si está en disco y no en TP, + violencia
Memoria virtual
-
- The diagram illustrates the flow of a page fault. It starts with a circle labeled "PAGER" with an arrow pointing to "SEGMENTATION FAULT". Below this, there is a memory map representation with a box containing a diagonal line pattern. An arrow points from the "SEGMENTATION FAULT" text to this box. Another arrow points from the bottom of the box to the text "PF.20, rwx ; U; P".