

Sistemas Operativos  
2º Curso – Grado en Ingeniería Informática

# Tema 2:

---

# Procesos e hilos

José Antonio Gómez Hernández, 2016.



# Contenidos

- ▷ Conceptos de partida
- ▷ Implementación de procesos/hilos en Linux:
  - Descriptor de procesos/hilos
  - Diagrama de estados y transiciones
  - Operaciones sobre procesos
- ▷ Planificación de procesos/hilos:
  - Tipos de planificadores y algoritmos de planificación básicos
  - El planificador de Linux
  - Ahorro de energía

1.

# Conceptos de partida

**Conceptos sobre procesos e hilos**

# Conceptos a repasar

- ▷ Qué es un proceso y un hilo
- ▷ Qué es y qué contiene la imagen de un proceso
- ▷ Qué es el Bloque de Control de Proceso (PCB)
- ▷ Estados básicos de procesos/hilos
- ▷ Transiciones posibles entre estados y eventos que las disparan
- ▷ Qué es y cómo se realiza en cambio de contexto (proceso).

# Dónde podemos revisarlo

- ▷ W. Stallings, *Sistemas Operativos. Aspectos internos y principio de diseño*, 5ª Ed., Prentice Hall, 2005.
  - Capítulo 3: 3.1 a 3.4
  - Capítulo 4: 4.1 y 4.2

Disponible en

<https://docs.google.com/file/d/0B9TsLZzbZBEYWXpLTF93NDNCTm8/edit?pref=2&pli=1>

# 2.

## Implementación de procesos/hilos en Linux

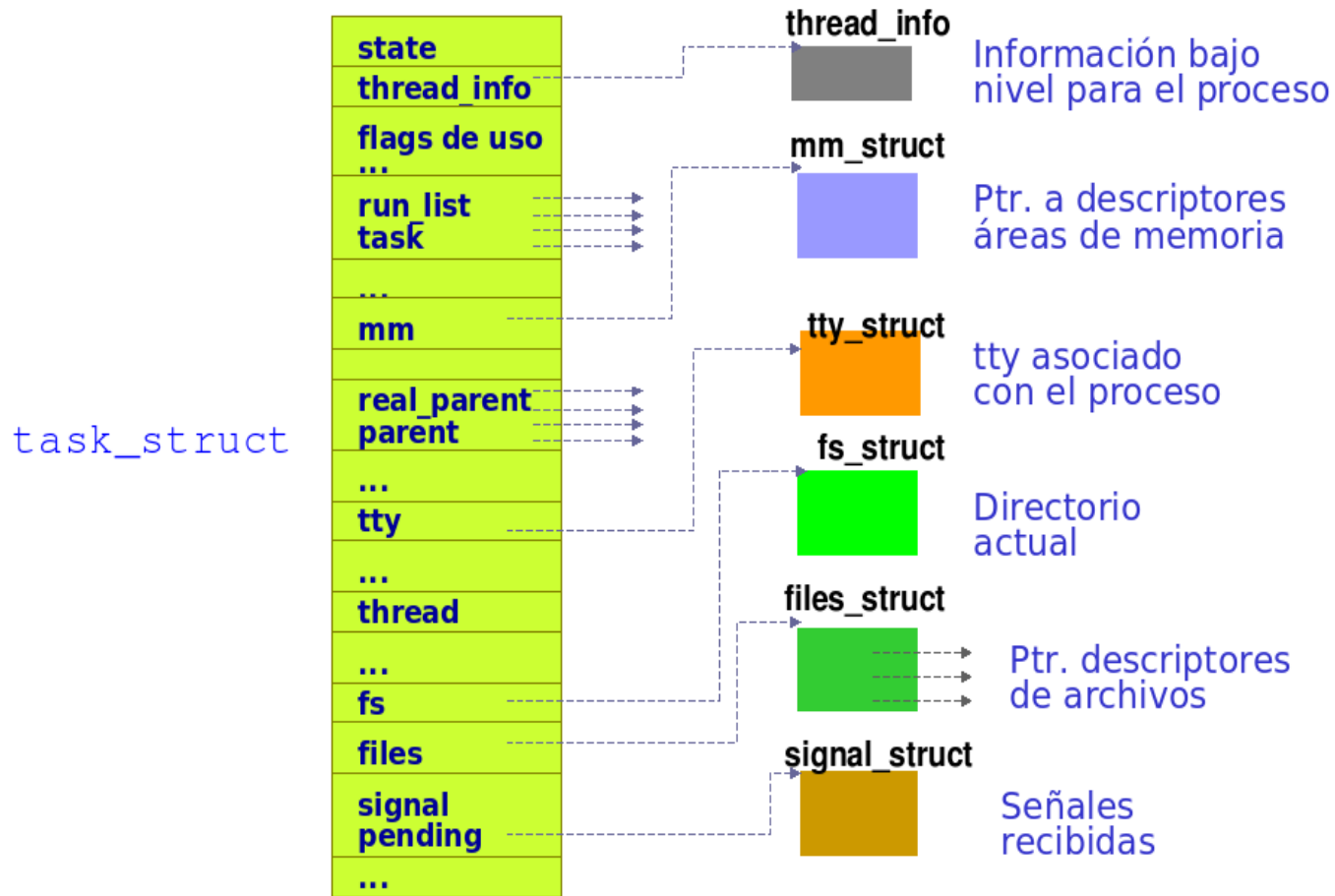
**Cómo se materializa un proceso**

# El Bloque de Control de Proceso

- > En Linux, al PCB se lo denomina **descriptor de proceso** y viene descrito por una estructura `task_struct`.
- > Definida en *include/linux/sched.h*  
(<http://lxr.linux.no/#linux+v4.7/include/linux/sched.h>)

Nota: las referencias al código fuente de Linux se realizan para la versión 4.7 del kernel (disponible en [lxr.linux.no](http://lxr.linux.no))

# Descriptor de proceso





# Comentarios

> Las sub-estructuras `mm_struct`, `files_struct`, `fs_struct`, `tty_struct`, `signal_struct` se desgajan de la principal por varios motivos:

- No se asignan cuando no es necesario. Por ejemplo, un “demonio” no tiene asignado terminal por lo que `tty_struct` → `NULL`)
- Permiten su compartición cuando sea necesario (volveremos sobre ellos al hablar de hilos y `clone()`).

# Estructura thread\_info

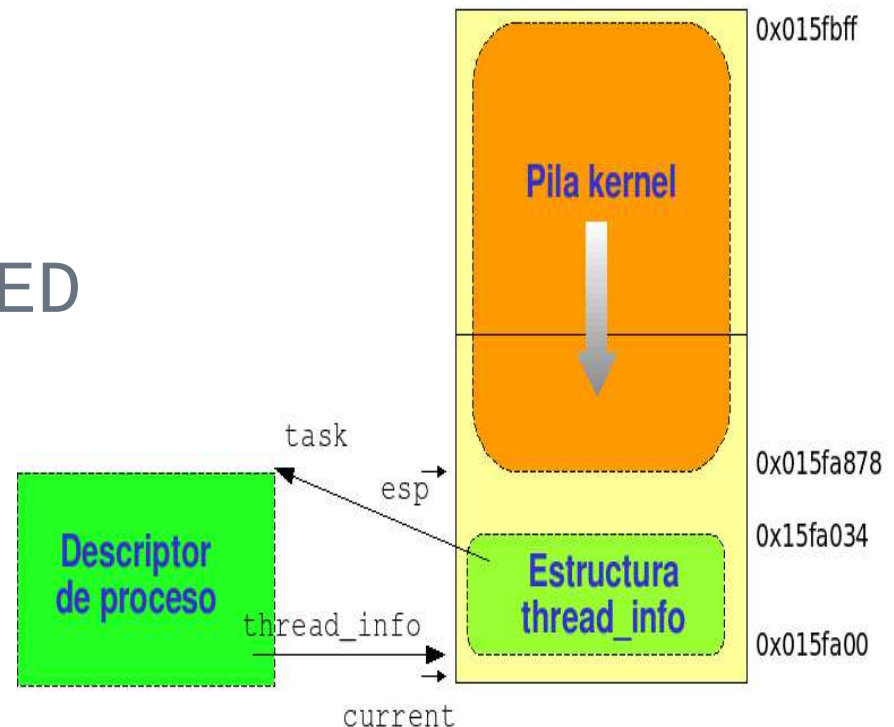
> Contiene información de bajo nivel sobre el proceso y permite acceder a la *pila kernel* del mismo.

> Algunos campos:

- TIF\_SIGPENDING
- TIF\_NEED\_RESCHED
- cpu
- preempt\_count

> Macros:

- current()



# Estados de los procesos

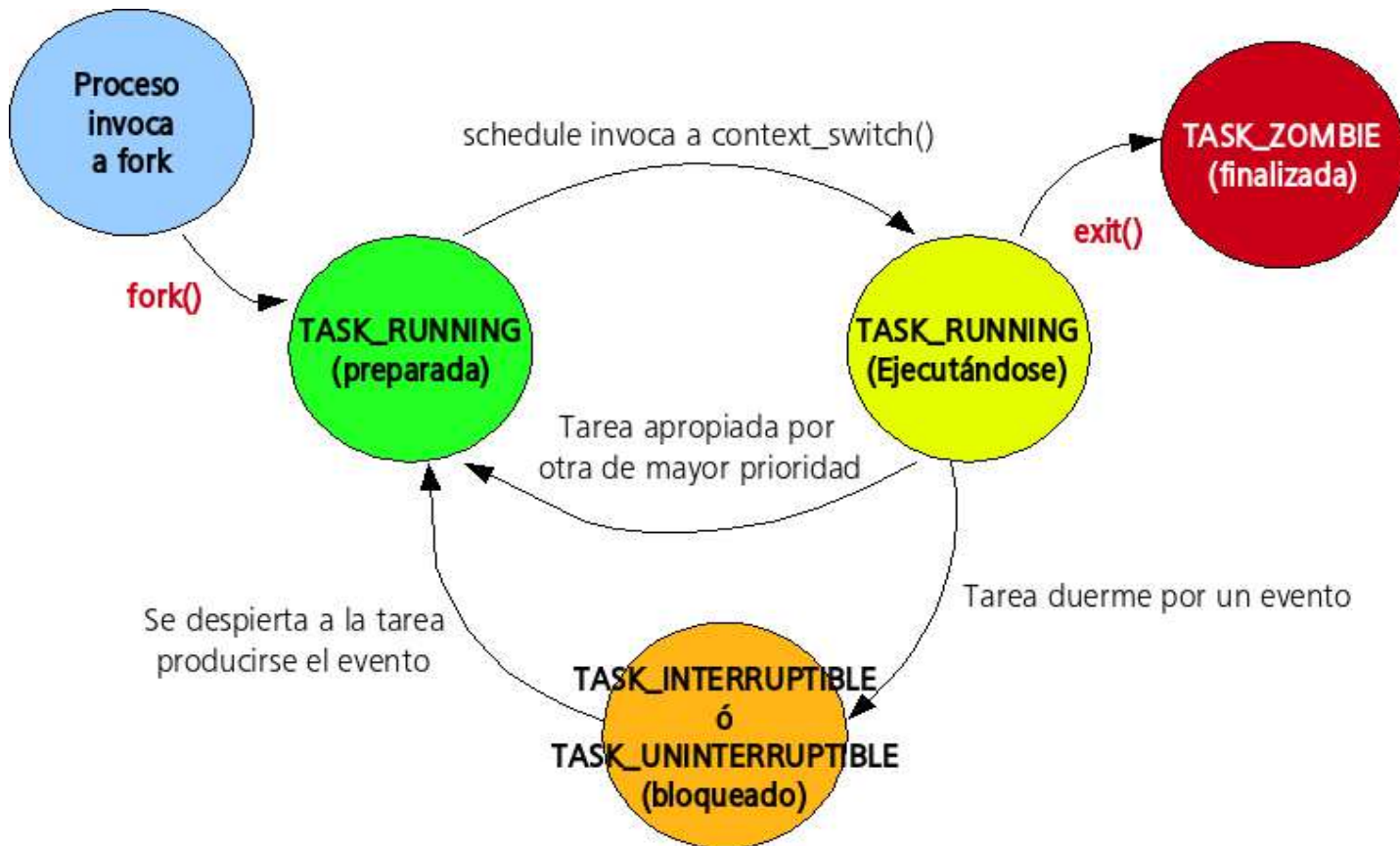
> El campo `state` almacena el estado:

- ◆ `TASK_RUNNING`: El proceso es ejecutable o esta en ejecución.
- ◆ `TASK_INTERRUPTIBLE`: el proceso esta bloqueado (dormido) de forma que puede ser interrumpido por una señal.
- ◆ `TASK_UNINTERRUPTIBLE`: proceso bloqueado no despertable por una señal.
- ◆ `TASK_TRACED`: proceso que esta siendo “traceado” por otro.
- ◆ `TASK_STOPPED`: la ejecución del proceso se ha detenido por algunas de las señales de control de trabajos.

> El campo `exit_state` almacena la condición en que ha finalizado:

- `EXIT_DEAD`: va a ser eliminado, su padre ha invocado `wait()`:
- `EXIT_ZOMBIE`: el padre aún no ha realizado `wait()`.

# Diagrama de estados de procesos/hilos



# Transiciones entre estados

- > **clone( )**: llamada al sistema para crear un proceso/hilo.
- > **exit( )**: llamada para finalizar un proceso.
- > **sleep( )**: bloquea/duerme a un proceso en espera de un determinado evento.
- > **wakeup( )**: desbloquea/desperta a un proceso cuando se ha producido el evento por el que espera.
- > **schedule( )**: planificador – decide que proceso tiene el control de la CPU

# Colas asociadas a estado

- > Existe una lista de procesos doblemente enlazada con todos los procesos del sistema y a la cabeza esta el *swapper* (PID=0, `task_struct_init_task`).
- > Los estados `TASK_STOPPED`, `EXIT_ZOMBIE`, ó `EXIT_DEAD` no están agrupados en colas.
- > Los procesos `TASK_RUNNING` están en diferentes *colas de procesos ejecutables*: una cola por procesador (ver §Planificación).

# Jerarquía de procesos

- > Todos los procesos forman parte de una jerarquía, con el proceso `systemd / init` (PID=1) a la cabeza.
  - Todo proceso tiene exactamente un padre
  - Procesos *hermanos* (*siblings*) = procesos con el mismo padre.
- > La relación entre procesos se almacena en el PCB:
  - `parent`: puntero al padre
  - `children`: lista de hijos.
- > Un proceso localiza a su padre con:  
`my_parent=current->parent`
- > Padre recorrer la lista de sus hijos:  
`list_for_children(list, &current>children)`

# Manipulación de procesos

- > Algunas llamadas para la manipulación de procesos:
- `clone( )`: crea un proceso o hilo desde otro con las características que se especifican en los argumentos.
- `exec( )`: ejecuta un programa dentro de un proceso existente a partir del ejecutable que se pasa como argumento. Al invocar a `exec()` el SO destruye el espacio de direcciones a nivel de usuario del proceso invocador, solo se queda el descriptor de proceso, y construye un nuevo ED de usuario a partir de la información del formato ELF del programa invocado.



# clone()

> El prototipo de la funcion en glibc es:

```
#define <sched.h>
```

```
int clone(int (*func()) (void *), void *child_stack,  
          int flags, void *func_arg, ... /*pid_t *ptid,  
          struct user_desc *tls, pid_t ctid */);
```

> El modelo de hilos de Linux es 1:1, es decir, cada hilo de usuario esta soportado por un hilo kernel.

## clone() (cont.)

- > Significado de algunos de los indicadores de creación:
  - CLONE\_FILES – hilo padre e hijo comparten los mismos archivos abiertos
  - CLONE\_FS – padre e hijo comparten la información del sistema de archivos
  - CLONE\_VM – padre e hijo comparten el espacio de direcciones de usuario
  - CLONE\_SIGHAND – comparten los manejadores de señales y señales bloqueadas
  - CLONE\_THREAD – ambos procesos/hilos pertenecen al mismo grupo (mismo GID).

# Ejemplo de clone

```
#include <stdio.h>  #include <unistd.h> #include  
<sys/types.h>  
#include <linux/unistd.h> #include <sys/syscall.h>  
#include <errno.h>  
#include <linux/sched.h> #include <malloc.h>  
  
int variable=3;  
  
int thread(void *p) {  
    int tid;  
    variable++;  
    tid = syscall(SYS_gettid);  
    printf("\nPID - TID del hijo: %d - %d, var hijo:  
          %d\n",getpid(),tid, var);  
    sleep(5);  
}
```

## Ejemplo (cont.)

```
int main() {
    void **stack;
    int i, tid;
    stack = (void **)malloc(15000);
    if (!stack)
        return -1;
    i = clone(thread, (char*) stack + 15000, CLONE_VM|
              CLONE_FILES|CLONE_FS|CLONE_THREAD|CLONE_SIGHAND,
              NULL); /* (1) */
    sleep(2);
    if (i == -1)
        perror("clone");
    tid = syscall(SYS_gettid);
    printf("\nPID - TID del padre: %d - %d, var padre
%d\n\n", getpid(), tid, var);
    return 0;
}
```

# Ejecución

> Si compilamos y ejecutamos el ejemplo tal como esta:

```
> ./clon
```

```
PID - TID del hijo: 7917 - 7918, var hijo: 4
```

```
PID - TID del padre: 7917 - 7917, var padre 4
```

> Si en (1) Si solo dejamos:

```
i = clone(thread, (char*) stack + 15000, NULL, NULL)
```

```
> ./clon2
```

```
PID - TID del hijo: 7971 - 7971, var hijo: 4
```

```
PID - TID del padre: 7970 - 7970, var Padre 3
```

> ¿Qué ha pasando?

# fork()

> `clone()` es una llamada endémica de Linux, en sistemas UNIX, la llamada para crear procesos (no hilos) es `fork()`.

> En linux, `fork()` se implementa como:

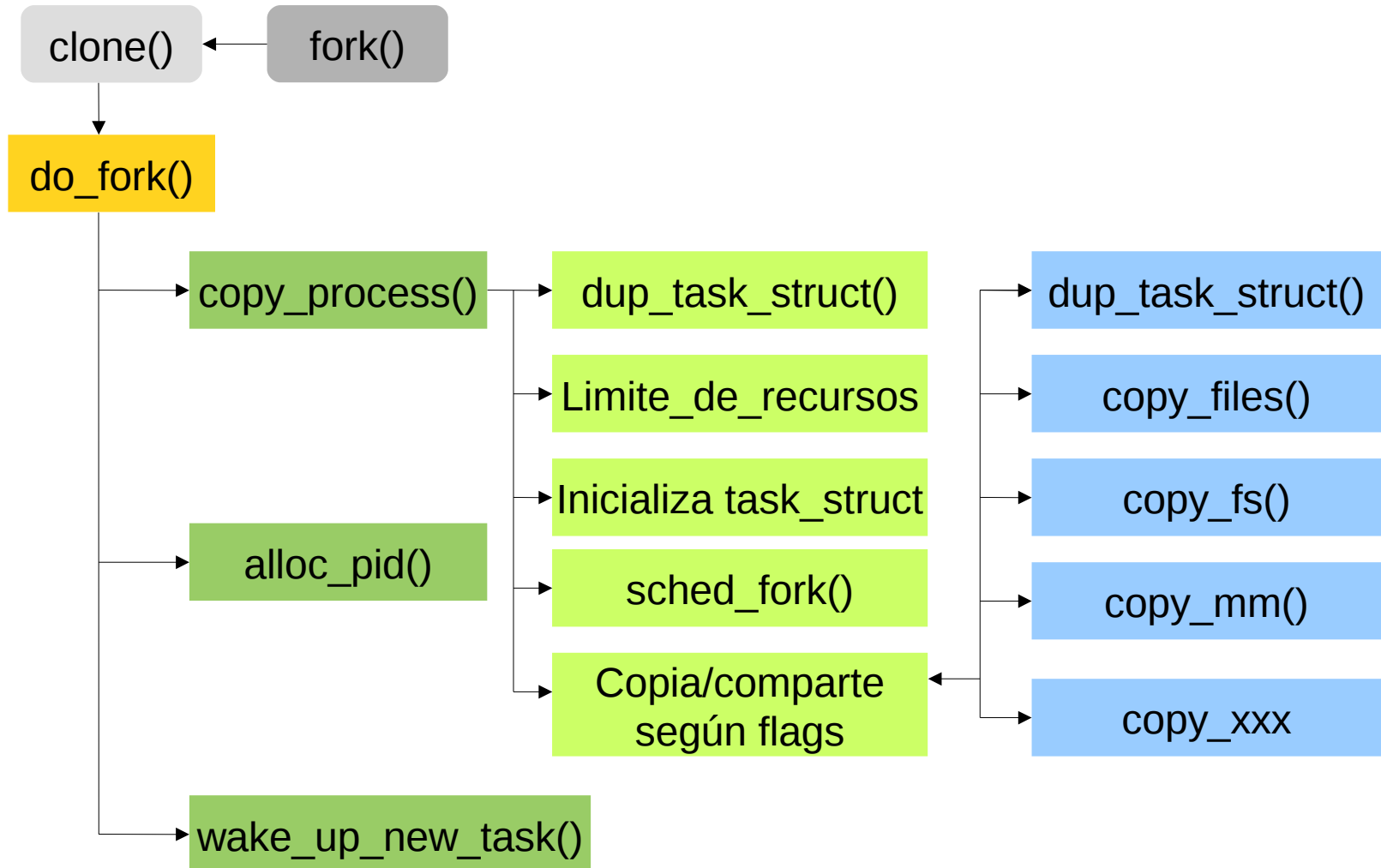
`clone(SIGCHLD, 0)`

Nota: El prototipo de la llamada al sistema `clone()` es

`clone(flags, stack, ptid, ctid, regs)`

# Clone: implementación

> La estructura del código de `clone()`:



# clone(): explicación

- > `do_fork()` esta implementada en *kernel/fork.c*.
- > El trabajo que realiza es:
  - `dup_task_struct`: copia el descriptor del proceso actual (`task_struct`, pila y `thread_info`).
  - `alloc_pid`: le asigna un nuevo PID
  - Al inicializar la `task_struct`, diferenciamos los hilos padre e hijo y ponemos a este último en estado no-interrumpible.
  - `sched_fork`: marcamos el hilo como `TASK_RUNNING` y se inicializa información sobre planificación
  - Compiamos/compartimos componentes según los indicadores de la llamada.
  - Asignamos ID, relaciones de parentesco, etc.



# Clone(): consideraciones

- > Clone debe hacer algunas comprobaciones:
  - Algunos indicadores no tienen sentido juntos, por ejemplo, `CLONE_NEWNS` y `CLONE_FS`.
  - Otros debe aparecer a la vez: `CLONE_THREAD` con `CLONE_SIGHAND`, o `CLONE_SIGHAND` con `CLONE_VM`.
- > Cuando aparece el flag `CLONE_xxx`, esto indica que la estructura `xxx_struct` debe compartirse (no copiarse).

Nota: El kernel asigna a cada estructura compartida un **contador de referencias** que lleva la cuenta que cuantos hilos la comparten. Cada vez que borramos una de las referencias a la estructura, decrementamos dicho contador; si este llega a cero, la estructura se libera. Esto elimina la necesidad de un recolector de basura.

# Crear un hilo de usuario

> Creamos un hilo con los indicadores:

```
clone( . , CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND, . )
```

> Con estos indicadores, el hilo creado comparte los recursos del llamador: memoria, archivos y manejadores de señales.

# Hilos kernel

- > Son hilos que no tienen espacio de direcciones de usuario. Por tanto, su descriptor tiene `task_struct->mm=NULL`.
- > Realizan labores de sistema sustituyendo a los antiguos *demonios* de Unix.
- > Solo se pueden crear desde otro hilo kernel con la función `kthread_create()`.

> Ejemplos:

> `ps -ef`

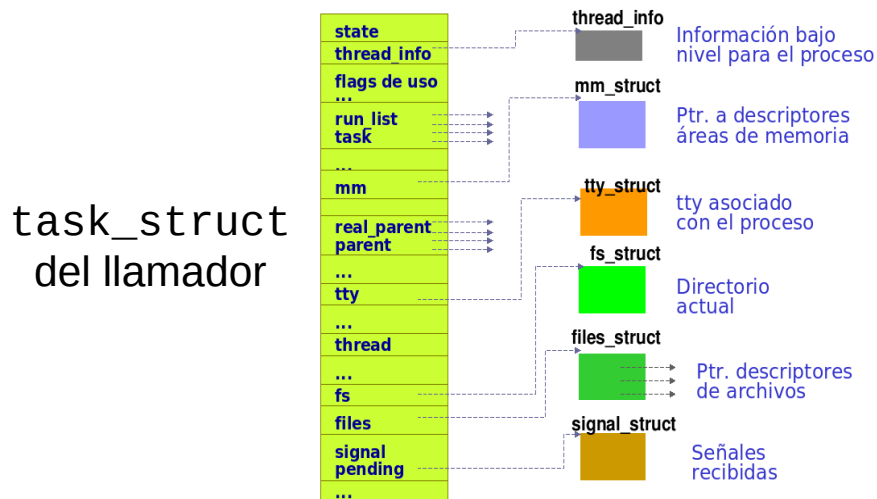
UID	PID	PPID	C	STIME	TTY	TIME	CMD
Root	1	0	0	08:54	?	00:00:01	.../systemd..
root	2	0	0	08:54	?	00:00:00	[kthreadd]
root	3	2	0	08:54	?	00:00:00	[ksoftirqd/0]
root	5	2	0	08:54	?	00:00:00	[kworker/0:0H]

# Actividad en grupo

> En el ejemplo anterior, hemos utilizado la función `clone()` con los argumentos siguientes:

`CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_THREAD | CLONE_SIGHAND`

Hacer un dibujo que represente los principales elementos de las `task_struct` de ambos procesos relacionados con los indicadores mencionados.



¿ task\_struct del nuevo hilo ?

# Terminar un proceso

> Se produce cuando un proceso:

- ♦ Invoca a voluntariamente a
  - ♦ `exit()` o `return()` -en el `main()`- finaliza el proceso primero a nivel de biblioteca.
  - ♦ `_exit()` - llamada al SO, si se invoca directamente no da la oportunidad de finalizar el proceso a nivel de biblioteca.
- ♦ Involuntariamente: recibe una señal y se aplica la acción por defecto, que es terminar.

# exit()

- > Finaliza el proceso `do_exit()` en *kernel/exit.c*:
  - ◆ Activa `PF_EXITING`
  - ◆ Decrementa los contadores de uso de `mm_struct`, `fs_struct`, `files_struct`. Si estos contadores alcanzan el valor 0, libera los recursos.
  - ◆ Ajusta el `exit_code` del descriptor, que será devuelto al padre, con el valor pasado al invocar a `exit()`.
  - ◆ Envía al padre la señal de finalización; si tiene algún hijo le busca un padre en el grupo o el init, y pone el estado a `TASK_ZOMBIE`.
  - ◆ Invoca a `schedule()` para ejecutar otro proceso.
- > Ya solo queda: la pila kernel, `thread_info` y `task_struct`, de cara a que el padre pueda recuperar el código de finalización. ¿Cuándo se libera el resto?

# wait()

- > `wait()`: llamada que bloquea a un proceso padre hasta que uno de sus hijo finaliza; cuando esto ocurre, devuelve al llamador el PID del hijo finalizado y el estado de finalización (código de finalización, coredump y señal).
- > Esta función invoca a `release_task()` que:
  - ♦ Elimina el descriptor de la lista de tareas.
  - ♦ Si es la última tarea de su grupo, y el líder esta zombi, notifica al padre del líder zombi.
  - ♦ Libera la memoria de la pila kernel, `thread_info` y `task_struct`.

3.

# Planificación

**Cómo y a quién asignar las CPUs**



# Contenido del apartado

- > Tipos de planificadores.
- > Tipos de planificación.
- > Criterios, algoritmos y métricas de planificación: Prioridades, RR, colas múltiples.
- > Planificación en Linux:
  - ◆ Planificador CFS (Completely Fair Scheduler)
  - ◆ Planificación de tiempo-real
  - ◆ Planificación en multiprocesadores
- > Ahorro de energía

# Trabajo individual

- > Tipos de planificadores - §9.1 del W. Stallings “Tipos de planificación del procesador”.
- > Criterios, algoritmos y métricas de planificación - §9.2 del W. Stallings “Algoritmos de planificación”. Ver:
  - FIFO
  - Prioridades
  - Round-Robin
  - Colas múltiples (con/sin realimentación)

# Tipos de planificadores

- > Planificador a:
  - **Largo plazo** – procesos por lotes
  - **Corto plazo** o scheduler
  - **Medio plazo** – gestor de memoria

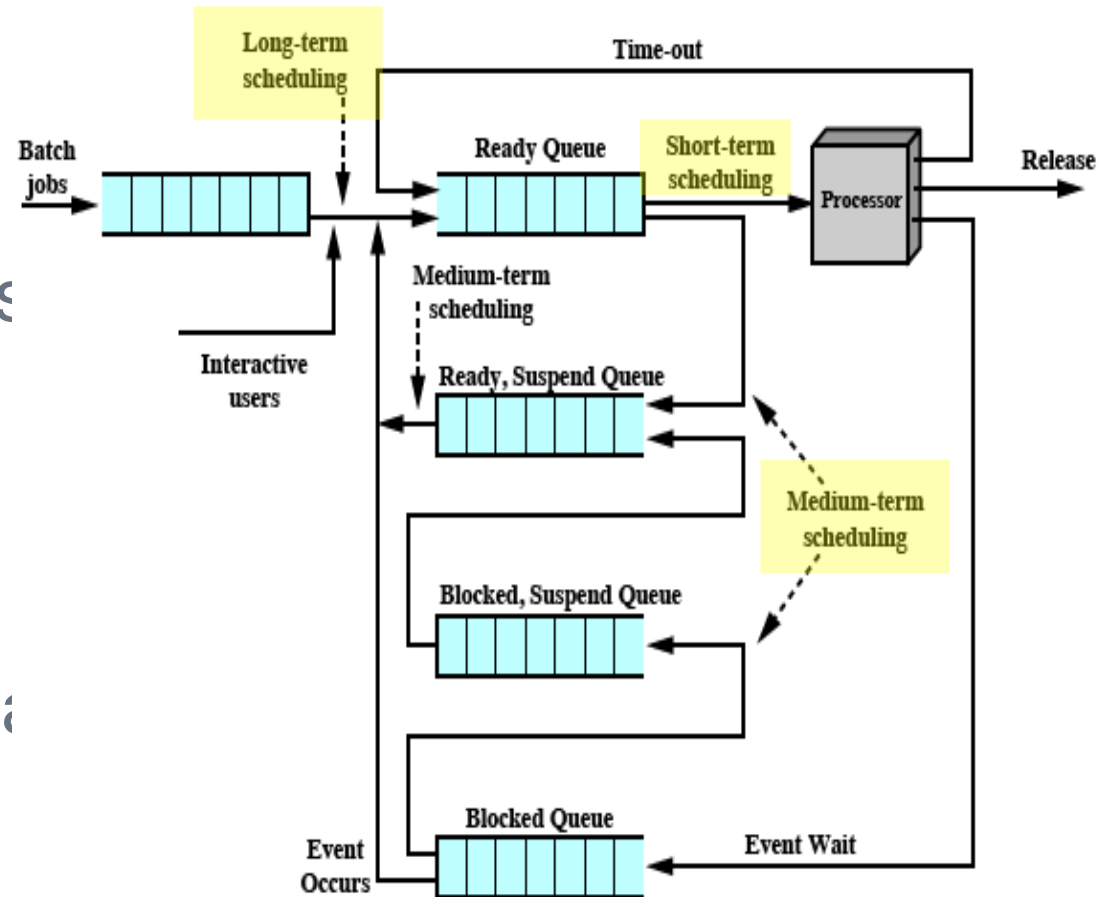


Figura del libro de W. Stallings

# Tipos de planificación

## > Planificación

### **apropiativa** (*preemptive*)

– al proceso actual se le puede retirar la CPU.

## > Planificación no

### **apropiativa** (*non*

*preemptive*) – al proceso actual NO se le puede retirar la CPU.

> La NO apropiación ha sido utilizada por los constructores de SOs como mecanismo de grano grueso de sincronización en modo kernel.

> Los kernel de tiempo-real necesitan ser apropiativos.