# Tema 2. PROCESOS E HILOS

1. GENERALIDADES SOBRE PROCESOS E HILOS
2. DISEÑO E IMPLEMENTACIÓN DE PROCESOS EN LINUX
3. PLANIFICACIÓN EN LINUX

Mientras no se diga lo contrario nos situamos en un entorno monoprocesador.

**Bibliografía punto 1**:
Cancela Solá, S. [*et al.*]; *Fundamentos de sistemas operativos : teoría y ejercicios resueltos*, S. International Thomson Editores
Stallings, W.; *Sistemas Operativos. Aspectos Internos y Principios de Diseño* (5/e), Prentice Hall
Tanenbaum, A.S.; "Sistemas Operativos Modernos" (3/e), Pearson Prentice Hall

**Bibliografía puntos 2 y 3**:
Love, R.; *Linux Kernel Development (3/e)*, Addison-Wesley Professional, 2010.
Mauerer, W.; *Professional Linux Kernel Architecture*, Wiley, 2008.

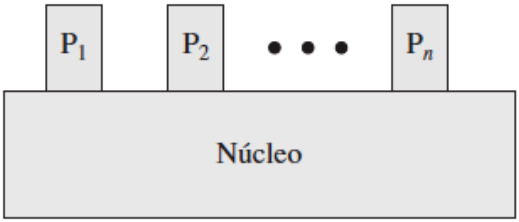


# 1. GENERALIDADES SOBRE PROCESOS E HILOS

## 1.1 Ejecución del SO [Stallings 3.4]

### Núcleo sin procesos

El núcleo del SO se ejecuta fuera de cualquier proceso, como una entidad separada que opera en modo privilegiado

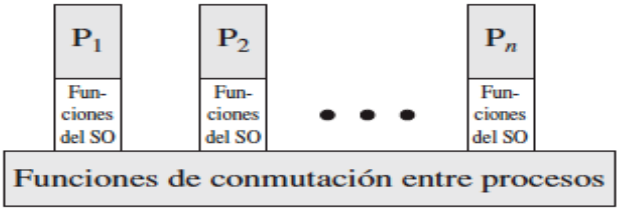



[Stallings Fig. 3.15 a]

**Ejecución dentro de los procesos de usuario:**

El SO se percibe como un conjunto de rutinas que un proceso de usuario invoca para realizar una determinada función,
siendo así que el software del SO se ejecuta en el contexto del proceso de usuario

Un proceso se ejecuta en modo privilegiado cuando se ejecuta el código del SO.



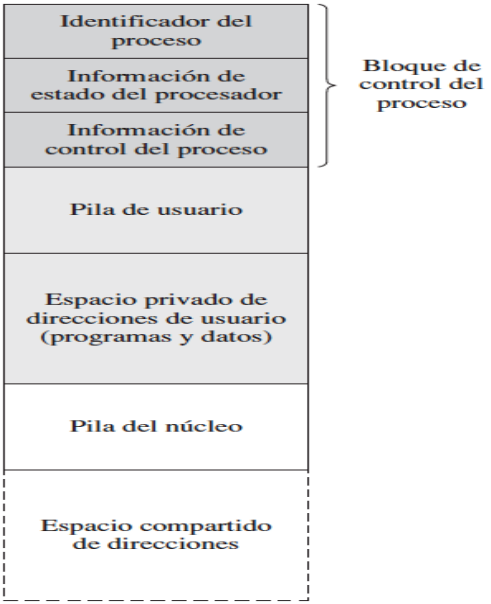

[Stallings Fig. 3.15 b]



[Stallings Fig. 3.16] Estructura típica de imagen de proceso en esta estrategia

**Se usa una pila de núcleo separada para manejar llamadas/retornos al/del modo núcleo**

**El código del SO y sus datos están en el espacio de direcciones compartidas y se comparten entre todos los procesos**



Cuando ocurre una llamada al sistema,

el procesador se pone en modo núcleo y el control se pasa al SO....

la ejecución continúa dentro del proceso de usuario actual

De esta forma, no se realiza un cambio de proceso, sino un cambio de modo dentro del mismo proceso.

El SO, tras realizar su trabajo,

- si determina que el proceso actual debe continuar con su ejecución, se realiza un cambio de modo y prosigue el proceso interrumpido.

- en caso contrario, se pasa el control a la rutina de cambio de proceso



## 1.2 Sobre la creación de procesos

En general, crear un proceso significa
asignarle el espacio de direcciones que utilizará
crear las estructuras de datos para su administración

Los sucesos comunes que provocan la creación de un proceso son:

- **en sistemas bach**: el planificador a largo plazo selecciona uno de los trabajos por lotes en espera para crear el proceso correspondiente

- **en sistemas interactivos**: cuando el usuario se conecta, el SO crea un proceso que ejecuta un intérprete de órdenes (o bien el programa ejecutable asociado al perfil de usuario)

- el SO puede crear un proceso para llevar a cabo un servicio solicitado por un proceso de usuario

Un proceso puede crear otro proceso mediante la llamada al sistema correspondiente

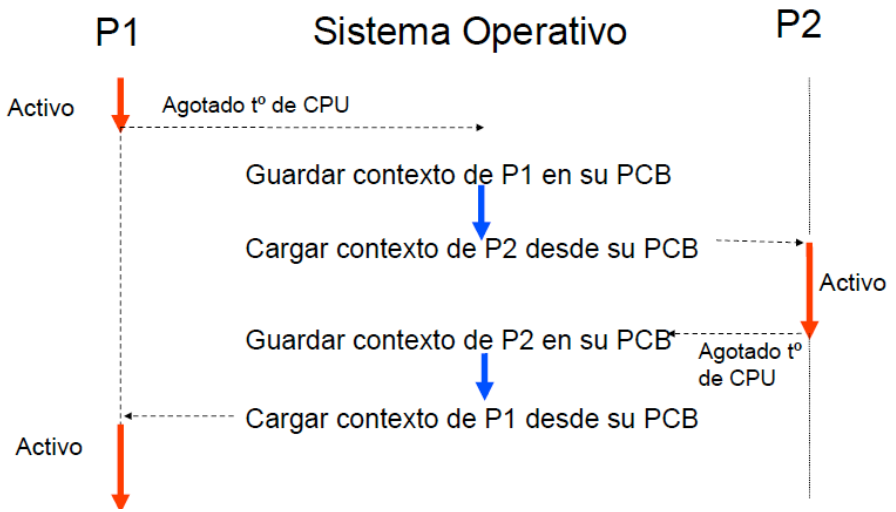
## 1.3 Cambio de contexto

Cuando un proceso está ejecutándose su PC, puntero a pila, registros, etc... están cargados en la CPU

Cuando el SO detiene a un proceso ejecutándose, salva los valores actuales de estos registros (**contexto**) en el PCB (Process Control Block) de ese proceso

La acción de conmutar la CPU de un proceso a otro se denomina **cambio de contexto**.





## 1.4 PCB´s y colas de estados

El SO mantiene una colección de colas que representan el estado de todos los procesos en el sistema

Típicamente hay una cola por estado

Cada PCB está encolado en una cola de estado acorde a su estado actual

Conforme un proceso cambia de estado, su PCB es retirado de una cola y encolado en otra

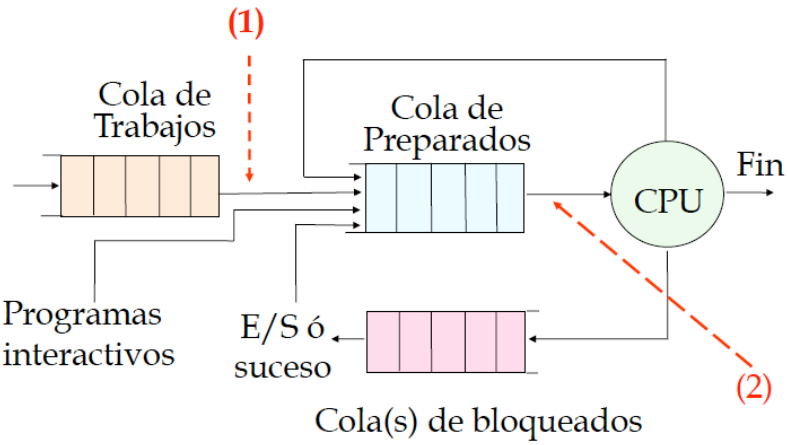


**Colas de estados**

**Cola de trabajos**: conjunto de trabajos pendientes de ser admitidos en el sistema (trabajos por lotes que no están en memoria)

**Cola de preparados**: conjunto de todos los procesos que residen en memoria principal, preparados y esperando para ejecutarse (estado ***preparado*** o ***ejecutable***)

**Cola(s) de bloqueados**: conjunto de procesos esperando un evento o un recurso actualmente no disponible (estado ***bloqueado***)






## 1.5 Planificación de procesos.

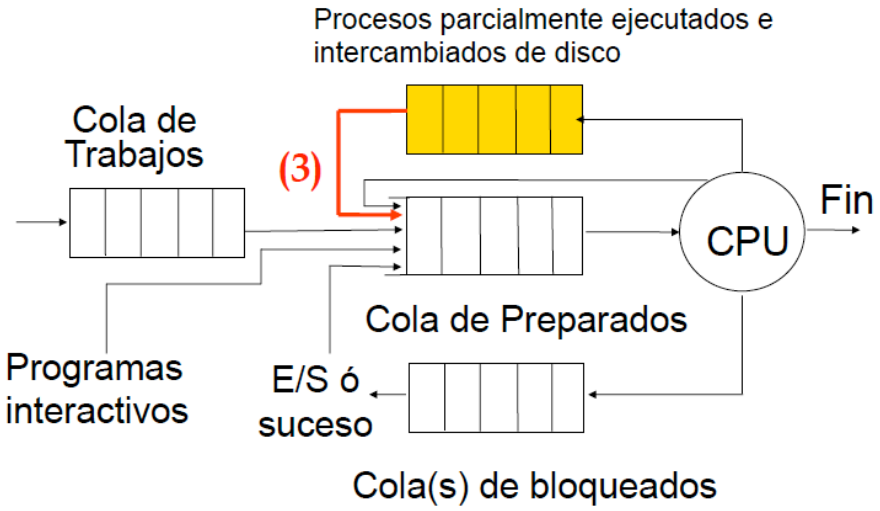
**Planificador:** Parte del SO que controla la utilización de un recurso

**Tipos de planificadores de la CPU:**

- **Planificador a largo plazo (o planificador de trabajos)**:
  Selecciona los trabajos que deben admitirse en el sistema (transición 1 en la figura anterior)

- **Planificador a corto plazo (o planificador de la CPU)**:
  Selecciona al proceso en estado *preparado* o *ejecutable* que debe ejecutarse a continuación y le asigna la CPU (transición 2 en la figura anterior)
  Es invocado muy frecuentemente

- **Planificador a medio**
  Se encarga de sacar/introducir procesos de/a MP (transición 3 en la figura siguiente)






**Comportamiento de un proceso: ráfagas de CPU y de bloqueo** [Tanenbaum 2.4]

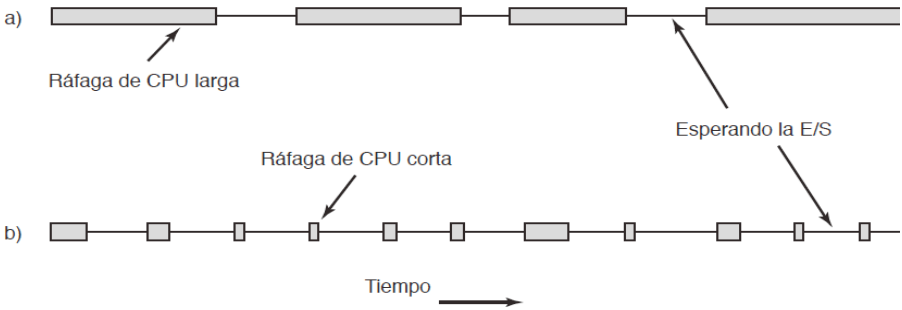
Normalmente un proceso alterna ráfagas de CPU con periodos de tiempo en que se encuentra en estado *bloqueado*

La ejecución del proceso comienza con una ráfaga de CPU.
Puede que realice una operación que le provoque pasar a estado *bloqueado* (por ejemplo realizar una petición de E/S)
En algún momento pasa a estado *ejecutable* (por ejemplo debido al fin de la E/S) y llegará a ser elegido para estar en ejecución
Finalmente, el proceso concluye con una solicitud al SO para finalizar la ejecución

**Ráfaga de CPU**: periodo de tiempo en que el proceso está en estado *ejecutándose*, haciendo uso de la CPU

**Ráfaga de bloqueo**: periodo de tiempo en que el proceso está en estado *bloqueado*

En la figura siguiente se ejemplifica esta alternancia particularizándose al caso de la realización de E/S como causa de bloqueo




[Tanenbaum Fig. 2-38]



**Procesos limitados por E/S – procesos limitados por CPU**

"**proceso limitado por E/S**": dícese del proceso que intercala pequeñas ráfagas de tiempo de CPU con largos periodos de espera,
es decir, frecuentemente realiza alguna operación que provoca que su estado cambie a *bloqueado* (por ejemplo E/S)

ráfagas de CPU cortas

"**proceso limitado por CPU**" : dícese del proceso que realiza cómputos durante largos periodos de tiempo sin realizar ninguna petición que le cambie a estado *bloqueado*

ráfagas de CPU largas

**Mezcla de trabajos**

Es importante que el planificador a largo plazo seleccione una buena mezcla de trabajos, ya que

si un nº alto de procesos fuesen limitados por E/S,
la cola de preparados estará casi siempre vacía y la CPU estaría infrautilizada

si un nº alto de procesos fuesen limitados por CPU,
los dispositivos de E/S (u otros recursos) estarían infrautilizados



**Sobre los términos "despachador" – "planificador a corto plazo"**
Algunos autores definen estos términos como sigue:

"**planificador a corto plazo**": parte del SO que decide a qué proceso darle el control de la CPU

"**despachador**": parte del SO que realiza las acciones adecuadas para efectuar el cambio de asignación de CPU entre dos procesos, tal como lo haya decidido el planificador a corto plazo.
Esto involucra:
- salvar el contexto del proceso actual y restaurar el contexto del nuevo proceso
- salto a la posición de memoria del nuevo proceso para su reanudación

En la asignatura utilizaremos ambos términos como **equivalentes**, refiriéndonos a la **parte del SO que realiza todas las acciones involucradas en la asignación de CPU a un nuevo proceso**

**Activación del planificador a corto plazo:**

El planificador a corto plazo puede activarse cuando...

a) el proceso actual no quiere seguir ejecutándose (finaliza o ejecuta una operación que lo bloquea)

b) un elemento del SO determina que el proceso actual no puede seguir ejecutándose pasándolo a estado *bloqueado* (ej: retiro de MP)

c) el proceso actual cambia de estado *ejecutándose* a *ejecutable* (por ejemplo porque agota el quantum de tiempo asignado)

d) un suceso cambia el estado de un proceso (distinto al actual) de *bloqueado* a *ejecutable*



**Una clasificación de las políticas de planificación:**

* Política de planificación *no apropiativa* (o *sin desplazamiento*):
  una vez que se asigna la CPU a un proceso, no se le puede retirar hasta que éste finalice o se bloquee

  No se incluye el apartado (d) de la lista anterior como causa de activar al planificador a corto plazo

* Política de planificación *apropiativa* (o *con desplazamiento*):
  aunque el proceso actual no pase a bloqueado, el SO puede apropiarse del procesador cuando lo decida

  Sí se incluye el apartado (d) de la lista anterior como causa de activar al planificador a corto plazo

**Sobre las políticas de planificación de la CPU:**
Objetivos:

- buen rendimiento (productividad)
- buen servicio

Para reflexionar sobre el comportamiento de las políticas de planificación se definen ciertas medidas asociadas a cada proceso:
(denotamos por t el tiempo de CPU del proceso)

- Tiempo de respuesta (T): tiempo total transcurrido desde que se crea el proceso hasta que termina
  También se le llama tiempo de finalización

- Tiempo de espera (E): tiempo que el proceso ha estado esperando en la cola de ejecutables
  $E = T - t$

- Penalización (P):
  $P = T / t$



Otras medidas interesantes son:

- tiempo del núcleo: tiempo empleado por el SO en tomar decisiones alusivas a planificación de procesos y haciendo los cambios de contexto necesarios
  en un sistema eficiente debe representar entre el 10% y el 30% del total de tiempo de la CPU

- tiempo en que la cola de ejecutables está vacía y no se realiza ningún trabajo productivo

En un entorno interactivo será prioritario asegurar una respuesta ágil del sistema frente a una buena utilización de la CPU

En un entorno no interactivo se primará el aprovechamiento de la CPU

## Algoritmo de planificación FCFS (First Come First Served) o FIFO

Los procesos son servidos según el orden de llegada a la cola de ejecutables

Algoritmo no apropiativo: cada proceso se ejecutará hasta que finalice o se bloquee

Fácil de implementar pero pobre en cuanto a prestaciones

Todos los procesos pierden la misma cantidad de tiempo esperando en la cola de ejecutables



El tiempo de espera es independiente del tiempo de CPU de la ráfaga

La penalización disminuye al aumentar el tiempo de CPU de la ráfaga

Descartado para entornos interactivos debido a que

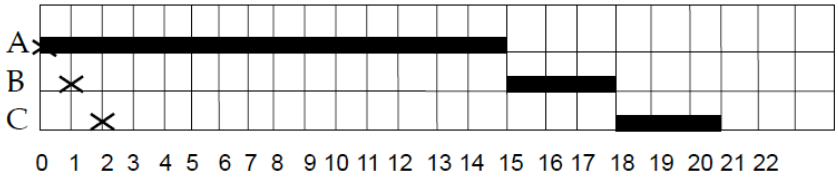
un proceso se adueña de la CPU

las ráfagas cortas quedan muy penalizadas (es característico de los procesos interactivos que tengan ráfagas cortas)

**Ejemplo: Algoritmo FCFS**

| Procesos | llegada | tiempo CPU | T (t.respuesta) | E (t. espera) |
|---|---|---|---|---|
| A | 0 | 15 | 15-0=15 | 0 |
| B | 1 | 3 | 18-1=17 | 14 |
| C | 2 | 3 | 21-2=19 | 16 |

Diagrama de ocupación de la CPU






## Algoritmo de planificación SJF (Shortest Job First) o "El más corto primero"

Algoritmo no apropiativo.

Cuando el procesador queda libre, selecciona el proceso tenga la siguiente ráfaga de CPU más corta.

Si existen dos o más procesos en igualdad de condiciones, se sigue FCFS.

Necesita conocer explícitamente el tiempo estimado de ejecución ¿Cómo? Es necesaria una estimación

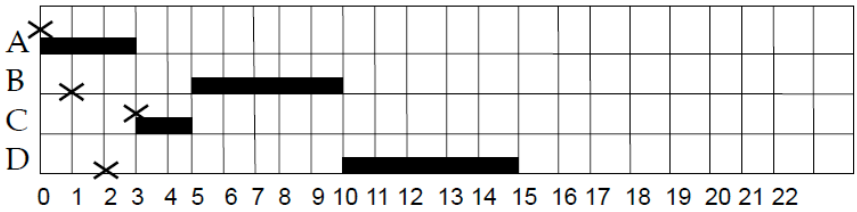
Tiempo medio de espera bajo.

Tiene solo utilidad teórica, ya que es necesario conocer el tiempo de CPU que durará una ráfaga antes de que sea ejecutada.

**Ejemplo: Algoritmo SJF**

| Procesos | llegada | Ráfaga real | Ráfaga estimada |
|---|---|---|---|
| A | 0 | 3 | 3 |
| B | 1 | 5 | 5 |
| C | 3 | 2 | 2 |
| D | 2 | 5 | 5 |

Diagrama de ocupación de la CPU

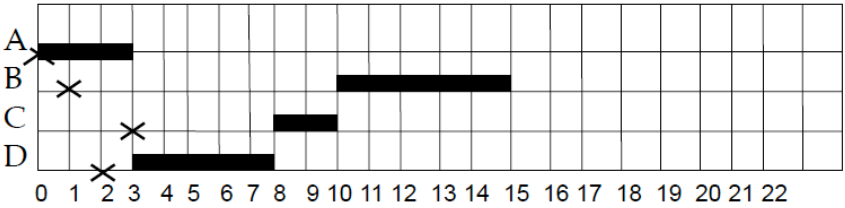





**Ejemplo: Algoritmo SJF**

| Procesos | llegada | Ráfaga real | Ráfaga estimada |
|---|---|---|---|
| A | 0 | 3 | 3 |
| B | 1 | 5 | 6 |
| C | 3 | 2 | 5 |
| D | 2 | 5 | 5 |

Diagrama de ocupación de la CPU



## Algoritmo de planificación SJF apropiativo o SRTF (Shortest Remaining Time First)

Cuando un proceso llega a la cola de ejecutables se comprueba si su tiempo de CPU es menor que el tiempo que le queda al proceso que está ejecutándose; casos:

si es menor: se realiza un cambio de contexto y se asigna la CPU al proceso que acaba de llegar
en caso contrario: continúa el proceso que estaba ejecutándose

Ráfagas cortas muy bien tratadas

Ráfagas largas muy mal tratadas

Respecto al conjunto de algoritmos, es el que proporciona la menor penalización en promedio

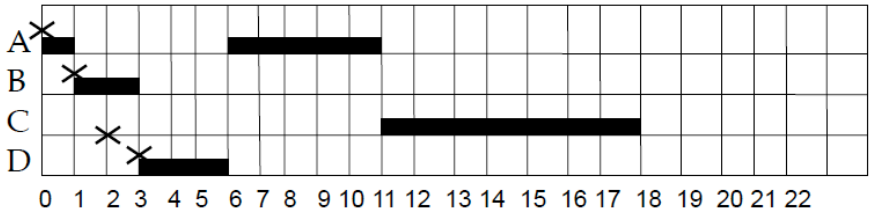
Mantiene la cola de ejecutables con la mínima ocupación posible



**Ejemplo: Algoritmo SRTF**

| Procesos | Tº llegada | Ráfaga real | Ráfaga estimada |
|---|---|---|---|
| A | 0 | 6 | 6 |
| B | 1 | 2 | 2 |
| C | 2 | 7 | 7 |
| D | 3 | 3 | 3 |

Diagrama de ocupación de la CPU



## Planificación por turnos, "Barrido Cíclico" o Round Robin (RR)

(Si no se especifica otra cosa, entendemos que aplicamos los algoritmos sin desplazamiento, es decir sin derecho preferente o de forma no apropiativa)

La cola de ejecutables se ordena por orden cronológico de llegada.

Se elige el proceso más antiguo, disfrutará de la CPU durante un máximo de tiempo que llamamos **quantum**

Si el proceso finaliza o se bloquea antes de agotar el quantum, liberal a CPU y se toma el siguiente proceso de la cola de ejecutables

En caso contrario, se le retira el control de la CPU y se coloca al final de la cola de ejecutables.



**Ejemplo: Algoritmo Round Robin (con valor de quantum=4)**

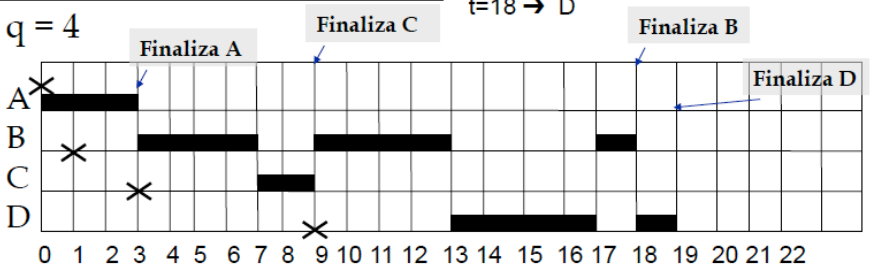
| Procesos | Tº llegada | Ráfaga |
|---|---|---|
| A | 0 | 3 |
| B | 1 | 9 |
| C | 3 | 2 |
| D | 9 | 5 |

Estado de la cola de ejecutables

t=0 ➔ A; t=3 ➔ BC; t=7 ➔ CB

t=9 ➔ BD; t=13 ➔ DB; t=17 ➔ BD

t=18 ➔ D



Repita el ejemplo anterior con valor de quantum = 1

Si un proceso X llega a la cola de ejecutables al mismo tiempo que el actual Y agota su quantum (y sigue ejecutable), estos eventos son ordenados tomando el evento "fin de quantum" el último.

Es decir, el SO considera la llegada de X como nuevo ejecutable antes que la resolución de "fin de quantum".

El tiempo de espera crece al aumentar el tiempo de CPU de la ráfaga

La penalización es independiente del tiempo de CPU de la ráfaga (excepto para ráfagas muy cortas en relación al quantum).



Consideraciones sobre el valor del quantum

Si el valor del quantum es muy grande (en relación al tiempo promedio de duración de las ráfagas) el algoritmo degenera en FCFS

Si el valor del quantum es muy grande se producen demasiados cambios de contexto (tiempo del núcleo muy alto)

Véanse las siguientes figuras de Stallings sobre la relación del tiempo de espera E y de la penalización P (aludida por el autor por el término "Tiempo de estancia normalizado") frente a t (tiempo de CPU de la ráfaga) para los algoritmos FCFS y RR

## E frente a t en los algoritmos FCFS y RR

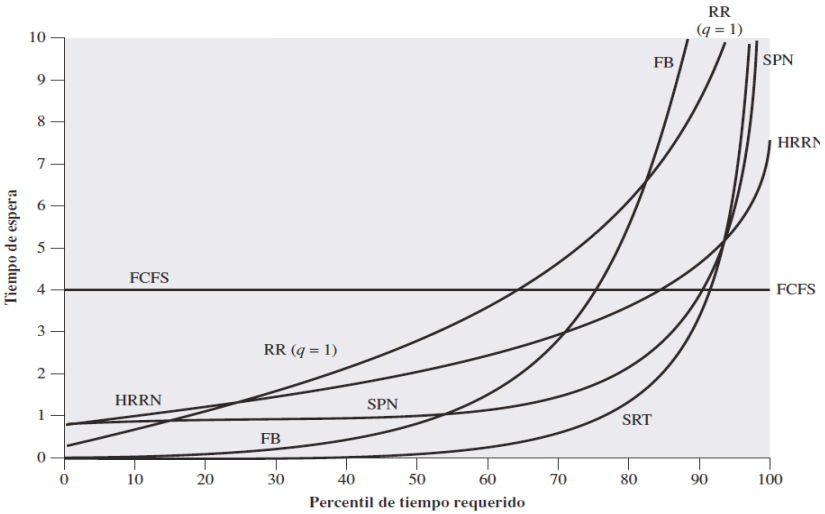



**Figura 9.15.** Resultados de la simulación para el tiempo de espera.

35

## P frente a t en los algoritmos FCFS y RR

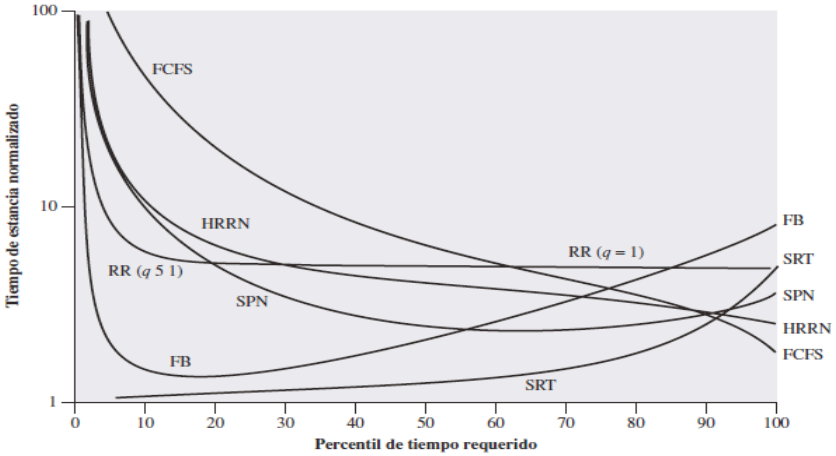



**Figura 9.14.** Resultados de la simulación para el tiempo de estancia normalizado.

[Stallings Figs 9.15 y 9.14]

36

## Algoritmo de planificación de Colas múltiples

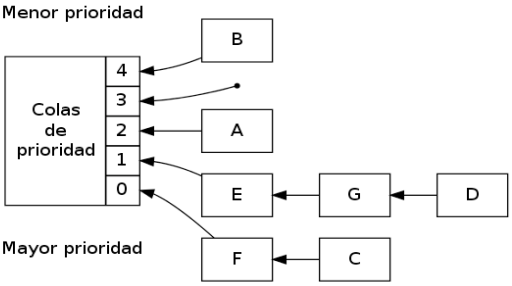
Cada proceso tiene asociado un valor de prioridad (número entero)

Asumimos el criterio de que a menor valor numérico de prioridad implica una mayor preferencia en la asignación de CPU

Se elige para ejecutar el proceso en estado ejecutable que tenga la máxima prioridad (menor valor numérico con el criterio que estamos usando)

Puesto que puede haber varios procesos ejecutables con el mismo nivel de prioridad, es necesario usar otro algoritmo para decidir entre ellos.

Cada nivel de prioridad tiene asociado un algoritmo para elegir entre los procesos de ese nivel de prioridad (FCFS, RR...)



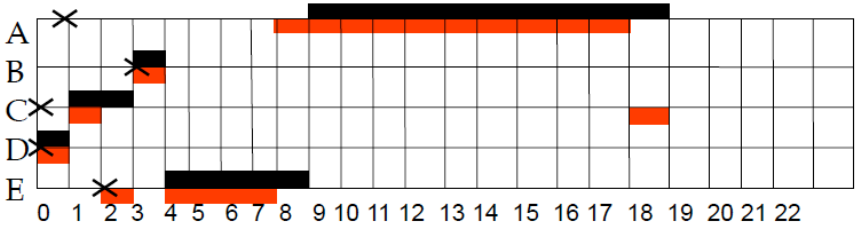

**Ejemplo: Algoritmo de colas múltiples en que todas las colas tienen asociado el algoritmo FIFO**

| Procesos | Tº llegada | Ráfaga | Prioridad |
|---|---|---|---|
| A | 1 | 10 | 3 |
| B | 3 | 1 | 1 |
| C | 0 | 2 | 3 |
| D | 0 | 1 | 2 |
| E | 2 | 5 | 2 |

No apropiativo

Apropiativo

Diagrama de ocupación de la CPU

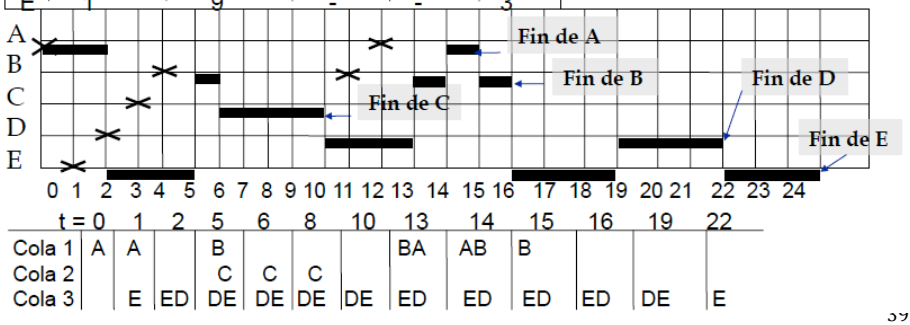


**Ejemplo: Algoritmo de colas múltiples NO APROPIATIVO en que todas las colas tienen el algoritmo RR con distintos valores de quantum**

| P | TºLlegada | Ráfaga | Bloqueo | Ráfaga | Cola |
|---|---|---|---|---|---|
| A | 0 | 2 | 10 | 1 | 1 |
| **B** | **4** | **1** | **5** | **2** | **1** |
| C | 3 | 4 | - | - | 2 |
| **D** | **2** | **6** | **-** | **-** | **3** |
| E | 1 | 9 | - | - | 3 |

El algoritmo de cada cola es **RR** con **q=1, 2 y 3** ms respectivamente.

El algoritmo entre colas es **prioridades no apropiativo**




| t = 0 | 1 | 2 | 5 | 6 | 8 | 10 | 13 | 14 | 15 | 16 | 19 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cola 1 | A | A | | B | | | BA | AB | B | | | |
| Cola 2 | | | | C | C | C | | | | | | |
| Cola 3 | | E | ED | DE | DE | DE | DE | ED | ED | ED | ED | DE | E |

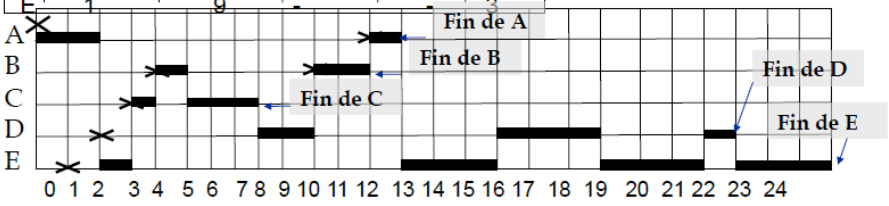


**Ejemplo: Algoritmo de colas múltiples APROPIATIVO en que todas las colas tienen el algoritmo RR con distintos valores de quantum**

| P | TºLlegada | Ráfaga | Bloqueo | Ráfaga | Cola |
|---|---|---|---|---|---|
| A | 0 | 2 | 10 | 1 | 1 |
| **B** | **4** | **1** | **5** | **2** | **1** |
| C | 3 | 4 | - | - | 2 |
| **D** | **2** | **6** | **-** | **-** | **3** |
| E | 1 | 9 | - | - | 3 |

El algoritmo de cada cola es **RR** con **q=1, 2 y 3** mlsg. respectivamente.

El algoritmo entre colas es **prioridades *apropiativo.***




| t = 0 | 1 | 2 | 3 | 4 | 5 | 7 | 8 | 10 | 11 | 12 | 13 | 16 | 19 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cola 1 | A | A | | | B | | | B | B | A | | | | | |
| Cola 2 | | | | C | C | C | C | | | | | | | | |
| Cola 3 | | E | ED | DE | DE | DE | DE | DE | ED | ED | ED | ED | DE | ED | DE | E |

Cuando las prioridades son fijas puede surgir el problema de la **inanición**:

un proceso puede estar esperando indefinidamente sin llegar a ejecutarse

ocurrirá si van apareciendo siempre procesos ejecutables de mayor prioridad.

Posibilidades para evitar el problema de la inanición:

- añadir un mecanismo de **envejecimiento**:
  el algoritmo aumenta la prioridad a un proceso cuando ha superado un determinado tiempo en espera sin ser ejecutado

- asegurar que cada cola disfruta de un determinado porcentaje de tiempo de CPU (que deberá repartir entre sus procesos)
  ejemplo con 2 colas:...



ejemplo con 2 colas:

cola de mayor prioridad:
asociada a los procesos interactivos
se rige por el algoritmo RR
deberá disfrutar del 80% del tiempo de CPU

cola de menor prioridad:
asociada a trabajos batch
se rige por el algoritmo FCFS
deberá disfrutar del 20% del tiempo de CPU

## Algoritmo de planificación de colas múltiples con realimentación o con traspaso

Se tienen $N$ colas de prioridad ($cola_1$ , $cola_2$ , ... $cola_{N-1}$) siendo la 1 la cola más prioritaria

Todas las colas tienen asociado el algoritmo RR con valores de quantum $q_1$ , $q_2$ , ... $q_{N-1}$ siendo $q_1 < q_2 < .. < q_{N-1}$

Cuando un proceso comienza como ejecutable, es asociado a $cola_1$

Un proceso puede cambiar de cola (**traspaso**). El proceso cambia de $cola_i$ a $cola_{i+1}$ cuando satisface determinado criterio definido en el algoritmo. Por ejemplo:

un proceso en $cola_i$ pasa a $cola_{i+1}$ cuando ha satisfecho en $cola_i$ cierto número de quanta, denotado por $m_i$ , en $cola_i$,

Cuando un proceso llega a $cola_{N-1}$ permanece en ella hasta que termina



Cuando un proceso se desbloquea volviendo a estar en estado ejecutable, el algoritmo puede decantarse por dos posibilidades:

(a) prosigue en la cola en que estaba antes de bloquearse

(b) entra en $cola_1$

Reflexiones:

- las ráfagas cortas están bien tratadas, particularmente si se aplica la versión con apropiatividad

- las ráfagas largas están tratadas acorde a su duración, habría que añadir una política de envejecimiento para que no se vean sistemáticamente postergadas

- el algoritmo consigue adecuar la asignación de CPU a la duración de las ráfagas, sin necesidad de estimación.

**Ejemplo: Algoritmo de colas múltiples con traspaso (no apropiativo)** concretado como sigue:

Tres colas gestionadas mediante RR:

- ***cola*$_1$** con quantum $q_1 = 2$ ms
- ***cola*$_2$** con quantum $q_2 = 4$ ms
- ***cola*$_3$** con quantum $q_3 = 8$ ms

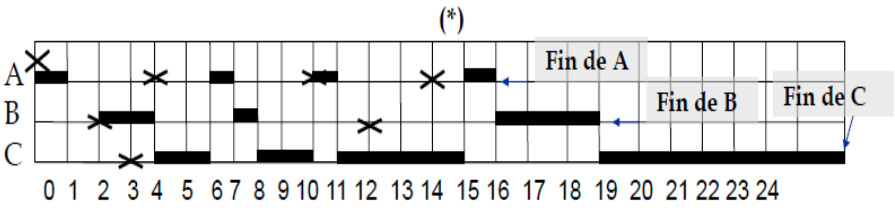
Un proceso en ***cola*$_i$** pasa a ***cola*$_{i+1}$** cuando ha satisfecho en ***cola*$_i$** 2 quanta de tiempo, o bien agota uno y se bloquea en el siguiente

Cuando un proceso se desbloquea prosigue en la cola en que estaba antes de bloquearse (a no ser que haya traspaso)



| P | TºLlegada | TºServicio total | Ráfaga | Bloqueo |
|---|---|---|---|---|
| A | 0 | 4 | 1 | 3 |
| B | 2 | 6 | 3 | 4 |
| C | 3 | 23 | - | - |

(*) Los procesos tienen un comportamiento **cíclico**




| t = | 0 | 1 | 2 | 4 | 6 | 7 | 8 | 10 | 11 | 15 | 16 | 19 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cola 1 | A | | B | CAB | ABC | BC | C | A | | A | | | |
| Cola 2 | | | | | | | | C | C | BC | BC | C | |
| Cola 3 | | | | | | | | | | | | C | |

# 2. DISEÑO E IMPLEMENTACION DE PROCESOS EN LINUX

En los puntos 2 y 3 de este tema nos basamos el kernel 2.6 de Linux.
(Se descargar los fuentes de www.kernel.org )

Nomenclatura usada para las rutas de archivos : aludiremos los archivos fuente expresando su ruta relativa desde el directorio donde hemos descargado todos los fuentes.

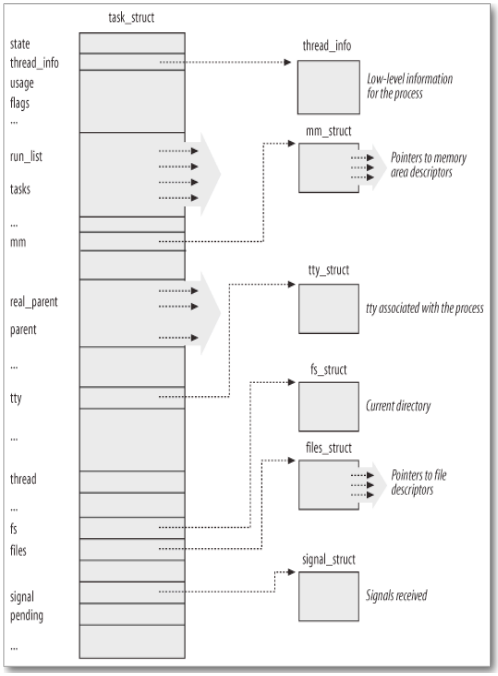


## 2.1 Representación de los procesos [Mauerer 2.3]

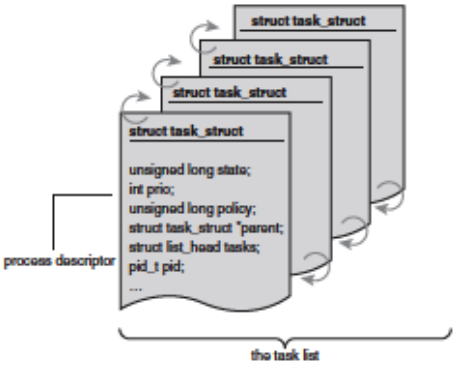
En Linux, un proceso es representado por estas dos estructuras:

el PCB que es una estructura del tipo `struct task_struct`

y una estructura del tipo `struct thread_info`.



El kernel almacena la lista de procesos como una lista circular doblemente enlazada llamada **lista de tareas** (task list).




Cada elemento en la task list es un descriptor de proceso de tipo struct task_struct (definido en `<include/linux/sched.h>`):



```
struct task_struct { /// del kernel 2.6.24
   volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
   void *stack;
   atomic_t usage;
   unsigned int flags;     /* per process flags, defined below */
   unsigned int ptrace;

   int lock_depth;         /* BKL lock depth */

#ifdef CONFIG_SMP
#ifdef __ARCH_WANT_UNLOCKED_CTXSW
   int oncpu;
#endif
#endif

   int prio, static_prio, normal_prio;
   struct list_head run_list;
   const struct sched_class *sched_class;
   struct sched_entity se;

#ifdef CONFIG_PREEMPT_NOTIFIERS
   /* list of struct preempt_notifier: */
   struct hlist_head preempt_notifiers;
#endif

   unsigned short ioprio;
   /*
    * fpu_counter contains the number of consecutive context switches
    * that the FPU is used. If this is over a threshold, the lazy fpu
    * saving becomes unlazy to save the trap. This is an unsigned char
    * so that after 256 times the counter wraps and the behavior turns
    * lazy again; this to deal with bursty apps that only use FPU for
```

```
     * a short time
     */
    unsigned char fpu_counter;
    s8 oomkilladj; /* OOM kill score adjustment (bit shift). */
#ifdef CONFIG_BLK_DEV_IO_TRACE
    unsigned int btrace_seq;
#endif
    unsigned int policy;
    cpumask_t cpus_allowed;
    unsigned int time_slice;

#if defined(CONFIG_SCHEDSTATS) || defined(CONFIG_TASK_DELAY_ACCT)
    struct sched_info sched_info;
#endif
    struct list_head tasks;
    /*
     * ptrace_list/ptrace_children forms the list of my children
     * that were stolen by a ptracer.
     */
    struct list_head ptrace_children;
    struct list_head ptrace_list;

    struct mm_struct *mm, *active_mm;

/* task state */
    struct linux_binfmt *binfmt;
    int exit_state;
    int exit_code, exit_signal;
    int pdeath_signal;  /*  The signal sent when the parent dies  */
    /* ??? */
    unsigned int personality;
    unsigned did_exec:1;
```



```
    pid_t pid;
    pid_t tgid;

#ifdef CONFIG_CC_STACKPROTECTOR
    /* Canary value for the -fstack-protector gcc feature */
    unsigned long stack_canary;
#endif
    /*
     * pointers to (original) parent process, youngest child, younger sibling,
     * older sibling, respectively.  (p->father can be replaced with
     * p->parent->pid)
     */
    struct task_struct *real_parent; /* real parent process (when being debugged) */
    struct task_struct *parent;  /* parent process */
    /*
     * children/sibling forms the list of my children plus the
     * tasks I'm ptracing.
     */
    struct list_head children;    /* list of my children */
    struct list_head sibling;     /* linkage in my parent's children list */
    struct task_struct *group_leader;  /* threadgroup leader */

    /* PID/PID hash table linkage. */
    struct pid_link pids[PIDTYPE_MAX];
    struct list_head thread_group;

    struct completion *vfork_done;          /* for vfork() */
    int __user *set_child_tid;          /* CLONE_CHILD_SETTID */
    int __user *clear_child_tid;        /* CLONE_CHILD_CLEARTID */

    unsigned int rt_priority;
    cputime_t utime, stime, utimescaled, stimescaled;
```

```
    cputime_t gtime;
    cputime_t prev_utime, prev_stime;
    unsigned long nvcsw, nivcsw; /* context switch counts */
    struct timespec start_time;         /* monotonic time */
    struct timespec real_start_time;    /* boot based time */
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-
    specific */
    unsigned long min_flt, maj_flt;

    cputime_t it_prof_expires, it_virt_expires;
    unsigned long long it_sched_expires;
    struct list_head cpu_timers[3];

/* process credentials */
    uid_t uid,euid,suid,fsuid;
    gid_t gid,egid,sgid,fsgid;
    struct group_info *group_info;
    kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
    unsigned keep_capabilities:1;
    struct user_struct *user;
#ifdef CONFIG_KEYS
    struct key *request_key_auth; /* assumed request_key authority */
    struct key *thread_keyring;  /* keyring private to this thread */
    unsigned char jit_keyring;   /* default keyring to attach requested keys to */
#endif
    char comm[TASK_COMM_LEN]; /* executable name excluding path
                      - access with [gs]et_task_comm (which lock
                        it with task_lock())
                      - initialized normally by flush_old_exec */
/* file system info */
    int link_count, total_link_count;
#ifdef CONFIG_SYSVIPC
```



```
/* ipc stuff */
    struct sysv_sem sysvsem;
#endif
/* CPU-specific state of this task */
    struct thread_struct thread;
/* filesystem information */
    struct fs_struct *fs;
/* open file information */
    struct files_struct *files;
/* namespaces */
    struct nsproxy *nsproxy;
/* signal handlers */
    struct signal_struct *signal;
    struct sighand_struct *sighand;

    sigset_t blocked, real_blocked;
    sigset_t saved_sigmask;       /* To be restored with TIF_RESTORE_SIGMASK */
    struct sigpending pending;

    unsigned long sas_ss_sp;
    size_t sas_ss_size;
    int (*notifier)(void *priv);
    void *notifier_data;
    sigset_t *notifier_mask;
#ifdef CONFIG_SECURITY
    void *security;
#endif
    struct audit_context *audit_context;
    seccomp_t seccomp;

/* Thread group tracking */
     u32 parent_exec_id;
```

```
    u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
   spinlock_t alloc_lock;

   /* Protection of the PI data structures: */
   spinlock_t pi_lock;

#ifdef CONFIG_RT_MUTEXES
   /* PI waiters blocked on a rt_mutex held by this task */
   struct plist_head pi_waiters;
   /* Deadlock detection and priority inheritance handling */
   struct rt_mutex_waiter *pi_blocked_on;
#endif

#ifdef CONFIG_DEBUG_MUTEXES
   /* mutex deadlock detection */
   struct mutex_waiter *blocked_on;
#endif
#ifdef CONFIG_TRACE_IRQFLAGS
   unsigned int irq_events;
   int hardirqs_enabled;
   unsigned long hardirq_enable_ip;
   unsigned int hardirq_enable_event;
   unsigned long hardirq_disable_ip;
   unsigned int hardirq_disable_event;
   int softirqs_enabled;
   unsigned long softirq_disable_ip;
   unsigned int softirq_disable_event;
   unsigned long softirq_enable_ip;
   unsigned int softirq_enable_event;
   int hardirq_context;
   int softirq_context;
```



```
#endif
#ifdef CONFIG_LOCKDEP
# define MAX_LOCK_DEPTH 30UL
   u64 curr_chain_key;
   int lockdep_depth;
   struct held_lock held_locks[MAX_LOCK_DEPTH];
   unsigned int lockdep_recursion;
#endif

/* journalling filesystem info */
   void *journal_info;

/* stacked block device info */
   struct bio *bio_list, **bio_tail;

/* VM state */
   struct reclaim_state *reclaim_state;

   struct backing_dev_info *backing_dev_info;

   struct io_context *io_context;
   unsigned long ptrace_message;
   siginfo_t *last_siginfo; /* For ptrace use.  */
...}
```

**ALGUNOS CONTENIDOS DE task_struct:**

* estado e información de ejecución tales como señales pendientes, Pid, puntero al padre y a otros procesos relacionados (se verá más adelante), prioridades, información sobre el tiempo de CPU..
* Información sobre asignación de memoria
* Credenciales del proceso como identificativos de usuario y de grupo
* Archivos abiertos
* Información sobre comunicación entre procesos
* Manejadores de señales usados por el proceso para responder a las señales

Dentro del kernel las tareas son referenciadas mediante un puntero a su task_struct.

La macro *current* proporciona un puntero al descriptor de proceso de la tarea que se está ejecutando actualmente.



## 2.2 Estados de un proceso [Mauerer 2.3]

La variable *state* de task_struct especifica el estado actual del proceso.
Valores: (definidas en <sched.h>)

**TASK_RUNNING:** proceso ejecutable, tanto si está actualmente ejecutándose o está esperando a que se le asigne CPU.
Si el proceso se está ejecutando puede estar tanto en el espacio de usuario como en el espacio del kernel.

**TASK_INTERRUPTIBLE**: proceso bloqueado o durmiendo.
la tarea no está lista para ejecutarse porque espera un evento.
cuando el kernel notifique al proceso (mediante una señal) que el evento ha ocurrido se calificará en el estado `TASK_RUNNING` y podrá reanudar su ejecución cuando el planificador le asigne la CPU.

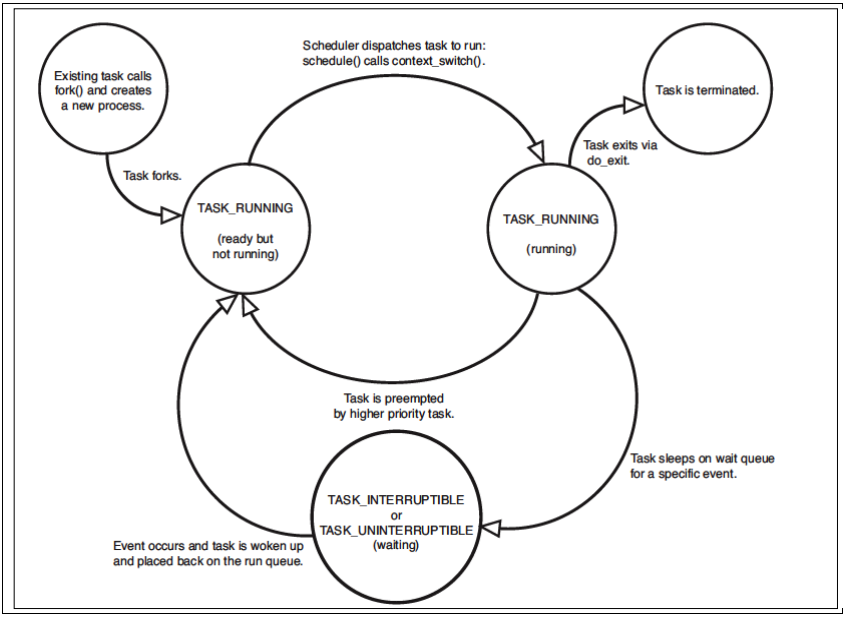
**TASK_UNINTERRUPTIBLE**: estado idéntico al anterior excepto que no es necesariamente una señal lo que despertará al proceso, sino que el propio código del kernel lo calificará como ejecutable cuando sea oportuno.

**TASK_STOPPED**: proceso parado o detenido, no se está ejecutando ni es elegible para ejecutarse. Ocurre cuando la tarea recibe señales como SIGSTOP o ciertas señales de depuración.

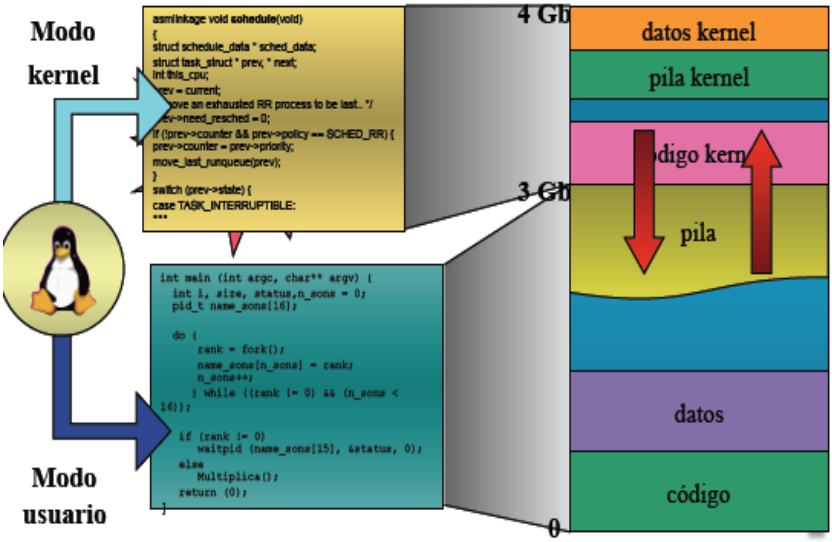
**EXIT_ZOMBIE**: una tarea está en estado zombie cuando ha terminado pero todavía el padre no ha tomado su valor de retorno,
éste debe permanecer almacenado hasta que el padre lo recoja con un wait, por tanto no se puede destruir todavía su task_struct.

**EXIT_DEAD:** es el estado al que pasa un proceso tras la la ejecución de wait por el padre, hasta que sea completamente eliminado del sistema.





## 2.3 Estructura interna de un proceso en linux






Pila: zona del espacio de direcciones lógicas de un proceso que se utiliza para gestionar las llamadas a función que se efectúa en el código del programa.

Está formada por un conjunto de capas con estructura LIFO.

Cada vez que se realiza una llamada a una función se crea un marco nuevo en la pila que contiene...

las variables locales de la función,
sus parámetros actuales,
una dirección de retorno,
y la dirección del marco anterior para poder eliminar éste.

Cada vez que una función retorna se elimina el marco actual.

Como el proceso puede estar trabajando en dos modos (usuario o kernel), habrá una pila para cada modo.

## 2.4 Contexto de un proceso [Love Pág. 29]

La ejecución normal de un proceso ocurre en el espacio de usuario (user-space);

--- Si se produce una llamada al sistema, o se genera una excepción,
.....entra en el espacio del kernel (kernel-space)
....y se dice entonces que el kernel "se está ejecutando en el contexto del proceso", o que está en el contexto del proceso.

--- Cuando se termina dicha llamada o tratamiento de excepción el proceso reanuda su ejecución en su espacio de usuario (a no ser que un proceso de más alta prioridad se haya convertido en ejecutable en ese espacio de tiempo en cuyo caso el planificador lo elegirá para asignarle CPU).

En contraposición a esto, cuando ocurre una interrupción el sistema no está ejecutándose en el contexto de proceso alguno, sino que está ejecutando una rutina de manejo de interrupción (que nunca tiene asociado un proceso en particular)



## 2.5 El árbol de procesos [Love Pág. 29-30]

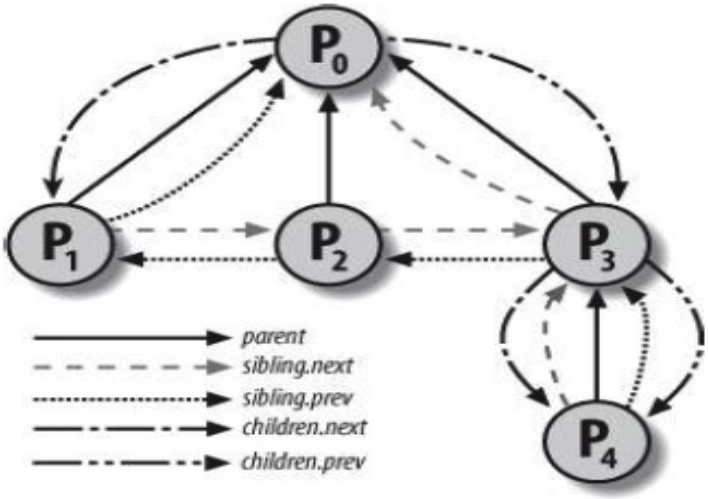
* En todos los entornos Unix hay una jerarquía de procesos definida como sigue:

Si el proceso P0 hace una llamada la sistema fork y así genera el proceso P1, se dice que P0 es el proceso padre y P1 es el hijo.

Si el proceso P0 hace varios fork general de varios proceso hijos P1,P2,P3, la relación entre ellos es de hermanos (sibling)

* Todos los procesos son descendientes del proceso init (cuyo pid es 1);
El kernel comienza init en el último paso del proceso de **arranque del sistema**.

El proceso **init lanza a los demás procesos** completando el proceso de arranque.




Ver Mauerer figura 1-11 (pag 21) y 2-6 (pag 63)



* **Cada task_struct tiene un puntero** ...

a la task_struct de su padre:

```
struct task_struct *parent
```

a una lista de hijos (llamada children):

```
struct list_head children;
/*apunta a la cabeza a lista de mis hijos*/
```

y a una lista de sus hermanos (llamada sibling):

```
struct list_head sibling
     /* ...a la lista de hijos de mi padre */
```

El kernel dispone de procedimientos eficaces para las acciones usuales de manipulación de la lista.

* Dado el proceso actual, es posible obtener el descriptor de proceso de su padre mediante ....este código:

```
struct task_struct *my_parent = current->parent;
```

* El descriptor de proceso del proceso **init** está almacenado estáticamente como **init_task**.

* El código siguiente pone de manifiesto la relación existente entre cualquier proceso y el **init**:

```
struct task_struct *task;
for (task = current; task != &init_task;
                                    task = task->parent);
    /* task now points to init */
```

* Recorrer todos los procesos en el sistema es fácil porque la lista de tareas es una lista circular doblemente enlazada



## 2.6 Hebras kernel [Love pag 33 y ss]

A veces es útil que el kernel realiza algunas operaciones en segundo plano, para lo cual se crean hebras kernel (procesos que existen únicamente en el espacio del kernel).

La principal diferencia entre hebras kernel y procesos normales es que las hebras kernel no tienen un espacio de direcciones (su puntero mm es NULL), se ejecutan únicamente en el espacio del kernel.

Por lo demás son planificadas y podrían ser expropiadas, como procesos normales.

Se crean por el kernel al levantar el sistema, mediante una llamada a clone().

Y como todo proceso, termina cuando realizan una operación exit o cuando otra parte del kernel provoca su finalización

## 2.7 Ejecutando llamadas al sistema para gestión de procesos

Nos centramos en las llamadas al sistema para gestión de procesos como fork, vfork, y clone.

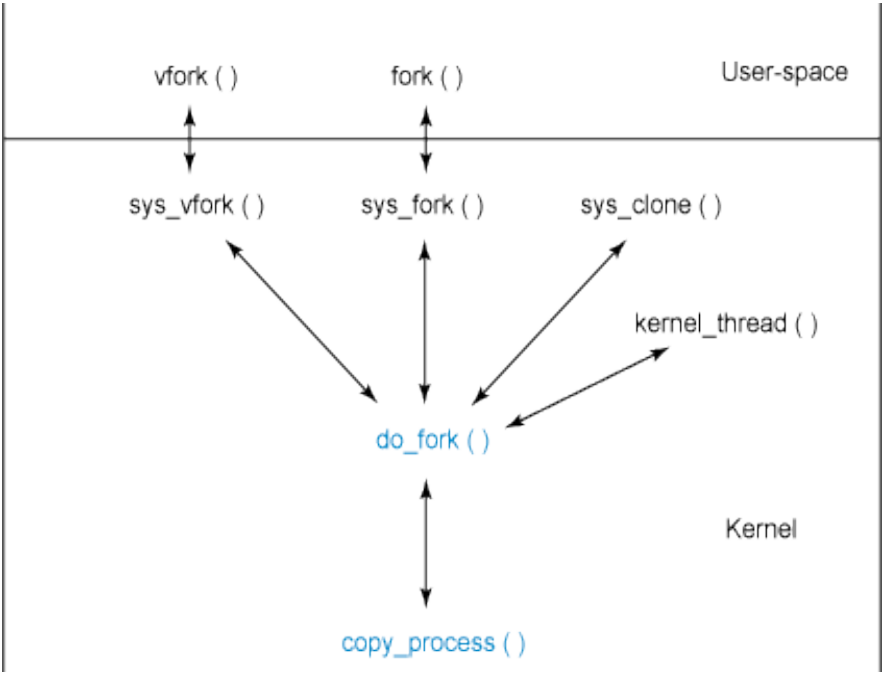
Normalmente estas llamadas se invocan a través de las librerías de C que realizan la comunicación con el kernel

El punto de entrada para fork, vfork y clone son las funciones sys_fork, sys_vfork y sys-clone;

su implementación es dependiente de la arquitectura puesto que la forma en que los parámetros se pasan entre el espacio de usuario y el espacio del kernel son diferentes en las distintas arquitecturas.

La labor de las anteriores funciones es extraer la información suministrada en el espacio de usuario (parámetros de la llamada) e invocar a la función do_fork (independiente de la arquitectura) que es quien realiza la duplicación de procesos.





## Creación de procesos [Love pag 32]

Situémonos en el espacio de usuario; se produce una llamada a alguna de las rutina de librería para crear un nuevo proceso como fork(), vfork() o clone();

Dado que todas ellas, básicamente, realizan la misma función que es crear un nuevo proceso (aunque varían en las características de éste), a grandes rasgos ocurre la misma secuencia de llamadas:

.... se transfiere el control a la función do_fork() del kernel (definida en <kernel/fork.c>)

... que a su vez llama a la función copy_process(), que realiza en sí la creación del nuevo proceso

tras el fin de copy_process, do_fork hará posible que el nuevo hijo se ejecute.



### Actuación de copy_process()

1. Se crea una nueva pila kernel, la estructura thread_info y la task_struct para el nuevo proceso con los valores de la tarea actual.

2. Para los elementos de task_struct del hijo que deban tener valores distintos a los del padre, se les dan los valores iniciales correctos (como por ejemplo datos para estadísticas).

3. Se establece el estado del hijo a TASK_UNINTERRUPTIBLE mientras se realizan las acciones siguientes.

4. Se establecen valores adecuados para los flags de la task_struct del hijo:
   pone a 0 el flag PF_SUPERPRIV
   (indica si la tarea usa privilegio de superusuario)
   pone a 1 el flag PF_FORKNOEXEC
   (indica si 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(), copy_process() duplica o comparte recursos como archivos abiertos, información de sistemas de archivos, manejadores de señales, espacio de direccionamiento del proceso.....

> Normalmente estos recursos son compartidos entre tareas de un mismo proceso (contrario al caso de que se creen nuevos recursos para el hijo con los valores iniciales que tenía del padre).

7. Finalmente copy_process() termina devolviendo un puntero a la task_struct del hijo.



## **Copy-on-Write** [Love pag 31]

Con esta técnica, en la creación de un nuevo proceso, al hijo no se le asigna nuevo espacio de memoria sino que las páginas del padre resultan ahora compartidas por ambos, padre e hijo.

Las páginas son marcadas de tal forma que si un proceso intenta escribir se producirá una excepción por "violación de protección";

> Linux resuelve esta excepción de una forma particular:
> hace una copia de la página para el proceso que generó la excepción en un nuevo marco de página
> y así cada proceso tiene una página privada

Si finalmente ninguno de los dos procesos escribe en la página, se ha ahorrado un marco de página

Si finalmente la página va a ser modificada, con esta técnica se posterga lo más posible la asignación de un nuevo marco de página

## **Terminación de procesos** [Love Pág. 36]

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()`***

esto puede ser **explícito** si el programador incluyó esa llamada en el código del programa, o bien la equivalente sentencia `return` dentro de `main`

o **implícito**, pues el compilador incluye automáticamente una llamada a `exit()` cuando `main` termina.

2) **recibe una señal** ante la que tiene la acción preestablecida de terminar

Independientemente de qué acontecimiento ha provocado el fin del proceso, el grueso del trabajo lo hace la función **do_exit()** definida en <linux/kernel/exit.c>



### Actuación de do_exit()

1. Establece el flag PF_EXITING de task_struct

2. Para cada recurso que esté utilizando el proceso (por ejemplo espacio de direcciones, archivos abiertos, ...), **se decrementa el contador correspondiente que indica el nº de procesos que lo están utilizando**;

si este contador vale 0 se realiza la operación de destrucción oportuna sobre el recurso, por ejemplo si fuera una zona de memoria, se liberaría.

por ejemplo, se libera el campo mm_struct de este proceso; si no hay otros procesos que estén usando este espacio de direcciones el kernel lo destruya.

3. El valor que se pasa como argumento a exit() se almacena en el campo exit_code de task_struct; Hay que almacenarlo por si el padre quisiera obtenerlo para tener información sobre cómo ha terminado el hijo.

4. Se envía al padre la señal `SIGCHLD` indicando la finalización de un hijo.

5. **Si aún tiene hijos**, se pone como padre de éstos al proceso init
(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

Puesto que este es el último código que ejecuta un proceso, do_exit nunca retorna.



# 3. PLANIFICACION EN LINUX

El algoritmo de planificación es por prioridades y apropiativo (con desplazamiento)

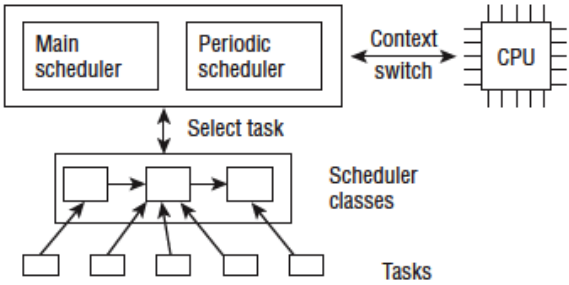
## 3.1 Una visión global de la planificación [Mauerer 2.5.2] [Love pág.46].

El planificador de Linux es modular, permitiendo que diferentes algoritmos de planificación se apliquen a diferentes tipos de procesos.

Esto se implementa con el concepto de clases de planificación.

El kernel dispone de las siguiente **clases de planificación**, cada una de las cuales tendrá asociado un conjunto de procesos: (de mayor a menor prioridad)

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



(Mauerer Fig. 2.13)

Cada clase de planificación tiene una prioridad;

el planificador principal va recorriendo las distintas clases en orden de mayor a menor prioridad, encontrando la primera que no está vacía, y ...

el método de planificación asociado determina el siguiente proceso a ejecutar.



En la figura anterior se muestra ...

una **parte del planificador que se activa periódicamente**,

el **planificador principal** (función `schedule` que se verá más tarde)
Elije el siguiente proceso a ejecutar y provocará un cambio de contexto;

Se activa cuando el proceso actual no desee seguir ejecutándose bien porque se bloquea o finaliza,

o por la llegada de un nuevo proceso ejecutable (creación o desbloqueo de uno preexistente) que resulta tener una mayor prioridad que el actual.

## 3.2 Estructuras de datos [Mauerer 2.5.2]

## a) Elementos en la task_struct para la planificación

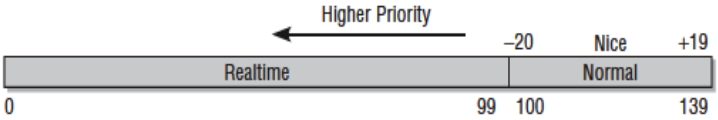
```
int prio, static_prio, normal_prio;
```

static_prio es la prioridad estática o nominal de un proceso.
Se asigna al proceso cuando es creado; es modificado por la llamada a `nice`.

Rango de valores:

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

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




normal_prio y prio son las prioridades dinámicas del proceso (se calculan a partir de static_prio y de la política de planificación del proceso, no se detallará más)



```
const struct sched_class *sched_class;
```

sched_class es la clase de planificación a la que pertenece el proceso

```
unsigned int policy;
```

policy es la política de planificación que se aplica al proceso. Valores posibles:

Políticas manejadas por el planificador de tiempo real....
SCHED_RR y SCHED_FIFO: métodos Round-Robin y FIFO respectivamente

Políticas manejadas por el planificador CFS....

SCHED_NORMAL: se aplica a los proceso normales en los que nos centramos (frente a los de tiempo real)

SCHED_BACH: tareas menos importantes (típicamente con gran proporción de uso de CPU para cálculos).

Los procesos de este tipo son considerados menos importantes por el planificador: nunca pueden desplazar a otros procesos

SCHED_IDLE: tareas de este tipo tienen un peso mínimo para ser elegidas para asignación de CPU



## b) Estructura `thread_info`

Contiene datos sobre los procesos dependientes de la arquitectura; mencionamos a continuación los relevantes para la planificación:

```
struct thread_info {
        struct task_struct *task; /* puntero a la task_struct del proceso */
        .....
        unsigned long flags; /* low level flags */
        __u32 cpu; /* CPU en que el proceso se está ekecitando */
        int preempt_count;
        .....}
```

flags: contiene varios flags específicos del proceso, por ejemplo:

flags: contiene varios flags específicos del proceso, por ejemplo:

> TIF_SIGPENDING está establecido si el proceso tiene señales pendientes
>
> TIF_NEED_RESCHED está establecido si se debe activar al planificador principal para que elija el proceso que debe ejecutarse

preempt_count: contador para implementar la apropiación en modo kernel (se verá más tarde)



## 3.3 El planificador periódico [Mauerer 2.5.4]

Función llamada automáticamente por el kernel con una frecuencia en el rango entre 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

Si se concluye que hay que replanificar,

> el planificador de la clase de planificador en cuestión activará el flag TIF_NEED_RESCHED asociado al proceso en su thread_info para expresar la necesidad de tomar una nueva decisión sobre replanificación,
>
> y el kernel la satisfará en el siguiente momento en que sea oportuno.

## 3.4 El planificador principal [Mauerer 2.5.4]

Se implementa en la función `schedule`, invocada en diversos puntos del kernel para tomar decisiones sobre asignación de la CPU.
Antes de afinar en su actuación hemos de reflexionar sobre los siguientes puntos

### Expropiación (Apropiación) [Love Págs. 62-64]

La expropiación se puede entender como el hecho de que el SO retira la asignación de la CPU al proceso actual aunque pudiera seguir ejecutándose cuando decide que hay otro proceso preferente.

En un **kernel expropiativo (o apropiativo)**, podría quitarse el control de CPU a un proceso mientras está ejecutando código kernel ,
es decir se puede expulsar a un proceso en cualquier punto de su ejecución, sea o no código kernel.



En un **kernel no expropiativo (no apropiativo)**, NO podría quitarse el control de CPU a un proceso mientras está ejecutando código kernel ,
es decir NO se puede expulsar a un proceso en cualquier punto de su ejecución, en concreto mientras el proceso está ejecutando código kernel; éste código se ejecuta entero hasta que termine

**Definiciones**:

**expropiación en modo privilegiado**: acción de retirar la CPU a una tarea que está ejecutando código kernel

**expropiación en modo usuario**: acción de retirar la CPU a una tarea que está ejecutando código usuario

## Sobre la expropiación en modo privilegiado:

Se podrá realizar si el kernel se encuentre en un estado seguro, concepto que en general se define así:

> el kernel está en un estado seguro si todas las estructuras del kernel están
> - o bien actualizadas y consistentes
> - o bien bloqueadas mediante algún mecanismo de sincronización.

El kernel Linux utiliza "locks" para delimitar las zonas de código del kernel en que se está en un estado seguro:

> si una tarea tiene bajo su control algún lock entonces el kernel no está en un estado seguro



Se utiliza un contador llamado **preempt_count** en la estructura thread_info; inicialmente vale 0

se va incrementando a medida que la tarea se hace con un lock

se decrementa cuando lo libera

El kernel Linux puede llevar a cabo la expropiación de la CPU de una tarea si ésta no tiene bajo su control ningún lock,

así se satisfarán los requisitos de reentrancia si se pasara el control ahora a otra tarea que también en un futuro pasara a ejecutar código kernel.

Se consigue así que el kernel de linux sea SMP-safe.

**La expropiación en modo privilegiado puede ocurrir...**

**(a) cuando se retorna de modo privilegiado a modo privilegiado:**
se evalúan el flag TIF_NEED_RESCHED y el contador preempt_count

Si el flag TIF_NEED_RESCHED está activo:

si preempt_count == 0 entonces el planificador será invocado.

si preempt_count != 0 entonces la tarea que está ejecutándose tiene en su poder algún lock y por tanto, no es seguro planificar otra tarea.

En este caso se deja la misma tarea que estaba ejecutándose.

En un momento futuro, cuando la tarea actual libere todos los locks y el contador retorna 0, se verificará el flag TIF_NEED_RESCHED y, si está activo, el planificador principal será invocado.



**(b) de forma explícita, dentro del código del kernel hay una llamada explícita a la función `schedule`**

No necesita codificación adicional para asegurar o comprobar en que estado se encuentra el kernel.

Se asume que en el código en que se invoca al `schedule` r se sabe si el estado es seguro, es decir, que no se tiene retenido ningún lock.

**(c) cuando el código del kernel vuelve a ser expropiativo**
**(es decir cuando se cumple nº de loks == 0)**

## Sobre la expropiación en modo usuario:

La apropiación en modo usuario se da cuando el kernel retorna a modo usuario y se encentra activado el flag TIF_NEED_RESCHED; entonces se llama al planificador y se elige para ejecutarse a un proceso distinto al actual.

Puesto que se está volviendo a modo usuario, todas las actualizaciones del kernel se han terminado y es correcto tanto pasar la CPU a otro proceso como al proceso actual.



**Resumen: cuándo se invoca a `schedule`**

- La función `schedule` es invocada de forma explícita cuando un proceso se bloquea o termina.

- al volver al espacio de usuario desde modo kernel:
  el kernel chequea el flag TIF_NEED_RESCHED;
  si está establecido llama a `schedule` .

- al volver a modo kernel desde modo kernel:
  Si el flag TIF_NEED_RESCHED está activo:
  si preempt_count == 0 el planificador será invocado.
  si preempt_count != 0 se deja la misma tarea que estaba ejecutándose.

- cuando el código del kernel vuelve a ser expropiativo (nº de loks == 0)

**Actuación de `schedule` :**

Actualiza estadísticas y limpia el flag TIF_NEED_RESCHED

Si el proceso actual estaba en un estado TASK_INTERRUPTIBLE y ha recibido la señal que esperaba, se establece su estado a TASK_RUNNING

Recorre las clases de planificación de mayor a menor prioridad, seleccionando la clase de planificación de más alta prioridad que no esté vacía. Llama al procedimiento particular de dicha clase de planificación encargado de seleccionar el siguiente proceso a ejecutar.

Si hay cambio en la asignación de CPU, se realiza el cambio de contexto llamando a `context_switch`

Resumen: **Cuándo se activa el flag TIF_NEED_RESCHED**
en el planificador periódico (como se mencionó anteriormente)
en la creación o desbloqueo de un proceso de más alta prioridad que el actual



## 3.5 Cambio de contexto [Love Pág. 62]

Función `context_switch`: interfaz con los mecanismos de bajo nivel dependientes de la arquitectura que se deben realizar cuando hay un cambio en la asignación de CPU.

Básicamente, `context_switch` realiza estas dos acciones:

Lama a switch_mm (declarada en <asm/mmu_contexto.h>) donde se realiza el cambio del mapa de memoria del proceso previo al proceso nuevo.

Llama a switch_to (declarada en <asm/system.h>) que realiza el cambio del estado del procesador correspondiente al proceso previo por el correspondiente al proceso nuevo.

Esto involucra salvar y restaurar la información de pila, los registros del procesador y demás información de estado dependiente del procesador que sea particular de cada proceso.

# 3.6 La clase de planificación CFS (Completly Fair Scheduler) [Love pag48-50] [Mauerer 2.6]

## Conceptos básicos de CFS

Objetivo básico de CFS o "Completly Fair Scheduler" si tuviésemos únicamente n procesos de la misma prioridad:

cada proceso debe recibir 1/n del tiempo de CPU

Una posibilidad sería implementar una política Round Robin en que el valor del quantum no es fijo sino que se calcula en función del número total de procesos ejecutables.

**Latencia:** menor periodo de tiempo en que se asegura que todos los procesos han sido elegidos para ejecutarse.

A menor valor de latencia se consigue un mejor reparto del tiempo de cpu entre los procesos pero un mayor coste en tiempo de cpu para cambios de contexto.



**Granularidad mínima**

Si el número de procesos tiende a infinito la proporción de tiempo asignada a cada proceso tiende a cero con la consiguiente elevación excesiva del coste en cambios de contexto.

Se introduce el concepto de **granularidad mínima** (en Linux 2.6 por omisión 1ms):

Es la cantidad de tiempo que se asegura que disfruta un proceso la CPU de forma consecutiva aunque el número de procesos tendiera a infinito
(habría algún momento de tiempo en que no se proporcionaría reparto equitativo).

Si contemplamos ahora procesos con distintas prioridades, CFS repartiría el tiempo de cpu de modo que todos los procesos de la misma prioridad tengan igual proporción de uso de CPU.

# Implementación de CFS en Linux 2.6.24

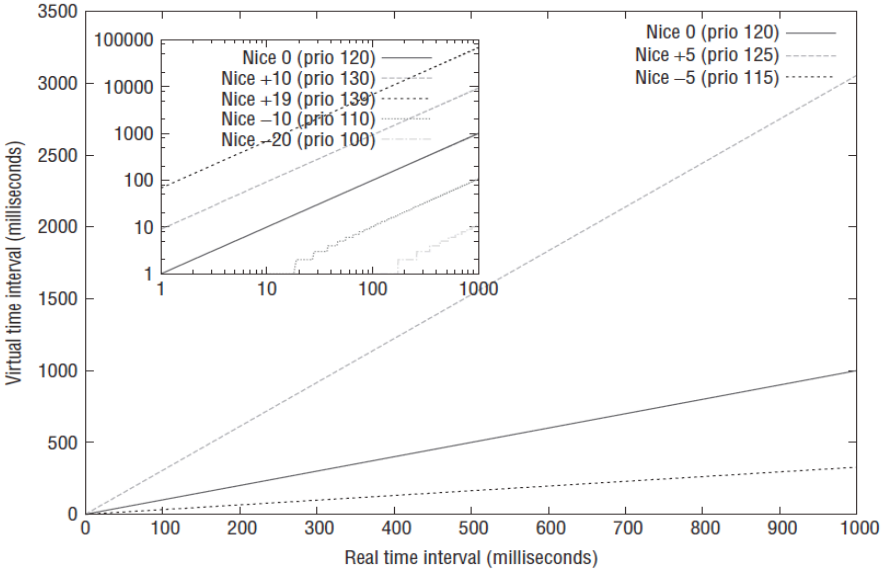
El valor vruntime (virtual runtime) de un proceso almacena el así llamado *tiempo virtual* que un proceso ha consumido:

**se calcula a partir del tiempo real que el proceso ha hecho uso de la CPU y de su prioridad estática:**

Cuando se decide qué proceso ejecutar a continuación, se elige el que tenga un valor menor de vruntime.

El valor vruntime del proceso actual se actualiza ...
periódicamente
cuando llega un nuevo proceso ejecutable
cuando el proceso actual se bloquea






[Mauerer Fig. 2.18]

## 3.7 La clase de planificación de tiempo real [Love Págs. 64-65] y [Mauerer 2.7]

Los procesos de tiempo real tienen son más prioritarios que los normales, y mientras existan procesos de tiempo real ejecutables éstos serán elegidos frente a los normales.

* Linux 2.6 tiene dos políticas de planificación de tiempo real:
  Roun-Robin (SCHED_RR): algoritmo de planificación Round Robin. Las tareas que siguen este tipo de planificación hacen uso de la CPU durante un intervalo de tiempo predeterminado. Una tarea con mayor prioridad siempre pasa por delante de una tarea menos prioritaria.

  FIFO (SCHED_FIFO): algoritmo FIFO; se ejecuta hasta que se bloquea o termina.
    Una tarea podría ejecutarse de forma indefinida.
    Puede ser expulsada de la CPU por otra tarea de mayor prioridad que siga una de las dos políticas de planificación de tiempo real.



* 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.

**Planificación en Tiempo real ligero (soft Real-Time) en Linux.**

Las políticas de planificación de tiempo real SCHED_RR Y SCHED_FIFO posibilitan que el kernel Linux pueda tener un comportamiento soft real-time.

En este tipo de comportamiento el kernel trata de planificar diferentes tareas dentro de un rango temporal determinado, pero no se garantiza la consecución de la planificación dentro del rango temporal, sino que simplemente se intenta.

## 3.9 Particularidades en SMP [Mauerer 2.8.1]

[Ver Introducción a SMP en el libro de Stallings apartado 4.2]

Aunque las estructuras de datos que se han visto están pensadas para adecuarse a un entorno SMP (Symmetric MultiProcessing o multiprocesamiento simétrico), será necesario incluir nuevos campos y nuevas estructuras de datos al pasar a SMP.

Para realizar correctamente la planificación en un entorno SMP, el kernel deberá tener en cuenta estas consideraciones adicionales:

- 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 comprobará que se da un equilibrio entre las cargas de trabajo de las distintas CPU. Si se detecta que una tiene más procesos que otra, se reequilibran pasando procesos de una CPU a otra.