TEMA 4 Procesos

Un proceso es una unida de actividad con una secuencia de instrucciones a ejecutar, un estado y un conjunto único propio de recursos asociado. Su memoria lógica contiene un código, unos datos y una pila. Tiene un contexto hardware y un contexto sofware.

Para aprovechar mejor los recursos limitados del hardware se utiliza la técnica de concurrencia y paralelismo, que consiste en repartir los procesos entre las CPU's para que parezca que se ejecutan a la vez.

En concurrencia: con el tiempo de CPU se multiplexan los procesos.
|--P1--|--P2--|--P3--|

Cuando tienes más de una CPU, se trabaja en paralelo entre CPUs y de forma recurrente dentro de esta CPU.
CPU1 |--P1--|--P3--|--P2--|

CPU2 |--P3--|--P1--|

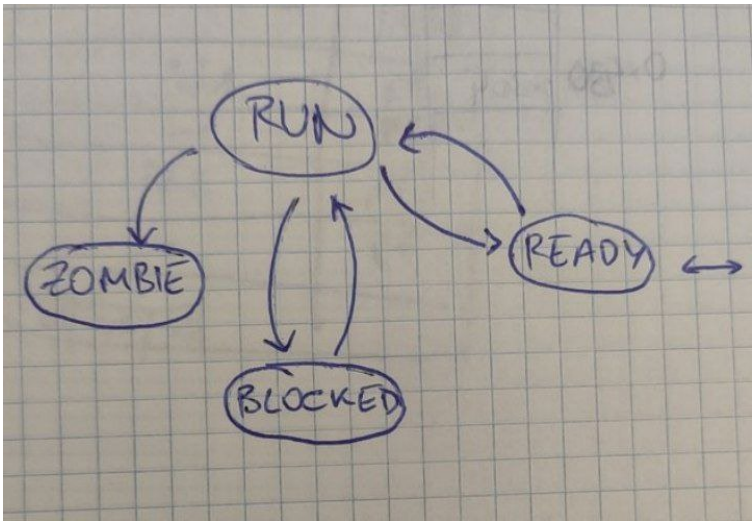
El usuario puede gestionar estos recursos a través de llamadas de sistema: creación, consulta y destrucción.

Cada proceso de sistema tiene:
· Una **PCB** (Process Control Block), que contiene:
- PID
- Estado
- Recursos asociados (págs de memoria, ficheros...)
- Estadísticas del proceso (cpu consumida, memoria ocupada...)
- Información de planificación (prioridad, quantum...)
- Contexto de ejecución (las @, ya que el contexto está en la pila).

· Una **pila de sistema** para cada proceso**.**

PCB Y ESTADOS DE UN PROCESO

El sistema tiene **un vector de PCBs** para procesos que se crearán o que se están ejecutando, este contiene todas las PCBs de que dispone y delimita el número de procesos máximo que podemos tener, y que son 1000.



Hay un estado READY en el que se pueden ejecutar. Una vez ejecutándose están en estado RUN, de aquí puede pasar a ready si se acaba el quantum, o a blocked, o a zombie. El estado ZOMBIE es el estado de estancamiento, ha liberado todos los recursos menos el PCB, cuyo padre todavía no ha leido el estado de finalización. Y BLOCKED si está realizando una operación de larga latencia, ej. esperando una interrupción E/S, puede pasar a run si le llega lo que esperaba.
Cada estado, menos RUN y ZOMBIE, tiene una estructura correspondiente, una cola de PCBs:

- Freequeue: es una lista de posiciones de PCBs libres para no tener que buscar en todo el vector (costoso), así solo se coge el primero de la cola.
- ReadyQueue: es una lista de los PCBs de los procesos que pueden seguir ejecutándose.
- WaitQueue: para los procesos bloqueados.

En el caso de los procesos ZOMBIE, sus PCBs están ocupadas en el vector, pero no están encoladas.

PILA DE PROCESO

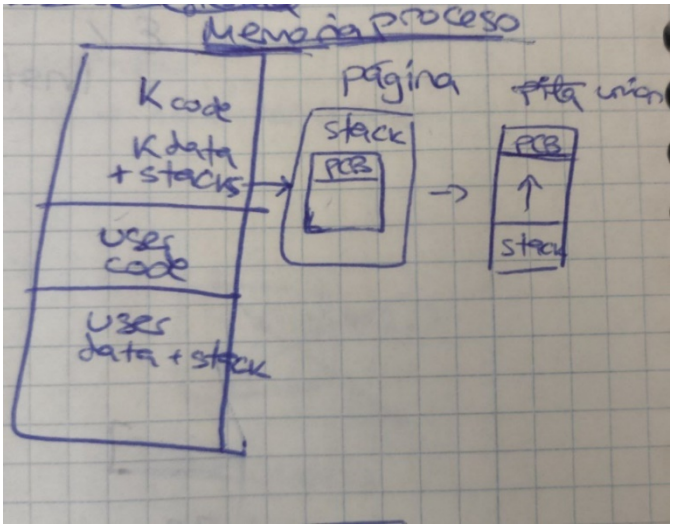
Por otro lado, tenemos también la pila de un proceso, que tiene tamaño de una página (4KB). Es en la cima de esta pila donde tenemos el PCB, ya que cuando pasa a estado de RUN se ha de poder acceder a su task_struct, pues ese estado no tiene una cola de PCBs con la que acceder a este (si está en ready lo encontramos en la readyqueue).
Para acceder a este PCB tenemos:

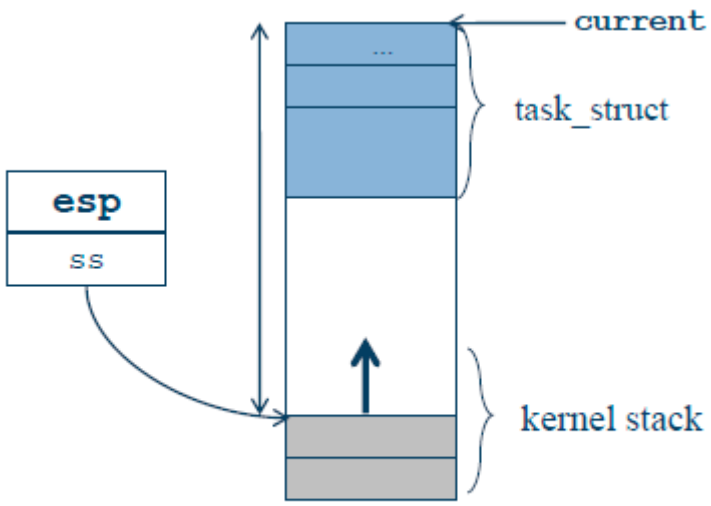
- Windows: el registro GS que apunta directamente al PCB del proceso.
- Linux: union task_union

El **union task_union** es la estructura que une los dos elementos del proceso, el PCB y la pila.

```
union task_union {
        unsigned long stack[1024]
        struct task_struct task
}
```

La pila de sistema y el PCB están en la misma página de memoria, y el PCB se encuentra en el top de la pila.





Task_union

El registro esp apunta a la mitad del union/pila, para acceder al PCB hay que lograr apuntar al top de la página. Como las páginas están alineadas a 4KB, el inicio de una página empezará por 0xXXXXX000, así que solo tenemos que aplicar una máscara a la @ que ya tiene de la pila el registro %esp:

%esp & 0xFFFFF000

```
Current()
{
        eax <- esp
        eax <- eax AND 0xFFFFF000
        ret
 }
```

CAMBIO DE CONTEXTO

En los sistemas con un solo thread, para ejecutar múltiples procesos de manera alternante, se realiza un cambio de procesos extremadamente rápido. Para poder hacer esto necesitamos hacer un cambio de contexto.
Pasamos el proceso viejo de RUN a READY, y el nuevo lo pasaremos de READY a RUN.
Este cambio de contexto se hace en una función del SO que se llama **task_Switch(*proceso nuevo). Todos los procesos han de pasar por aquí para poder entrar en la CPU.**

Este cambio generalmente realiza dos pasos:

- Guarda el contexto de ejecución del proceso actual.
- Restaura el contexto de ejecución del nuevo proceso.

```
void task_switch (union t_union *new)  #puntero al union del nuevo proceso
{
      push ebp
      ebp <- esp
_____
      current() -> task.kernel_esp = ebp
      esp <- new -> task.kernel_esp
_____
      pop ebp
      ret
}
```

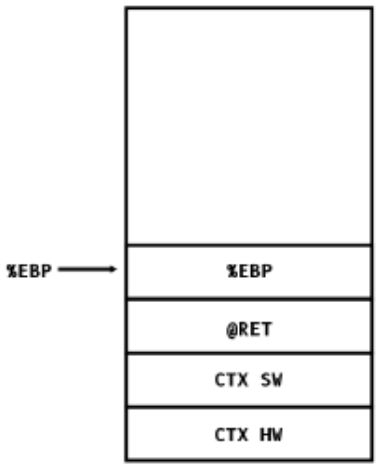
1- GUARDA EL CONTEXTO DEL PROCESO ACTUAL

Primero de todo, al ser una ejecución de alto nivel del kernel, **utiliza el reloj (clock_routine) para llegar al task_switch, puesto que interviene el planificador, que se invoca dentro de esta rutina.**
Al saltar la interrupción de reloj, accede a la IDT, consulta GDT desde donde accede a TSS, un registro que se utiliza para el cambio de contexto. Aquí encuentra la @base de la pila de sistema y empila el contexto hardware. La @base porque al pasar de modo user a sistema esta pila estará vacía.

Ejecuta el handler del reloj, recordamos:

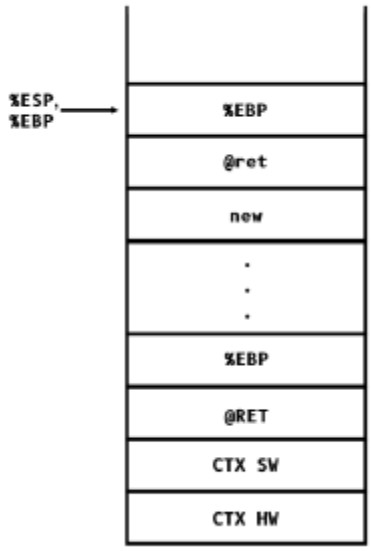
- SAVE ALL (guarda contexto SW en la pila)
- CALL clock_routine
- EOI
- RESTORE ALL
- IRET

En SAVE ALL guarda en la pila de sistema el Ctx Sw, y en el CALL la @ de retorno.
Dentro de la rutina de reloj, hace push ebp, mov ebp<-esp, y la pila de sistema queda así:



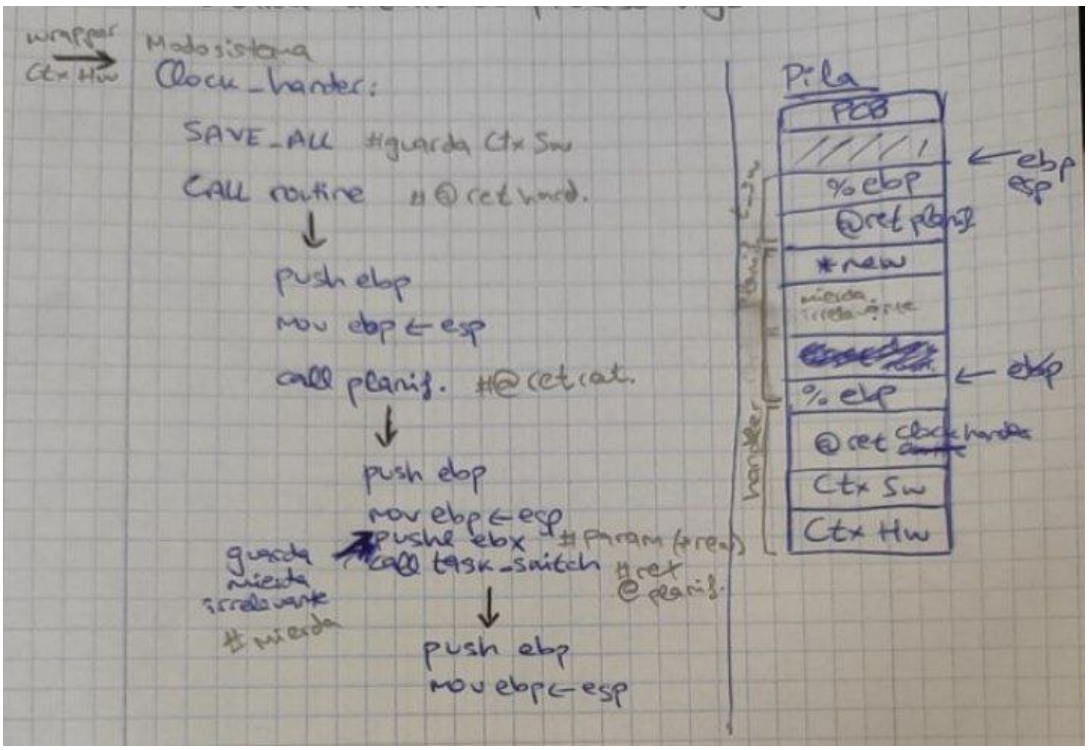
Ahora en la rutina llama al planificador, que empila muchas cosas (no relevante) y decide hacer el cambio de contexto. Llama a task_switch.

En la cima de la pila de sistema, después de todo esto, estará el parámetro de la función task_switch, que era el puntero al PCB del nuevo proceso new (struct task_struct *new), y la dirección de retorno del planificador, el que ha llamado a esta función, y que cuando llamamos a task_switch hará push ebp, mov ebp <- esp, por lo tanto la pila quedará así al inicio de la función:



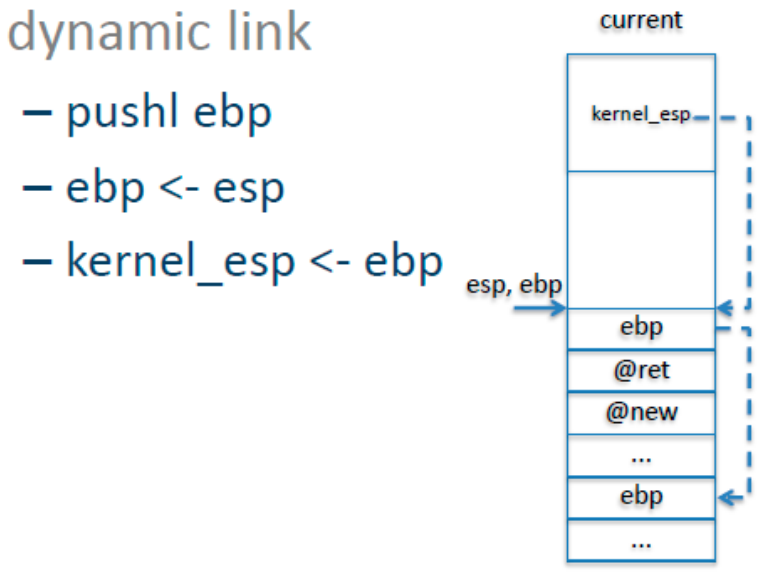
Recordemos que esta pila se encuentra en la página (union) donde también tenemos la PCB del proceso current.
La pila del proceso new será muy parecida, ya que este proceso usó task_switch para entrar en la CPU (suponemos que es un proceso que ya existía, estuvo RUN y se sacó de allí).

2- TASK SWITCH, CAMBIO Y RESTAURACIÓN DE CONTEXTO

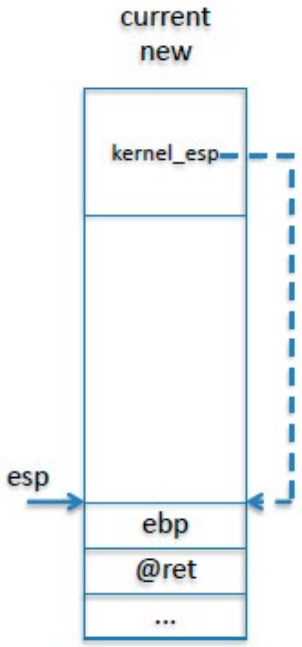


```
1  inner_task_switch (union task_union * new)
2  {
3    PUSH EBP
4    EBP <- ESP
5
6    set_cr3(new -> task.dir);
7    tss.esp0 <- &new->stack[1024]
8
9    (current() -> task.kernel_esp) <- esp;
10   ESP <- (new -> task.kernel_esp);
11
12   POP EBP
13   RET
14 }
```

Cambiar puntero de tabla de páginas: En primer lugar, cambiamos el **CR3** para apuntar al nuevo directorio de páginas del nuevo proceso, ya que las traducciones son únicas para cada uno, el espacio lógico y físico no es el mismo. Al escribir en el CR3 se hace un flush del TLB por lo que habrá una ráfaga de fallos de TLB, por esto es la línea más costosa.
Cambiar puntero de pila de sistema: El registro **TSS.esp0** sirve para encontrar la pila de sistema del proceso actual, por lo que también hay que cambiar su @, ahora apunta a la **base de la pila** de sistema de new (&new->stack). A la base, porque solo se consulta por la CPU cuando va a saltar a sistema, por lo que la pila tendría que estar vacía, y no se vuelve a consultar hasta que salga a usuario y vuelva a entrar.
Guarda el estado del proceso current: dentro de la PCB que aún es current crea una nueva variable kernel_esp para guardar el puntero esp actual, la cima de la pila de sistema de current (si no hiciera esto perdería la referencia de esta pila dentro del union), esta cima contiene el ebp.
La pila queda así:



Cambia el valor esp (top de pila de sistema): Como sabemos que el proceso new en su dia pasó por el task_switch, también tendrá esta variable kernel_esp, ahora el registro esp de la CPU tiene que guardar el contenido de esta variable de new para apuntar a su pila de sistema (esp <- new.kernel_esp).



Recupera ebp: Haciendo pop ebp recupera el valor de ebp, haciendo que ebp ahora apunte a la base de la pila que había antes del task_switch.
Retorna: a la dirección de retorno que queda que lleva de nuevo al planificador, de ahí viene el @ret al handler, finalmente sale al modo usuario.

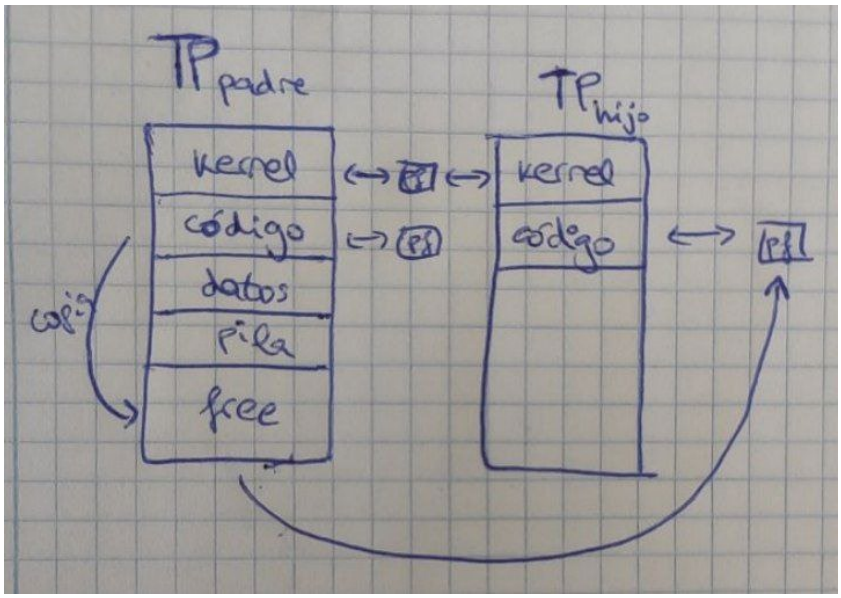
CREACIÓN DE PROCESOS

La creación de procesos se hace con la función **int fork()** que crea un proceso a través de una copia de su código, datos y pila del padre. Este proceso devuelve el pid del hijo al padre, y al hijo un 0. En caso de error le pasa al padre un -1.

1- **Mira si hay recursos:**
   · **Asigna PCB:** mira si hay PCB disponible y le asigna el primero de la FreeQueue.
   · **Asigna tabla de páginas:** la asigna un directorio de páginas.
   · **Asigna memoria física:** bloque de direcciones físicas
2- **Inicializa datos de sistema (contexto de ejecución):** inicializa el union del hijo. Llena el task_Struct asignado y una pila de sistema copiando el contenido de la PCB y la pila (union) del padre al hijo.
3- **Copia la memoria:** Una vez el hijo tiene su tabla de páginas asignada se realiza la copia de datos (código, datos y pila). Esta tabla de páginas del padre tiene varios segmentos definidos: mapeos del **kernel,** segmento de **código,** el de **datos,** y el de la **pila.** La tabla del hijo está vacía. Depende del segmento se copiará de cierta manera:
   - **Segmento de kernel y código de usuario:** las páginas físicas de estos segmento son compartidos por todos los procesos, pues el kernel es el mismo para todos, y el código es

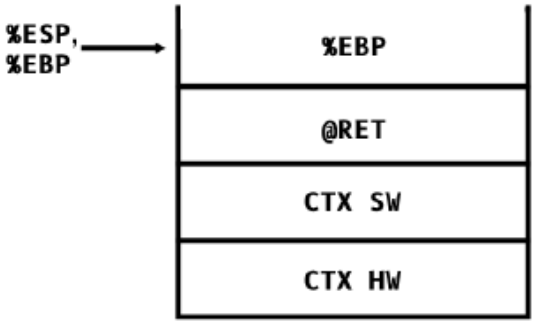
común entre padre e hijo. Por eso las traducciones de estas páginas se copian exactamente igual (apuntan a las mismas páginas físicas).

- **Segmento de datos y pila:** las demás páginas físicas han de ser diferentes y no compartidas, tienen copias, pero son independientes y cada proceso accede a las suyas, si intenta acceder a otra página saltará un error. Por lo tanto, queremos que el proceso hijo tenga unas **páginas físicas diferentes pero que contengan una copia** de su contenido. Para eso, se mira la tabla del padre y por cada entrada lógica asigna una página física al hijo, así **tantas páginas físicas como tenga el padre**.
  Una vez tiene ya las páginas físicas asignadas, se va a copiar el contenido.
  El problema: es que la **CPU trabaja con @lógicas, y no puede acceder directamente a las páginas físicas** si no es a través de la traducción de la TLB. Además, la CPU solo tiene un registro CR3, y ya apunta a la tabla del padre, por lo que la TLB no puede mirar la tabla del hijo.
  La solución: en la tabla del padre tenemos páginas libres sin asignar a físicas. Por casos como este **un proceso no puede ocupar toda la tabla de páginas.** Además, al estar en modo sistema el que ejecuta estas acciones es el sistema y no el padre, por lo que **el sistema sabe qué @físicas le ha asignado al hijo**. Lo que hace es **asignar las páginas del hijo a las entradas lógicas libres de la TP del padre de manera temporal. Así, traduce estas entradas a las @físicas del hijo,** luego borra el mapeo temporal de la TLB. Finalmente hace un **set_Cr3,** que hace flush de la TLB y quitar los mapeos temporales de allí también.

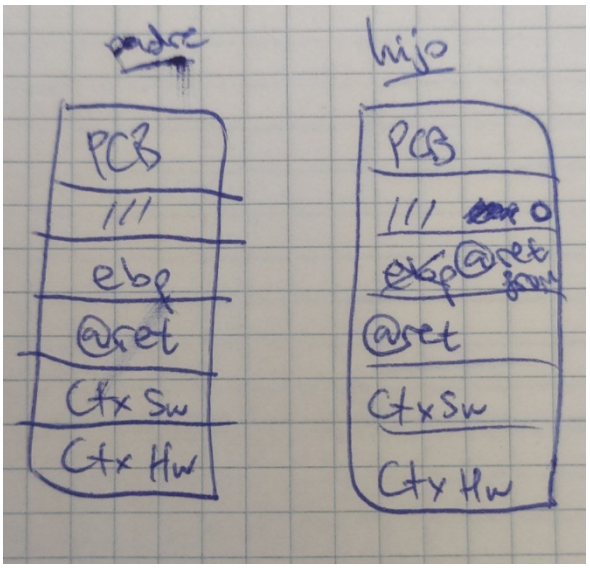


4- **Actualiza otras estructuras:** vector de signals, tabla de canales, etc. Por ejemplo, esta tabla de canales la copia igual que del padre. Hay una tabla de canales para cada proceso, y una de ficheros abiertos para todo el sistema.
5- **Asigna PID:** número aleatorio que no de información real sobre el proceso. Si está ocupado incrementa secuencialmente.
6- **Prepara contexto ejecución del hijo:** Actualmente el contexto de ejecución lo ha heredado del padre haciendo una copia de su union. El PCB del hijo ha de ser compatible con el proceso task_switch, puesto que va a pasar a ejecutarse a través de este.

La pila de sistema del padre, que ha pasado por el task_switch para hacer fork, es así:



El hijo tiene lo mismo, por lo que tiene lo necesario para volver, pero, el kernel_esp guardado en la PCB apunta al ebp de la pila, que todavía guarda la @ del ebp padre, y eso no puede ser, hay que cambiarlo.

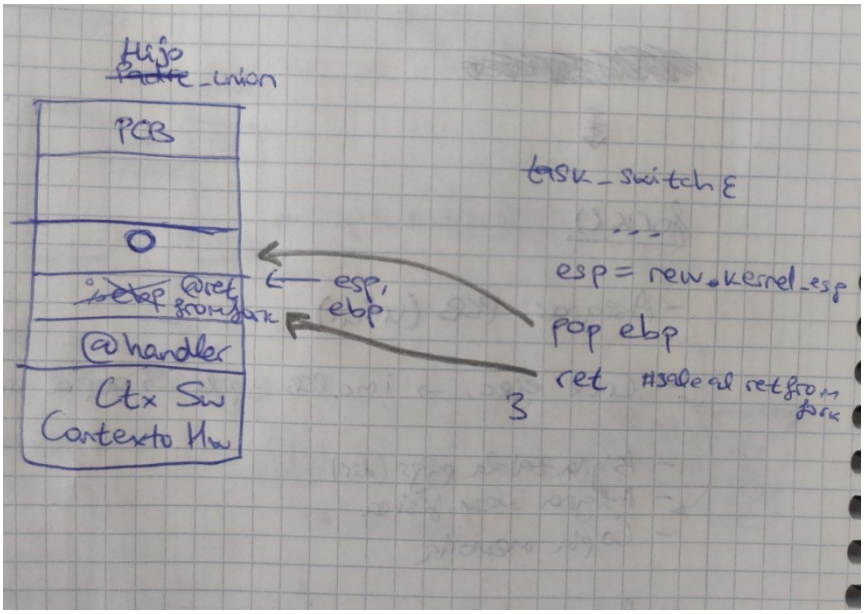


Además, tenemos que devolver el valor 0 en eax, por lo que tenemos que modificar el contexto sw antes de volver a modo usuario, ya que ahora mismo eax tiene lo que se salvó en su momento.

Esta modificación del eax del hijo la haremos en una función llamada ret_from_fork():

```
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 }
```

Para que el hijo ejecute esto antes de nada, hay que modificar su pila.
Sabemos que el task_switch, después de cambiar el contexto/pilas, hace: pop ebp, ret. Por lo tanto, ponemos un valor 0 donde estaba el ebp padre, y debajo la @ret_from_fork para que salte allí.



Ahora kernel_esp apunta al 0, y cuando salga de task_switch saltará al ret_from_fork(), cambiará eax y retornará al handler.

Por último, nos fijamos en el flujo de ejecución de cada proceso, padre e hijo.

USER ➡ FORK ➡ HANDLER ➡ SYS_FORK ➡ HIJO

CLOCK ➡ HANDLER CLOCK ➡ CLOCK ROUTINE ➡ SCHEDULER ➡ INNER TASK SWITCH ➡ RET FROM FORK ➡ HANDLER SYSCALLS ➡ MODO USUARIO (PROCESO HIJO)

El proceso padre llama al fork, entra en modo sistema, accede al handler, ejecuta sys_fork y crea al hijo. Y sale por el handler a modo usuario.
Sin embargo, el proceso hijo entró por el esquema de cambio de contexto, a través de la rutina de reloj hasta el task_switch, y al ejecutar el fork sale por el handler_syscall para salir a modo usuario.
El problema evidente es que al entrar por el handler de reloj, y no salir por él, no se ejecuta el EOI, y ya no podríamos ejecutar más interrupciones. Por este motivo el EOI se ejecuta antes del call en el handler del reloj.

7- **Encola el proceso en readyqueue**
8- **Devuelve el PID al padre**

DESTRUCCIÓN DE PROCESOS

Cuando un proceso acaba lo hace con sys_exit(), libera todos sus recursos menos la PCB, estos procesos están en estado ZOMBIE mientras el padre espera el código de finalización del hijo, 0 si ha ido bien, negativo si ha ido mal. El sys_exit() tiene un parámetro para guardar este código dentro de la PCB y devolverlo al padre, libera los recursos (TP, tabla de canales...), e invoca al planificador para que ejecute otro proceso. Cuando el padre haga wait_pid() y reciba este código, liberará el PCB del hijo y por fin morirá.
El PCB del hijo es almacenado en el vector de PCB's mientras se encuentra en este estado transitorio.

PROCESOS INICIALES

Estos procesos se crean e inicializan en tiempo de boot.

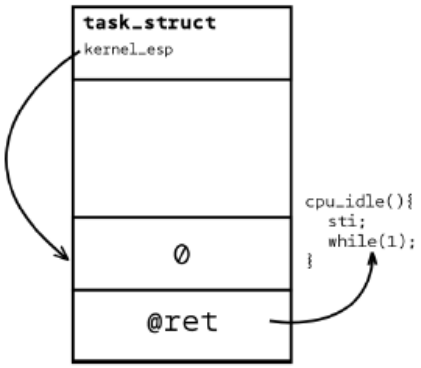
Idle():
Se ejecuta cuando no hay procesos en ready que ejecutar, y solo contiene un while(1) para mantener la CPU en marcha.
Su inicialización consta de:

- Reserva PCB y PID=0. Idle_task = @PCB.
- Inicializa espacio de direcciones: asignar DIR
- Prepara contexto: 0 y @ret

Cuando llega una interrupción de reloj, mira si está corriendo idle, si está ejecutándose mira si la cola ready está vacía, si lo está, sigue ejecutándolo, si no, pasas a ejecutar un proceso.

Realmente, idle se utiliza para hacer limpieza en el SO, y es muy complejo puesto que al solo ejecutarse en modo sistema, no tiene contexto de usuario, ni nunca estará en la cola de ready. Por esto el **sistema tiene un puntero directo a idle para encontrarlo, *idle_task,** cada vez que lo necesite.
Al ejecutarlo, **no guarda ningún contexto, pero guarda lo necesario para ser compatible con task_switch**, por lo que tendrá en su pila un ebp y un @ret que apunta al código while(1), y su kernel_esp apunta a un falso ebp.

Su task_union queda así:



Su PID = 0, es el primer PCB del vector y no puede cambiarse.
La TP es parte del kernel de la pila y el directorio se crea a mano.

Init():
Es el primer proceso que se ejecuta en modo usuario.
Saltar a init se hace de forma especial, sin fork.

Su inicialización consta de:

- Reserva de PCB (union) y PID=1
- Inicializa espacio de direcciones: asigna DIR y mapea traducciones
- Pasar a ejecutar proceso: Actualizar TSS y CR3. En el boot, se indica que la pila y TP del proceso a ejecutar es el de init.

Se hace set_cr3(init->dir) para indicar la TP, y tss.esp0 = &init->stack[1024] para indicar la base de la pila de init. En el MSR 0x175 se le pone la pila init, para que al leer el registro salte a este proceso.
Es el único proceso que no accede a la CPU por el task_switch.

PLANIFICACIÓN

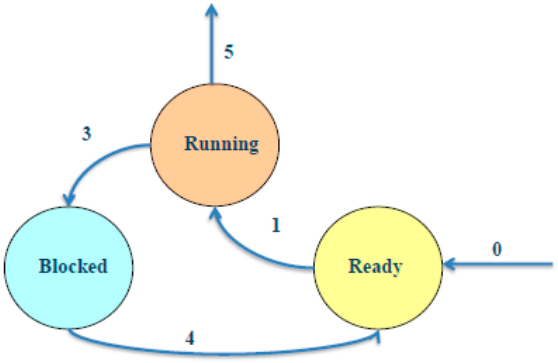
La política de planificación es la que decide cuando se ejecutarán los procesos, y el que lo lleva a cabo es el algoritmo de planificación.

Hay tres **clases de planificadores**:

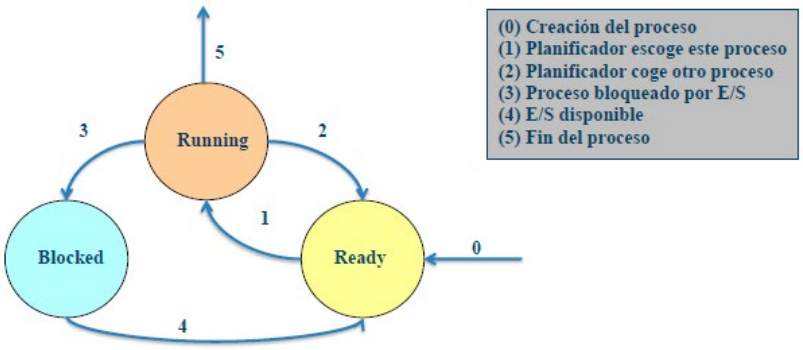
- Corto plazo: procesos pendientes de la CPU. Decide cuando se va a ejecutar un proceso (cuando pasará de reday a run) y cuál irá a continuación. Multiplexa el tiempo de CPU entre todos los procesos.
- Medio plazo: Cuando está en idle o tiene la memoria física saturada hace housekeeping, hace limpieza y libera la memoria.
- Largo plazo: decide si el proceso de usuario se puede crear o no a partir de los recursos disponibles.

Hay tres **políticas de planificación a corto plazo**:

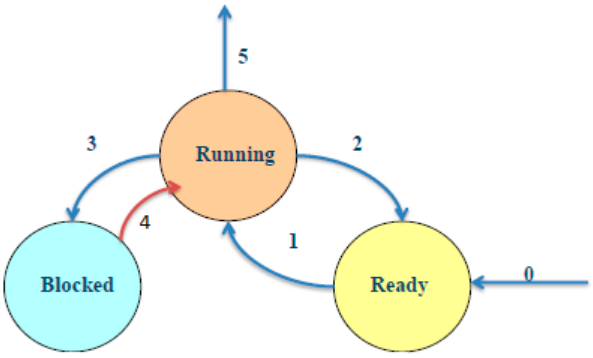
- No apropiativa: la cola de ready usa un algoritmo FIFO**. Nunca expulsa al proceso**. Va bien cuando solo hay procesos con muchas operaciones E/S que alteran mucho entre run y blocked. Los de cálculo ocupan la CPU hasta que acaban, por lo que si son mixtos y tienen cálculo de CPU y operaciones E/S, va fatal, porque con esta política, los procesos de cálculo se quedan al principio y los de E/S al final.



- Apropiativa diferida: Utiliza el algoritmo Round Robin, **puede expulsar a un proceso y dárselo a otro (apropiación).** Se define un **quantum (5-8ms) por proceso que indica el tiempo que puede estar el proceso en modo run.** Si lo gasta, el SO lo saca de la CPU al modo ready. **Se comprueba el quantum restante con cada tick del reloj**. El quantum no ha de ser muy largo porque si no sería un FIFO, ni muy corto para que no haga demasiados task_Switch (ineficientes por el flush de TLB). Aquí aparece la necesidad de dar prioridad a los procesos de administrador sobre los de usuario.



- Apropiativa inmediata: **Igual que el diferido, pero se efectúa al momento en que se produce un cambio.** Se determinan prioridades sobre los procesos, si entra en ready un proceso con más prioridad que el current, se le saca y mete al primero. Igual que si un proceso blocked termina su espera, o si se crea un proceso nuevo, y tienen mayor prioridad que el current, se le saca del estado de run y se mete al primero.
  Las prioridades se asignan:
  · Estática: se crea el proceso y se le asigna prioridad que no puede cambiar. El problema de **starvation** implica que los de más baja prioridad nunca entran en la CPU.
  · Dinámica: se le asigna una prioridad inicial y va cambiando. Si lleva mucho tiempo en ready se le va subiendo la prioridad (**aging**) hasta que en algún momento entre a la CPU.



Un SO actual no solo tiene una cola de ready, tiene varias **colas multinivel retroalimentadas**, según la prioridad. Para los procesos de **E/S** tiene colas con la prioridad tan baja que sean FIFO,

para los de **CPU** usa quantum intermedio. Para los procesos **mixtos**, usa un quantum bajo y se va cambiando según las necesidades. Así, las prioridades son dinámicas y los procesos al salir de la CPU van a una cola u otra según las necesidades.

THREADS

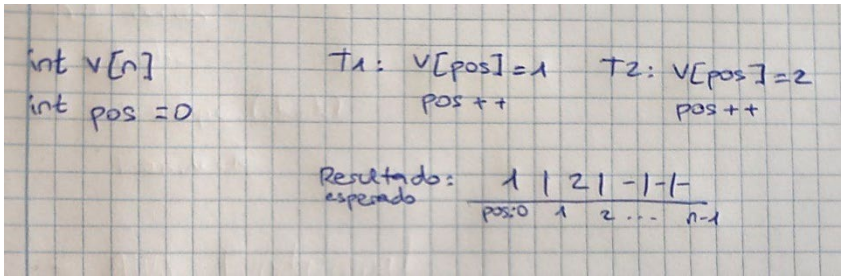
Los procesos no comparten memoria pero sí sistema de ficheros, por lo que usan mecanismos como las pipes para pasarse información unos a otros, pero no es eficiente porque es una llamada a sistema que se lee cuando el otro proceso hace read, es lento. Por esto se crean los threads, que pertenecen a un proceso y comparten entre sí todos los recursos de este. Un recurso se asigna al proceso al que pertenece cada thread, que es el que lo pide. Por esto un proceso es un contenedor de recursos y planificador de threads, así ya no tienen que comunicarse los procesos, porque comparten los recursos.

El PCB pasa a ser el TCB y los recursos son los mismos que un proceso, TCB con su TID (identificador) y pila de sistema por cada thread, y pila de usuario. Tendrán su contexto de ejecución y su TLD, su parte de memoria para sus variables propias. Hay dos niveles de planificación, de threads y de procesos, cuando se hace un cambio de thread no se hace flush de TLB, pero sí cuando se cambia de proceso. Matar al thread principal sí libera memoria pero la creación de threads no alocata memoria.

En Linux, la implementación de threads no cambia el planificador porque ya no existen los procesos, seguimos utilizando task_union, pero en Windows se implementa el planificador de dos niveles, uno para procesos y otro para threads, para intentar minimizar el número de flush de TLB, que no se hacen en caso de cambio de thread, solo de proceso.

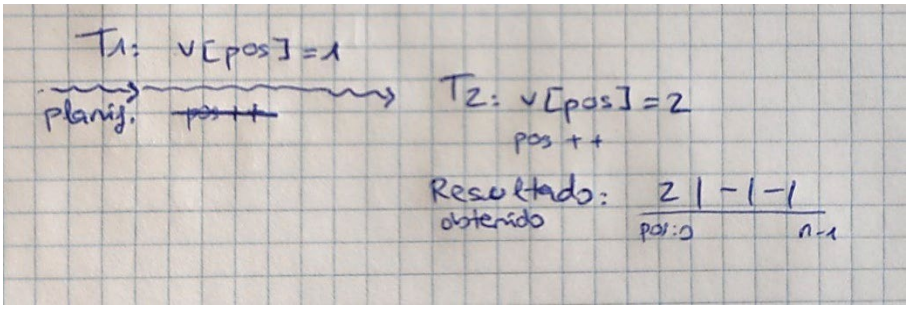
En la compartición de recursos hay un problema de **Race Conditions:**

Imaginemos que tenemos dos variables globales, declaradas antes de la función de un thread (para que pueda utilizarlas), un vector int v[n] y un pointer int pos = 0. Queremos que 2 threads utilicen estas variables.



El planificador puede quitar un thread para meter otro, y como no sabemos cómo funciona este, puede empezar a verse cosas raras.

Es posible que el planificador irrumpa en mitad del T1, y pase lo siguiente: pone un 1 en v[0]. Antes del pos++ que le sigue, el planificador quita a T1 y casualmente pone al T2, en v[pos], pos sigue siendo 0 por lo que sustituye el 1 por el 2. Ahora se ve esto:



Según como se están ejecutando estos threads el resultado puede no ser correcto, pues se está ejecutando una variable global, pero el procesador puede saltar en cualquier momento. Esto se llama race condition, los cuales el planificador no detecta.

Para solucionarlo, el programador ha de detectar estas zonas problemáticas y "marcarlas" para asegurar su fiabilidad. El planificador aún puede sacarlo de la CPU, pero al menos otro thread no puede entrar a ese segmento de código hasta que no acabe el thread anterior.

Mecanismos de exclusión mutua:
El marcaje de estas zonas se llama exclusión mutua y se hace a través de semáforos. Un **semáforo** es una variable de sistema que indica si se puede entrar a una región de código, si se puede, entra, y si no, bloquea el thread. Su estructura es esta:

```
struct sem_t{
        int count,  #contador. indica la cantidad de procesos simultaneos que permite
        struct list_head blocked,  #lista de espera de threads que quieren acceder a la zona restringida
}
```

Las syscalls (o en modo sistema) deshabilitan las interrupciones, sin la interrupción de reloj no puede saltar el planificador que interrumpa la llamada, por lo que en modo sistema no saltará.

```
# inicializa la cola del semaforo
int sys_sem_init(sem_t *s, int value)
{
        s->count = value;       # inicializa el contador
        initializeListHead (&s->blocked)      # inicializa lista vacía
}

# marca el inicio de una zona de exclusión mutua, si el contador es menor que cero bloquea el thread que quiere entrar
int sys_sem_wait(sem_t *s)
{
        s->count--;
        if(s->count<0){
            block(current(), &s->blocked);  --->> list_add(&s->blocked, current());   #encola el thread
```

```
                                        sched_next();          # pasa al siguiente
        }

}
```

marca el final de la zona, desbloquea un thread que esté bloqueado para que siga ejecutándose, solo uno para que no haya race condition. Si es <= 0 es que hay un thread esperando.

```
int sys_sem_post(sem_t *s)
{
        s->count++;
        if(s->count <= 0)
        {
                unblock(&s->blocked);  -->> l = list_first(&s->blocked);   # saca el primero
                                              list_del(c);
                                              list_add(&ready_queue,l);    # lo mete en ready
        }
}
```

Siguiendo el ejemplo de antes, quedaría así:

```
int v[n];
int pos=0;

sem_t s;
sem_init(&s,1);

T1: sem_wait(&s);                T2: sem_wait(&s);
      v[pos] = data;                   v[pos] = data;
      pos++;                           pos++;
      sem_post(&s);                    sem_post(&s);
```

El T2 quedaría bloqueado, al acabar T1, sacaría T2 de la cola y lo enchufaría en ready.

Cuando el semáforo se inicializa con el valor 1 se hace un semáforo de **EXCLUSIÓN MUTUA**. Cuando el sem_init(&s, N) sería para **limitar el número de threads que puede ejecutarlo**, pero ya no se usa porque se usan otras herramientas como buffering.

Cuando lo pongo a 0, es de **SINCRONIZACIÓN**, y lo uso para sincronizar los threads, marcar un orden de ejecución entre estos. Si al init le ponemos (&s,0), el contador pasaría de 0 a -1, y se quedaría bloqueado en la cola del semáforo. El T2 quedaría de -1 a -2, y también se quedaría bloqueado.

En T1 no hay sem_wait, por lo que se ejecutaría, hasta que al final desbloquea la ejecución de T2, que sí fue bloqueado antes de empezar.
Se puede producir un **deadlock,** que consiste en que dos threads acceden a dos mismos recursos, y los bloquean entre sí provocando un ciclo de espera infinito.Se pueden evitar permitiendo quitarle recursos al thread, o ordenando las ejecuciones, o conseguirlo de manera atómica.

En Linux, la implementación de threads no cambia el planificador porque ya no existen los procesos, seguimos utilizando task_union, pero en Windows se implementa el planificador de dos niveles, uno para procesos y otro para threads, para intentar minimizar el número de flush de TLB, que no se hacen en caso de cambio de thread, solo de proceso.

```
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