SOA - Resums

Alvaro Moreno Ribot

December 24, 2021

Contents

1 Syscalls **3**

1.1 Las llamadas al sistema desde punto de vista hardware 3

1.2 Boot 5

1.3 Punto de vista Software 5

1.3.1 Paso de parámetros de un wrapper a una rutina de servicios. 7

1.3.2 ¿Cómo se hace en Linux? 8

1.4 Interrupciones y excepciones 11

2 Sistemas de memoria **12**

2.1 La fragmentación 12

2.2 El TLB 14

2.3 Bits extra del TLB 16

2.4 Paginación 18

3 Procesos **21**

3.1 El ciclo de vida de un proceso 21

3.2 La pila de un proceso 21

3.3 Multiplexación de procesos 22

3.4 La llamada `fork()` 27

3.4.1 Implementación de `fork()` 27

3.5 `sys_exit()` 34

3.6 Los procesos `idle()` e `init()` 34

3.6.1 `idle()` 34
3.6.2 `init()` 36
3.7 Planificación 36
3.7.1 Políticas y algoritmos de planificación 37
3.8 Threads 39
3.8.1 Race conditions 40

1 Syscalls

Existen 3 métodos para cambiar de nivel de privilegios:

Excepciones. Se generan cuando algo en el **código de usuario** va mal (divisiones entre 0, acceso memoria incorrectos, etc.) - son asíncronas, ya que no se sabe cuando van a ocurrir.

Interrupciones Hardware. Se generan cuando en el hardware pasa algo. Por ejemplo se mueve ratón, el disco duro acaba de leer, etc. El hardware se comunica con el Sistema Operativo a través de interrupciones (también asíncronas, pues no sabemos cuando van a ocurrir).

Excepciones Software. También llamadas *syscalls*. Se generan cuando el usuario le pide cosas al sistema (abrir ficheros, escribir, etc.). Son síncronas, ya que el usuario las provoca y sabe cuando van a ocurrir.

1.1 Las llamadas al sistema desde punto de vista hardware

Normalmente el usuario está ejecutando código y en algún punto se produce una llamada al sistema o una excepción y esta llega a la CPU.

En este momento la CPU recibe una petición de cambio de privilegios. Para esto, la CPU va a acceder a una tabla llamada **IDT** (Interrupt Descriptor Table) que tiene 256 entradas. Cada una de las entradas contiene una dirección de memoria y un nivel de privilegios.

La IDT está dividida en secciones:

- 32 entradas para **excepciones**.
- 16 entradas para **interrupciones hardware**.
- El resto son para **interrupciones software**.

La dirección que se encuentra en la entrada de la IDT contiene la rutina de servicio que procesará el evento y es una dirección de Sistema Operativo. El nivel de privilegios nos dice el nivel necesario para ejecutar esa rutina de servicio. Para las interrupciones y excepciones hardware el nivel de privilegios siempre será 0 (PL = 0). Para las interrupciones software el nivel de privilegios es 3, ya que provienen del usuario.

¿Para qué sirve este nivel de privilegios? Para que el usuario no pueda lanzar una interrupción o excepción haciendose pasar por el hardware.

La IDT es un vector en memoria y la estructura la define el hardware. *¿Cómo sabe el procesador dónde está la IDT?*

- Se podría tener un registro que apunta al inicio de la tabla.
- Se puede encontrar en una posición fija de memoria.

Ahora la CPU debe validar la dirección de memoria para saber si es correcta y puede saltar a esa dirección para ejecutarla. Así que coge esa dirección y la pasa por la **GDT** (Global Descriptor table), una tabla muy compleja que a grosso modo tiene un mapa de la memoria de la máquina y nos dice qué direcciones están asignadas y a quién pertenecen (usuario, sistema...). La CPU hace un check de la dirección con la GDT para validar que el usuario o malware no ha manipulado la IDT.

Una vez se han hecho los checks con la GDT, el procesador debe ejecutar la rutina de servicio. En los registros **CS** y **EIP** (CS:EIP) se encuentra la dirección de la siguiente instrucción que se debe ejecutar del código de usuario. Además el usuario tiene una pila, dónde se guardan los frames de activación. Para saber dónde está la cima de esta pila de usuario se usan los registros **SS** y **ESP** (SS:ESP).

El *registro de segmento* (**C**ode **S**egment y **S**tack **S**egment) nos dice dónde está el bloque en memória y EIP y ESP son relativos a los registros de segmento, ya que nos marcan la posición dentro del segmento.

Entonces tenemos que la CPU sabe: dónde se encuentra la cima de la pila del usuario y la siguiente instrucción a ejecutar (de usuario). Ahora la CPU va a saltar a código de sistema para ejectuarlo.

Para hacer esto, una vez hecho lo anterior, se va a acceder a una nueva estructura llamada **TSS** (Task State Segment), que es una estructura en memoria que nos da información sobre el proceso que se está ejecutando actualmente en la CPU. La CPU va a mirar el campo **ESP0** de la TSS, que contiene la dirección de la cima de la pila de sistema para ese proceso.

Cada proceso tiene dos pilas: Una para el modo usuario y otra pila aparte que se usa exclusivamente en modo sistema. Eso se hace ya que el sistema no se fia *nunca* del usuario para evitar posibles problemas de seguridad. Si tengo 200 procesos tengo 400 pilas (200 usuario y 200 sistema).

ESP0 tiene la dirección de la cima de la pila de sistema para ese proceso que se está ejecutando actualmente en la CPU. Ahora el procesador va a guardar dentro de la **pila de sistema** la info que necesita el procesador para poder volver a modo usuario. *¿Cuál es esa información?*

- Dónde está la pila de usuario.
- El estado del procesador.
- La siguiente instrucción a ejecutar.

*Imágen del contexto hardware en la pila de sistema.

A esto le llamamos el *contexto hardware.* Una vez se ha guardado, el procesador ya puede empezar a ejecutar código de sistema.

En CS:EIP (que apunta la siguiente instrucción a ejecutar) se va a guardar la dirección a la primera instrucción de la rutina de servicio (código de sistema), para que sea la siguiente instrucción a ejecutar para el procesador. En PSW:PL se va a guardar un *00* indicando que está en *máximos privilegios.*

Ahora ya se puede pasar a ejecutar la rutina de servicio de la interrupción con la pila de sistema y máximos privilegios. Esto lo hace todo el procesador sin ninguna acción del SO.

1.2 Boot

En tiempo de boot, el Sistema Operativo debe inicializar las estructuras (IDT, GDT, TSS). Estas estructuras solo se pueden modificar con el máximo nivel de privilegios, por lo que solo lo puede hacer el SO.

1.3 Punto de vista Software

Cuando nos encontramos en modo usuario y en algún punto queremos hacer una llamada al sistema (por ejemplo un `write(fd,*buffer,tambuff);`)

Cuando usamos el `write` realmente no hacemos una llamada al sistema porque realmente es una función. En Linux se encuentra en *libc.c* y en Windows en *kernel32.dll* (librerías del sistema). Dentro de estas librerías se encuentra la implementación de estas funciones. Así ***write*** es una función implementada en las librerías del sistema y dentro de esta función es dónde se hace la llamada al sistema.

A esto se le llama ***wrapper***. Un *wrapper* oculta el código realmente necesario para ejecutar la syscall. La ventaja de usar *wrappers* es la portabilidad que nos dan; en C siempre podemos usar la función write independientemente del Sistema Operativo, luego se va a comunicar con las librerias del sistema que cada una tendrá su propia implementación.

No todos los Sistemas Operativos funcionan igual, por eso debemos tener una forma de traducir el **write** estándar a una syscall. El *wrapper* nos asegura la compatibilidad del código de usuario con cualquier sistema a través de los **wrappers estándar** (posix).

Estos *wrappers* se programan en ensamblador. Para entrar al nivel de máximos privilegios en ensamblador tenemos dos instrucciones: **INT** (más completa) y **SYSENTER** (más rápida).

La macro de ensamblador **INT** se le debe pasar un parámetro que se define el número de la entrada de la IDT que se debe utilizar para saltar al modo de máximos privilegios (SO). Al ser interrupciones software solo usaremos el rango

de la IDT que se corresponde con las interrupciones software. Si se intenta usar una interrupción Hardware o una excepción, el procesador no nos va a dejar, ya que venimos de un nivel de privilegios de usuario (PL=3).

Existen 3 formas de trabajar con la IDT:

Una entrada de la IDT por servicio. Esto se usa en sistemas embeded con pocos servicios o Sistemas Operativos pequeños porque esto nos va a limitar el número de servicios disponibles al número entradas que hay en la IDT.

Un conjunto de servicios por entrada en la IDT. Por ejemplo; la entrada 35 de la IDT corresponde a todos los servicios del disco. De este modo el parámetro de INT nos va a dar el número de conjunto y en el registro **%eax** encontraremos el número de servicio. De este modo podemos ofrecer un número de servicios ilimitado, pero con este método debemos tener un handler para cada uno de los servicios y cuantos más puntos de entrada al sistema más verificaciones debemos hacer y esto nos obliga a replicar mucho código. Este método no se usa.

Una única entrada en la IDT. Este es el sistema usado actualmente. Solo hay una entrada en la IDT y un único punto de entrada al SO. Con el registro %eax se indica el número de servicio. Por otro lado, existe una entrada para cada una de las excepciones e interrupciones hardware.

Una vez se ejecuta INT se salta al handler (`handler_syscall`) indicado en la entrada de la IDT que se ha pasado por parámetro. Una vez se salta al handler ya nos encontramos en modo Sistema Operativo. Existe **un único handler para todas las llamadas al sistema.** El handler se escribe en ensamblador y luego se pasa a la ejecución de funciones en C.

Los handlers se encargan del *context management*; es dónde se guarda toda la información necesaria para volver al modo usuario. Desde el modo Sistema Operativo, cuando queremos volver al modo usuario, necesitamos haber guardado antes toda la información necesaria para hacer este cambio de contexto de vuelta a usuario.

Por eso lo primero que va a hacer el handler va a ser ejecutar la macro **SAVE_ALL**, que guarda todo el **contexto software**, es decir, la mínima información necesaria para volver al modo usuario desde el punto de vista del Sistema Operativo.

Previamente se ha guardado el **contexto hardware** como ya hemos visto. La Tabla 1 nos muestra como se encuentra la pila con el contexto hardware y el contexto software.

EBX	**CTX SW**
ECX	
EDX	
ESI	
EDI	
EBP	
EAX	
DS	
ESI	
GS	
EIP	**CTX HW**
CS	
PSW	
ESP	
SS	

Table 1: Contextos Hardware y Software

El contexto hardware y software están guardados. Ahora se deben verificar los datos de usuario para asegurar que no vamos a invocar un servicio que no existe. Si la verificación es correcta se pasa a ejecutar la rutina de servicio. Como en un sistema operativo existen muchos servicios, no es eficiente hacer un `switch; case` de %eax con todas las entradas, ya que no sería para nada eficiente.

Para hacerlo se inicializa un vector llamado `SYSCALL TABLE` que tiene tantas entradas como servicios hay en el sistema operativo y dentro de cada una de las entradas se encuentra la dirección de la rutina de alto nivel escrita en C que atiende al servicio. Con %eax obtenemos el número de rutina que se corresponde con la `SYSCALL TABLE`:

`call xsyscall.table(0, EAX, 4);` dónde cada uno de los parámetros se corresponde con offset inicial, posición y tamaño de las posiciones respectivamente.

1.3.1 Paso de parámetros de un wrapper a una rutina de servicios.

Se puede hacer de 3 formas:

Pasar los parámetros a través de registros. Este método lo usaba Linux.

Pasar los parámetros con un buffer. Este método lo usaba Windows. Se usa la pila de usuario para pasar los parámetros al sistema operativo y se le indica dónde están estos parámetros dentro de la pila de usuario.

Forma mixta. Algunos por registros y otros por buffer.

1.3.2 ¿Cómo se hace en Linux?

Cuando un compilador compila un código siempre lo transforma de la misma forma. El frame de activación que crea siempre es igual (los parámetros de derecha a izquierda y la dirección de retorno). Para pasar los parámetros al sistema operativo lo haremos asignando cada parámetro a un registro de igual forma que se crea el frame de activación: `foo(ebx, ecx, edx, esi, edi, ebp...);`

En el wrapper (pondremos el *write* como ejemplo) se va a guardar cada variable con su respectivo registro como hemos visto anteriormente. Si se hace de este modo, cuando se llega al código de la rutina de servicio ya se puede trabajar sin hacer ningún paso extra, ya que estará guardado en el contexto software en ese mismo orden: **cuando creamos el contexto software estamos guardando el contexto y a la vez creando el frame de activación de la función de sistema:**

```
int sys_write(int fd, void *buf, size_t size){
    push ebp;
    ebp <- esp;
    VERIFICACIONES
    .
    .
    .
    esp <- ebp;
    pop ebp;
    ret;
}
```

ebp	
@ret handler	
EBX	**CTX SW**
ECX	
EDX	
ESI	
EDI	
EBP	
EAX	
DS	
ESI	
GS	
EIP	**CTX HW**
CS	
PSW	
ESP	
SS	

Table 2: Contextos Hardware y Software. El ebp de arriba la pila, apunta a la pila de usuario.

El wrapper permite que como programadores nos podamos olvidar de toda la parte de sistema y deja esta tarea al propio Sistema Operativo. En %eax se pone el valor de retorno de la llamada al sistema. Una vez se ejecuta el *ret* se vuelve al handler de la syscall y ahora, desde el handler, se debe volver a modo usuario.

Para volver a modo usuario vamos a tener que hacer un `RESTORE_ALL` y un `IRET`.

El `INT` hace justo lo contrario que el `IRET`: restaura la siguiente instrucción, la palabra de procesador, apunta ESP a la pila de usuario y cambia la palabra de estado del procesador a 3 (nivel de mínimos privilegios) y salta a la siguiente instrucción (la de después del `INT` en el wrapper). Al final del wrapper, se hace un `RET` para volver a nuestro código.

En %eax tendremos el valor de retorno de la función; pero hay un problema: **cuando se hace un `RESTORE_ALL` se machaca el %eax con el valor del %eax que se había guardado antes de saltar a sistema**. Si nos acordamos, ese %eax antiguo contenía el número de la rutina de servicio.

Para solucionarlo, como el handler está programado en ensamblador, podemos ver los valores de los registros y modificarlos. Una vez de vuelta al handler, se va a machacar el valor de %eax de la pila actual (contiene el número de servicio) por el %eax nuevo que contiene el valor de retorno:

```
handler_syscall:
    SAVE_ALL
    [VERIFICACIONES]

    // llamada a la syscall
    call xsyscall.table(offset, %eax, sizepos)

    // machacamos el %eax de la pila con el
    // nuevo %eax antes de restaurarlo
    24(%esp) <- %eax
    RESTORE_ALL
    IRET
```

Nos podríamos plantear modificar el `RESTORE_ALL` y el `SAVE_ALL` en vez de usar la solución planteada. Pero queremos ahorrar líneas de código y por lo tanto todos los handlers van a usar las mismas macros para evitar duplicaciones y más posibilidades de errores. Recordemos que para las interrupciones e excepciones hardware **sí** se debe guardar el valor de %eax. Por ese motivo se soluciona en el propio handler de las llamadas al sistema y no en las macros.

El fallo de una llamada al sistema devuelve el valor `-1` y en `errno` el código de error. Pero el kernel no trabaja así, en el modo sistema se pondrá un valor igual o mayor a `0` si todo va bien y si va mal se pondrá el valor negativo del código de error. Por ese motivo, en el wrapper, debemos solucionarlo para devolver lo que se espera:

```
    int write(fd, *buffer, size){
        // se pone en cada registro el
        // valor correspondiente
        ebx <- fd;
        ecx <- buffer;
        edx <- size;

        mov %eax, #;
        int 0x80;

        // si eax es menor que 0, significa que
        // hay error y en eax hay el código de error.
        // por eso negamos %eax y devolvemos -1.
        if(%eax < 0){
            errno = -%eax;
            return -1;
        }
        ret
    }
```

1.4 Interrupciones y excepciones

Los handlers de las interrupciones o excepciones hardware no usan la syscall table, ya que es para las llamadas al sistema. Un handler de una excepción o interrupción HW no hace verificaciones de datos, ya que se confía siempre que el HW va a pasar datos válidos; además no tiene que guardar el %eax como pasaba antes. Lo que sí que hacen es un `EOI` para indicar que se ha acabado de tratar la interrupción y se pueden recibir nuevas.

Las rutinas de servicio para estos también son más sencillas porque no devuelven nada ni tienen ningún parámetro. La pila, después del `push ebp` para una interrupción HW es **exactamente igual** que la vista anteriormente: Contexto Hardware, Contexto Software + pila de rutina con `@ret` y `ebp`.

Para las excepciones es lo mismo prácticamente: no hay parte de usuario y no hay `EOI`. Lo demás es igual.

Sin embargo hay **algunas** excepciones que cambian la pila del sistema. Algunas a la hora de crear el contexto hardware pasan un parámetro, por lo que tienen un parámetro añadido al contexto hardware; por ejemplo la dirección que ha causado la excepción. Por este motivo, a veces, en el handler se va a tener que incrementar en 4 el contenido de `ESP` para anular el parámetro de la pila antes de hacer el `IRET`.

2 Sistemas de memoria

En los primeros sistemas, la memoria física se correspondía de forma idéntica con las direcciones de memoria del programa. El principal problema de ese método era que no existían los **niveles de privilegios**, por lo que los programas se debían fiar del SO para que sus espacios de memoria no fuesen manipulados.

Para solucionar esos problemas, Intel creó la MMU. Esta pequeña memoria guardaba la dirección física de inicio del sistema operativo. La MMU verificaba si la dirección que el usuario quería usar era más pequeña que la del sistema operativo y si no lo era, detectaba que el proceso quería entrar en el espacio de memoria del sistema operativo y lo mataba.

A medida que avanzó el tiempo, las CPUs eran más potentes y podían ejecutar más de 1 programas a la vez. Apareció un problema con como se guardaban las cosas en memoria. Si se seguía con ese método, todos los desarrolladores tenían que ponerse de acuerdo para no pisarse entre ellos en memoria. Aquí es donde nació la idea de direcciones lógicas. Todos los programas ahora se compilarían empezando en la dirección de memoria `0`. Las direcciones de programa, ahora, son `offsets` desde el inicio de su respectivo espacio en memoria física. Esto son las direcciones lógicas.

Ahora la CPU, cuando va a ejecutar un programa, lo aloja en memoria dónde le parece mejor. Por eso, a partir de ahora va a ser necesario hacer traducciones de las direcciones lógicas (del programa) a direcciones físicas.

Para estas traducciones, se aplican mejoras en la MMU. Ahora hay dos nuevos registros; uno contiene la **dirección física inicial** dónde se ha puesto el programa. El segundo registro contiene el **tamaño de este proceso en memoria**; para hacer la traducción la MMU va a coger la dirección lógica (que no deja de ser un offset desde el inicio del programa) y se la va a sumar a la dirección física inicial que guarda el primer registro. De este modo se puede hacer la traducción.

Aparte de traducir, la MMU también debe **proteger** al Sistema Operativo y además a los procesos entre ellos. Para hacer esta protección se usa el segundo registro; comprueba si la dirección lógica obtenida es más pequeña que el tamaño del programa.

2.1 La fragmentación

Con este método aparece un gran problema: a medida que los procesos mueren, van quedando **huecos de distintos tamaños** entre los espacios asignados de otros procesos.

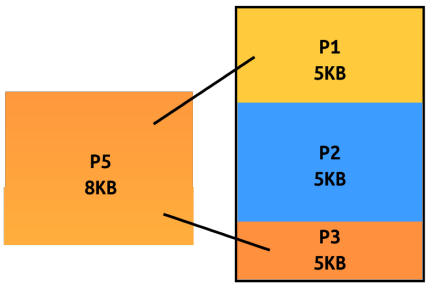

Figure 1: Problema de fragmentación

Como vemos en la figura 1, una vez liberados P1 y P3 existe espacio suficiente en memoria para P5; sin embargo P5 no va a poder ser asignado, ya que con el modelo actual los espacios deben ir **consecutivos en memoria**. No se pueden aprovechar los espacios liberados.

Para solucionar este problema se mantienen las direcciones lógicas para los procesos, pero ahora desde el punto de vista lógico el programa se va a ver dividido en segmentos del mismo tamaño (esto es un hecho lógico, virtual; en realidad no ocurre nada). A cada uno de estos segmentos se les llamará **página lógica**, y contiene un conjunto de direcciones lógicas consecutivas.

En memoria física hacemos lo mismo. Se divide de forma virtual en segmentos del mismo tamaño, iguales que las páginas lógicas. Estos segmentos se van a llamar **páginas físicas** o **frames**.

—— *Una página es un conjunto de direcciones consecutivas.*

Los programas siempre van a tener las direcciones lógicas de forma consecutiva (como hasta ahora), pero a la hora de alojar el programa en memoria física, no vamos a tener porque alojarlo de forma consecutiva. Las páginas lógicas se van a alojar en páginas físicas, sin necesidad de que sean páginas consecutivas (ver Figura 2).

Para hacer estas traducciones se usa el Translation Lookaside Buffer (TLB).

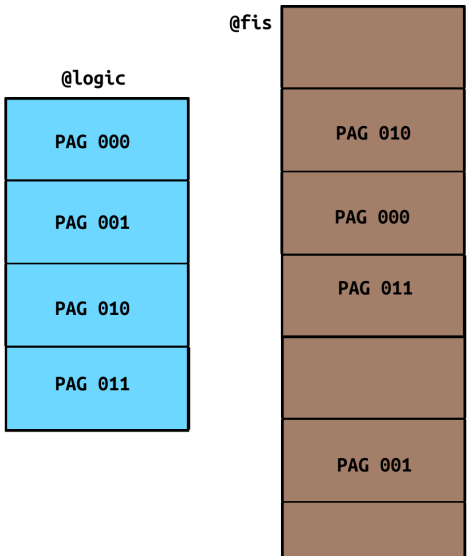

Figure 2: Páginas lógicas y físicas

2.2 El TLB

El TLB es una estructura hardware de la CPU. Hay un TLB por cada procesador. El TLB es una extensión de la MMU; una tabla de asociaciones de páginas lógicas a físicas. Normalmente suele tener 512 o 1024 entradas. El TLB debe ser capaz de traducir muy rápido: una dirección por ciclo de la CPU.

Las páginas normalmente son de 4KB y sabemos que dentro del espacio lógico todas las direcciones son consecutivas. Si calculamos:

$$\log_2(4KB) = 12$$

Vemos que se necesitan 12 bits para indexar todo el espacio de direcciones de una página. Una dirección lógica tiene 32 bits, de estos 32 bits vamos a usar los 12 bits de menor peso para indexar las direcciones dentro de una página.

De esta forma, como las **páginas lógicas y físicas tienen el mismo tamaño**, cuando hacemos la traducción y empezamos a generar una dirección física a partir de una lógica, se podrán usar esos 12 bits de menos peso de la dirección lógica **sin necesidad de hacer nada** para la dirección física. Estos 12 bits son el offset dentro de una página.

Los 20 bits restantes van a servir como el identificador de página (lógica o física).

Estos 20 bits van a pasarse al TLB y este va a buscar si existe alguna entrada con ese ID de página lógica:

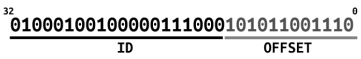

Figure 3: Dirección

- **Si existe:** Es un **hit** de TLB. El ID de página física asociado se pone como la parte alta de la dirección traducida a física junto al offset de 12 bits (que no cambia). Todo esto ocurre en un ciclo de CPU.
- **Si no existe:** Es un **miss** de TLB. En este caso el TLB no es capaz de traducir la dirección, por eso va a acceder al registro **cr3** que apunta a la tabla de páginas del proceso en memoria, allí buscará el ID de página física correspondiente y lo va a guardar en el TLB.

El registro **cr3** contiene la dirección de inicio de la tabla de páginas del proceso actual y existe un **cr3** por procesador.

Hay una tabla de páginas en memoria por proceso, y esta es un vector **con tantas entradas como páginas lógicas hay en el proceso**; dónde cada entrada contiene el identificador de la página lógica asociada. En una máquina de 32 bits, con identificadores de 20 bits, tendremos 2^{20} entradas en la tabla de páginas (son muchísimas, el TLB son solo entre 512 y 1024).

El TLB, durante un **miss**, debe calcular el offset para llegar a la posición correspondiente dentro de la tabla de páginas del proceso y encontrar la dirección de la página física asociada. Una vez encontrada, se busca una entrada del TLB que se pueda sustituir (con un algoritmo como puede ser LRU) y se guarda la asociación en el TLB. Ahora, desde el TLB, ya que todas las traducciones se hacen desde allí, se puede hacer la traducción de esta página que había producido un **miss**.

En el momento que se hace el salto a modo protegido (antes de saltar a modo usuario, en el tiempo de boot), se activa el **nivel de privilegios** y la **paginación**. Esto significa que antes del salto, **no tenemos paginación ni privilegios**. En el momento que se activa la paginación, **todo** va a trabajar con direcciones lógicas (usuario y **sistema**).

Las tablas de páginas se encuentran en direcciones lógicas (el **cr3 contiene una dirección lógica**). Cuando se va a acceder a la tabla de páginas del proceso, el TLB debe pasar "por sí mismo" otra vez, para traducir la dirección lógica que se encuentra en **cr3**. Por eso, sin el TLB no podríamos tener paginación; porque para acceder al cr3 necesitamos hacer una traducción.

Dentro del TLB hay un espacio reservado que siempre hace referencia a la memoria del sistema operativo.

Por este motivo, que todo funciona con direcciones lógicas cuando estamos en modo protegido, no se puede hacer la traducción usando solo la tabla de páginas.

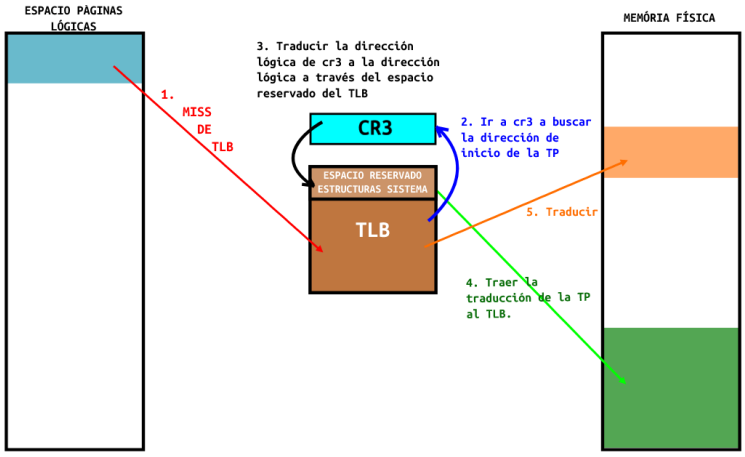

Figure 4: Esquema de traducción de un miss de TLB

Necesitamos algún elemento capaz de traducirnos estas direcciones.

2.3 Bits extra del TLB

Permisos

Se le añaden nuevos campos a la tabla de páginas, entre otros, los permisos R/W/X para cada página. El TLB añade 3 bits también para eso, y ahora al traer las asociaciones de páginas después de un **miss de TLB**, también se van a traer los permisos.

Nivel de privilegios

Además de esto, cada página va a tener dos bits adicionales que representarán el **nivel de privilegios** necesario.

En la tabla de páginas del proceso existen unas entradas asociadas a la memoria del Sistema Operativo. Siendo esto así, podría ser una vulnerabilidad, ya que se podría acceder a la memoria del Sistema Operativo (con fuerza bruta, por ejemplo). Para solucionar esto se añade el **nivel de privilegio** para **cada entrada de la tabla de páginas**. Este conjunto de páginas que se corresponden con la memoria del sistema operativo, van a necesitar un nivel de privilegios superior para poder ser accedidas.

Así que, para cada página lógica no solo va a tener los permisos de Lectura, Escritura y Ejecución; sino que además va a limitar el nivel de privilegios desde el cual se puede acceder a esta.

Presencia

Cuando se monta la imagen lógica de un proceso en memoria, en un trozo vamos a tener el código, en otro trozo los datos y al final de todo, la pila. De forma intencionada, se dejan espacios libres entre ellos. Esto se hace porque sabemos que la pila, en algún momento va a crecer y le dejamos espacio para que pueda crecer. Los datos también van a ir creciendo, por eso también dejamos espacio.

Figure 5: Estructura de la pila de un proceso

Los espacios libres son páginas lógicas sin páginas físicas asignadas. Las páginas libres no están asignadas al código, datos o pila. Simplemente no están asignadas. Solo se asignan las páginas que van a contener algo. Lo esquematizamos de esta forma simplificada para que se entienda. Cuando se necesita más espacio, se van asignando páginas físicas de forma dinámica a medida que esos segmentos van creciendo.

Como vemos, hay segmentos del espacio lógico del proceso sin memoria física asignada. Esto significa que cuando vamos a buscar una página lógica, debemos poder saber si tiene o no una página física asignada; ya que si no la tiene vamos a tener que generar un error.

De esto último se ocupa el **bit de presencia**. Nos indica si el contenido de la página es válido o no. Cuando tenemos un fallo de TLB, vamos a la entrada correspondiente de la tabla de páginas y antes de copiar todos los datos de la entrada, lo que hace el procesador es mirar el bit de presencia.

Si el bit de presencia es **1**, significa que la página es válida y que se puede copiar.

Si el bit de presencia es **0**, significa que la página no se encuentra en la tabla de

páginas (ni tampoco en el TLB). En este caso se lanza una excepción de **Page Fault**: se está accediendo a una dirección lógica que no tiene ninguna dirección física asignada (en ZeOS sale un aviso).

En un Sistema Operativo moderno, desde el handler de page fault se hace una llamada a un daemon llamado **daemon_pager** (paginador). El paginador monitoriza el estado de la memoria física de la máquina para hacer una gestión eficiente de esta. Es el responsable de saber qué ha pasado con la página que ha generado el **page fault**.

Los procesos, en los sistemas actuales, pueden consumir memoria virtual siendo la suma de su memoria superior a la memoria física. Esto nos permite extender la memoria física de la máquina usando el **disco**. Lo que se hace es pasar páginas de memoria al disco y a la inversa.

Cuando al paginador le salta la excepción de página le llega a través de un parámetro el identificador de la página que ha causado esa excepción; el paginador tiene un historial de qué páginas tienen su contenido **en el disco**. Cuando el paginador lo considera, puede volcar las páginas de un proceso en el disco; cuando esto pasa, el **bit de presencia** se pone a cero. El bit de presencia realmente indica si una página lógica está asignada a una página física **de memoria** o no. Pero eso no significa que la página lógica no pueda estar asignada a una **página virtual en disco**.

El paginador, cuando le llega una excepción, va a buscar en esa estructura para saber si está en el disco. En el caso que la página no aparezca tampoco en esta estructura, va a lanzar un **segmentation fault**; la página no esta asignada a **nada**. Por el contrario, si se encuentra dentro de esa estructura, el paginador (que también guarda un listado de las páginas físicas libres) va a cargar la página virtual asociada del disco y la va a volcar en la primera página física libre que encuentre y va a quitar esa página de la lista de páginas físicas libres. Se modifica la tabla de páginas con la nueva asignación y pone el bit de presencia a 1.

Ahora el TLB ya es capaz de hacer la traducción. Esto se le llama **fallo de página** y es una operación muy costosa.

2.4 Paginación

Problema

Cuando hay un fallo de TLB, este consulta al registro **cr3** (que contiene la dirección de inicio de la tabla de páginas) y busca en la tabla la asociación de la página lógica y la física para traerla al TLB. Esto se hace con los 20 bits de mayor peso de una dirección de 32 bits, el **indicador de página**.

Eso significa que la tabla de páginas tiene 2^{20} entradas. Una dirección de 32 bits ocupa 4 Bytes. Eso significa que la tabla de páginas tiene un tamaño total

de 2^{20} x 4 Bytes. Aproximadamente cada tabla de páginas ocupa 4 MB en memória.

Tenemos una tabla de páginas por proceso, si tenemos 100 procesos significa que estamos usando 400 MB para guardar tablas de páginas. **Eso es inviable**.

Además, como ya hemos visto antes, hay trozos en la tabla de páginas que están libres de traducciones y solo serán usados en supuestos casos de crecimiento de ciertos datos.

Solución

Para solucionar este problema, se hace una optimización. Ahora tendremos una **tabla de páginas multinivel** (2 niveles).

Dividimos la tabla de páginas (de 2^{20} entradas) en segmentos de 2^{10} entradas cada uno (4KB), de esta forma optimizamos más el espacio, ya que se puede destinar segmentos mucho más pequeños que antes.

Se añade, además, una nueva estructura llamada **directorio de páginas** dónde cada entrada contiene la dirección del segmento de la tabla de páginas asociado. También hay un bit de validez que nos indica si el segmento existe o no.

El directorio tiene 2^{10} entradas (4KB). Gracias a añadir esta estructura, podemos hacer una gestión mucho más óptima de la memoria.

Básicamente, en vez de hacer una división solo por páginas dónde cada proceso tiene una tabla de páginas de 2^{20} entradas en la que, los espacios libres no se pueden optimizar, vamos a tener una doble división dónde cada proceso podrá ocupar de 4KB en 4KB. Un proceso pequeño, por ejemplo de 2KB, ahora solo va a ocupar los 4KB del directorio de páginas y 4KB de un segmento. En total 8KB. Pudiendo llegar a un máximo de 2^{20}*4KB, 4GB.

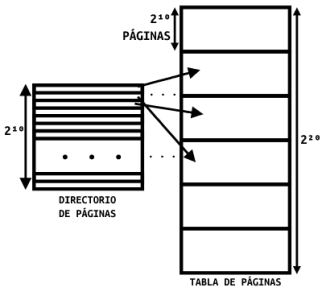

Figure 6: Nuevo modelo. Directorio + tabla de páginas segmentada.

El TLB solo accede a la tabla de páginas cuando hay un fallo, como hemos visto antes en la definición el problema.

Ahora, las direcciones de 32 bits se dividen de forma distinta. Los 12 bits de menor peso siguen siendo el offset, pero los 20 bits del identificador se dividen: los 10 primeros indexan dentro del directorio y los 10 siguientes indexan dentro de la tabla de páginas. Ahora el registro **cr3** contiene la dirección inicial del directorio de páginas.

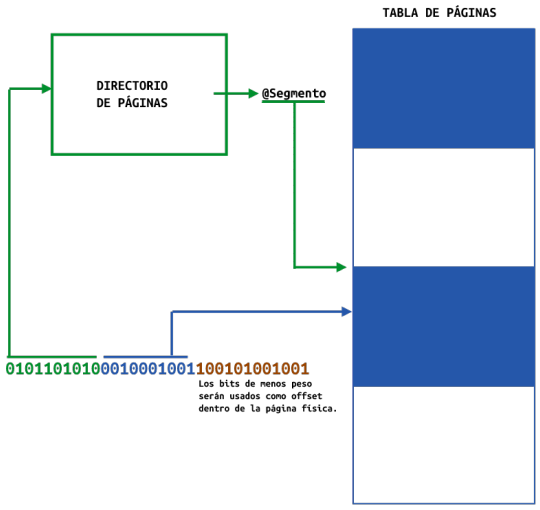

Figure 7: Esquema de una traducción multinivel.

3 Procesos

3.1 El ciclo de vida de un proceso

Tenemos estructuras de *management* que gestionan estos estados. En concreto tenemos un **vector de PCB** con todos los PCB de los procesos del sistema, este vector también determina el número máximo de procesos que puede tener el sistema.

Ready
: En este estado se encuentran todos los procesos que están listos para ejecutarse. Ya sea que se van a ejecutar por primera vez o que ya se han ejecutado anteriormente y pueden seguir ejecutándose. Todos estos procesos están en la *readyqueue*, que encadena los PCB's de los procesos del vector mencionado anteriormente.

Run
: En algun momento el planificador va a cambiar un proceso del estado de **ready** a este estado. En **run** no hay ninguna estructura. De este estado se puede pasar a **ready**, si expira el *quantum* por ejemplo. También se puede pasar a **blocked** si se está realizando alguna operación de larga latencia (p.e entrada del usuario).

  En algún momento un proceso en **run** va a pasar al estado **zombie**, hasta que alguien lea su estado de finalización.

Blocked
: En este estado también se tiene una cola de PCB's de los procesos bloqueados esperando un determinado recurso. Cuando se termina la operación de larga latencia, el proceso vuelve a **ready**.

Zombie
: En este estado se encuentran todos los procesos que ya han liberado todos los recursos **excepto el PCB**, dónde hay el estado de finalización que debe leer el padre del proceso. Una vez leído, el proceso muere y desaparece.

3.2 La pila de un proceso

La pila de un proceso suele ocupar el tamaño de una página (4KB). Dentro de la misma pila, en la cima, se encuentra el `task_struct` del proceso (o PCB). Esto es así ya que cuando un proceso se encuentra en estado de **run**, es necesario poder acceder al `task_struct` del proceso.

Esto es así ya que, por ejemplo en ready podemos saber que un proceso está en ese estado porque lo encontramos en la cola de **ready**. Para blocked lo encontraremos en la cola de **blocked**. Para los porcesos **zombie** normalmente

existe una lista dentro del `task_struct` del padre dónde se encolan los procesos en estado **zombie**. Pero para los procesos en estado **run**, no se guarda el PCB en ningún sitio; pero de igual forma vamos a necesitar acceder al PCB para hacer las gestiones de este.

La forma en que se consigue acceder al PCB de un proceso en **run** depende del sitema operativo. En **Windows** se utiliza el registro GS que apunta directamente al EPCB (el PCB del proceso); hay un registro que siempre apunta al PCB del proceso en ejecución.

En **Linux** es distinto; como tenemos el `task_struct` solapados dentro de la misma página, lo que sabemos que va a pasar siempre es que **esp**, que en modo sistema apunta a la cima de la pila de sistema del proceso, va a estar apuntando una dirección dentro de la página lógica dónde tenemos el `task_struct`. Para eso podemos usar el registro **esp** para hallar el PCB.

El inicio del task_struct va a estar siempre al **inicio de una página lógica**, eso es así porque conocemos como se ha programado el sistema operativo. El inicio de una página lógica siempre esta alineado a 4KB (tamaño de página), así que el task_struct va a estar siempre alineado a 4KB también. Esto significa que el inicio de una página lógica va a ser siempre igual: `0x_____000`. Siempre va a acabar con tres ceros (12 bits a cero).

Para calcular el inicio de la página, solo necesitamos el registro **esp** y hacer una máscara: `%esp & 0xFFFFF000`. De esta forma podemos hallar siempre el task_struct de la pila del proceso en ejecución. A esta máscara la llamamos "función `current()` ".

3.3 Multiplexación de procesos

En un sistema donde solo se puede ejecutar un thread a la vez; en estos casos, para simular la ejecución de múltiples procesos a la vez lo que se hace es **multiplexar** el tiempo de CPU de forma muy rápida.

Esta multiplexación debe ser extremadamente rápida, ya que va a pasar de forma muy repetida.

Para hacer esto se hace lo que llamamos un **cambio de contexto**. Se pasa un proceso que está en **run** y lo pasamos a **ready** y un proceso que no se está ejecutando va a pasar a **run**.

Este cambio de contexto ocurre en una función del sistema operativo llamada `task_switch` (cambio de tarea). Todos los procesos deben pasar por esta función para poder entrar en la CPU; ya sea un proceso nuevo o un proceso que está esperando. Esta función recibe un único parámetro: el puntero al `task_union` del proceso que se quiere poner en ejecución.

¿Como llegamos al `task_switch`?

Al saltar la interrupción de reloj el procesador accede a la IDT, a la GDT y a la TSS (estructuras del sistema) para hallar la dirección de la pila de sistema. Una vez hallada, el procesador empila el contexto hardware.

Ahora se va a ejecutar el handler de la interrupción de reloj, dónde lo primero que se hace es un `SAVE_ALL`, por lo que se va a guardar el contexto software en la pila.

Dentro de este handler, antes o después se va a hacer una llamada a la rutina de servicio de la interrupción de reloj. Con esta llamada, se añade a la pila de sistema la dirección de retorno (`@ret`).

Ahora nos encontramos dentro de la `clock routine`, por lo que lo primero que se hace es un `push ebp` y a continuación se hace que **ebp** apunte al **ebp** que acabamos de empilar. Hasta ahora, la pila está como se puede observar en la Figura 8.

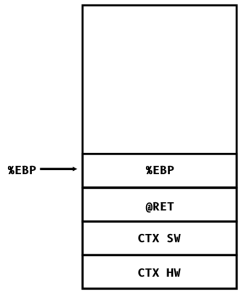

Figure 8: Esquema de la pila de sistema una vez se ha entrado enla clock_routine

Dentro de la rutina de reloj, en algún punto se va a invocar el planificador. El planificador es algo muy complejo por lo que va a empilar muchas cosas en la pila (que no nos interesan por ahora). El planificador decide que se debe hacer un cambio de contexto: se debe multiplexar el proceso.

Para hacer esto se va a llamar a la función `task_switch`, sabemos que en la cima de la pila de sistema, después de todo lo que ha empilado el planificador, habrá los parámetros de la función (en este caso, solo uno: ***new**), la dirección de retorno del código desde el cual se ha llamado a task_switch (en este caso el código del planificador) y cuando ejecutemos la función se va a hacer otro `push ebp` y `esp` y `ebp` apuntarán a la cima de esta pila de sistema. Ver la Figura 9.

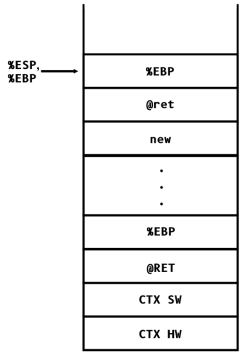

Figure 9: Esquema de la pila de sistema del proceso current() ecuando llega al task_switch

Esta página lógica que vemos en la figura 9 y que contiene la pila de sistema del proceso `current()`, debemos recordar que en su cima; al inicio de la página, tenemos el task_struct (el PCB) del proceso cuya dirección se puede hallar con una máscara en **esp**.

La pila de sistema del proceso `new` (hablamos de procesos que ya habían estado en CPU y el planificador los sacó), va a ser muy parecida a la que hemos visto en la figura 9 ya que este proceso fue sacado de la CPU usando también task_switch y por lo tanto la pila en ese momento tenía la misma estructura que la que hemos visto, con distintos valores.

El cambio de contexto

1. Debemos tener en cuenta que el espacio de direcciones lógico de `current()` es distinto al espacio de direcciones lógicas de `new`. Por lo tanto, el mapeo de direcciones lógicas a físicas que tiene `current()` será distinto al de `new`. Si no hacemos nada con el registro **cr3**, este estará apuntando a la tabla de páginas de `current()`, pero al salir a modo usuario vamos a querer usar la tabla de páginas del proceso `new` por lo que este **cr3** debe ser modificado para que apunte a la tabla de páginas de `new`.

   Así que dentro del `task_switch`, también debemos modificar el registro **cr3** como podemos ver en la línea 6 de la figura 10. **Cuando se escribe en cr3, se hace un flush del TLB de forma automática**, por lo que la traducción de lógicas a físicas de `current()` se pierde y se irá cargando con las traducciones de new.

   Cuando pasa esto, se van a generar unas ráfagas de fallos de TLB. Por esto el task_switch es más lento de lo que podría ser.

2. Cada vez que llega una interrupción se accede a la IDT, luego a la GDT y finalmente a la TSS para encontrar la dirección de la pila de sistema

```
1  inner_task_switch (union task_union * new)
2  {
3    PUSH EBP
4    EBP <- ESP
5
6    set_cr3(new -> task.dir);
7    tss.esp0 <- &new->stack[1024]
8
9    (current() -> task.kernel_esp) <- esp;
10   ESP <- (new -> task.kernel_esp);
11
12   POP EBP
13   RET
14 }
```

Figure 10: Pseudocódigo de la función task_switch

del proceso actual. Como se está haciendo un cambio de contexto, nos tenemos que asegurar que después de salir a modo usuario, la pila de sistema a la cual apunte el campo **esp0** de la TSS sea la del proceso `new`.

Dentro del `task_switch` se tiene que hacer que la pila de sistema actual que se utilice en modo sistema sea la de `new`. De este modo cuando se vuelva a saltar a modo sistema, la pila de sistema será la de `new`.

Eso se hace con lo que podemos ver en la línea 7 de la figura 10 dónde se hace que el campo **esp0** de la TSS sea la dirección de la **base de la pila** de `new`.

¿Por qué la base de la pila? Se debe tener en cuenta que cuando se salta a modo usuario a modo sistema, la pila de sistema va a estar vacía. Eso es así porque cuando hemos saltado a modo usuario se ha tenido que desempilar toda la pila de sistema (para ir volviendo hasta el modo usuario). Así que la base y la cima de la pila de sistema, en esta situación, coinciden.

3. Se guarda el estado de `current()` ya que va a ser sacado de la CPU.

   Para esto, dentro del `task_struct` de current, se crea una nueva variable llamada `kernel_esp` dónde se guarda el valor del **esp** actual de forma que esta nueva variable va a apuntar a la cima de la pila de sistema de `current()` que es dónde hay el **ebp** que guardamos dentro del `task_switch`. Ver línea 9 del código en la figura 10.

   Haciendo esto, estamos guardando todo el contexto de ejecución de `current()` con una sola línea de código, porque la pila de sistema del proceso contiene todos los frames de activación de las funciones hasta llegar al `task_switch`.

Además, en el mismo rangos de 4KB, hay el `task_struct` del proceso `current()`.

Por definición, sabemos que en el `task_struct` de `new` también tenemos esta variable `kernel_esp` ya que cuando se sacó de la CPU se hizo lo mismo.

4. En el registro **esp** del procesador se va a cargar el `kernel_esp` de `new`. Ver línea 10 del código en la figura 10.

   El registro **esp**, que antes apuntaba a la cima de la pila de sistema de `current()` ahora apuntará a la cima de la pila de sistema de `new`. Con esto, la pila de sistema pasa a ser la de `new`.

5. Se hace un `pop ebp` para recuperar el **ebp** que había antes de entrar en el `task_switch` y un `ret` para recuperar el código que se estaba ejecutando (planificador). El `task_switch` **siempre** acaba con `pop ebp` y `ret` porque `task_switch` siempre espera que en la cima de la pila de sistema contenga `ebp` y una dirección de retorno.

   Después va a ocurrir otro `ret` que ejecutará el **handler de la interrupción de reloj** (dónde empezó todo).

   Cuando al final se salga al **modo usuario**, se va a salir pero en el proceso **new.**

6. Una vez completados los pasos anteriores, y desempile todo su contexto de ejecución (incluidos el contexto software y hardware) de la pila, al hacer el `iret` dentro del `clock_handler`, el procesador ya estará ejecutando `new`. El cambio de pila es lo que hace que se cambie el proceso en ejecución. Son los valores de la pila de sistema; sus contextos y sus frames de activación lo que determinan cuál es cada uno de los procesos.

   Nota:

   (a) En la pila de `new` habrá guardado también un puntero a "`new`" (le llamaremos `oldnew` para aclararnos), igual que en `current()`. Este `oldnew` es el proceso que se utilizó en su momento para sacar a `new` de la CPU.

Todo esto se hace con menos de 20 instrucciones en ensamblador. Es muy rápido.

¿Cómo se preserva el valor de los registros seguros durante el `task_switch`?
La función `task_switch` que hemos estado hablando en realidad se llama `inner_task_switch` y la función `task_switch` realmente es un **wrapper** de la función `inner_task_switch` dónde se hace un push de los registros que se quieren guardar (edi, esi, etc.) y dentro de este wrapper es dónde se va a hacer una llamada a `inner_task_switch`.

3.4 La llamada `fork()`

En Linux la creación de procesos es a través de la llamada `fork()`. Esta llamada crea una copia del proceso padre, replicando el código, datos y pila al proceso hijo. **No comparten código, datos y pila** sino que tienen una **copia**. Esta función devuelve al padre **pid** del hijo y al hijo le devuelve 0. De esta forma podemos repartir trozos de código entre el padre y el hijo. En caso de error devuelve -1 y en `errno` hay la clase de error.

En Windows es completamente distinto. La llamada `fork()` no existe, por el contrario hay un **proceso cargador**. Al crear un proceso en Windows siempre se hace para ejecutar código nuevo. Para crear un proceso en Windows se usa `create_process()` al que se le pasa nombre del ejecutable que se va a usar para crear el proceso. Es muy complejo. En SOA, nos centramos en `fork()`.

3.4.1 Implementación de `fork()`

Asignar un `union task_union`

Es decir, le asignamos un PCB libre de la `freequeue`. Notar que el `union task_union` no es solo el `task_struct`, sino que también contiene la pila del proceso.

Asignar al proceso hijo un contexto de ejecución

Inicializamos inicial de su `task_struct` y su pila de sistema. Para esto, los 4KB que ocupa el `union task_union` del proceso padre se copiarán al `union task_union` del proceso hijo. De esta forma se inicializa el `task_struct` y la pila de sistema del proceso hijo.

Asignar tabla de páginas al proceso hijo

Se le asigna una **tabla de páginas** (un directorio de páginas en realidad) al proceso hijo.

Copia de memoria

.

Una vez se tiene asignada una tabla de páginas al proceso hijo, se empieza a realizar la copia de memoria del proceso padre al proceso hijo (codigo, datos y pila).

Esta copia se hace de una forma determinada. En la **tabla de páginas** del proceso padre, se pueden diferenciar **varias secciones** bien definidas: una primera

sección dónde se encuentran los mapeos de las direcciones lógicas a físicas del **kernel**. Otro segmento corresponde al **código**, otro a los **datos** y por último el que corresponde a la **pila**.

La tabla de páginas del proceso hijo está vacía por el momento. Dependiendo de qué segmento se va a copiar se va a hacer de una forma u otra:

- **Segmento de kernel:** el kernel tiene asignadas unas páginas físicas dónde guarda su kernel, pila, datos, etc. Se busca que todos los procesos compartan las mismas páginas físicas de kernel; el kernel debe ser el mismo independientemente del proceso que se ejecute.

  Para eso, en el `fork()`, la parte de la tabla de páginas que corresponde al kernel se copia **exactamente igual** del padre al hijo. Se iteran una a una las tablas de páginas correspondientes al kernel y se copian idénticas al la tabla de páginas del proceso hijo.

- **Segmento de código, datos y pila:** Los procesos no comparten memoria; los procesos tienen **copias** de código, datos y pila del proceso padre, pero no lo **comparten**.

  Cada una de las entradas, tienen asignadas páginas físicas. Para el proceso hijo queremos que tenga asignada **otras** páginas físicas distintas y que se copie el **contenido** del padre al hijo.

  Para hacer eso, se va a mirar la **tabla de páginas** del padre y, por cada entrada que tenga una página física del padre se va a asignar una página física de la misma entrada al hijo: **se asignan tantas páginas físicas como tiene el padre, al hijo.**

  Una vez el hijo tiene páginas físicas asignadas, se debe empezar a copiar el contenido del padre al hijo. Esto conlleva un **problema**: El procesador no tiene acceso directo a las páginas físicas de los procesos porque estamos trabajando con paginación y se deben **traducir** las direcciones a través del hardware; usando el **TLB**.

  El registro `cr3` apunta a la cima de la tabla de páginas del proceso padre y solo hay **un registro cr3**. Esto significa que el TLB no ve los mapeos que estamos creando en la tabla de páginas del hijo. **No se va a poder hacer la traducción para las páginas del hijo**. Las traducciones las hace el **hardware** y **no el software**.

  ¿Cómo se va a poder escribir en la memoria física que tiene asignado el proceso hijo si el TLB solo puede ver la tabla de páginas del proceso padre?

  Para solucionar este problema debemos tener en cuenta que, por definición, sabemos que **en la tabla de páginas del proceso padre hay páginas libres** (sin asignar a página física). Se define que un proceso no pueda ocupar todas sus páginas para casos como este.

  Estamos en `sys_fork()` por lo que quien está ejecutando estas instrucciones es el sistema (no el padre!) **desde el contexto de ejecución del**

padre. Para hacer esta copia de datos, como estamos en **modo sistema** tenemos acceso a los `task_struct` del padre y el hijo y a las tablas de páginas del padre y el hijo. El sistema operativo sabe qué páginas se están asignando al **proceso hijo** y también tiene un listado de las páginas libres en la tabla de páginas del padre. Lo que se va a hacer es que las páginas físicas que se están asignando al proceso hijo se van a asignar de forma temporal, a la vez, a los distintos espacios libres de la tabla de páginas del padre.

Ahora, a través de las direcciones lógicas que se quieren copiar del padre se va a acceder al su contenido en las páginas físicas y se van a ir copiando a las páginas físicas del hijo a través de este **mapeo temporal** que se ha creado dentro de la tabla de páginas del padre.

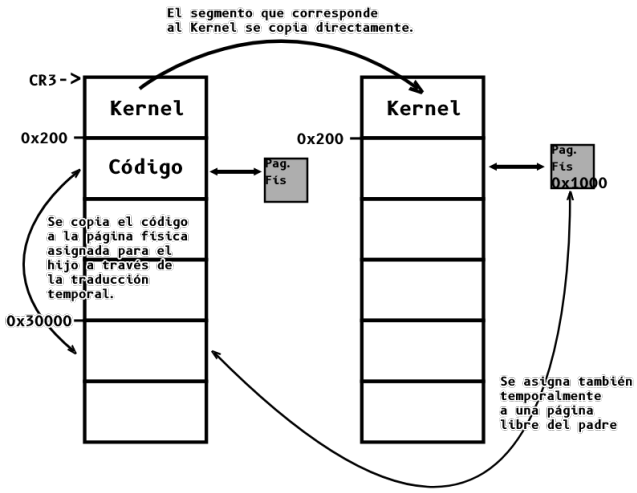

Figure 11: Esquema de la copia de datos de padre a hijo

Una vez se ha hecho esto para todas las páginas lógicas, se eliminan los mapeos temporales de la tabla de páginas del padre y se hace un `set_cr3()` para hacer un **flush** del TLB y quitar todos los mapeos temporales de allí también.

Actualización de las estructuras del hijo

Los procesos tienen muchas estructuras distintas que deben ser inicializadas (vector de signals, tabla de canales del pcb, etc).

Para inicializar la tabla de canales del hijo, por ejemplo, se copian las páginas de la tabla de canales del padre a la tabla de canales del hijo y se incrementa el número de referencias de ese fichero en la tabla de ficheros abiertos. Hay una **tabla de canales** por proceso y **una tabla de ficheros abiertos** en todo el sistema.

Se deben inicializar todas las estructuras del proceso hijo.

Para asignar un **PID** a un proceso hijo se puede hacer de varias formas:

- **En ZeOS** asignamos el PID 1000 al primer hijo de `init()` y se va incrementando de forma secuencial.
- **En un SO real** se debe evitar que el **PID** de información acerca del proceso. En los primeros sistemas operativos, el **PID** era la dirección de memoria dónde se encontraba el PCB del proceso. Eso es una vulnerabilidad. El **PID** debe identificar el proceso sin dar información.

  Actualmente en los sistemas modernos se genera un número aleatorio, si está libre, se asigna este número. Si está ocupado se incrementa de forma secuencial.

Actualizar el contexto de ejecución del hijo

Actualmente el proceso hijo tiene asignado un contexto de ejecución *temporal* cuando hemos hecho una copia del `union task_union` del padre al del hijo. Pero el proceso hijo no puede tener los mismos valores que el padre en todos los campos; por ejemplo, debe tener un **PID** distinto. Además, el contexto se debe preparar para que el proceso hijo se ponga en marcha.

Todos los procesos entran en la CPU a través del `inner_task_switch`. Los procesos nuevos también siguen este camino.

El `task_struct` del proceso hijo debe ser compatible con el código del `task_switch` que hemos visto en la Figura 10. El proceso padre, cuando empieza a ejecutar `sys_fork()`, tiene un esquema similar a este:

Antes de ir al handler de las llamadas al sistema, se va a guardar el contexto hardware. Una vez en el handler de las llamadas al sistema, se guarda el contexto software (con el (SAVE_ALL) y la dirección de retorno al handler. Dentro del `sys_fork()` se hace un `push ebp` y mueve `esp` a `ebp` para crear el enlace dinámico.

Para el `task_switch` lo importante es la **cima de la pila de sistema**. Si nos fijamos en el código del `task_switch` (Figura 10) cuando acabamos, en las dos últimas instrucciones, el `task_switch` se espera que haya un `ebp` y una dirección de retorno en la cima de la pila de sistema.

Eso significa que en la pila de sistema del hijo, **ya hay** lo necesario para volver, ya que es una copia del padre. Pero para que se ejecute el código del hijo, lo

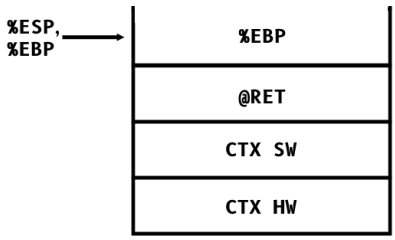

Figure 12: Esquema de la pila en `sys_fork()`

que se va a hacer es que el `kernel_esp` del proceso hijo apunte al `ebp` que hay dentro de su pila. Si no hacemos esto, el `kernel_esp` del proceso hijo estará apuntando al `ebp` de la pila del proceso padre!

Por otra parte, el `sys_fork()` debe devolver el **PID** del hijo al proceso padre y un **0** al proceso hijo. Por este motivo, dentro del `sys_fork()` (que lo está ejecutando el padre) debo devolver el **PID** del proceso hijo.

Pero para el proceso hijo debo devolver un **0**. Tal y como está la pila de sistema del hijo, actualmente somos incapaces de devolver un **0** ya que si el `kernel_esp` está apuntando al `ebp` de la cima de la pila y cuando se haga el proceso de volver al modo usuario no se devuelve en ningún momento un **0**; se devuelve lo que había en **eax** en el contexto software.

Para solucionar esto en un sistema operativo real, nos aprovechamos de que, cuando un proceso pasa de **ready** a **run** por primera vez, nos interesa hacer una ejecución de un código específico antes de que este empiece a ejecutar código de usuario para acabar de hacer algunas inicializaciones y, entre otras cosas, poder devolver un **0**. Para esto, se va a "trucar" un poco el contexto de ejecución del proceso hijo para que **la primera vez** que pase a **run** se pueda ejecutar este código específico pudiendo así además devolver un **0**.

Este código específico se va a encontrar dentro de una función llamada `ret_from_fork()`, una función que solo se ejecutará la primera vez que el proceso pase a **run**.

En ZeOS, lo único que se va a hacer es que el proceso cree el dynamic link y a continuación se guarda en **eax** un cero y se hace un `ret`. En alto nivel esto sencillamente sería un `return 0;`

Necesitamos retocar el contexto de ejecución del hijo para que pueda ejecutar

```
1 int ret_from_fork()
2 {
3   PUSH EBP
4   EBP <- ESP
5   // Inicializaciones en un SO real
6   EAX <- 0
7   POP EBP
8   RET
9 }
```

Figure 13: Código alto nivel de `ret_ from_fork()`

el `ret_ from_fork()` antes de pasar a ejecutar código de usuario por pirmera vez. Sabemos que el `task_switch` espera un `ebp` y una dirección de retorno en la cima. Lo que haremos será poner un **0** (`"fake ebp"`) en la cima de la pila y machacar el `ebp` que teníamos en la pila para poner la dirección de `ret_ from_fork()`. El `kernel_esp` que antes apuntaba al `ebp` que hemos machacado haremos que apunte al **0** que se acaba de guardar en la cima de la pila.

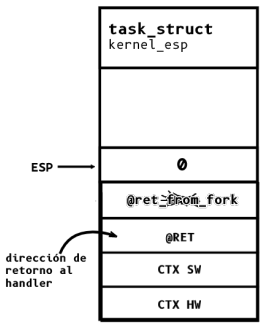

Figure 14: Esquema de la pila de `sys_fork()` del proceso hijo modificada.

Cuando el proceso hijo vaya a pasar a usuario por primera vez, cuando se ejecute el `task_switch`, se va a cambiar a la pila de sistema del hijo (línea 10 del código de la Figura 15) por lo que `esp` apuntará al **0** que se ha guardado en la pila del hijo (que se recuperará como **ebp** en el `task_switch` con el `pop ebp`) y a continuación, con el `ret` se saltará a `ret_ from_fork()` (ver figura 14).

Ahora ejecutará el `ret_ from_fork()` (ver figura 13) y al final se hará un `ret` a la siguiente dirección de la pila que es el `@ret` de retorno al handler. Así salta al handler con el valor de `eax` a cero.

A continuación se pueden ver dos esquemas de los flujos de la creación de un

```
inner_task_switch (union task_union * new)
{
  PUSH EBP
  EBP <- ESP

  set_cr3(new -> task.dir);
  tss.esp0 <- &new->stack[1024]

  (current() -> task.kernel_esp) <- esp;
  ESP <- (new -> task.kernel_esp);

  POP EBP
  RET
}
```

Figure 15: Pseudocódigo de la función task_switch

proceso hijo (en **ready**) y de como este proceso hijo llega a la ejecución.

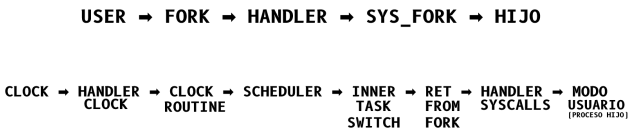

Figure 16: Creación proceso hijo (arriba) y llegada del nuevo proceso a ejecución (abajo)

Viendo estos flujos, se puede notar alguna cosa que no encaja: **entramos por el handler del reloj y se sale por el handler de las llamadas al sistema**. Esto es algo particular cuando se ejecuta un proceso nuevo.

Los handlers de las interrupciones hardware (como es el **handler del reloj**) deben ejecutar un `EOI` antes de hacer el `IRET`. Pero con lo que hemos visto hasta ahora, este `EOI` no se ejecutaría. Si no llega el `EOI` no van a llegar más interrupciones de reloj y por lo tanto no se podría planificar más: se deshabilitan las interrupciones hardware.

Para solucionar esto, debo asegurarme que el `EOI` se ejecute antes. Por este motivo, en ZeOS el `EOI` va antes de la llamada a la rutina de servicio de la interrupción de reloj.

Encolar el proceso en la cola de ready

Devolver el PID del hijo al proceso padre

3.5 `sys_exit()`

Cuando un proceso acaba, se liberan todos sus recursos menos el PCB. Esto es debido a que `sys_exit()` tiene un parámetro que es un codigo de finalización. Los procesos hijos notifican a los padres un código de finalización (0 si todo ha ido bien, negativo si ha ido mal).

Cuando se hace `sys_exit()` se guarda en el PCB el código de finalización y se liberan todos los recursos menos el PCB. En algún momento el padre a través de la llamada `wait_pid()` va a leer el código de finalización del PCB del hijo para saber si se debe hacer alguna cosa. El `wait_pid()` libera entonces el PCB del hijo.

Cuando un proceso en **run** ejecuta `exit()`, aparece un nuevo estado que llamamos **zombie**: un proceso que se le han liberado todos los recursos y solo tiene asignado un PCB. El proceso saldrá de **zombie** liberando el PCB cuando el padre haga `wait_pid()`.

¿Qué se debe hacer en `sys_exit()`? Se deben liberar los recursos del proceso (liberar tabla de páginas, la tabla de canales, etc.). Además debe poner dentro del PCB del proceso el código de finalización. Por último llama al scheduler para que busque y ejecute otro proceso.

¿Dónde se guarda el PCB del hijo mientras está en el estado de zombie? Pregunta de exámen.

3.6 Los procesos `idle()` e `init()`

Estos procesos se crean manualmente en **tiempo de boot** porque son procesos especiales del sistema.

3.6.1 `idle()`

Es un proceso de sistema que, en ZeOS solo ejecuta un `while(1);`. Este proceso `idle()` se ejecuta cuando la cola de ready está vacía y no hay procesos para ejecutar. No se puede parar la **CPU** porque si se para también se va a parar el fetch de instrucciones y por lo tanto no se va a ejecutar nada más.

Para mantener la CPU en marcha, se crea este proceso que se ejecuta en modo sistema esperando interrupciones de reloj. Cuando llega una interrupción de reloj el **planificador** mira si se está ejecutando el proceso `idle()`. En este caso, mira directamente si la cola de ready está vacía: si está vacía `idle()` se

sigue ejecutando y en caso contrario se saca a `idle()` y se pasa a ejecutar el proceso que esta en **ready.**

En un sistema operativo real, `idle()` no es un bucle infinito. En un sistema operativo real `idle()` se utiliza para lo que se llama *housekeeping*. Cuando salta `idle()` se aprovecha para hacer limpieza en el sistema operativo (memoria, procesos, etc.). En un sistema operativo real, `idle()` es muy complejo.

Este proceso se crea de una forma especial; es un proceso que siempre se ejecuta en **modo sistema**. Esto significa que no tiene contexto de usuario, **nunca se salta a modo usuario desde** `idle()`. Además nunca va a estar en la cola de ready. Por esto, en el sistema existe un puntero apuntando al `union task_union` de `idle()` para que se pueda encontrar cuando sea necesario.

Cuando se crea el contexto de ejecución de `idle()` se hace de una forma especial: no es necesario que tenga un **contexto hardware** (ya que es la mínima información que necesita la CPU para volver al modo usuario). Tampoco hay **contexto software** (ya que es la mínima información que necesita el sistema para volver a modo usuario). Lo único que hay en la pila de `idle()` es la información mínima para que se pueda pasar a ejecutar a través de `inner_task_switch`; ya que **todos** los procesos del sistema pasan a ejecutarse a través de esta función.

Las dos últimas instrucciones de `inner_task_switch` son `pop ebp` y `ret`. Esto significa que en la pila debe haber al menos un `ebp` y una dirección de retorno.

Sabiendo todo esto, para preparar el contexto de `idle()`, se va a hacer de forma que **nunca** salte a modo usuario y se pueda poner en funcionamiento a través de `task_switch`:

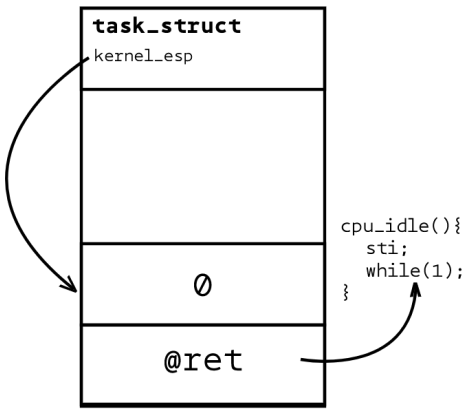

Figure 17: Esquema del `union task_union` de idle

El `kernel_esp` de `idle()` va a apuntar a un `"fake ebp"` (**0**) y la dirección de retorno va a ser la de la función `cpu_idle()` que es dónde se encuentra todo el código de la función `idle()`. El `"fake ebp"` es para que el `task_switch` pueda hacer el `pop`.

3.6.2 `init()`

Es el proceso dónde se salta por primera vez al modo usuario, siendo el primer código de usuario que se va a ejecutar.

Es parecido a `fork()` en cuanto a estructura; se le debe asignar un `union task_union`, un `pid=1` y se le asigna memoria (tabla de páginas que todos los demás procesos heredan). Para saltar a `init()` se hace de una forma especial.

En el boot, se le debe indicar al sistema operativo y el hardware que el proceso actual a ejecutar es `init()`. Para hacer esto, se debe indicar que la tabla de páginas y la pila de sistema a utilizar es la de `init()`.

Una vez se ha asignado todo lo que se le debe asignar, se hace un `set_cr3(init->dir)` para indicar que se debe usar la tabla de páginas de `init()` y en los campos `esp0` de la TSS y en el `MSR 0x175` se les pone la pila de sistema de `init()`. Cuando se salte a modo usuario, el proceso actual va a ser `init()`. Es el único proceso que no se pone en marcha a través del `inner_task_switch`.

3.7 Planificación

Existen tres clases de planificadores:

- **Long term** - Decide si el proceso que crea el usuario se puede crear o no a partir de los recursos disponibles, la seguridad, etc.
- **Mid term** - Es muy complejo. Cuando la máquina no está saturada (el proceso `idle()` está en ejecución) o cuando la máquina tiene la memoria física saturada este planificador hace *housekeeping*. Intenta liberar toda la memoria física (llevandola a disco) para poder ejecutar más procesos.
- **Short term** - Decide cuando se va a ejecutar un proceso y cuál es el proceso que va a ejecutar a continuación. Multiplexa el tiempo de CPU entre todos los procesos y decide cuando se pasa un proceso de Ready a Run.

El *cuando* es lo que llamamos **política de planificación**. El *quién* es el algoritmo de planificación.

3.7.1 Políticas y algoritmos de planificación

No apropiativa
El sistema operativo **no** se apropia de la CPU para meter otros procesos. Normalmente se usa un algoritmo de tipo **FIFO** o **FCFS** (First Called First Served). Con este algoritmo la cola de **ready** se ordena por orden de llegada de los procesos.

Esta política funciona muy bien cuando la carga de procesos es **muy interactiva**: hay procesos con muchas operaciones de entrada/salida o trabajo con el disco. Esto es así porque estos procesos van a cambiar mucho entre los estados **run** y **blocked** (esperando entrada del usuario, lectura del disco, etc.).

Si los procesos son mixtos: hay cálculo con la CPU y entrada/salida o lecturas a disco, esta política va muy mal. Los procesos de cálculo siempre están en RUN y no abandonan la CPU hasta que acaban. Con esta política y este tipo de procesos (mixtos) se da lugar al *efecto convoy* ya que los procesos interactivos o de disco se quedan al final de la cola (a medida que van pasando a **blocked**) y los de cálculo con CPU al principio.

Apropiativa diferida
Se puede tomar la decisión de que un proceso en run abandone la CPU y pase de **run** a **ready**.

Normalmente se hace uso del algoritmo *Round Robin*. Se define un *quantum* por proceso que indica cuanto tiempo puede estar un proceso ejecutándose en estado **run** hasta que el sistema operativo lo saque. El *quantum* suele ser de entre 5 y 8 milisegundos.

En cada tick de reloj se hace una comprobación para ver si el *quantum* del proceso actual ha expirado. Si esto es así, se saca de **run** y se mete en **ready**.

Si el *quantum* es demasiado largo (p.e 2s), este Round Robin se va a convertir en un FCFS y no nos dará ninguna ventaja. Por este motivo los *quantums* no deben ser muy largos. Por otro lado, tampoco pueden ser muy pequeños porque, sino se harían demasiados `task_switch` y no sería óptimo.

Pero aparece una nueva necesidad: No todos los procesos son iguales. Un proceso de un **administrador** es más importante que el proceso de un usuario corriente. Por lo tanto, el proceso del administrador debería poder entrar antes en la CPU sin tener que esperar otros procesos menos importantes.

Apropiativa immediata
Permite el paso de **run** a **ready** como hemos visto antes, pero añade el paso de **blocked** a **ready**.

Imaginamos que un proceso de un administrador entra en la máquina (**ready**) y es más prioritario que el proceso actual (**run**). Al de **run** se le puede sacar a

ready para poder meter el nuevo proceso más prioritario en **run**.

Cuando un proceso **blocked** acaba su espera, se va a mirar si su prioridad es mayor que la del proceso en **run** y, si es así se va a sacar el de **run** para entrar el de **blocked**.

Además, cuando entra un **nuevo** proceso también se va a mirar si es más importante que el que está en **run** y si es así, se cambia.

Para esto se necesita un nuevo parametro en la `task_struct` del proceso: la prioridad.

Las prioridades pueden ser asignadas de dos formas:

- **Estática**: Al crear un proceso se le asigna una prioridad y no cambia. Esto puede llevar a una situación de ***starvation***; dónde un proceso de baja prioridad nunca entra en la CPU porque van llegando nuevos procesos de más prioridad.
- **Dinámica**: El proceso tiene una prioridad inicial que puede cambiar durante su ejecución. Si ese proceso lleva mucho tiempo en **ready** se le irá subiendo poco a poco la prioridad **(aging)** hasta que antes o después acabará entrando en **run**.

  Para esto, en la interrupción de reloj, se mira la cola de **ready** para ir incrementando las prioridades de esos procesos.

Los sistemas actuales incluyen una mezcla de todo: *quantum*, round robin y si hay empate FCFS.

¿Cómo se implementa?

En un Sistema Operativo actual, no hay una sola cola de **ready**. Hay colas multinivel; varias colas de ready según la prioridad y el planificador mira estas colas para tomar la decisión de qué proceso se va a ejecutar. Cada una de las colas implementa:

- **Colas menos prioritarias:** Procesos interactivos con mucha E/S -¿ FCFS. Si un proceso tiene mucha prioridad pero requiere mucha E/S el sistema operativo va a bajar la prioridad hasta que se ejecuten con FCFS ya que sabemos que van a estar abandonando la CPU constantemente.
- **Prioridades altas:** Round Robin con *quantum*. Son procesos intensos en CPU normalmente.
- **Intermedios**: Round Robin con *quantum* más bajo. Se les van subiendo y bajando las prioridades dependiendo de sus características.

Estas colas se extienden a lo que se llama **colas multinivel retroalimentadas**. Esto significa que hay prioridades dinámicas y los procesos al salir de la CPU van a una cola u otra; la CPU va tomando estadísticas y según lo que ve decide una cosa u otra de forma dinámica.

3.8 Threads

Hasta ahora, los procesos son lo que realmente se ejecuta. Este modelo es bastante limitante. Muchas veces, los procesos van a ser cooperativos y van a necesitar compartir información entre ellos. Con lo explicado hasta ahora, para compartir información, los procesos deben recurrir al sistema operativo; los procesos **no** comparten memoria, pero comparten el **sistema de ficheros**. El sistema operativo ofrece mecanismos, como las pipes, para poder pasar informaicón de un proceso a otro.

Pasar la información a través del sistema operativo no es eficiente y introduce bastante overhead en la ejecución de los procesos, porque si se quiere enviar información entre procesos no queda otro remedio que hacer una llamada al sistema (pipe), el sistema la procesa y se la pasa al otro proceso cuando este hace un read. Esto no es eficiente.

Por esta razón se crean los **threads**. Los threads pertenecen a un proceso y comparten todos los recursos del proceso entre ellos. Los threads son los que ejecutan las llamadas al sistema para pedir recursos. Pero estos recursos no se asignan a un thread, sino que se asignan al proceso al que pertenece el thread. A partir de ahora un proceso es un **contenedor de recursos** y el planificador ahora planifica **threads** de procesos.

Gracias a esto, todo el movimiento de un proceso a otro desaparece porque comparten los recursos. Ya no se comunican con pipes, sino que lo hacen a través de la memoria del proceso. Los threads no pueden mover a otro proceso.

Lo que nos tenemos que plantear en este punto es **¿cómo impactan los threads en los servicios que se tienen en el sistema operativo y qué modificaciones se deben hacer?**

Información necesaria para la ejecución de un thread

En Linux el proceso "deja de ser real": realmente el proceso no está guardado en ninguna estructura; el `task_union` que se estaba utilizando para guardar los datos del proceso pasa a ser del **thread**. La información del thread se utiliza el `task_union`, dónde hay el `task_struct` **del thread** y la pila **del thread** ya que lo que se ejecuta ahora es el thread.

En Windows dentro del PCB del proceso hay una lista de los TCB (Thread Control Block) que son la información de cada uno de los threads de un proceso.

El thread debe tener un **TID** (Thread Identifier); en Linux este atributo es único en el sistema. Esto es debido a que no existe realmente el proceso, todo son **threads** deben tener un identificador global. En Windows los **TID** son locales la proceso, el **TID** dentro del proceso es único, pero se puede repetir en otro proceso. Para identificar un thread de forma global en Windows se usa el **PID**.**TID** (Identificador del proceso y identificador del thread).

Cada thread debe tener una **pila de sistema** y una **pila de usuario**. Tamién es propio de **cada thread** la variable `errno` ya que ahora es el thread que ejecuta las llamadas al sistema. Cada thread va a tener un **contexto de ejecución** propio. Por último los threads tienen lo que llamamos **TLD** (Thread Local Storage), dónde se van a guardar variables locales de ese thread (por ejemplo el `errno`).

Como en Linux el **proceso** no existe y son los **threads** los que tienen el `task_union`, el planificador no se debe modificar porque ya se usaban los `task_union` para moverlos entre los distintos estados. Ahora en vez de `task_union` de procesos serán de threads.

Por otro lado en Windows, como tenemos una estructura a dos niveles, se debe hacer una planificación a dos niveles: primero planifica threads y luego procesos. El planificador siempre va a intentar que el siguiente thread que se ejecute pertenezca al mismo proceso que el `current()`. Esto es una optimización porque, en el `inner_task_switch` una de las cosas menos eficientes es el `set_cr3` ya que hace un flush del TLB; esto significa que al meter un nuevo proceso en CPU se va a hacer un flush. Para evitar esto, si se hace un cambio a otro thread del mismo proceso no se debe hacer el flush del TLB. Por esto siempre se intenta cambiar por un thread del mismo proceso. Pero, cuando el proceso tenga que abandonar la CPU para dejar ejecucion a otro proceso o porque ha acabado, es cuando se usa el segundo nivel para seleccionar un thread de otro proceso.

Como no se debe alocatar memoria, la creación de threads es muy rápida. Para eliminarlos también es muy rápido, solo se debe eliminar el PCB o el `task_union` correspondiente (o el TCB en Windows). Si se mata el thread principal, sí se deberá liberar la memoria.

3.8.1 Race conditions

```
0    while(pos<N){
1        data[pos] = datos;
2        pos++;
3    }
```

Un problema que surje a raíd de la compartición de datos entre los threads son las *race conditions*.

Se puede dar el caso en el que, entre una instrucción y otra de un thread el

planificador lo intercambie por otro thread.

Imaginemos que el código de arriba, que inicializa un vector, se ejecuta usando varios threads. Como no se puede saber cuándo el planificador va a sacar un thread de la CPU podría pasar que el planificador sacara un thread justo antes de ejecutar la línea 2. Ahora el **thread 2** ejecuta este mismo código y lo que va a hacer es machacar los datos del **thread 1** y aumentar pos. Cuando vuelva a ser ejecutado **thread 1**, este se quedó en la línea 2 del código, por lo que ahora va a ejecutarla: incrementa la variable pos sin hacer nada con los datos dejando una posición del vector vacía. La inicialización no está siendo correcta y va a tener espacios vacíos.

La compartición de los datos de variables globales compartidas entre threads provoca que la ejecución de los programas no tenga los resultados esperados. Para solucionarlo, se deben detectar estas secciones de código y modificándolas manualmente (no existen formas automáticas) para que se ejecuten en **exclusión mutua** (de forma atómica). El planificador aún puede sacar el proceso de la CPU, pero la exclusión mutua garantiza que ningún otro thread va a entrar en ese segmento de código hasta que el thread que lo tiene acabe.

Mecanismos de exclusión mutua

El mecanismo que ofrece el sistema operativo para marcar zonas de exclusión mutua son los **semáforos**. El semáforo es una estructura bastante simple:

```
struct sem_t{
    int count;
    list_head blocked;
}
```

En la estructura `list_head` es dónde se van a bloquear todos los threads que están esperando para entrar dentro de la zona de exclusión mutua (para que no estén en la cola de ready y el planificador no los pueda poner a ejecutar).

Las llamadas al sistema deshabilitan las interrupciones. Sin la interrupción de reloj el planificador no se ejecuta, por lo que una llamada al sistema **no puede ser interrumpida**. Por ese motivo, los mecanismos para trabajar con estas zonas de exclusión son **llamadas al sistema**.

`sem_init()`

Esta llamada de inicialización de un semáforo recibe como parámetros un puntero a un semáforo (`sem_t *s`) y un valor inicial del contador. Sencillamente lo que hace es inicializar el contador con el valor que se le pasa e inicializa la lista para que sea una lista vacía:

```
sem_init(sem_t *s, int count){
    s -> count = count;
    INIT_LIST_HEAD(&s->blocked);
}
```

`sem_wait()`

Esta llamada sirve para marcar el inicio de una zona de exclusión mutua. Con esta llamada se indica que se ha entrado dentro de una zona de exclusión mutua y que ningún otro thread puede entrar.

En caso de que ya haya un thread ejecutando esta zona, se va a bloquear a la espera de poder entrar en la zona de este semáforo.

```
int sem_wait(sem_t * s){
    s -> count--;
    if(s->count < 0){
        list_add_tail(&current()->list, &s->blocked);
        sched_next();
    }
}
```

Si el contador es menor que **0** significa que el proceso se debe bloquear. Para bloquearse se añade el mismo a la cola del semáforo (blocked) y a continuación llama al planificador para que ejecute otro thread.

`sem_post()`

Marca el final de las zonas de exclusión mutua. Incrementa el valor del contador y si este valor es menor o igual a cero, significa que hay otro thread esperando a entrar, así que lo que se hace es sacar el primero de la cola del semáforo (blocked) y se añade a la cola de ready.

```
sem_post(sem_t * s)
{
    s->count++;
    if(s->count <= 0){
        list_head *l = list_first(&(s->blocked));
        list_del(l);
        list_add_tail(l, &ready_queue);
    }
}
```

Tomando de nuevo el ejemplo de la inicialización de un vector que hemos visto al principio, usando semáforos quedaría de la siguiente forma:

```
int v[10^10];
int pos;

sem_t s;
sem_init(&s, 1);

sem_wait(&s);
v[pos] = data;
pos++;
sem_post(&s);
```

¿Y si se inicializa el contador del semáforo en `sem_init()` con un valor de 0? Si se hace esto, como el `sem_wait()` lo primero que hace es decrementar el contador, lo que pasaría es que nunca entraría ningún thread a ejecutar la zona de exclusión y quedarían todos bloqueados. Esto se usa para crear semáforos que van a servir para **sincronizar** threads; en concreto para secuencializar el orden de ejecución de instrucciones de dos threads.

Si tenemos dos threads T1 y T2 dónde T2 **consume** datos que genera T1, no se puede permitir que las instrucciones de T2 se ejecuten **antes** que las de T1. T2 debe ejecutarse cuando ya existan los datos que ha generado T1. En este caso, se necesita **sincronización**: los threads se deben ejecutar de forma **secuencial** (primero T1 y después T2).

T2 debe esperar en el caso que la última instrucción de T1 no se haya ejecutado aún.

```
int v[10^10];
int pos;

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

THREAD 1	THREAD 2
A	sem_wait(&s, 0);
B	A1
C	B1
D	C1
E	D1
F	E1
sem_post(&s);	F1

Figure 18: Sincronización de threads con semáforos.

Para hacer esto, se usa un `sem_wait()` con un semáforo con el **contador a 0** al inicio de las instrucciones de T2. En T1 no se pone ningún `sem_wait()`, pero al

final de la ejecución de T1, después de ejecutarse la última instrucción habrá un `sem_post()`. Este `sem_post()` desbloqueará a T2 para que se siga ejecutando.