

→ ¿Qué pasos sigue un computador al iniciarse?

- (1) Resetea todos sus dispositivos (HW, periféricos...)
- (2) Comprueba que estén "pinchados" → CPU, Memoria, BIOS
- (3) Vuelca el contenido de la BIOS (Basic I/O's System) en una dirección específica de memoria (configurada por fabricante) ①

② Todos los discos duros booteables, su primer sector es el MBR (Master Boot Record) 512B que contienen código para bootear el sistema.

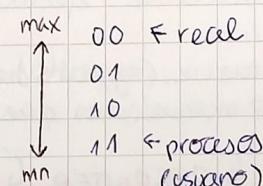
Hoy en día los OS son tan tachos que el MBR te manda a un Boot Loader, que es más extenso y permite que se cargue el sistema.

(4) Se carga el OS en el Kernel y se emplean a ejecutar instrucciones

- ↳ Inicializa estructuras HW
- ↳ Inicializa estructuras SW
- ↳ Inicializa todo el HW  
- (mas pasos, depende)
- ↳ Crea el proceso INIT  
↳ Es el primer proceso usuario (pid: 1) y es el que permite hacer todo. Se crea en tiempo de Boot porque es el que resalta la máquina

- ↳ Salta a modo protegido  
- activa los niveles de privilegio ③  
- activa la paginación de memoria
- ↳ Salta a modo usuario y al proceso INIT

proc



proc

HW

④ → Para que el usuario no toque cosas que pueden "romper" la máquina

NIVELES DE PRIVILEGIO La palabra de estado del procesador PSW (32b)

contiene 2 bits que son el CPL (Current Privilege Level): XX

Cada instrucción tiene tambien un nivel de privilegio (IPL) y al trazar el fetch se compara el CPL con el IPL si  $CPL \leq IPL \rightarrow \text{proceed}$ .

⑤ 1 → Para ejecutar cualquier código, debe ser volcado en memoria primero.

¿Cómo se ve en ensamblador?

call foo(3, 4);  
...

9. Hace la función, deja resultado en eax

1. Se empilan los parámetros de derecha a izquierda ←

2. Hace el call a la función

3. La función crea un link dinámico

5. Devuelve al contexto y a la función que la ha llamado

(main)

push \$4  
push \$3  
call foo

foo:

push ebp;  
ebp ← esp  
... # ebp + 8 + 4 · i → parámetro i de la función.  
eax ← res  
esp ← ebp  
pop ebp  
ret

## TEMA 2: LLAMADAS A SISTEMA

Hay 3 formas de cambiar lde nivel de privilegio, en x86

- ▶ Excepciones : cuando funciona mal el código de usuario, las lanza el propio S.O.
- ▶ Interrupciones hardware : ejecuta código del S.O (más privilegios)
- ▶ Interrupciones software (syscalls) : tiene que saltar a nivel 0 de privilegios. Hay 3 instrucciones que hacen eso
  - ↳ INT, ↳ SYSENTER ↳ SYSCALL

¿Cómo funciona "INT \$num"?

1. Accede a la Interrupt Descriptor Table : IDT \*1

Donde cada posición contiene : → la @ donde empieza el código  
                                  ↓ que handlea esa syscall  
                                  que nivel de privilegios requiere.

- \*1: La IDT es un vector de 256 posiciones (0 a 255), estructura HW
- ↳ De la posición (0 a 31) → Excepciones
  - ↳ De la posición (32 a 47) → Ints HW
  - ↳ De la posición (+48 a 255) → Ints SW

Cada vez se inicializa en una dirección diferente, cada y su primera @ está en el registro r IDT

- El \$num nos dice que posición acceder  $\rightarrow rIDT + $num \cdot size$   
El nivel de privilegio es, desde qué nivel se puede acceder  
(normalmente desde el 3 se puede).
  - Compara el CPL\*<sup>1</sup> con el PL de la entrada del IDT
  - El procesador accede a la Global Descriptor Table (GDT)  
que también se sabe donde está por un registro rGDT.
- Aquí, hacemos un chequeo para ver si la dirección dada por la IDT es válida y efectivamente señala a código de sistema.
- Entonces accede a la (TSS) Task State Statement, tabla que guarda el estado de la tarea actual. También tiene un rTSS

Hay un campo: esp()  $\rightarrow$  contiene la @ de la base de la pila del sistema (\*)<sup>2</sup>

program counter	EIP
	CS
stack pointer	PSW
	ESP
	SS

- El procesador guarda el contexto hardware del (proceso) la pila de usuario del código que estaba ejecutando antes de hacer la INT.
- El procesador cambia el CPL en la PSW a 00.  
También cambia SS:ESP para que apunte a la cima de la pila de sistema y CS:EIP para que apunte al código de sistema.

Entonces ya hemos cambiado de contexto y listos para ejecutar la instrucción (código SO) de la INT \$num.

- (\*)1: Current privilege level       $\hookrightarrow$  pila de usuario
- (\*)2: Todos los procesos (todos) tienen 2 pilas       $\hookrightarrow$  pila de sistema  
esto se hace para que los usuarios no puedan modificar de forma fatal la pila de sistema
- (\*)3: Página 1: En el paso (4)  $\rightarrow$  Inicializar estructuras hardware
  - $\hookrightarrow$  la IDT; GDT no están hardcodeadas en una dirección específica, se randomiza por seguridad.
  - $\hookrightarrow$  la TSS se inicializa en tiempo de boot pero se va modificando

Realmente, desde modo usuario, no haces la llamada a sistema, sino que (llamamos a una función WRAPPER, y esa función si que llama al sistema. Se implementan en ensamblador

Se hace para que el código sea portable y compatible.

En Linux: libso.so y Windows: ntdll.dll

1. La función llama a la función de sistema que primero salta al syscall - handler()

ebx	} CX SW
ecx	
edx	
esi	
edi	
ebp	
eax	
ds	
es	} CX HIW
fs	
gs	
eip	
cs	
psw	
esp	
ss	

2. El handler guarda el contexto de software
3. Hace unos chequesos (\*)<sup>1</sup>
4. Hace CALL a la función real de sistema  
se llama sys\_fución (ex: syswrite)

Intentan que la cabecera sea la misma que la del wrapper (mismos parámetros)

- ↳ La función
1. chequea los parámetros
  2. chequea si tiene recursos para hacerlo
  3. Hace lo que se pide

→ Hoy en día hay más servicios que las 256 entradas en la IDT, para concentrar el código y hacerlo más seguro hacen que haya solo una entrada a la IDT.

Siempre salta el mismo handler siempre que hace los mismos chequesos (\*)<sup>1</sup>

Se pone en el %eax el número de servicio y se busca en la syscall-table, una estructura software que contiene tantas entradas como servicios.

↳ call \*syscall-table (offset, position, size) <sup>offset</sup> %eax <sup>position</sup> %eax <sup>size</sup> %eax

## TEMA 3: GESTIÓN DE MEMORIA

Originalmente no se traducían las direcciones (@) de memoria. Se usaban las @ físicas y solo se podía ejecutar un ~~programa~~ programa a la vez.

- Luego se añaden 2 regjs en una MMU en CPU,
- ↳ uno con @ física del inicio del proceso
  - ↳ uno con el tamaño que ocupa en memoria



Así se traducían los offsets, y no se toca código que no toca.

! QUEDABA FRAGMENTADOS espacios libres en memoria!

- ↳ Se intenta implementar vía software: Reubicación dinámica (redistribuir los programas en memoria para que estén consecutivos).

Funciona superbien hasta que los programas empiezan a crecer, queremos ejecutar varios a la vez...~

Por fin se modifica la MMU por una TLB (translation lookaside buffer) porque creamos páginas & frames (de una medida fija), que se guardaran de manera no consecutiva en memoria. Asociamos cada una de estas pl's

• Página lógica (pl)

• Página / frame físico (pf)

con una pf con la TLB.

Queremos traducir en el menor tiempo posible:

Un TLB es una estructura de unas 1024 ~ 2048 entradas, que guarda las parejas de asociaciones pl  $\leftrightarrow$  pf. ( $(id(pl) \leftrightarrow id(pf))$ )

- \* Si una @ tiene 32 bits  $n = \log_2(\text{medida página})$   $\rightarrow 32 - n$  → de más peso son el (id) de página  
→ los n bits de menos peso son el offset dentro.

id página	offset	ejemplo páginas 4K
1211	0	$\log_2(4096) = 12$ bits

- Dentro de un solo TLB no van a caber todas las páginas que necesiten los procesos  
↳ cada vez que se crea un proceso se crea una tabla de páginas (tp) con tantas entradas como páginas lógicas necesita el programa

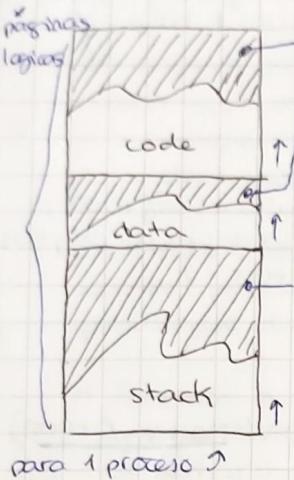
La @ inicial de la Tabla de Páginas del proceso actual, está en el registro %CR3

Es decir, hasta ahora

→ HIT TLB: La @pl está en el tlb y se traduce

→ MISS TLB: La id pl está en la tabla de páginas, y se lleva al tlb en el siguiente ciclo → HIT

Cuando el sistema operativo monta la imagen del proceso en memoria (ipm)

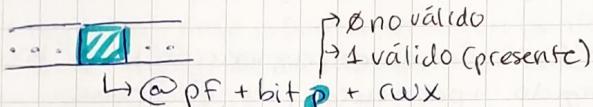


Detrás dejan 'gaps' entre secciones, en caso de que esas secciones crezcan, (datos con malloc, stack de manera automática....)

Como estos gaps, pese a tener id pl, no tienen asignadas páginas físicas (hasta que se usan)

↳ Se añade bit de presencia 'p' a cada entrada de la tabla de páginas

\* También añadimos bits de permisos 'rwx'

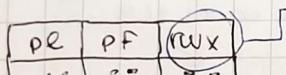


↳ @pf + bit P + rwx

Entonces, en caso de MISS TLB, → Mira @pl en la tabla de páginas, si  $p = \emptyset$  dejas de intentar → acceso incorrecto → PAGEFAULT ✗

También se le añade los 3 bits de 'rwx' a cada entrada del tlb:

↳ Si falla por permisos incorrectos → general protection fault



Memoria virtual → Extender memoria física con el disco.

\* El procesador debe tener los datos en memoria física para poder usarlos.

→ Cambiamos el significado del bit  $p = \emptyset$  → no tiene pf asignada

↳ está en disco (mem virtual)

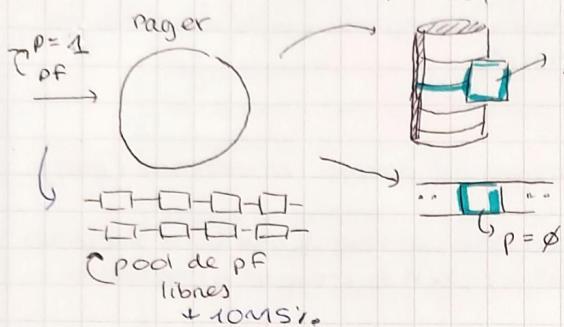
✗ El pagefault que salta va a el PAGINADOR / PAGER (proceso de sistema) que atiende los pagefaults para resolverlos

El paginador siempre intenta que un 10v15 y. esté libre en todo momento en mem física. Estudia como está distribuida la memoria, junto al SO y procesos para saber como distribuir la memoria virtual.

Es el que determina el grado de multiprogramación: cuantos procesos se pueden tener en ejecución a la vez.

Cuando el ordenador está en idle el paginador aprovecha para hacer swap-out de varios procesos o partes de procesos.

6) El swap-out: corre la tabla de páginas buscando alguna entrada con  $P = 1$  (que están en mem física) y mueve el contenido a disco, apuntando el proceso del que viene y su id pL.



Coje la tabla de páginas y libera la página que ha movido + bit  $p = \emptyset$

MISS TLB 2

→ Entonces; si buscamos id pL en la tabla de páginas y  $p = \emptyset$ , salta el paginador, busca en disco y si está, se la lleva a memoria física y a la tabla de páginas → vuelve a hacer MISS TLB → se lleva la página al TLB → HIT :)

• El SO trabaja con @físicas o @lógicas?

4) En tiempo de boot (antes de activar paginación) → @ físicas (empreza 0x000)  
4) (salto a modo protegido → paginación) → @ lógicas

Los procesos de sistema no tienen tabla de páginas, en cada ipm tiene una zona de Kernel en un pequeño rango. Todos los ipms apuntan a la misma zona de páginas físicas.

Evitamos que los procesos puedan acceder poniendo un bit de privilege level (esta vez solo 2 opciones  $\emptyset$  o 1), y también se pone en el TLB:

PLC(página lógica)	PF	rwX	$pL$ (privilege level)
n bits	n bits	(3bits)	(1bit)

Es decir, todas las tablas de páginas de todos los procesos, tienen mapeado el kernel para que se pueda traducir siempre.

## TEMA 4: GESTIÓN DE PROCESOS

El SO identifica procesos a través de el PCB - Process Control Block  
Todos los procesos vivos tienen un PCB.

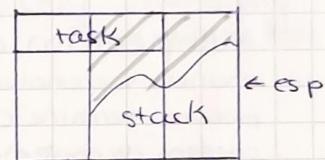
- \* En Windows  $\rightarrow$  EPROCESS, apuntado por el registro  $\&.GS$  (no se usa para otra cosa)
- \* En Linux  $\rightarrow$  struct task\_struct; (contenido en union task-union):  

```
union task-union {  
    unsigned long stack[1024];  
    struct task_struct *task;  
};
```

En principio en una unión se van machacando valores, pero el programador del SO evitó que la pila llegue a machacar la struct tasks (la pila crece des de abajo).

$\rightarrow$  Sabemos que el registro  $\&.esp$  apunta en algún punto de la página, y sabemos que la  $\&$  inicial de la task struct estará alineado con el tamaño de la página (4096)

$$@task struct [0] = \.esp \& 0x FFFF F000$$

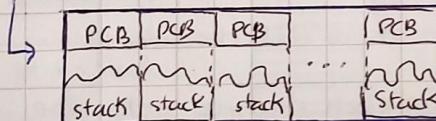


El programador evita que la cima de la pila toque las direcciones más bajas (la pila crece des de las más altas), para no machacar el task struct.

- \* Existe una función current()  $\rightarrow$  devuelve puntero \*union task-union del proceso actual.

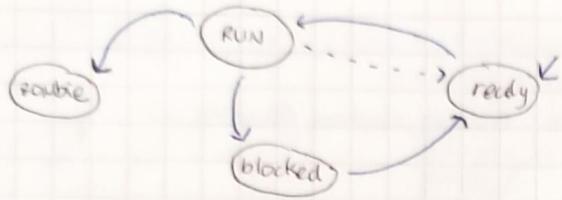
### ESTRUCTURAS DE LA GESTIÓN DE PROCESOS

- (1) Vector de PCBs (estático en zEOS, dinámico en sistemas modernos)  
 $\hookrightarrow$  se llama 'task' en zEOS.



Contiene todos los PCBs que puede tener el sistema, libres o no.

- (2) [OPCIONAL] Freequeue  $\rightarrow$  Contiene un acceso rápido a los PCBs dentro del vector que están libres. S
- (3) Readyqueue  $\rightarrow$  Listado de procesos candidatos a usar la CPU, ya sean nuevos o no.  
Un proceso se desencola de la readyqueue



Si un proceso no está encolado en ningún sitio, probablemente sea current

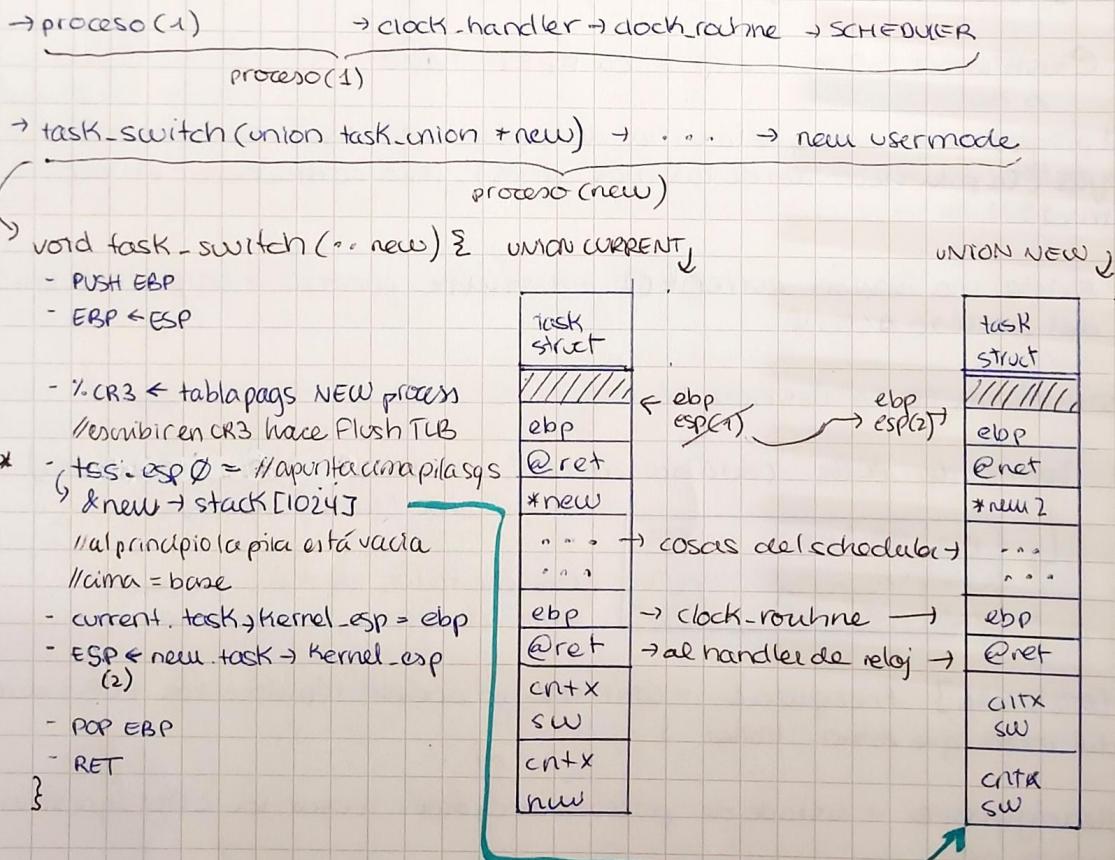
\* bloqued: muchas colas (hasta miles) con todos los procesos.

¿Cómo pasamos RUN ↔ READY?

↳ Multiplexación de procesos: no queremos ejecutar los procesos de manera secuencial ya que tendríamos que esperar siempre que un proceso termine (o muerto). → los "paramos / resumimos" con cambios de contexto, y queremos que sean lo más rápidos posibles.

Usando el sistema Round Robin: definimos un quantum en CPU y los vamos echando.

\* timeline cambio de procesos:



## CREACIÓN DE PROCESOS

- en windows → create process : crea des de \$
- en : → int fork() : crea procesos a través de la copia del proceso y si hace falta mutar

## INT SYS\_FORK()

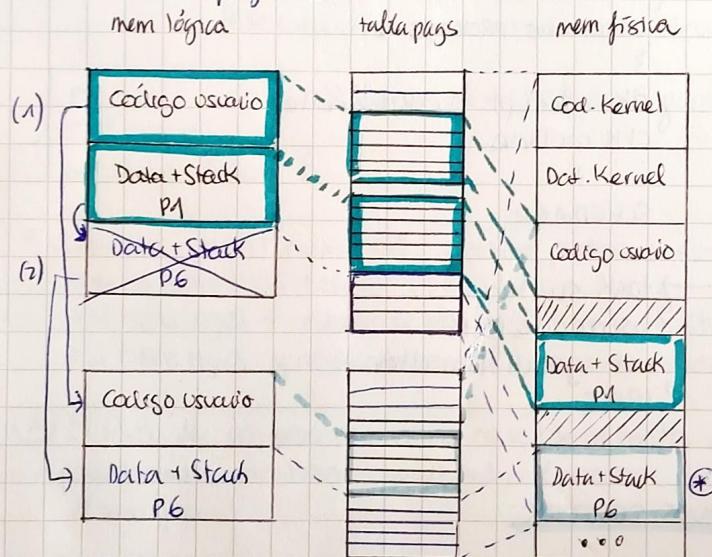
1. Obtiene PCB libre (de la Freequeue)
  2. Asigna PID: generado de manera pseudocaleatoria
  3. Hereda datos de sistema: copia task-union (PCB + Pila) (\*)
  4. Asigna directorio (tabla de páginas) [vacía]
  5. Hereda datos de usuario (\*)
  6. Actualiza task-union hijo (\*)
  7. Inserta proceso en la cda de READY
  8. Devuelve PID del nuevo proceso creado y \$ al hijo.
- } Si no quedan recursos: devuelve error y deshace todo

pointer → task union padre / hijo  
 ↓                           ↓

3 → Hereda datos del sistema: COPY\_DATA (padre(), hijo(), task-union-size)

5 → Hereda datos de usuario: → [CODE] [DATA] [STACK]

(1) [CODE] → Hereda: Compartidos entre padre/hijo → actualizamos tabla de páginas



(2) [Data + Stack]: Privados  
 ↳ creamos zona de mem nueva para el hijo (tan grande como él)

↳ Buscamos frames libres en la memoria logica para guardar las páginas de [data+stack] del proceso hijo

↳ Permitimos temporalmente al padre acceso a los frames del hijo,

↳ Copiamos los datos del padre al hijo

→ Quitamos el acceso del padre a los frames del hijo.

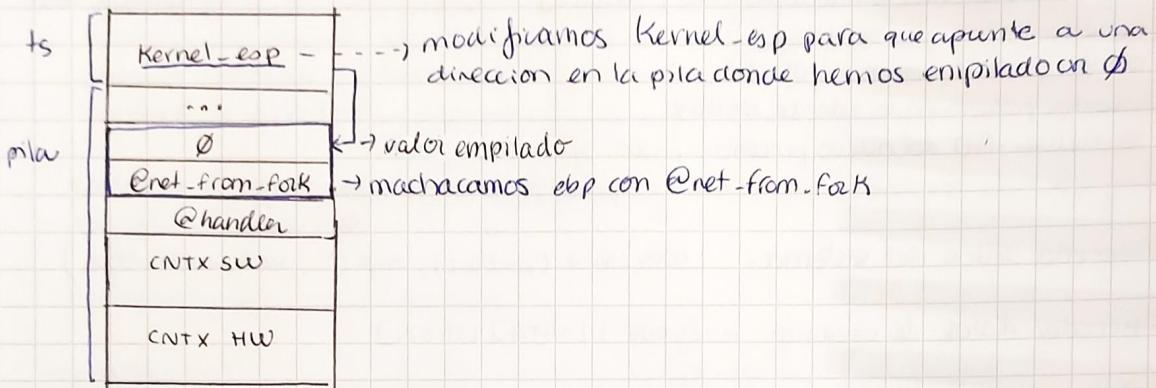
\*) Data + Stack P6 inicialmente es solo una copia de Data + Stack P1

6 → Actualizar task

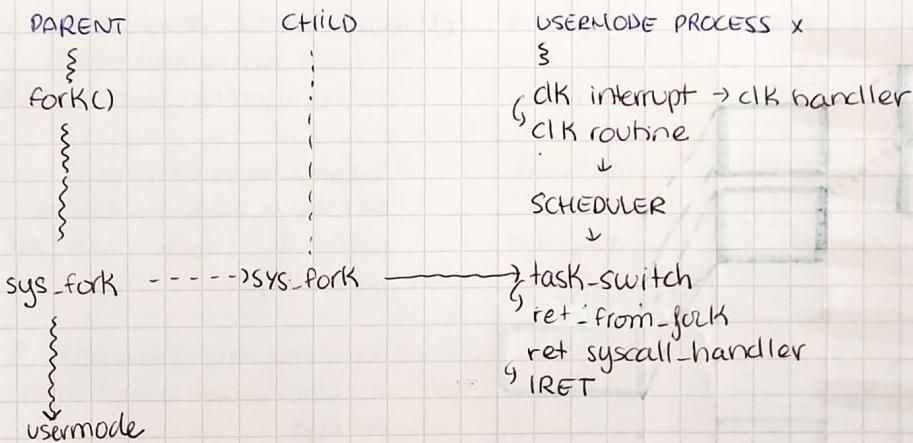
→ pid + preparar para cambio de contexto → debido a que el task-switch no distingue entre procesos que ya han estado en CPU o que se acaban de crear, debemos preparar el resto de contexto

- Kernel-esp , en ebp + dirección de retorno en la pila
- ↳ Debido a que sabemos exactamente como está la pila (copia exacta a la del padre)
- ↳ Dónde retorna el hijo? → función especial ret\_from\_fork() (gestiona la devolución de resultado del hijo: eax = φ ... )

modificamos la pila de sistema del hijo:



TIMELINE :



#### ► PROCESOS QUE NO SE CLEAN A TRAVÉS DE FORK:

##### • INIT: Primer proceso usuario:

- ↳ Reserva PCB + Asigna PID + Inicializa espacio de direcciones.
- ↳ Crea la imagen del proceso.
- ↳ Hace que init sea current
  - ↳ %CR3 ← Tabla páginas init
  - ↳ +SS, ESP φ ∈ @base pila sistema init + MSR[175]

- ↳ Prepara la pila de sistema para que pueda salir a modo usuario

ESP: CS	→ 1 instrucción usermode
ESP: SS	→ @user stack

- ↳ Return GATE para iniciarla  
(no entra des de el task switch!)

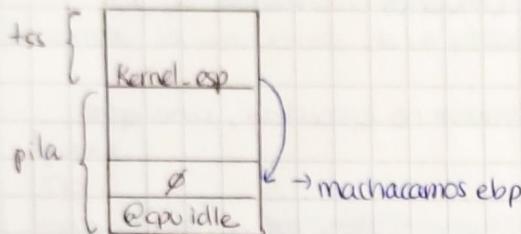
\* IDLE: Proceso que solo ocupa la CPU si no hay otro proceso candidato. Solo se ejecuta en modo sistema. Nunca vuelve a modo usuario.

\* No forma parte de la ready-queue. variable global idle\_task

↳ 1. Reserva PCB + inicializa variable

↳ 2. Asignar PID( $\emptyset$ ), + tabla de páginas: solo necesita el Kernel mapeado

↳ 3. Prepara contexto para cuando la política de planificación lo ponga en ejecución.

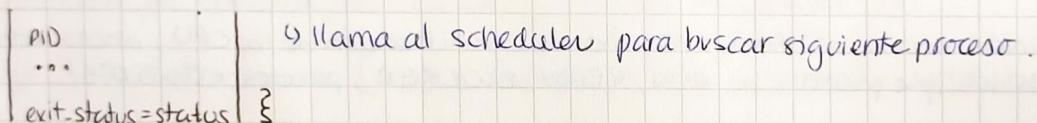


\* DESTRUCCIÓN DE PROCESOS Y ESTADO ZOMBIÉ: cuando un proceso termina, ya no se va a ejecutar más.

↳ int sys\_exit(int status)

↳ libera todos los recursos asignados excepto el task-union

↳ se crea un campo en el task\_struct (PCB)



\* LLAMADA WAIT / WAITPID: Proceso padre se sincroniza con el fin de sus hijos.

→ Si no tiene hijos → error (<0)

→ Si tiene hijos → estado de finalización (exit\_status) + PID + libera PCB hijos

→ Si tiene hijos y no acaban → se bloquea esperando finalización

(\*) El PCB de un proceso zombié no se libera hasta que su padre hace waitpid, si el padre muere sin hacer waitpid, el proceso init "adulta" a sus hijos y hace el waitpid.

bucle copia de datos de fork:

int N = num frames que ocupa region datos

int P = inicio\_region\_disponible mem logica padre

int INI = inicio región datos' pache

frames[N]

for ( int i =  $\phi$ ; i < N; ++i ) {  
 pt pacie  
 ↓  
 id logica  
 ↓

set\_ss\_page ( $\downarrow$  get\_pt(current()),  $(p + i * \text{PAGESIZE}) \gg 12$ , frames[i])  
 $\downarrow \text{src}$   $\leftarrow \text{loc} +$   $\text{size}_2$ )

copy-data((void \*) (i \* PAGE\_SIZE), (void \*) (p + i \* PAGE\_SIZE), PAGE\_SIZE)

`del_ss_page(get_pt(current), (P + i * PAGESIZE))` (2)

3