Scheduling en linux

Objetivos:

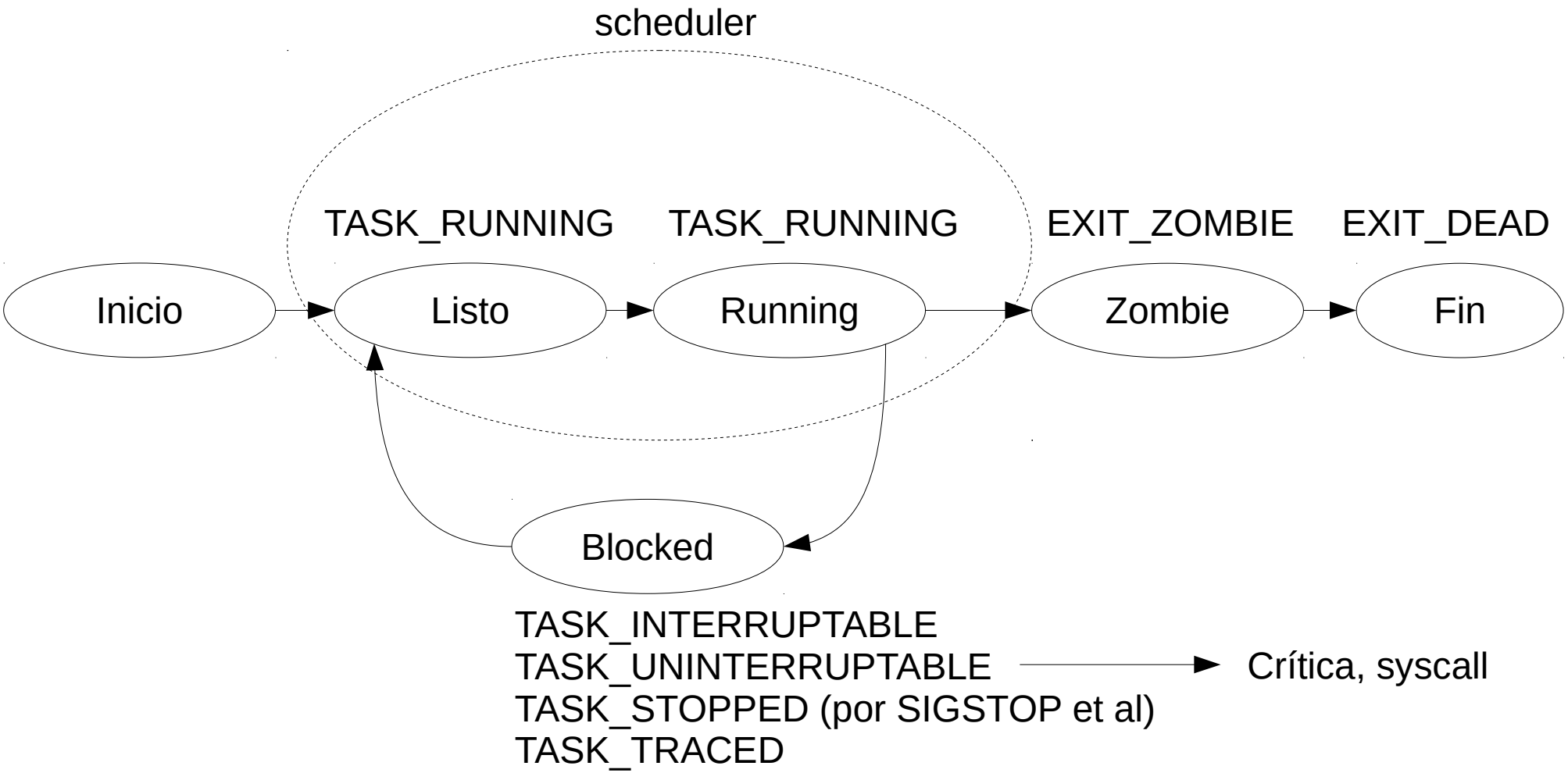
- Debe funcionar (o adaptarse bien) en distintos entornos (servidores, móbiles, escritorio, etc.).
- Optimizar rendimiento sin perjudicar procesos interactivos.

Tipos de procesos

- Interactivos: suelen estar durmiendo (bloqueados) por mucho tiempo esperando un evento generado por el usuario. Una vez recibido el evento, el proceso debe reaccionar lo más pronto posible.
- Batch: CPU intensivos y cuyo resultado puede demorarse (aunque no indefinidamente).

  Ej.: un compilador

Estados


scheduler
TASK_RUNNING
TASK_RUNNING
EXIT_ZOMBIE
EXIT_DEAD
Inicio
Listo
Running
Zombie
Fin
Blocked
TASK_INTERRUPTABLE
TASK_UNINTERRUPTABLE
Crítica, syscall
TASK_STOPPED (por SIGSTOP et al)
TASK_TRACED

Tipos de scheduling

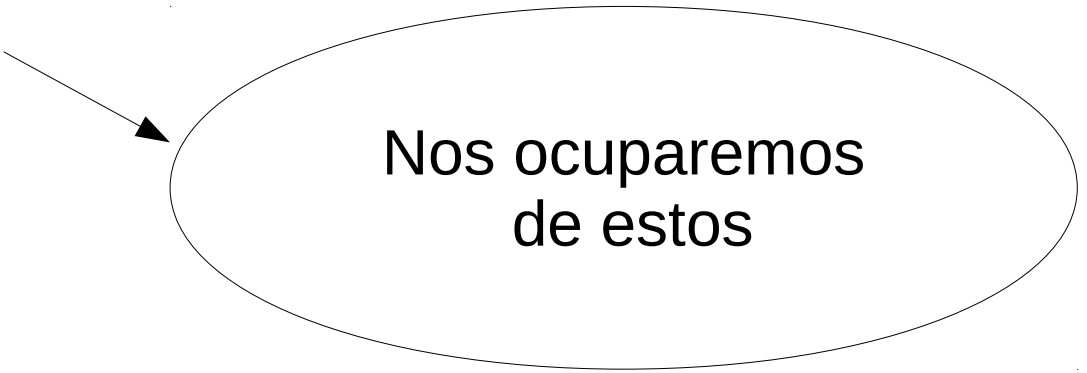
- Para tiempo real (RT):

  - SCHED_FIFO: sólo puede ser reemplazada por otra tarea de mayor prioridad

  - SCHED_RR: usa quantum

- Para procesos normales:

  - SCHED_OTHER



Prioridades

- Cada proceso tiene una prioridad **estática** que se puede fijar con *nice()* y *set_priority()*. Los hijos e hilos heredan esta prioridad.
- Además está la prioridad **dinámica**, utilizada internamente por el planificador. Está basada en la prioridad estática pero puede variar según el tipo del proceso (ver diap. Anterior) y el estado actual del sistema
- Escala:

  0 (mayor prioridad) a 99: procesos RT

  100 a 139 (menor prioridad): procesos normales

Un poco de historia

- Kernel 1.2: basado en RR
- Kernel 2.2: Aparece el concepto de clases: real time/non-preemptible/non-real-time y soporte para SMP
- Kernel 2.4: Planificador basado en épocas
  - Cada tarea tiene un time/slice por cada época
  - Si no usa todo su time/slice se lo benficia en la sig. época
  - Problema: O(N), poco escalable

N: nro. de procesos listos

Scheduler O(1)

- Kernel 2.4: scheduler “O(1)”, desarrollado por Ingo Molnar

  Dos colas (**runqueue**) por cada prioridad **dinámica**: procesos activos y procesos expirados (2 arreglos de 140 colas)

  - Mientras haya procesos activos, quitar uno de la cola de procesos activos de mayor prioridad. Cuando el proceso termine su “time-slice” se lo pasa a la cola correspondiente de expirados.
  - Cuando no haya más proceso activos, intercambiar los activos con los expirados (array switch).

Scheduler O(1)

- Procesos interactivos: un proceso recibe un priority-bonus basado en su “average sleep time” (si muchas veces esperó un evento).
- Procesos batch: se los penaliza (bajando la prioridad)

- Problena: heurística compleja y propensa a fallos. El código se hizo grande y difícil de mantener.

CFS

- Kernel 2.6.23: CFS. Ingo Molnar implementa un nuevo planificador basado en un patch de Con Kolivas que implementaba un Rotating StairCase Deadline Scheduler (RSDL).
- Se lo llama CFS (Completely Fair Scheduler) pues intenta simular lo que ocurriría en una “CPU multitarea ideal y precisa”
- Toma como base un “fair clock” (cfs_rq->fair_clock) calculado en base al tiempo real dividido por el número de procesos.

CFS

- Cada tarea tiene un *wait_runtime* asociado: el tiempo que pasó esperando por la CPU.
  - Se incrementa cuando está esperando el CPU
  - Se decrementa cuando usa la CPU
  - El cambio se realiza cuando alguno de los procesos en espera tiene mayor *wait_runtime* que el que está en ejecución.
  - Notar que no se utiliza el concepto de *quantum*.

CFS: Implementación

- Se usa un Red-Black Tree (RBT) pues las búsquedas, la inserción y el borrado son

  O(log n).

  Cada nodo *p* representa una tarea. Está ordenado por:

  rq->fair_clock – p-> wait_runtime

- La elección es sencillamente tomar el nodo de más a la izquierda

CFS: prioridades

- Solución elegante: se asocia un peso a cada prioridad (usado para decrementar el wait_runtime), de forma que los procesos de baja prioridad ven pasar el tiempo más rápido que los de alta prioridad (su *wait_runtime* expira más rápido).

CFS en kernel 2.6.24

- Kernel 2.6.24: reestructuración, en vez de utilizar *rq->fair_clock*, las tareas se persiguen entre sí.
- Cada tarea tiene un *vruntime* (virtual runtime) que se incremente con el reloj real cada vez que la tarea usa el CPU.
- Ahora el árbol se ordena por *vruntime* y se elige la tarea de menor *vruntime*.
- Además se agregó “group scheduling”

CFS en kernel 2.6.24

- Group scheduling
  - Usuario A: 24 procesos
  - Usuario B: 1 proceso
  - ¿cuánta CPU recibe el usuario B?
  - Lo mismo puede aplicarse para jerarquía de procesos (ej.: varios procesos “hermanos” lanzan varios hijos)

Multiprocesadores

- Cada procesador tiene sus propia runqueue.
- Cada runqueue lleva una medida de su carga (load).
- En el momento de elegir puede darse una migración: se mueve un proceso de la runqueue de un procesador -sobrecargado- a otro -posiblemente idle o con poca carga-.
- Problema: cachés