Tema 2 – Procesos y Hebras

Generalidades sobre Procesos, Hilos y Planificación

Diseño e implementación de procesos e hilos en Linux

Planificación de CPU en Linux

Sistemas Operativos
Alejandro J. León Salas, 2019
Lenguajes y Sistemas Informáticos

Objetivos

- Conocer la implementación en Linux de procesos y hebras.
- Conocer el funcionamiento básico del planificador de Linux.

Bibliografía

- [Lov2010] R. Love, *Linux Kernel Development (3/e)*, Addison-Wesley Professional, 2010.
- [Mau2008] W. Mauerer, Professional Linux Kernel Architecture, Wiley, 2008.

Contenidos

- Implementación de proceso/hebra en Linux: *task*
- Planificación de CPU en Linux

Implementación del concepto proceso/hebra (*process*/*thread*) en Linux: *task* (`task_struct`)

Nos basamos el kernel 2.6 de Linux.
Podemos descargar los fuentes de www.kernel.org

- El núcleo identifica a los procesos (tareas - tasks) por su **PID**
- En Linux, proceso es la entidad que se crea con la llamada al sistema ***fork*** (excepto el proceso 0) y ***clone***
- Procesos especiales que existen durante la vida del sistema:
  - **Proceso 0**: creado “a mano” cuando arranca el sistema. Crea al proceso 1.
  - **Proceso 1 (Init)**: antecesor de cualquier proceso del sistema

PCB en Linux: `struct task_struct`

- En Linux una tarea (*task*) engloba el concepto de proceso y hilo.

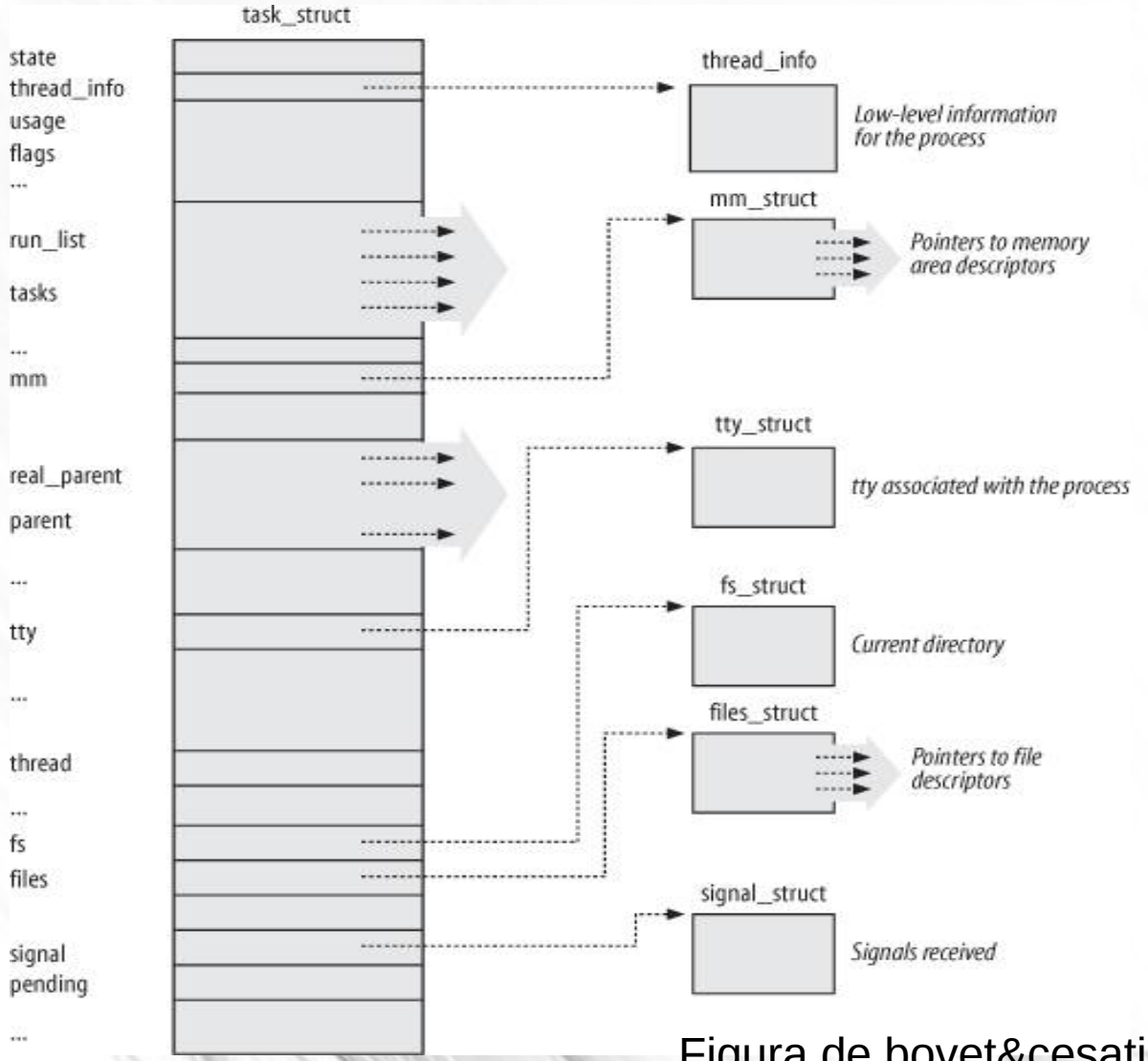


Figura de bovet&cesati

PCB en Linux: `struct task_struct`

- El Kernel almacena la lista de procesos como una lista circular doblemente enlazada: `task list`
- Cada elemento es un descriptor de proceso (PCB) definido en `</include/linux/sched.h>`

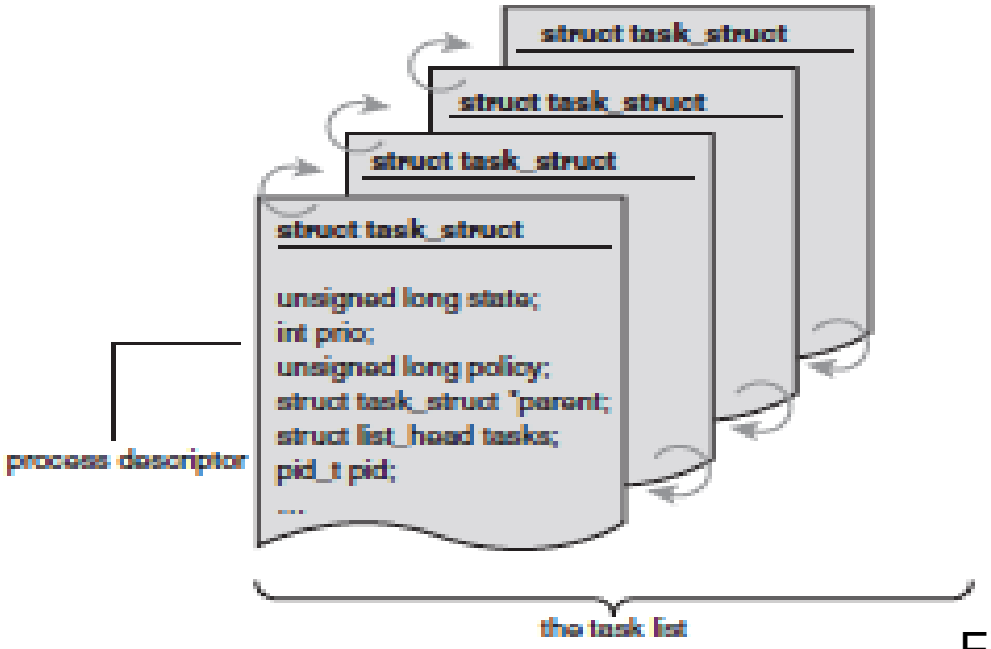


Figura de bovet&cesati

PCB en Linux: `struct task_struct`

```
struct task_struct {  /// del kernel 2.6.24
    volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
    /*...*/
/* Informacion para planificacion */
    int prio, static_prio, normal_prio;
    struct list_head run_list;
    const struct sched_class *sched_class;
    struct sched_entity se;
/*...*/
    unsigned int policy;
    cpumask_t cpus_allowed;
    unsigned int time_slice;
/*...*/
```

PCB en Linux: `struct task_struct`

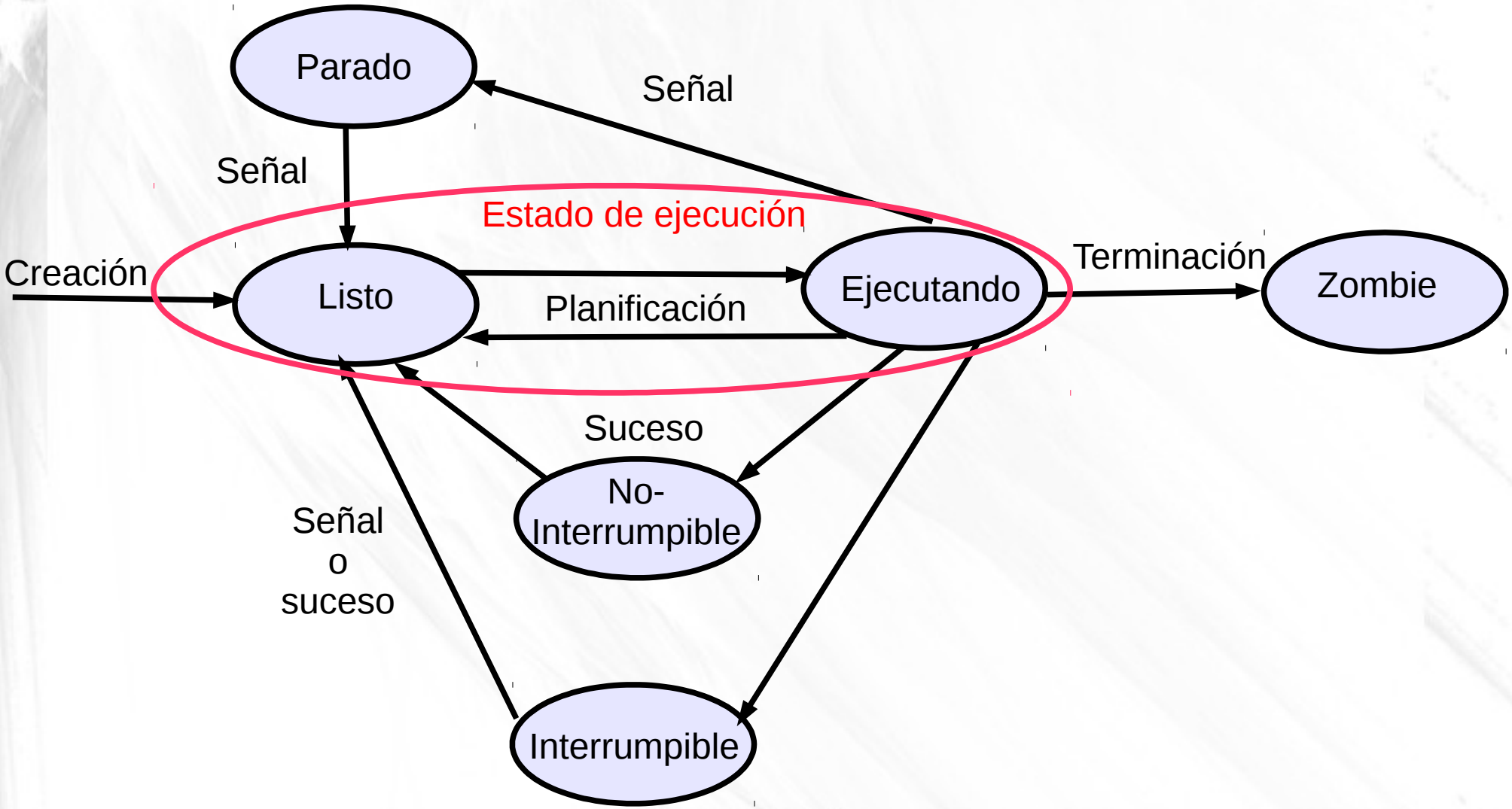
```
//* Memoria asociada a la tarea */
    struct mm_struct *mm, *active_mm;
/*...*/
    pid_t pid;
/* Relaciones entre task_struct */
    struct task_struct *parent; /* parent process */
    struct list_head children; /* list of my children */
    struct list_head sibling; /* linkage in my parent's children list */
/* Informacion para planificacion y señales */
    unsigned int rt_priority;
    sigset_t blocked, real_blocked;
    sigset_t saved_sigmask; /* To be restored with TIF_RESTORE_SIGMASK */
    struct sigpending pending;
/*...*/
}
```

Estados de una tarea (*task*)

- La variable state de task_struct especifica el estado actual de un proceso.

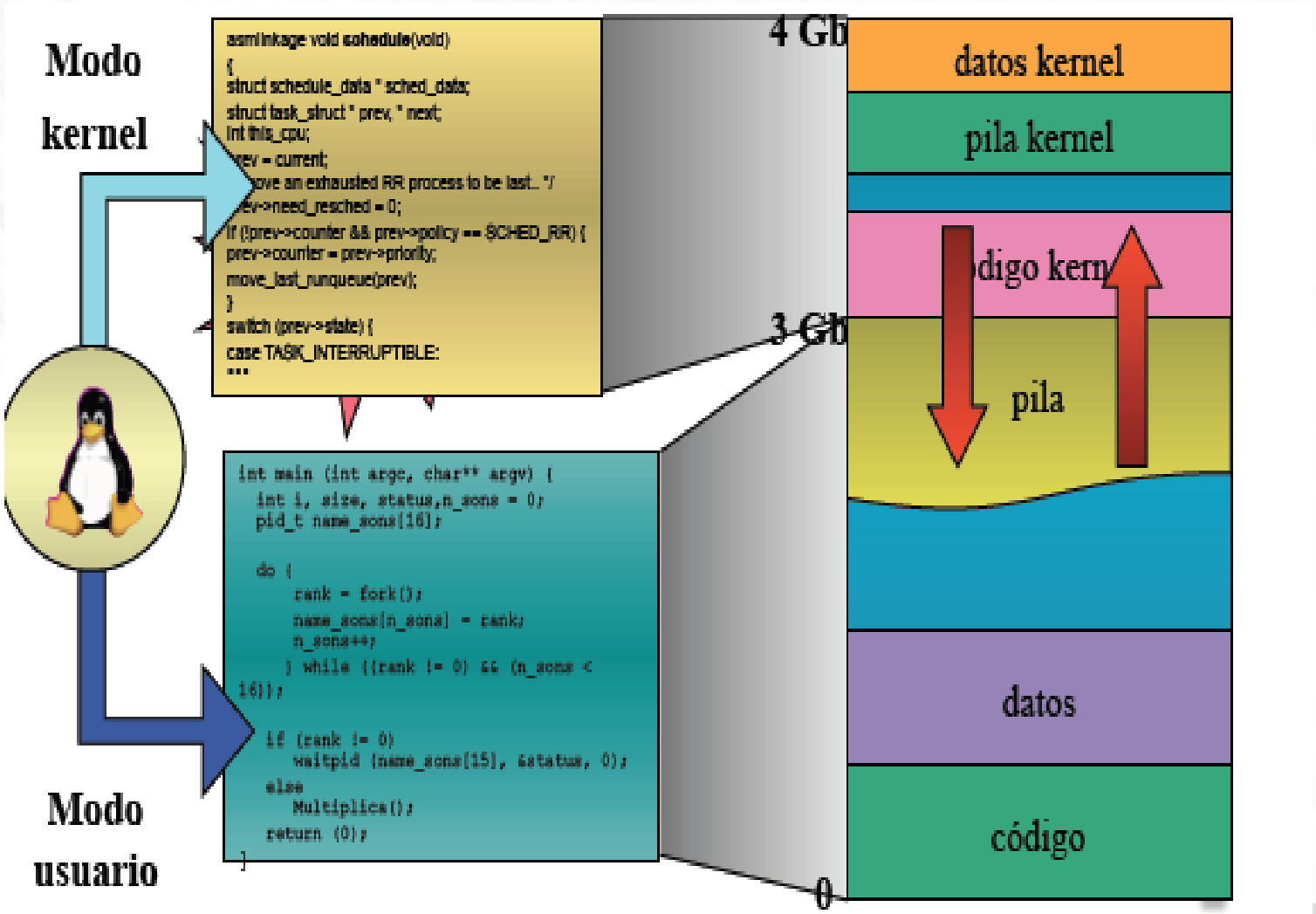
Ejecución **TASK_RUNNING**	Se corresponde con dos: ejecutándose o preparado para ejecutarse (en la cola de procesos preparados)
Interrumpible **TASK_INTERRUPTIBLE**	El proceso está bloqueado y sale de este estado cuando ocurre el suceso por el cual está bloqueado o porque le llegue una señal
No interrumpible **TASK_UNINTERRUPTIBLE**	El proceso está bloqueado y sólo cambiará de estado cuando ocurra el suceso que está esperando (no acepta señales)
Parado **TASK_STOPPED**	El proceso ha sido detenido y sólo puede reanudarse por la acción de otro proceso (ejemplo, proceso parado mientras está siendo depurado)
TASK_TRACED	El proceso está siendo traceado por otro proceso
Zombie **EXIT_ZOMBIE**	El proceso ya no existe pero mantiene la estructura task hasta que el padre haga un wait (**EXIT_DEAD**)

Modelo de estados para *tasks*



Espacio de direcciones en el contexto de un proceso

- Acceso a memoria en modo *user* y modo *kernel* dentro del contexto de un proceso.



Árbol de procesos

- Si el proceso P0 hace una llamada al sistema fork(), genera el proceso P1, se dice que P0 es el proceso padre y P1 es el hijo.
- Si el proceso P0 hace varias llamadas al sistema fork() genera varios procesos hijo y la relación entre ellos es de hermanos (*sibling*)
- Todos los procesos son descendientes del proceso `init` (cuyo PID es 1)

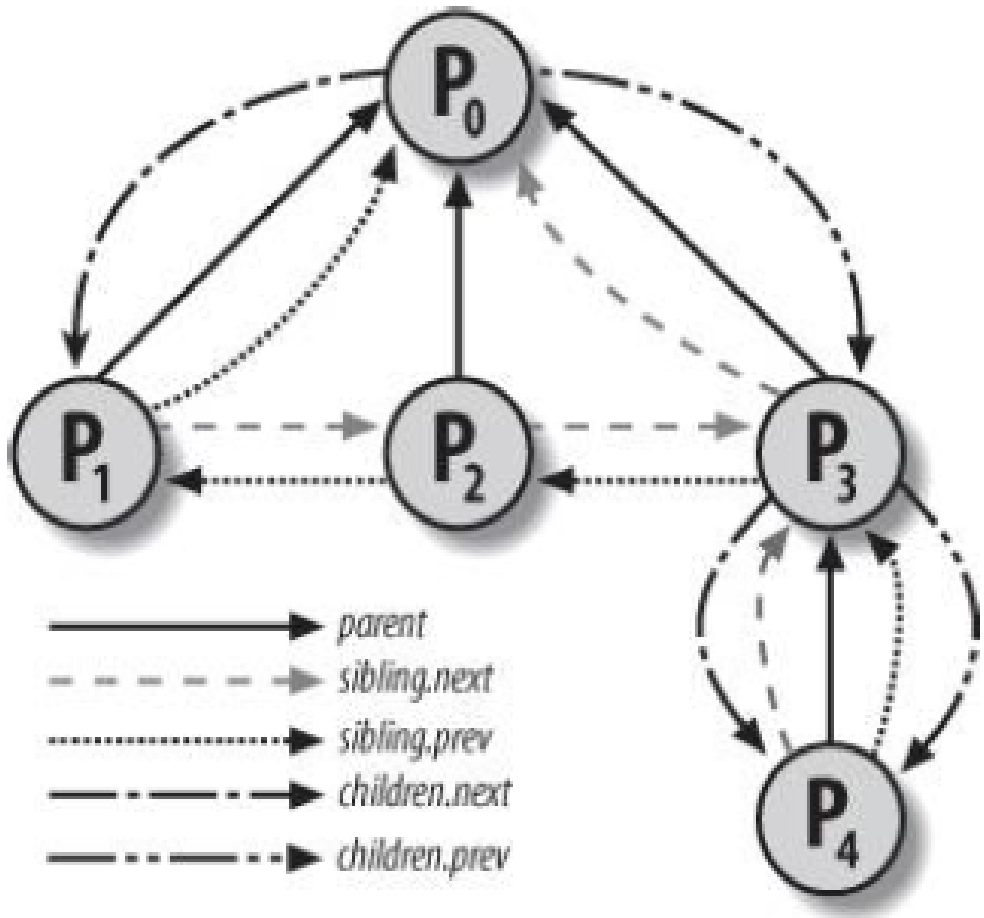


Figura de Bovet & Cesati

Árbol de procesos

- Cada task_struct tiene un puntero...
  - a la *task_struct* de su padre:

    struct task_struct *parent
  - a una lista de hijos (children):

    struct list_head children;
  - y a una lista de sus hermanos (sibling):

    struct list_head sibling

- El kernel dispone de procedimientos eficaces para las acciones usuales de manipulación de la lista. (Para una explicación detallada de list_head ver [Mau08 1.3.13])

Implementación de hilos (*threads*)

- Desde el punto de vista del kernel no hay distinción entre hebra y proceso: *task* = *process*/*thread*.
- Linux implementa el concepto de hebra como un proceso sin más, que simplemente comparte recursos con otros procesos.
- Cada hebra tiene su propia **task_struct**.
- La llamada al sistema clone() crea un nuevo proceso o hebra dependiendo del parámetro flags.

```
#define _GNU_SOURCE

#include <sched.h>

int clone ( int (*fn) (void *), void *child_stack, int flags, void *arg);
```

clone() *system call flags*

Flag	Meaning
CLONE_FILES	Parent and child share open files.
CLONE_FS	Parent and child share filesystem information.
CLONE_IDLETASK	Set PID to zero (used only by the idle tasks).
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PARENT	Child is to have same parent as its parent.
CLONE_PTRACE	Continue tracing child.
CLONE_SETTID	Write the TID back to user-space.
CLONE_SETTLS	Create a new TLS for the child.
CLONE_SIGHAND	Parent and child share signal handlers and blocked signals.
CLONE_SYSVSEM	Parent and child share System V SEM_UNDO semantics.
CLONE_THREAD	Parent and child are in the same thread group.
CLONE_VFORK	vfork() was used and the parent will sleep until the child wakes it.
CLONE_UNTRACED	Do not let the tracing process force CLONE_PTRACE on the child.
CLONE_STOP	Start process in the TASK_STOPPED state.
CLONE_SETTLS	Create a new TLS (thread-local storage) for the child.
CLONE_CHILD_CLEARTID	Clear the TID in the child.
CLONE_CHILD_SETTID	Set the TID in the child.
CLONE_PARENT_SETTID	Set the TID in the parent.
CLONE_VM	Parent and child share address space.

Figura de bovet&cesati

Hebras *kernel* (*kernel threads*)

- A veces es útil que el kernel realice operaciones en segundo plano, para lo cual se crean hebras kernel.
- Las hebras kernel no tienen un espacio de direcciones (su puntero mm es NULL)
- Se ejecutan únicamente en el espacio del kernel.
- Son planificadas y pueden ser expropiadas.
- Solo se pueden crear por una hebra kernel mediante:

```
#include <include/linux/kthread.h>

struct task_struct *kthread_create_on_node(int (*threadfn)(void *data),

void *data, int node, const char namefmt[], ...);
```

- Terminan cuando realizan una operación `do_exit` o cuando otra parte del kernel provoca su finalización.

Creación de tareas (*tasks*): task_struct

fork() → clone() → do_fork() → copy_process()

- Actuación de **copy_process()**:

  1. Crea la estructura thread_info (**pila kernel**) y la task_struct para la nueva tarea con los valores de la tarea actual.

  2. Asigna valores iniciales a los campos de la task_struct de la tarea hija que deban tener valores distintos a los de la tarea padre.

  3. Se establece el campo estado de la tarea hija TASK_UNINTERRUPTIBLE mientras se realizan las restantes acciones.

Creación de tareas (*tasks*): task_struct

4. Se establecen valores adecuados para los flags de la task_struct de la tarea hija:

   flag PF_SUPERPRIV=0 (la tarea no usa privilegio de superusuario)

   flag PF_FORKNOEXEC=1 (el proceso ha hecho fork() pero no exec())

5. Se llama a alloc_pid() para asignar un PID a la nueva tarea.

6. Según cuáles sean los flags pasados a clone(), duplica o comparte recursos como archivos abiertos, información de sistemas de archivos, manejadores de señales, espacio de direccionamiento del proceso,...

7. Se establece el estado de la tarea hija a TASK_RUNNING.

8. Finalmente copy_process() termina devolviendo un puntero a la task_struct de la tarea hija.

Terminación de tareas (*tasks*): task_struct

- Cuando un proceso termina, el kernel libera todos sus recursos y notifica al padre su terminación.
- Normalmente un proceso termina cuando:

  1. Realiza la llamada al sistema exit().

  De forma **explícita:** el programador incluyó esa llamada en el código del programa, o

  De forma **implícita:** el compilador incluye automáticamente una llamada a exit() cuando main() termina.

  2. **Recibe una señal** que tiene la disposición por defecto de terminar al proceso (Term).

- El trabajo de liberación lo hace la función do_exit() definida en <linux/kernel/exit.c>

Actuación de do_exit()

1. Activa el flag PF_EXITING de task_struct

2. Para cada recurso que esté utilizando el proceso, se decrementa el contador correspondiente que indica el nº de procesos que lo están utilizando. Si contador==0 → se realiza la operación de destrucción oportuna sobre el recurso, por ejemplo si fuera una zona de memoria, se liberaría el descriptor de memoria (mm_struct) correspondiente.

3. El valor que se pasa como argumento a exit() se almacena en el campo exit_code de task_struct (Esta es la información de terminación para que el padre pueda hacer un wait() o waitpid() y recogerla)

4. Se manda una señal al padre indicando la finalización de su hijo.

Actuación de do_exit()

5. Si el proceso aún tiene hijos, se establece como padre de dichos hijos al proceso init (PID=1).

Nota: Dependiendo de las características del grupo de procesos al que pertenezca el proceso, podría ponerse como padre a otro miembro de ese grupo de procesos.

6. Se establece el campo exit_state de task_struct a EXIT_ZOMBIE.

7. Se llama a schedule() para que el planificador elija un nuevo proceso a ejecutar.

Nota: Puesto que este es el último código que ejecuta un proceso, do_exit() nunca retorna.

Contenidos

- Implementación de proceso/hebra en Linux: *task*
- Planificación de CPU en Linux

Planificación de CPU en Linux

- Planificador modular: clases de planificación
- Planificación de tiempo real
- Planificación neutra o limpia (**CFS**: *Completely Fair Scheduling*)
- Planificación de la tarea "*idle*" (no hay trabajo que realizar)

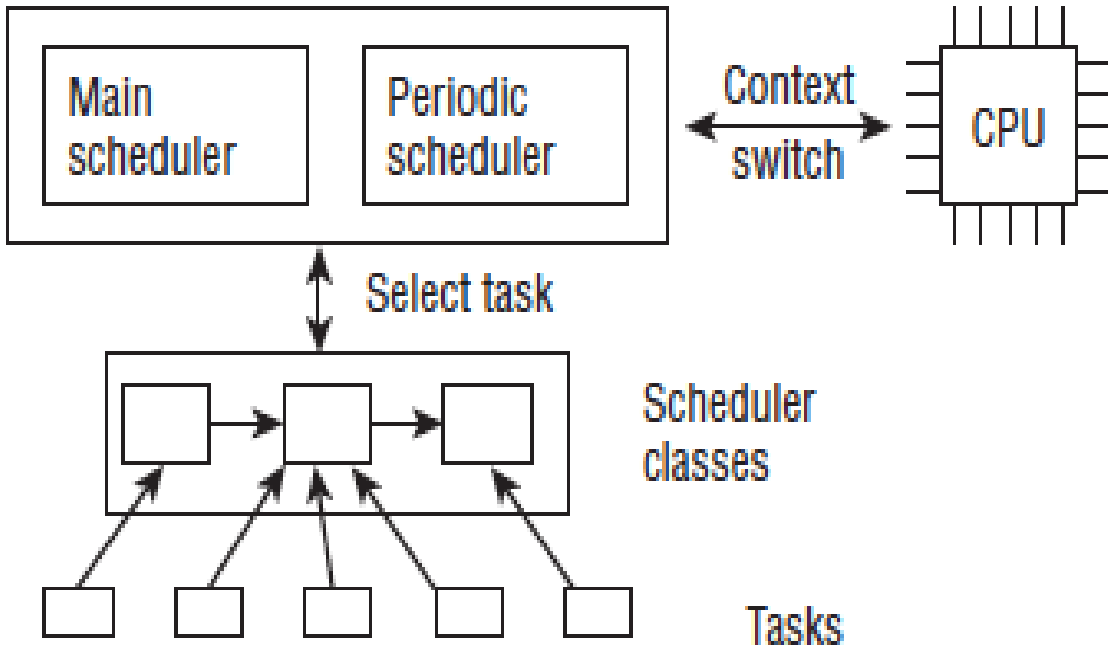


Figura de bovet&cesati

Planificación de CPU en Linux

- Cada clase de planificación tiene una prioridad.
- Se usa un algoritmo de planificación entre las clases de planificación por prioridades apropiativo.
- Cada clase de planificación usa una o varias políticas para gestionar sus procesos.
- La planificación no opera únicamente sobre el concepto de proceso, sino que maneja conceptos más amplios en el sentido de manejar grupos de procesos: Entidad de planificación.
- Una entidad de planificación se representa mediante una instancia de la estructura sched_entity.

Políticas de planificación

unsigned int policy; // política que se aplica al proceso

- Políticas manejadas por el planificador CFS – fair_sched_class:

SCHED_NORMAL: se aplica a los procesos normales de tiempo compartido

SCHED_BATCH: tareas menos importantes, menor prioridad. Son procesos batch con gran proporción de uso de CPU para cálculos.

SCHED_IDLE: tareas de este tipo tienen una prioridad mínima para ser elegidas para asignación de CPU

- Políticas manejadas por el planificador de tiempo real – rt_sched_class:

SCHED_RR: uso de una política Round-Robin

SCHED_FIFO: uso de una política FCFS

Prioridades

- Siempre se cumple que el proceso que está en ejecución es el más prioritario.
- Intervalo de valores de prioridad para static_prio:

  [0, 99] Prioridades para procesos de **tiempo real**.

  [100, 139] Prioridades para los procesos normales o regulares.

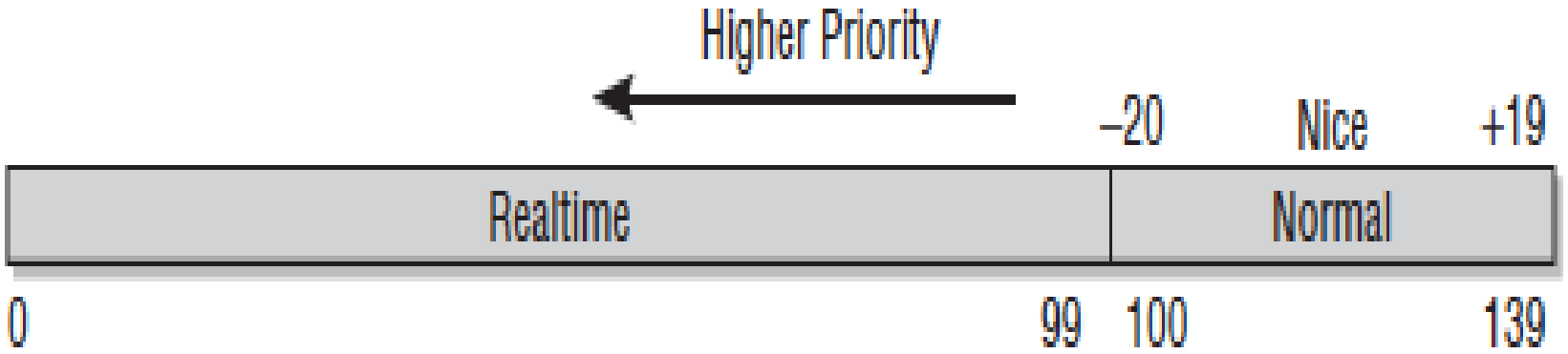


Figura de bovet&cesati

El planificador periódico

- Se implementa en scheduler_tick(), función llamada automáticamente por el kernel con frecuencia (HZ) constante cuyos valores están normalmente en el rango 1000 y 100Hz)
- Tareas principales:
  - Actualizar estadísticas del kernel.
  - Activar el método de planificación periódico de la clase de planificación a que corresponde el proceso actual (task_tick()). Cada clase de planificación tiene implementada su propia función task_tick() (contabiliza el tº de CPU cosumido).
- Si hay que replanificar, el planificador de la clase concreta activará el flag TIF_NEED_RESCHED asociado al proceso en su thread_info, y provocará que se llame al planificador principal.

El planificador principal: schedule()

- Se implementa en la función schedule(), invocada en diversos puntos del kernel para tomar decisiones sobre asignación de la CPU.
- La función schedule() es invocada de forma explícita cuando un proceso se bloquea o termina.
- El kernel chequea el flag TIF_NEED_RESCHED del proceso actual al volver al espacio de usuario desde modo kernel.
- Ya sea justo antes de volver de una llamada al sistema, de una rutina de servicio de interrupción (RSI) o de una rutina de servidio de excepción (RSE); si está activo este flag se invoca a schedule().

Actuación de schedule()

- Determina la actual runqueue y establece el puntero prev a la task_struct del proceso actual.
- Actualiza estadísticas y limpia el flag TIF_NEED_RESCHED.
- Si el proceso actual va a un estado TASK_INTERRUPTIBLE y ha recibido la señal que esperaba, se establece su estado a TASK_RUNNING.
- Se llama a pick_next_task de la clase de planificación a la que pertenezca el proceso actual para que se seleccione el siguiente proceso a ejecutar; y se establece next con el puntero a la task_struct de dicho proceso seleccionado.
- Si hay cambio en la asignación de CPU, se realiza el cambio de contexto llamando a context_switch() ¡Nuestro dispatcher()!

La clase de planificación CFS

- **Idea general**: repartir el tiempo de CPU de forma imparcial, garantizando que todos los procesos se ejecutarán y, dependiendo del número de procesos en cola, asignarles más o menos tiempo de uso de CPU.
- Mantiene datos sobre los tiempos consumidos por los procesos.
- El kernel calcula un **peso** para cada proceso. Cuanto mayor sea el valor de la prioridad estática de un proceso, menor será el peso que tenga.
- vruntime (*virtual runtime*) de una entidad es el **tiempo virtual** que un proceso ha consumido y se calcula a partir del tiempo real que el proceso ha hecho uso de la CPU, su prioridad estática y su peso.

La clase de planificación CFS

- El valor vruntime del proceso actual se actualiza:
  - Periódicamente (el planificador periódico ajusta los valores de tiempo de CPU consumido)
  - Cuando llega un nuevo proceso ejecutable.
  - Cuando el proceso actual se bloquea.
- Cuando se tiene que decidir qué proceso ejecutar a continuación, se elige el que tenga un valor menor de vruntime.
- Para implementar esto CFS utiliza un red black tree (rbtree), que es una estructura de datos árbol binario que almacena nodos identificados por una clave, vruntime, y que permite una búsqueda eficiente por valor de clave.

La clase de planificación CFS

- Cuando un proceso va a entrar en estado bloqueado:

1. Se añade a una cola asociada con el motivo del bloqueo.
2. Se establece el estado del proceso a TASK_INTERRUPTIBLE o a TASK_NONINTERRUPTIBLE según esté clasificado el motivo del bloqueo.
3. Se quita del rbtree de procesos ejecutables la referencia a la task_struct del proceso.
4. Se llama a schedule() para que se elija un nuevo proceso a ejecutar

- Cuando un proceso vuelve del estado bloqueado:

1. Se cambia su estado a ejecutable, TASK_RUNNING.
2. Se elimina de la cola de bloqueo en que estaba.
3. Se añade al rbtree de procesos ejecutables.

La clase de planificación de tiempo real

- Se define la clase de planificación `rt_sched_class`.
- Los procesos de tiempo real son más prioritarios que los normales, y mientras existan procesos de tiempo real ejecutables éstos serán elegidos frente a los normales.
- Un proceso de tiempo real queda determinado por la prioridad que tiene cuando se crea. El kernel no incrementa o disminuye su prioridad en función de su comportamiento.
- Las políticas de planificación de tiempo real `SCHED_RR` y `SCHED_FIFO` posibilitan que el kernel Linux pueda tener un comportamiento *soft real-time* (Sistemas de tiempo real no estricto).
- Al crear el proceso también se especifica la política bajo la cual se va a planificar y existe una llamada al sistema para cambiar la política asignada.

Planificación de CPU en SMP

- Para realizar correctamente la planificación en un entorno SMP (multiprocesador), el kernel deberá tener en cuenta:
  - Se debe repartir equilibradamente la carga entre las distintas CPUs.
  - Se debe tener en cuenta la afinidad de una tarea con una determinada CPU.
  - El kernel debe ser capaz de migrar procesos de una CPU a otra (puede ser una operación costosa).
- Periódicamente una parte del kernel deberá comprobar que se da un equilibrio entre las cargas de trabajo de las distintas CPUs y si detecta que una tiene más procesos que otra, reequilibra pasando procesos de una CPU a otra.