Concepto y definición de proceso
Descripción de procesos
- Imagen de un proceso

Estados de proceso
- Modelo de 3 estados
- Modelo de 5 estados
- Modelo con estado suspendido

Gestión de procesos
- Modo de ejecución
- Creación de procesos
- Terminación de procesos
- Cambio de contexto

Threads
- Tipos de Threads
- Ventajas

Procesos y threads en Linux: POSIX API
- Daemons
- POSIX API
- Gestión de señales
- Procesos y threads en linux
- PCB Linux

Planificación de procesos
- Prioridad
- Planificadores de procesos
  - Clasificación
  - CPU Scheduler (planificador corto plazo)
  - Criterios cuantitativos de planificación
  - Algoritmos de planificación
    - Preemption: Expulsión de proceso
    - FCFS (First Come First Serve)
    - Round Robin
    - SJF (Shortest Job First)
    - SRTF (Shortest Remaining Time First)
    - No expropiativa por prioridad
    - Expropiativa por prioridad
    - Cola multinivel
    - Cola multinivel con realimentación
- Planificación en Linux

Preguntas teoría

Concepto y definición de proceso

A la hora de ejecutar una instancia de un determinado programa, el SO (cargador) se encarga de buscar un espacio en memoria y reservarlo para este programa, así cómo instanciar el PC para que apunte a la primera línea del código del programa. Con esto se crea un **proceso**

Un **proceso** (Task/Job) es un programa en ejecución que es susceptible de ser planificado por un SO. De esta forma, solventamos el problema de la concurrencia. Muchos procesos comparten un procesador, una memoria y unos dispositivos de E/S.

- Simplicidad/modularidad: syscalls
- Velocidad/Eficiencia: Organizar los programas en procesos nos permite exprimir al máximo la concurrencia
- Seguridad/Privacidad: Separación de los hijos de ejecución

Descripción de procesos

El SO gestiona el uso de los recursos del sistema por parte de los procesos. Necesita almacenar información sobre el estado actual de cada proceso y los recursos que utiliza. Para ello, el SO emplea unas tablas con información sobre el estado del sistema con referencias cruzadas:

- Procesos: Puntero a la imagen del proceso
- Memoria: Memoria física y virtual
- Dispositivos E/S: Estado y disponibilidad
- Ficheros: Estados, atributos

Imagen de un proceso

Contiene el código, datos e información de estado del proceso (PCB) en un momento dado de su ejecución. Son gestionados por la memoria virtual del SO. Partes de la imagen:

- PCB (Bloque de control de proceso):
  - PID: Identificador de proceso
  - UID: Identificador de usuario
  - Contexto: Registros en uso de la CPU
  - Estado: Running, blocked, prioridad,...
  - Puntero a tablas de memoria, tablas de E/S, tablas de ficheros,...
- Stack
- Heap
- Data
- Text

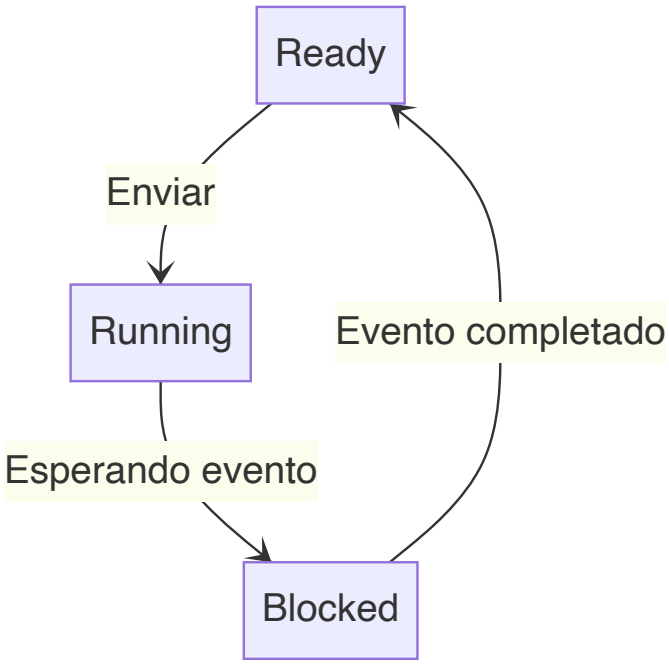
Estados de proceso

Para llevar un control de todos los procesos en ejecución actualmente en la CPU, el SO utiliza la cola de procesos. En función del estado de cada proceso, entrarán o saldrán de la cola y se les asignará una prioridad.

Modelo de 3 estados

- Running: En ejecución, uno por cada hilo
- Blocked: Esperan la recepción del evento que quieren
- Ready: Memoria cargada y el proceso está listo para ser ejecutado, por lo que se añade a la cola de

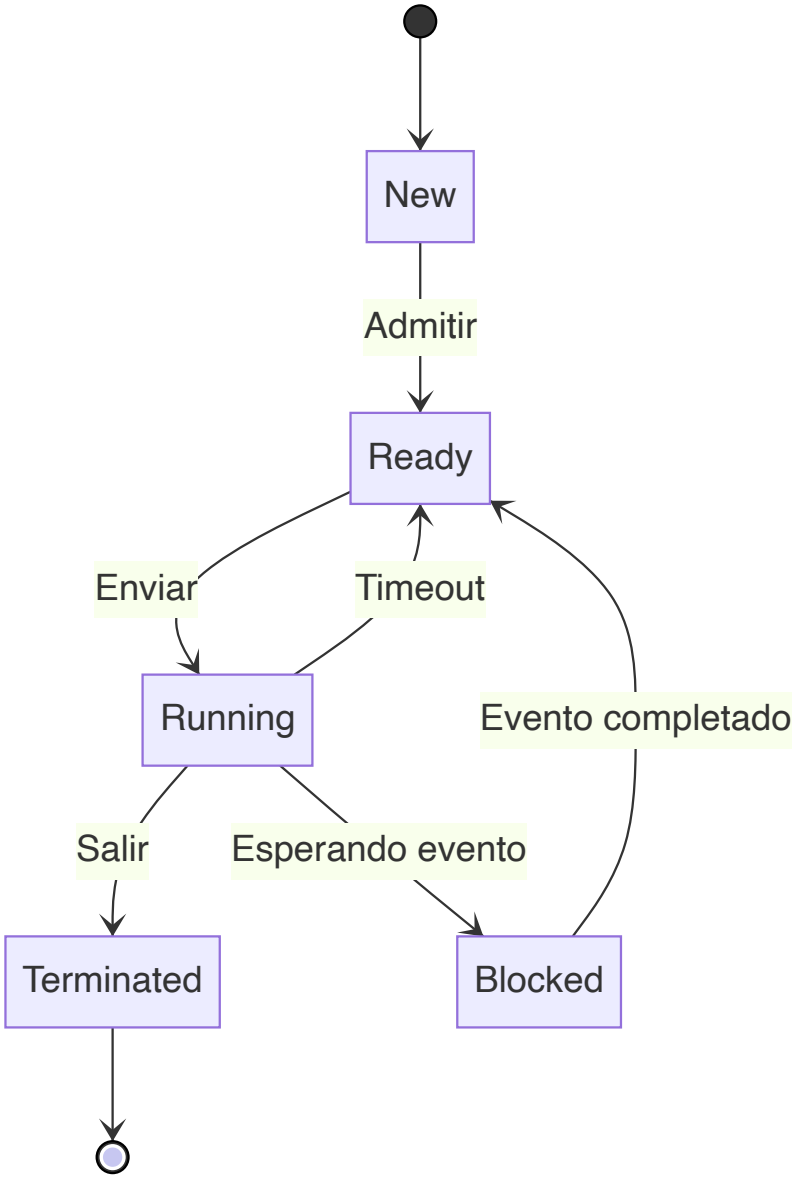
procesos



Modelo de 5 estados

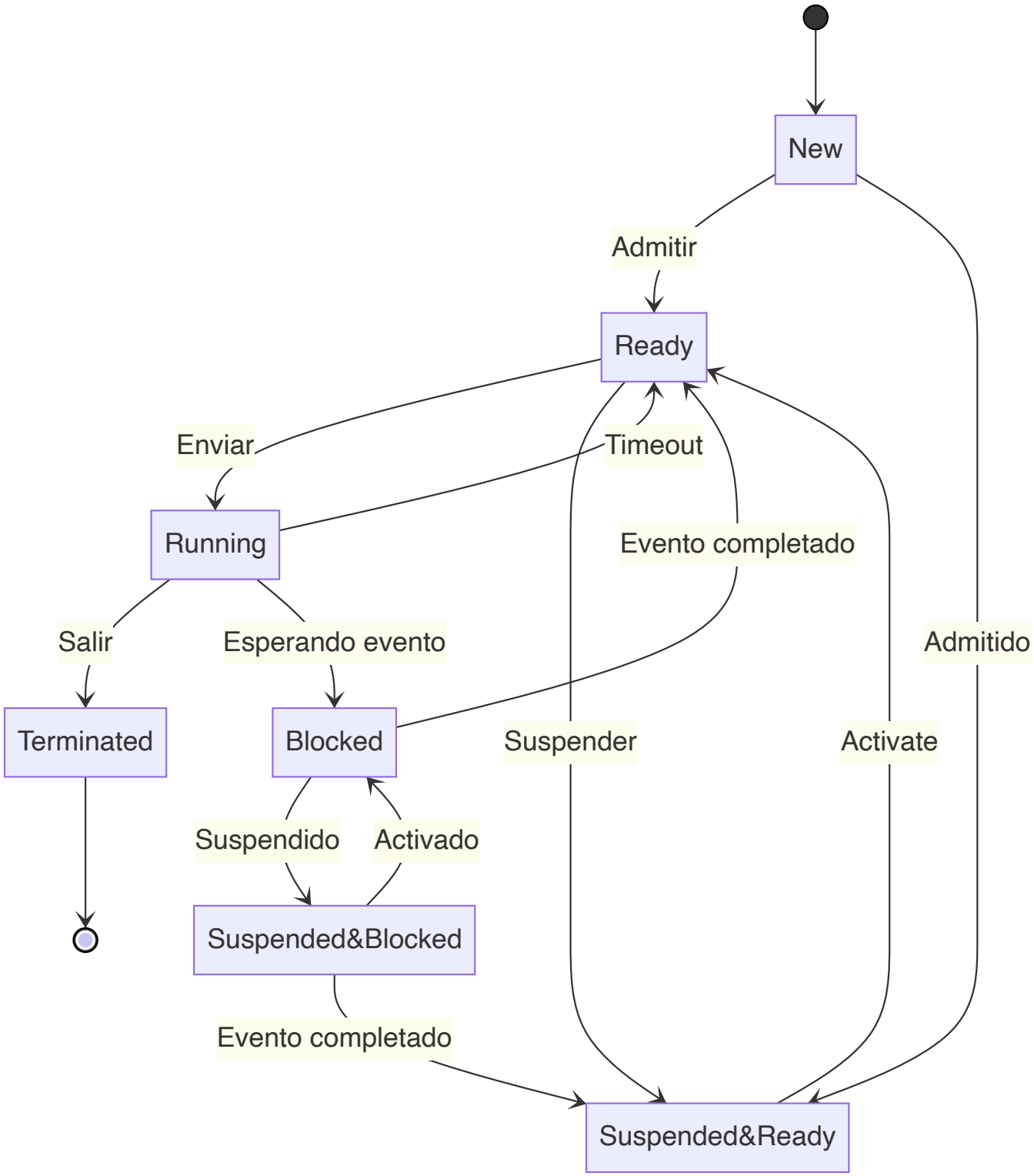
Para aumentar la eficiencia utilizamos varias colas donde clasificar los procesos en espera, de tal forma que si un evento es de tipo *mouse* solo tendríamos que ir a la cola donde el SO almacena los procesos relacionados con el evento *mouse*

- New: Procesos nuevos no cargados en memoria. Se reserva su PCB pero no se carga la imagen todavía. Es un estado transitorio que aparece en ciertos SO de tiempo real con límite de procesos
- Terminated: Procesos que terminaron su ejecución por alguna razón. Tras realizar contabilidad (core dump) son eliminados de la lista de procesos. Si un proceso padre no reclama la salida de su hijo puede permanecer en estado Zombie



Modelo con estado suspendido

Puesto que las interrupciones E/S suelen ser mucho más lentas que la CPU, conservar el proceso en la memoria puede que no sea lo más eficiente. Esos recursos podrían ser usados por otros programas. Para ello, el estado suspendido lo que hace es guardar la memoria en el disco (Swap) y suspender el proceso. El uso de Swap permite aumentar el grado de multiprogramación del SO



Gestión de procesos

Modo de ejecución

- Modo usuario: Menos privilegios, subconjunto de instrucciones máquina, registros inaccesibles, zonas del mapa de memoria prohibidas
- Modo kernel: Modificar registro de estado, zona de memoria de E/S, timer, instrucciones privilegiadas,...

Creación de procesos

Un proceso puede crear otros procesos y mantener una relación de parentesco:

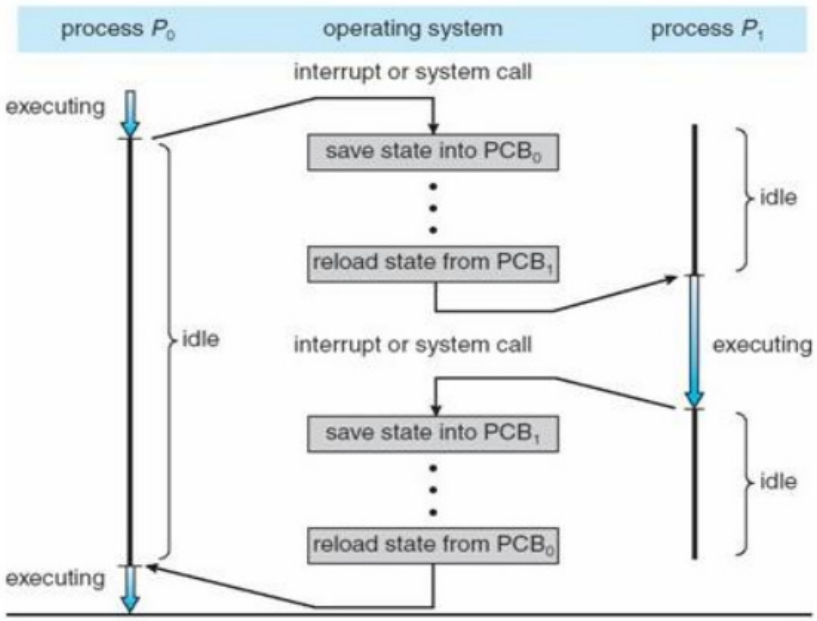
- Sus espacios de direcciones pueden ser **clonados** (una copia exacta a la del padre) o **diferentes** (un programa distinto al padre)
- El padre y el hijo se ejecutan de forma concurrente. El padre espera al hijo
- El cargador del SO crea una nueva entrada en la tabla de procesos, reserva memoria y asigna un identificador para el hijo

Terminación de procesos

- Llamada al sistema para salir (`exit()`): Llama al padre y le pasa su valor de retorno
- Forzar la ejecución (`abort()`): Exceso de recursos, tarea innecesaria, limpieza del SO (padre termina antes que el hijo)

Cambio de contexto

Un SO multitarea hace que el procesador pase de ejecutar un proceso a ejecutar otro por medio de un *cambio de contexto*. Puede ocurrir con **interrupciones o excepciones**. El SO almacena el estado actual del proceso en la PCB y continua con la ejecución de otro proceso. Este proceso se acelera con hardware o con el uso de *threads* (evitamos hacer el cambio de contexto)

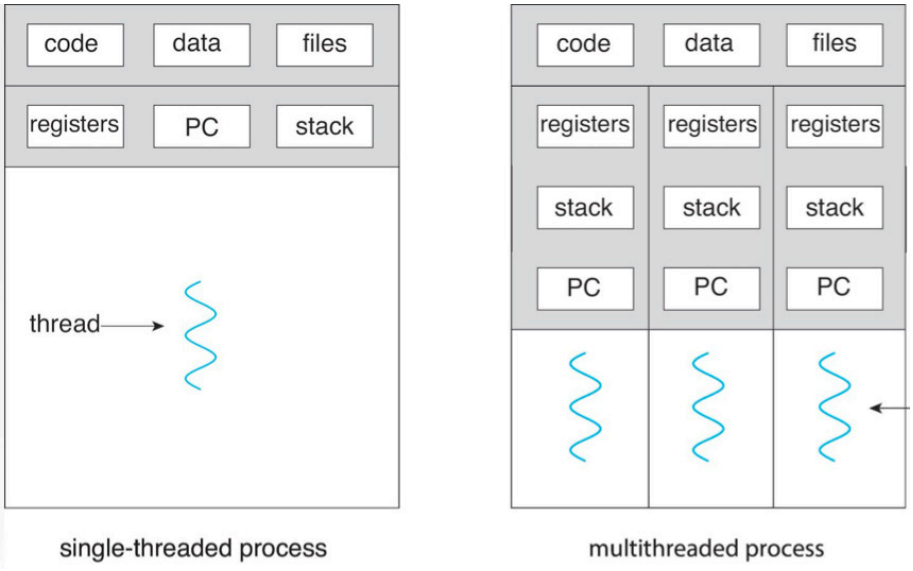


Threads

Un proceso tiene dos aspectos que se pueden separar:

- Planificación: Tiene un estado y un código a ejecutar determinado por su contexto (registros, puntero de pila,...)
- Propiedad de recursos: Asignación de memoria física, dispositivos E/S, ficheros abiertos,...

Si separamos ambas unidades, definimos la propiedad de recursos al *proceso* y la planificación a los *threads*. De esta forma, dentro de un único proceso podemos tener varios hilos de ejecución a la vez



Tipos de Threads

- De usuario: Creados por una librería y no controlados por el SO. Aunque son rápidas, las llamadas al sistema pueden bloquear ambos hilos indiscriminadamente
- De kernel: Creación y gestión más rápida que con procesos pero más lenta que threads de usuario, pero el SO puede planificarlos

Ventajas

- Gran capacidad de respuesta: Un hilo puede ejecutarse indistintamente de otro aunque esté bloqueado (importante para GUIS)
- Se pueden compartir recursos entre hilos de forma sencilla
- Escalabilidad: Puede aprovechar las arquitecturas multicore

Procesos y threads en Linux: POSIX API

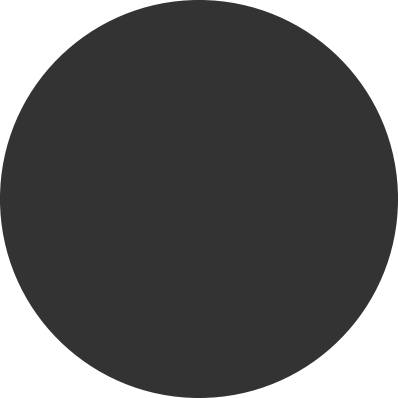
En linux los procesos se mantienen en un árbol (relaciones jerárquicas padre-hijo)

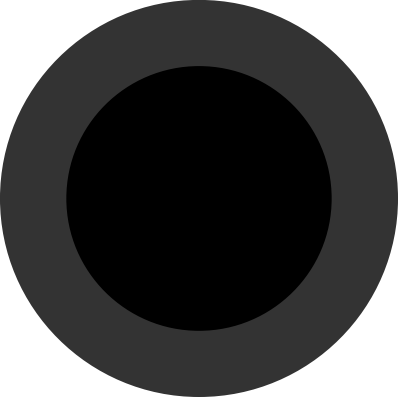
Daemons

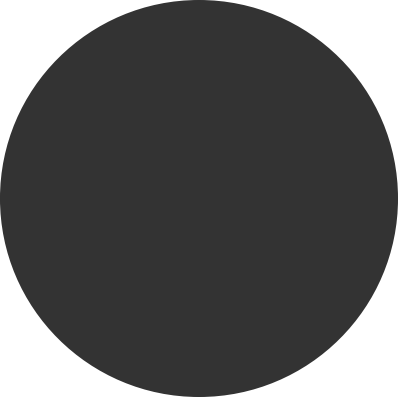
Son procesos especiales del SO que se ejecutan en segundo plano. No son interactivos, y no suelen estar asociados a una terminal. Se activan por tiempo (cron) o por eventos. Se arrancan al iniciar al sistema, no mueren y lanzan otros procesos o threads para realizar tareas

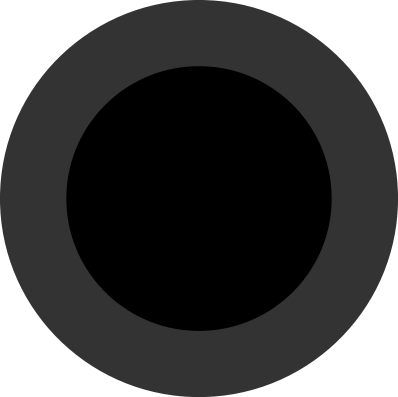
POSIX API

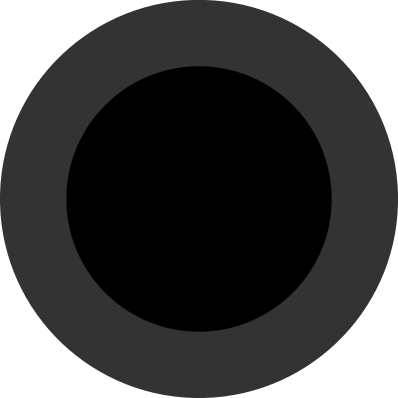
```
1  // Gestión de procesos
2  ////////////////////////////////////////////////
3
4  /**
5   * Duplica el proceso padre. Exactamente iguales y que hacen lo mismo.
```











```
 * returns: El padre recibe la dir del hijo mientras que el hijo
 *  recibe 0. En caso de error -1. Se generan 2^n procesos
 */
pid_t fork (void);

/**
 * El mismo proceso ejecuta otro programa
 * file: nombre del archivo ejecutable
 * argv: argumentos para el ejecutable
 * returns: -1 en caso de error y no retorna en caso contrario
 */
int execvp(const char*file,const char * argv[]);

/**
 * salir del proceso. Liberan los recursos del proceso
 * status: devolver al padre
 */
void exit (int status);

/**
 * Espera al hijo
 */
pid_t wait (int* status);

// Gestión de señales
////////////////////////////////////////////////

/**
 * envía al proceso pid la señal sig
 */
int kill (pid_t pid , int sig);

/**
 * Nos permite establecer el comportamiento de nuestro proceso al recibir una
 * señal determinada
 */
int sigaction(int sig , struct sigaction *act , struct sigaction *oact);
/**
 * detiene el proceso hasta la recepción de una señal
 */
int pause (void);

/**
 * El so enviará una señal SIGALRM pasados esos segundos
 */
unsigned int alarm (unsigned int seconds);
/**
 * Nos permite modificar la máscara de señales de un proceso
 */
```

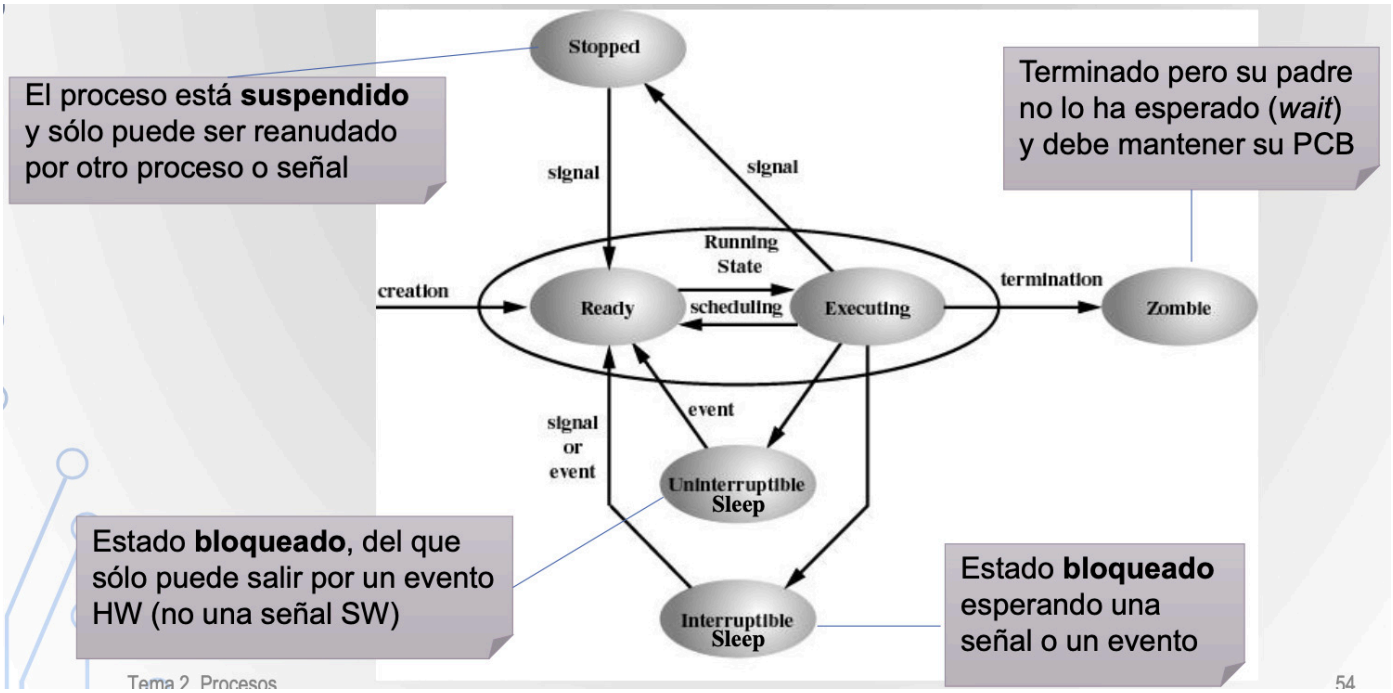
```c
void sigprocmask(int how, const sigset_t *set, sigset_t *oset);

// Syscalls para threads
////////////////////////////////////////////


/**
 * Crea un proceso ligero que ejecuta la función func con
 * argumento arq y atributos attr
 */
int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*func)
(void *), void *arg);

/**
 * Suspende la ejecución de un proceso ligero padre hasta que termina
 * el proceso ligero hijo con identificador thid. Devuelve el estado de
 * terminación del proceso ligero hijo
 */

int pthread_join (pthread_t thid, void **value);

/**
 * Permite a un proceso ligero finalizar su ejecución, indicando el
 * estado de terminación del mismo
 */
int pthread_exit (void *value);

/**
 * Devuelve el identificador del thread que ejecuta la llamada
 */
pthread_t pthread_self (void);
```

Cuando un proceso se queda sin padre, se denomina proceso *Zombie*. systemd adoptará a estos procesos

Gestión de señales

Las señales de interrupciones a un procesos son enviadas o bien de un proceso a otro (syscall kill) o bien del SO a un proceso. Existen varias tipos de señales (`kill -l` muestra todas las señales disponibles). El proceso debe estar preparado para recibir la señal y actuar en consecuencia. En caso de no estar preparado, muere o ignora la señal

Procesos y threads en linux



PCB Linux

La PCB del kernel linux para los procesos se almacena en una estructura de datos de tipo `task_struct`. Se compone de los siguientes elementos:

- Estado: Runnable, stopped, zombie...
- Estado de la CPU: Registros
- Información de planificación
- Identificadores: De procesos, de usuario y de grupo
- IPC: inter process communication
- Vínculos con padre, hijos y hermanos
- Tiempos y temporizadores: Hora de creación del proceso, tiempo de CPU consumido,...
- Puntero a tablas de memoria: Mapa de memoria virtual

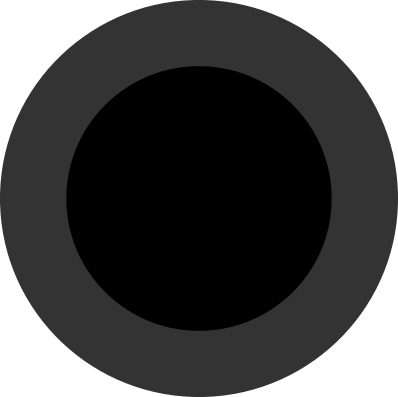
Planificación de procesos

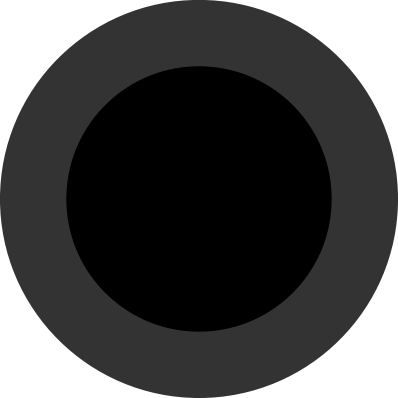
El planificador de la CPU se encarga de maximizar las prestaciones del computador tomando decisiones sobre la gestión de los procesos en ejecución. Se basa en los siguientes criterios:

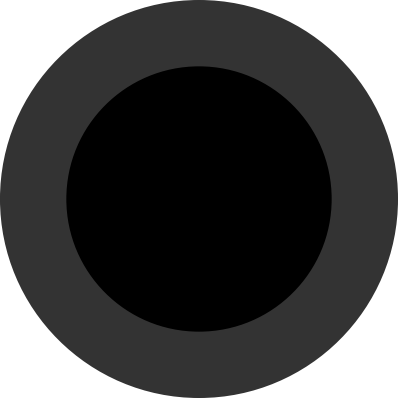
- Maximizar el rendimiento: Mantener la CPU siempre ocupada
- Ser estadísticamente predecible: Estima la duración de un trabajo
- Ser imparcial: No debe de discriminar los procesos por otros
- Aprovechar los recursos disponibles

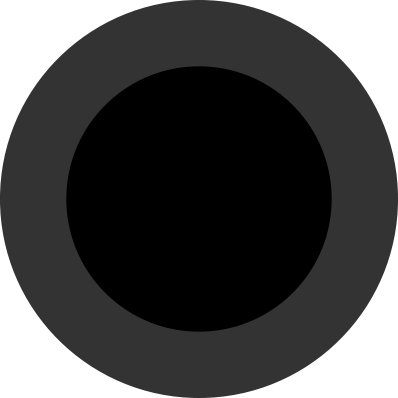
Prioridad

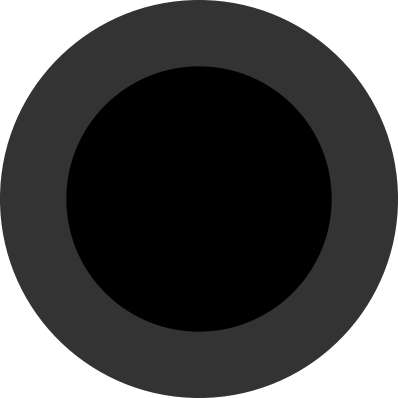
Indica la preferencia de ejecución de un proceso sobre otro. Se indica utilizando un número y se asigna en función de diversos criterios (externos o internos). Las prioridades se pueden establecer de forma estática (fáciles de implementar pero no adaptativas) o dinámicas (los procesos que llevan mucho tiempo esperando aumentan de prioridad)

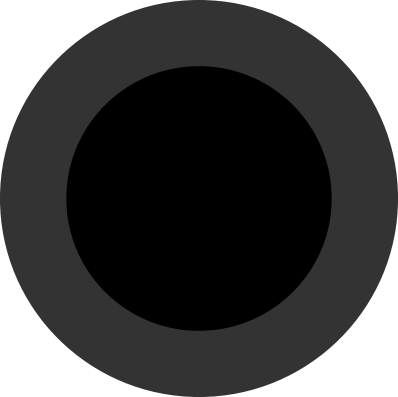


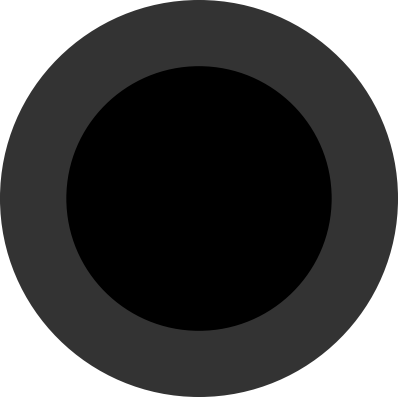


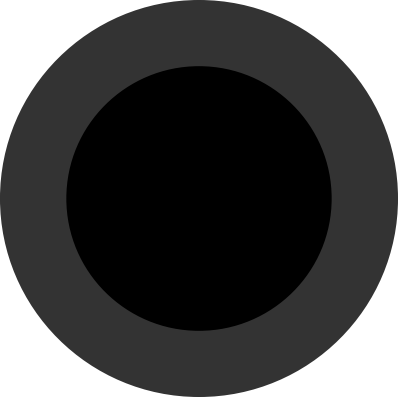


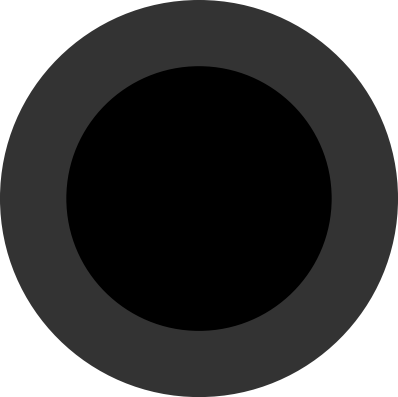


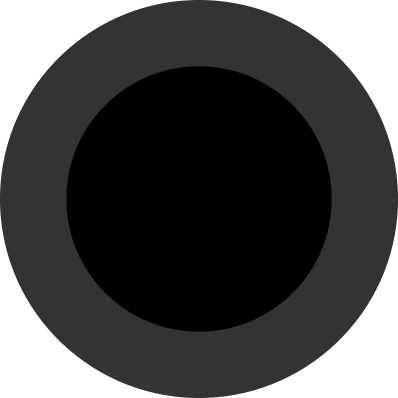


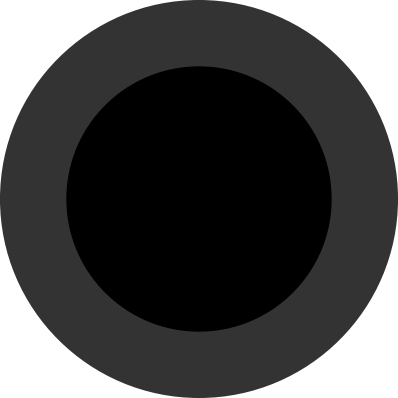


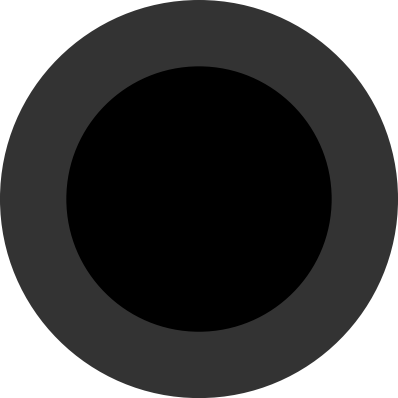












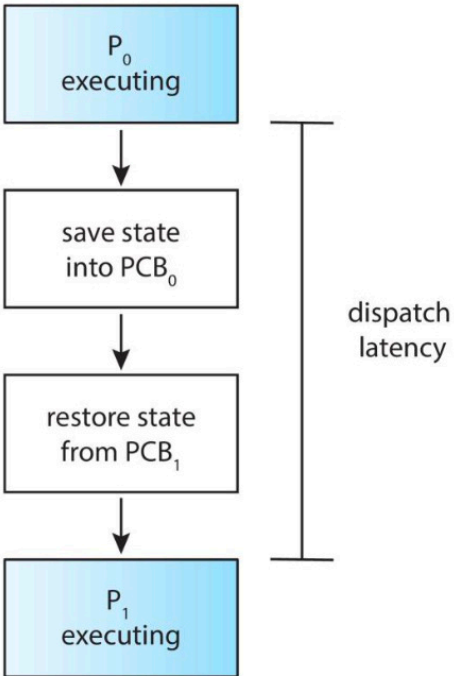
Planificadores de procesos

Clasificación

- Largo plazo: Decide la admisión de procesos en la cola de ready. Es importante en sistemas de tiempo real, clusters de computadores,...
- Medio plazo: Puede controlar la ocupación de la memoria y la carga del sistema. No es relevante en SSOO modernos con memoria virtual
- Corto plazo: Cola de ready. Uso muy frecuente

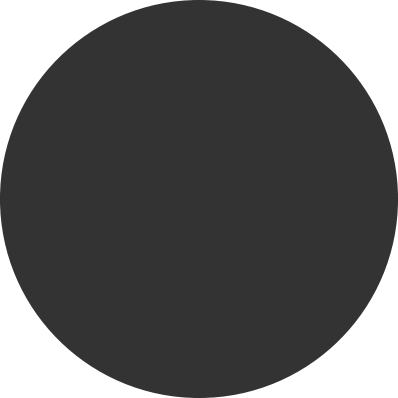
CPU Scheduler (planificador corto plazo)

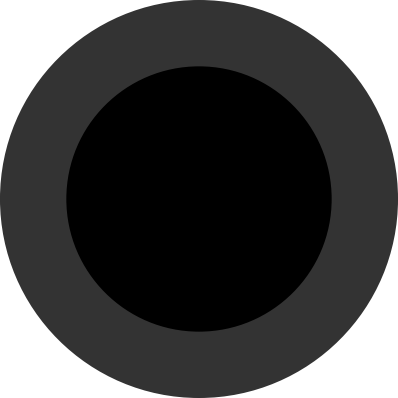
Se encarga de seleccionar que proceso ha de ejecutarse a continuación en la CPU. Es un proceso ligero (no puede acaparar la CPU) y se asegura de que siempre esté trabajando.

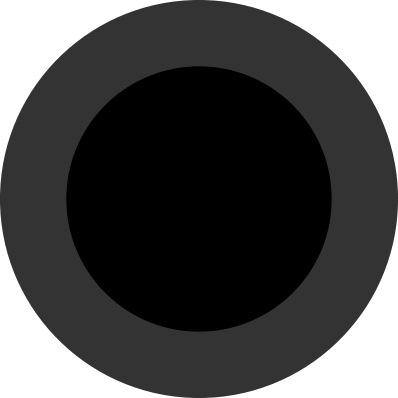


Dispatcher: Un módulo del planificador de corto plazo encargado de hacer el cambio de contexto (dispatch latency)

Criterios cuantitativos de planificación







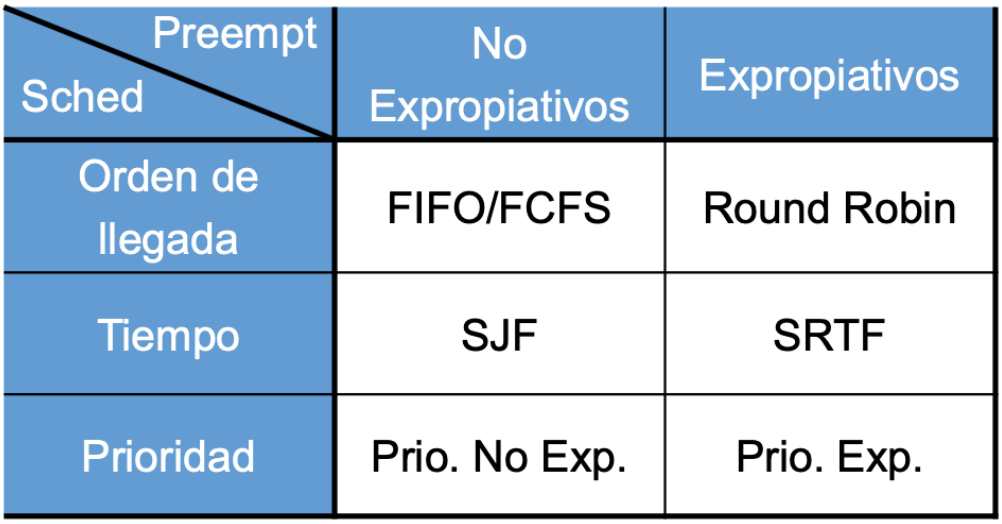
Nombre	Expresión	Descripción
Uso CPU	$0 - 100\%$	Porcentaje tiempo con la CPU ocupada
Productividad (throughput)	n_{proc}/T	Trabajos terminados por unidad de tiempo
Tiempo de retorno	t_r	Tiempo desde que lanzas un programa hasta que termina
Tiempo de espera	$t_e = t_r - t_{running} - t_{blocked}$	Tiempo en el que el proceso está esperando
Tiempo de respuesta		Desde que pides algo al SO hasta que responde

Algoritmos de planificación

En la política de planificación se componen de dos partes: Preemption (políticas de expulsión) y Schedule (orden de la cola de 'ready')

Preemption: Expulsión de proceso

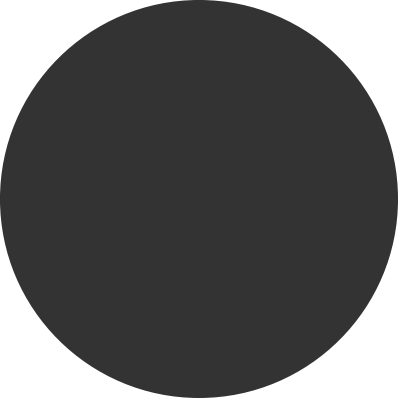
- No expropiativo (multiprogramación): Solo considera expulsar un proceso cuando termina o inicia E/S.
- Expropiativo (multitarea): Considera todas las situaciones posibles, permitiendo que un proceso pueda expropiar a otro menos prioritario. Esto aumenta el número de cambios de contexto

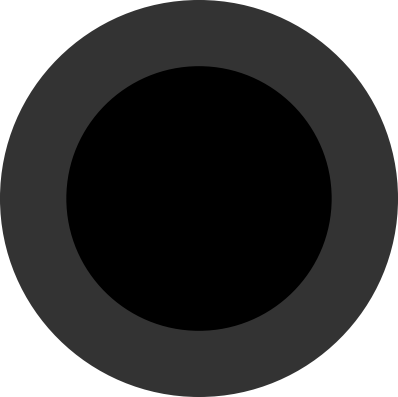


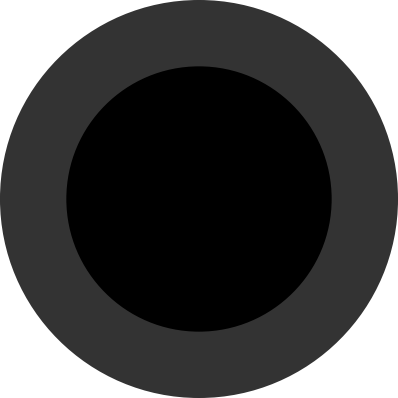
FCFS (First Come First Serve)

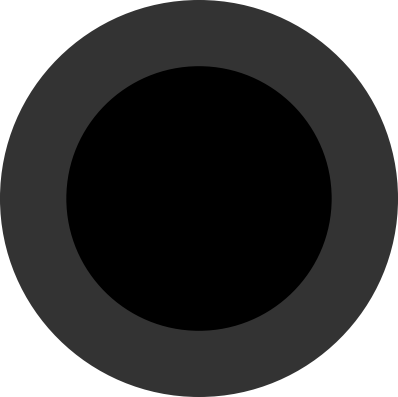
No expropiativo. Ordena la cola usando un algoritmo FIFO

- Ventajas: Sencillo
- Inconvenientes: Elevados tiempos de espera y retorno, bajo índice de ocupación de recursos. Efecto *convoy*. No realiza ningún tipo de optimización









Round Robin

Expropiativo, el proceso se expulsa cuando termina, realiza E/S o por **Quantum**. Ordena la cola usando un algoritmo FIFO y otorga $1/N$ del tiempo de CPU a cada uno de los N procesos de forma circular. El tiempo de CPU para cada proceso se denomina Quantum (Q) y es un indicativo del rendimiento del algoritmo. **Cuanto mayor sea Q, más parecido será a un algoritmo FCFS**. Cuanto menor sea el tiempo de ejecución, la CPU ejecutará demasiados cambios de contextos

$$t_{respuesta} \leq N * Q$$

- Ventaja: aumenta la interactividad
- Inconvenientes: Aumenta el número de cambios de contexto

SJF (Shortest Job First)

No expropiativo. Se planifica por estimación en el tiempo de ráfaga priorizando a aquellos procesos que tienen el tiempo de ejecución más corto

- Ventaja: Reduce los tiempos medios de ejecución y retorno
- Inconveniente: Afecta a los procesos largos y *starvation*

SRTF (Shortest Remaining Time First)

Expropiativo, el proceso se expulsa cuando termina, realiza E/S o por **prioridad**. Se ordena la cola por el mínimo de tiempo de ráfaga. Si llega un proceso a la cola con un tiempo de ráfaga aún menor, se evalúa y se ejecuta el de menor tiempo

- Ventaja: Aumenta la productividad
- Inconvenientes: Discriminación de procesos largos y *starvation*

> Estimación de ráfagas: La función estimación del tiempo de ráfaga puede ser cualquier función de estimación de series temporales donde $t_{n-1} = \alpha * t_n + (1 - \alpha) * t_n$

No expropiativa por prioridad

Se asigna una prioridad (`int`) a cada proceso. Se ordena la cola por orden de prioridad, dejando a los procesos con mayor prioridad a la cabeza. Solo se extraen a los procesos cuando realizan E/S o terminan

- Inconvenientes: Puede producir *starvation*, procesos con baja prioridad puede que no se ejecuten nunca

Expropiativa por prioridad

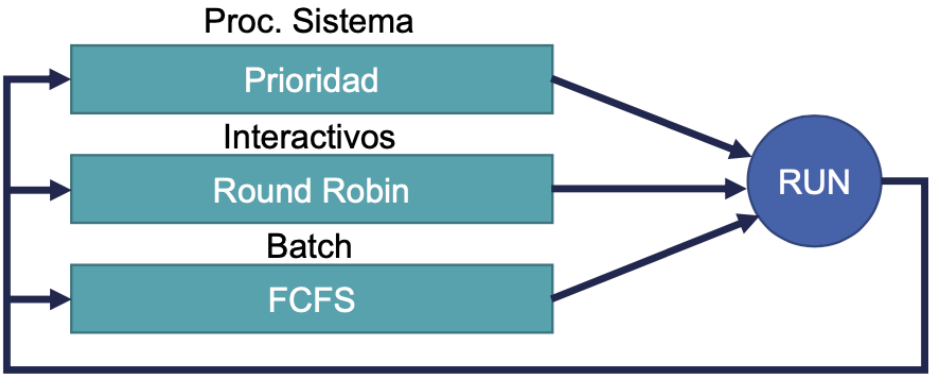
Igual que la no expropiativa por prioridad, solo que si un proceso nuevo llega a la cola, revaluamos e introducimos aquel proceso con mayor prioridad

- Inconvenientes: Puede producir *starvation*, procesos con baja prioridad puede que no se ejecuten nunca

Cola multinivel

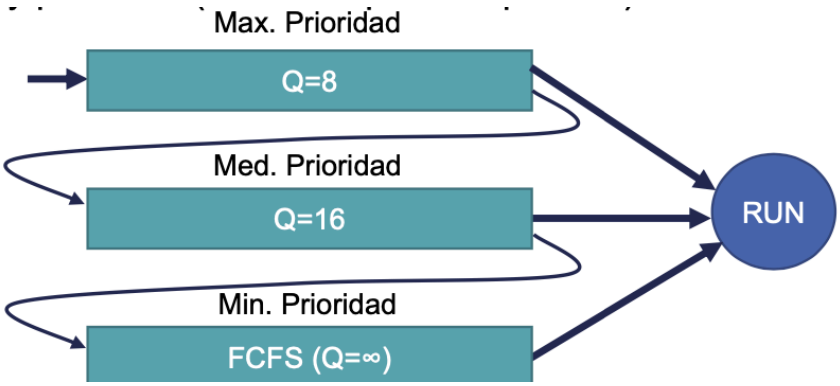
El SO mantiene varias colas dependiendo de que tipo de tarea realicen, de tal forma que clasifica los procesos en colas con distintas características.

- Cada cola se organiza con un algoritmo simple
- Las colas se planifican con 'Expropiativa con prioridad', otorgando la mayor prioridad a la cola del sistema



Cola multinivel con realimentación

Los procesos se clasifican automáticamente. Los procesos entran por la cola de máxima prioridad, si agotan el Quantum de la cola bajan de rango (pasan a una cola con menos prioridad). Si realizan E/S suben de rango (pasan a una cola superior). Su mayor inconveniente es que un proceso que tiene ráfagas de CPU largas puede acabar en la cola de mínima prioridad. Para solucionarlo, podemos añadir más condiciones a la hora de cambiar de capas

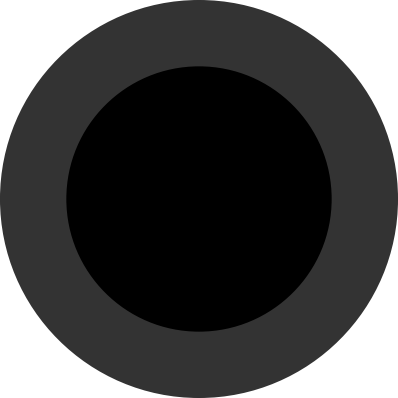


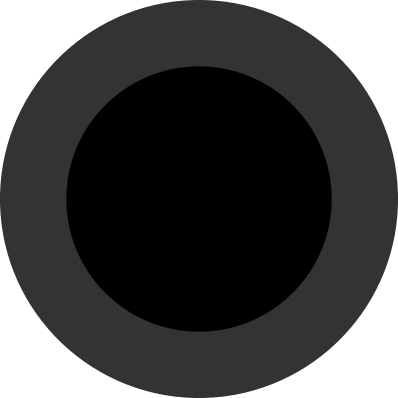
Planificación en Linux

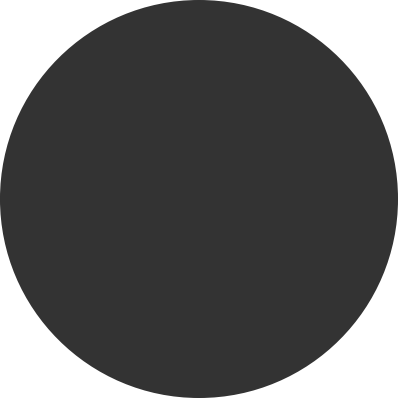
- Anterior a v2.5: Variación de UNIX
- v2.5: Algoritmo O(1) con dos prioridades y dos colas. La primera cola contiene los procesos activos y la segunda los procesos que han agotado su tiempo de ejecución. Cuando termina de ejecutar todos los procesos de la cola de activos, intercambia los punteros de las colas
- v2.6: Completely Fair Scheduler. Mantiene dos clases (por defecto y tiempo real). Mantiene un tiempo de ráfaga virtual y ordena la cola con un árbol rojo-negro

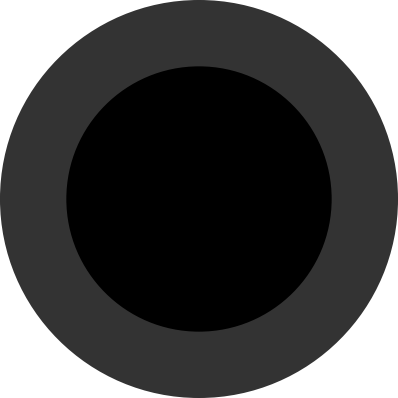
Preguntas teoría

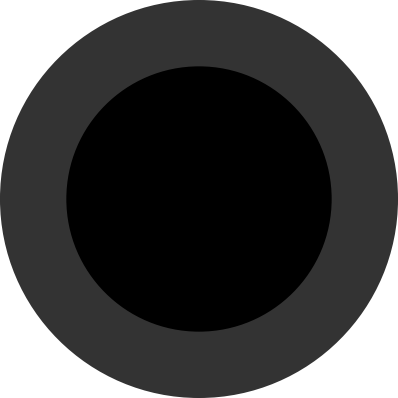
- **Un código binario cargado en memoria**: Puede no ser un proceso, sino una librería o rutina
- **Siempre que se produce un cambio de modo en la CPU (kernel/usuario) se debe de realizar un**

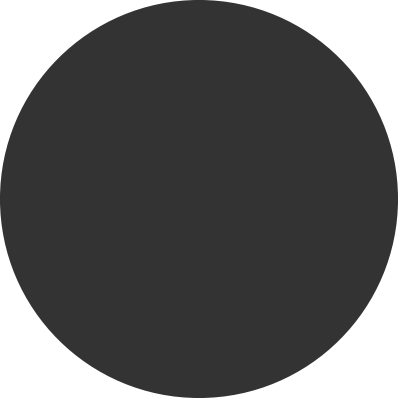


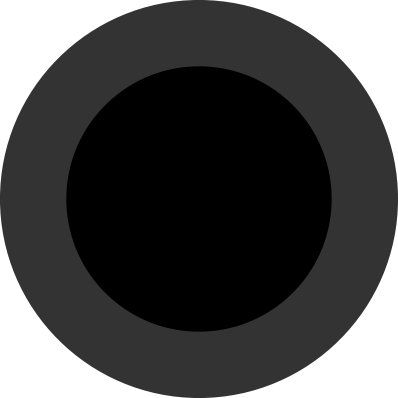












cambio de contexto: Falso

- **Multiprocesamiento Simétrico**: Un SSOO que soporta Multiprocesamiento simétrico puede ejecutarse en cualquier procesador de la máquina
- Si existen muchos procesos ready, el tiempo de respuesta de los procesos aumenta
- El tamaño de la memoria restringe el número de procesos que se pueden tener en ready
- HRRN: Dar prioridad a los procesos que lleven más tiempo esperando
- Tiempo de retorno: Tiempo desde que se lanza un proceso hasta que termina
- Nice UNIX: Prioridad estática asociada a un proceso de tiempo compartido