Scheduling

Planificación de CPU: ¿Cómo podemos gestionar los recursos que tenemos para lograr *multitasking*?

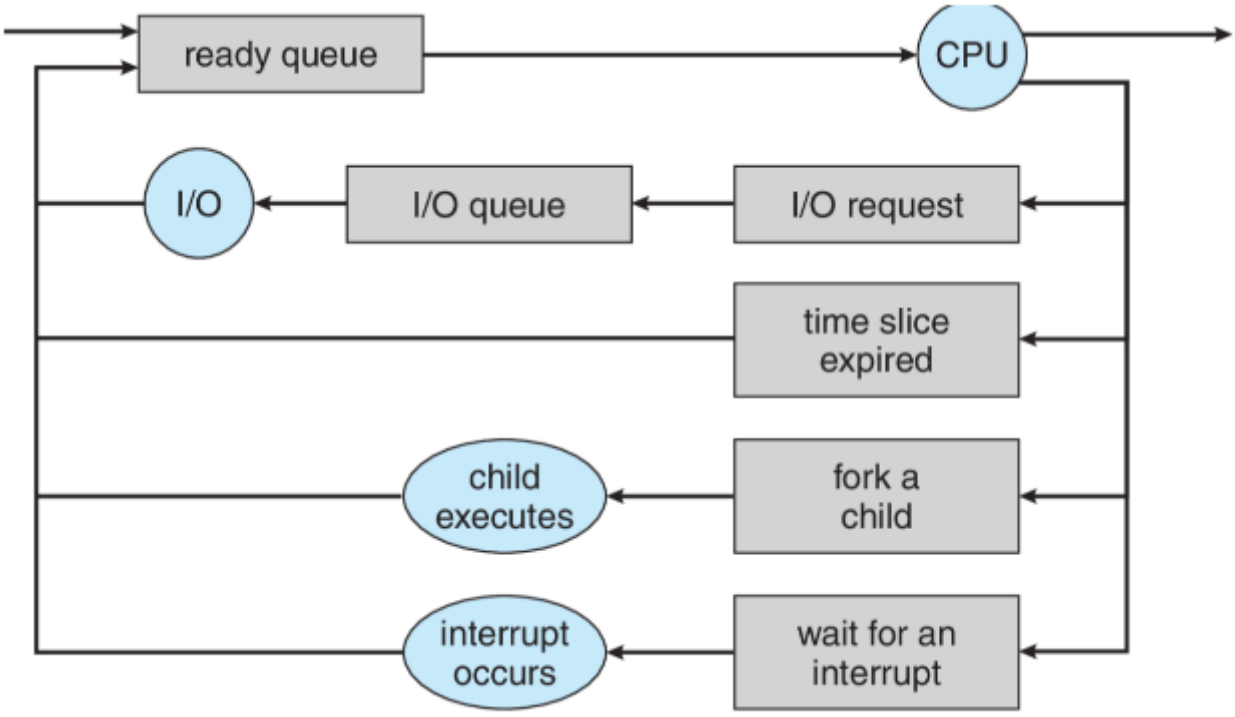
Tenemos:

- Múltiples procesos en memoria (multiprogramación) ordenados en una tabla de PCBs. Si no tuvieramos multiprogramación no necesitaríamos planificar.
- Algunos procesos en estado *ready*.
- CPU que puede atender un solo proceso a la vez.

Queremos:

- Multitasking: asignar tiempo a múltiples procesos.

Gestionamos los recursos a través del *scheduler*, el cual puede ser visto como un sistema de manejo de colas.



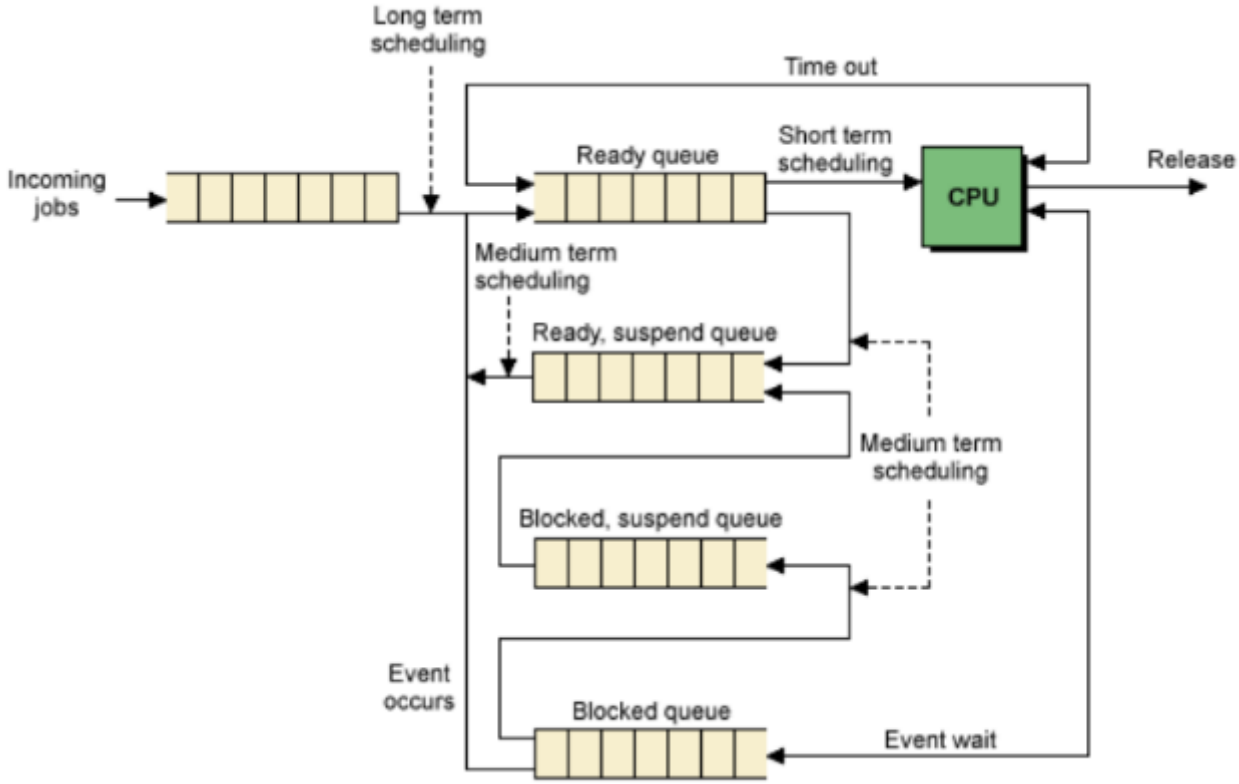
PROF

Recordemos que, en el caso de la *syscall* fork() para Linux, el padre también sigue ejecutando.

Niveles de *scheduling*

- *Long-term scheduler*:
  - Admite procesos en la cola *ready*.
  - Determina el grado de multiprogramación (cantidad de procesos en memoria).
- *Medium-term scheduler*:
  - Modifica temporalmente el grado de multiprogramación.

    - Ejecuta *swapping* copiando memoria RAM al disco y del disco a la RAM.
- *Short-term scheduler* (aka *dispatcher*):
    - Selecciona un proceso de la cola *ready* para ejecutar.
    - Ejecuta el cambio de contexto.

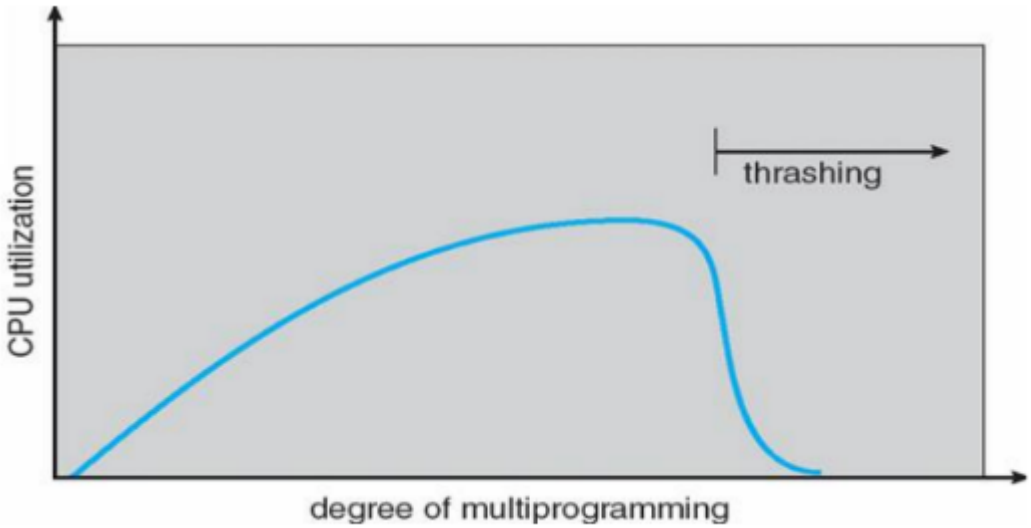


¿Y si estamos siempre haciendo *scheduling* en vez de ejecutar programas?

Scheduling es importante para el *multitasking*... pero *scheduling* y *context switch* son solo *overhead*:

- ¿Qué pasa si el *scheduler* o el *context switch* toman más tiempo de lo que toma el proceso?
- ¿Qué pasa si se le asigna poco tiempo a cada proceso?
- ¿Qué pasa si hay muchos procesos *ready*?
  La contención o atascamiento de procesos se refleja en el concepto de *thrashing*:

PROF

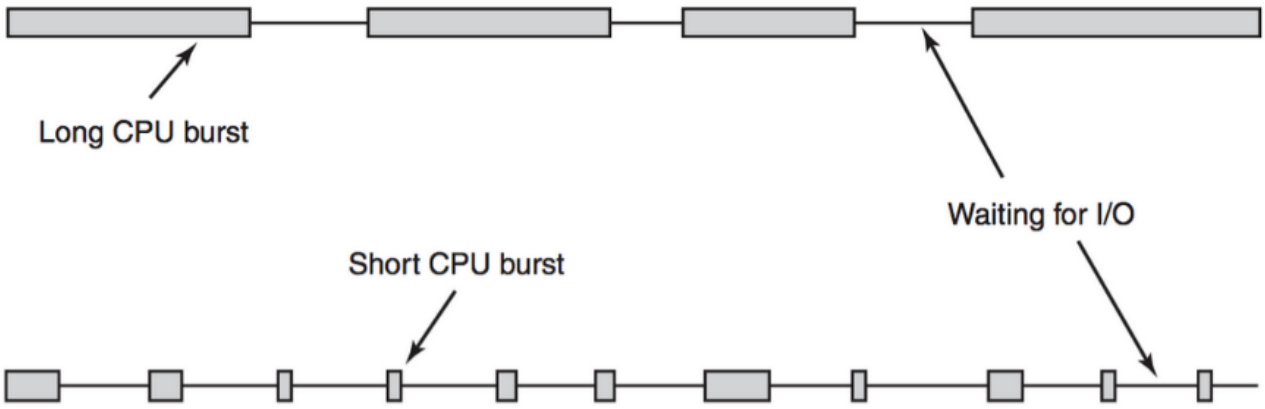


Modelo de ejecución de un proceso

No todos los procesos se comportan igual. Los procesos en general alternan entre dos fases:

- Uso de CPU (*CPU brust*)
- Espera por I/O (*I/O-brust*)

y suelen estar dominados por uno u otro, de modo que, a los procesos fuertemente dominados por el uso de CPU se le conoce como *CPU intensive* mientras que a los procesos fuertemente dominados por espera de I/O se les conoce como *I/O intensive* o procesos interactivos.

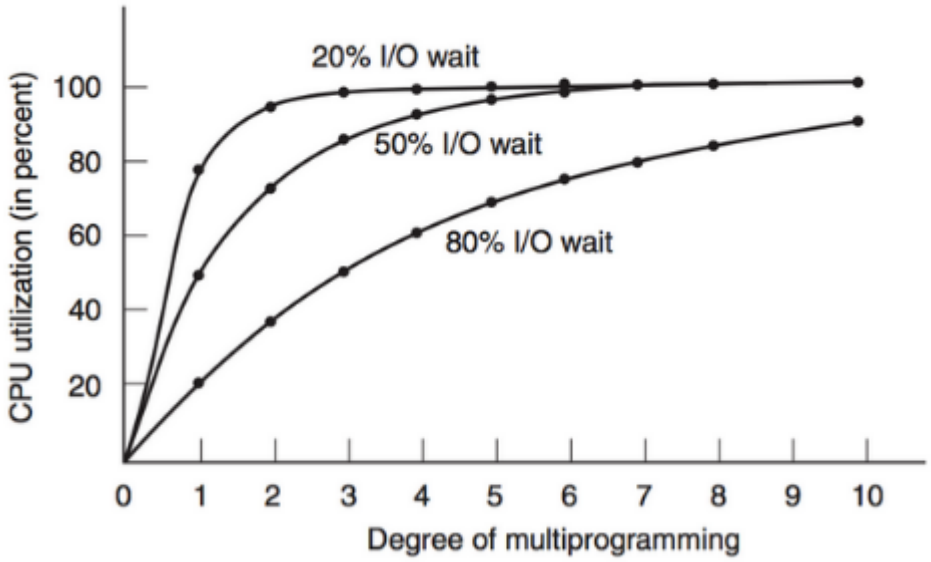


PROF

Una CPU bajo el 100% está subutilizada.

El tiempo que gasta un proceso en espera obviamente influye en la utilización de la CPU:

- p es el porcentaje de tiempo en espera de I/O.
- p^n es la probabilidad que n procesos estén esperando por I/O.
- Utilización de la CPU es $CPU_n=1-p^n$
  Una vez que llego al 100% ya no tiene mucho sentido agregar más procesos.



Tipos de *scheduling*

Podemos clasificar las políticas de *scheduling* según tipo de interrupción y según objetivo.
Según tipo de interrupción:

- *Preemptive* (expropiativo):
  - Utiliza interrupciones (ej: de reloj *timer*) para decidir cuando sacar un proceso de ejecución.
  - Hoy en día los OS suelen utilizar *schedulers* de tipo expropiativo.
- *Non-Preemptive* (colaborativo o no-expropiativo): Permite que un proceso ejecute hasta que:
  - El proceso deja voluntariamente la CPU, ó
  - El proceso se bloquea en I/O, ó
  - El proceso termina.

PROF

Según objetivo:

- *Batch*: trabajo por lotes. Sin interacción.
  - Mantener la CPU lo más ocupada posible.
  - Minimizar *turnaround time*: tiempo desde envío hasta término.
  - Maximizar *throughput*: número de trabajos por unidad de tiempo.
- *Interactive*:
  - Minimizar tiempo de respuesta.
  - Satisfacer usuarios.
  - Gran parte de los OS implementan *schedulers* interactivos.
- *Real Time*:
  - Tiempo de respuesta debe ser predecible.
  - Alcanzar *deadlines*.
  - Un buen ejemplo es un sistema digital de frenado de un auto.

Todos los tipos de _scheduler_ tienen el objetivo de _fairness_: que todos los procesos tengan un tiempo razonable de ejecución.

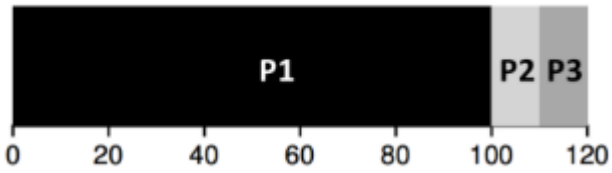
Algoritmos de *Scheduling*

Algoritmos de *batch scheduling*

First-come, First-served (FCFS)

Orden de llegada. Cola FIFO

	Llegada	CPU-burst	Término	Turnaround
P1	0	100	100	100
P2	1	10	110	109
P3	2	10	120	118



Importante entender la tabla y el gráfico.

Turnaround promedio de 109. Observar la diferencia de *turnaround* si el orden de llegada hubiese sido P2 en $t=0$, P1 en $t=1$ y P3 en $t=2$. En este caso el *turnaround* sería de 79.

Si bien es sencillo de implementar, depende mucho del orden en el que llegan los procesos:

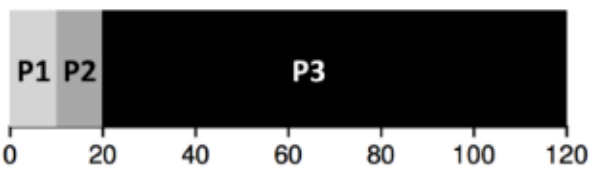
- *Non-preemtive*
- Simple
- Poco predecible. *Convoy effect*

PROF

Shortesr Job First (SJF)

El más corto primero.

	Llegada	CPU-burst	Término	Turnaround
P1	0	10	10	10
P2	2	10	20	18
P3	1	100	120	119



Turnaround promedio de 49.

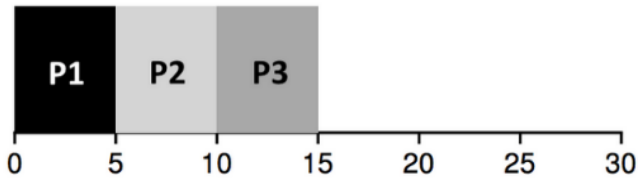
- El algorítmo es óptimo 😃
- No sabemos cuánto demora cada *CPU-brust* 😦

- Esta versión es *non-preemptive*. La versión *preemptive* es *Shortest Remaining Time Next* la cual escoge al que le queda menos tiempo. Este tiempo se debe estimar lo que puede agregar *overhead*.
- Posible inanición (*starvation*) de procesos largos x.x (si me demoro mucho puede que nunca me toque)

Algoritmos de *scheduling* interactivos

Para este tipo de algoritmos, lo que más importa es que los tiempos de respuesta sean mínimos. En otras palabras, estos algoritmos tienen como métrica principal el *response time*. El tiempo de respuesta es el tiempo que pasa desde que el usuario envía un trabajo hasta que el trabajo comienza a ejecutarse.

	Llegada	CPU-burst	Término	Turnaround	Inicio ejec.	T.Respuesta
P1	0	5	5	5	0	0
P2	1	5	10	9	5	4
P3	2	5	15	13	10	8

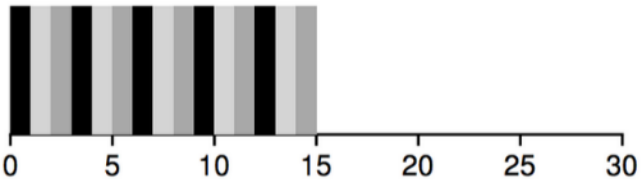


- Turnaround time promedio: 9
- Response time promedio: 4

Round Robin (RR)

- Asigna un tiempo de ejecución (*quantum*, *time slice*) fijo a cada proceso.
- Cuando el tiempo de ejecución de un proceso termina, el proceso es suspendido y se pasa al siguiente proceso en la cola.
- El tiempo de ejecución de cada proceso es el mismo y se asigna en forma circular.
- De tipo *preemptive*.

	Llegada	CPU-burst	Término	Turnaround	Inicio ejec.	T.Respuesta
P1	0	5	13	13	0	0
P2	0.1	5	14	13.9	1	0.9
P3	0.2	5	15	14.8	2	1.8



- Turnaround time promedio: 13.9
- Response time promedio: 0.9

PROF

- Con n procesos, cada uno recibe $\frac{1}{n}$ de CPU.
- Ningún proceso espera más de $(n-1)\times q$ para ejecutar.
- Altamente dependiente de la elección de q:
  - Si q es muy grande, el algoritmo se comporta como FCFS.
  - Si q es muy pequeño, puede generarse *thrashing* por muchos cambios de contexto.

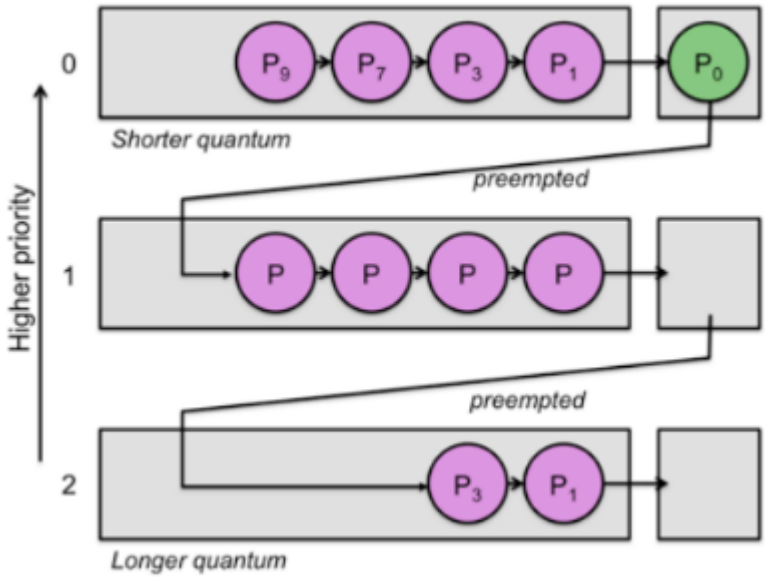
Priority Scheduling

- Cada proceso tiene una prioridad.

- Cuando hay prioridades iguales, se aplica FCFS o RR.
- SJF es un caso particular de este algoritmo.
- Prioridades pueden ser estáticas o dinámicas.
- Puede producirse inanición (*starvation*) de procesos de baja prioridad.
- Para evitar *starvation* se puede utilizar *aging*. Este mecanismo incrementa artificialmente la prioridad de los procesos que llevan más tiempo esperando.

Multi-level Feedback Queue Scheduling

- Buscamos optimizar el *turnaround time* y minimizar el *response time*.
- Se utilizan múltiples colas de procesos con diferentes niveles de prioridad.
- Tiene varias reglas:
    1. Si $priority(A)>priority(B)$, ejecutar A.
    2. Si $priority(A)=priority(B)$, ejecutar A y B con RR.
    3. Los procesos se ingresan en la cola con mayor prioridad.
    4. Si un proceso utiliza todo su tiempo de ejecución, se pasa a la cola de menor prioridad.
    5. Después de un tiempo S, todos los procesos se mueven a la cola con mayor prioridad.

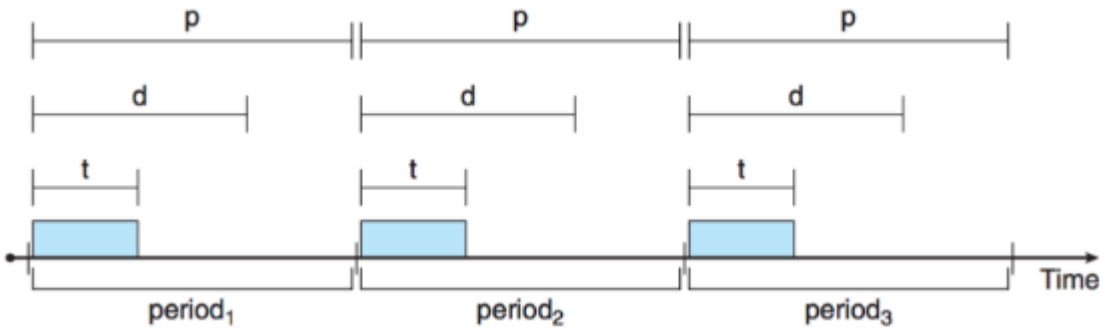


PROF

Algoritmos de *real time scheduling*

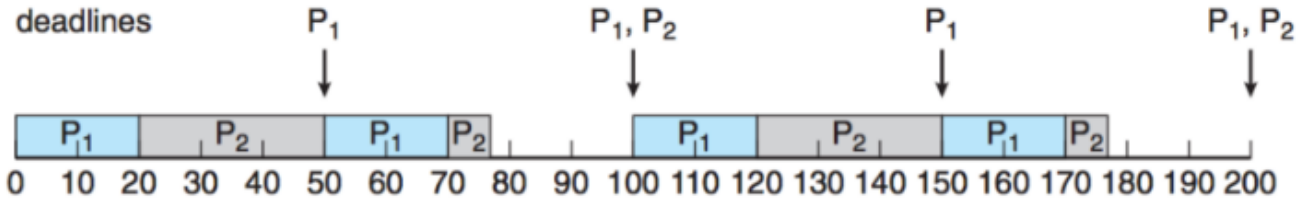
Procesos de tiempo real

- Poseen *deadlines* y períodos de ejecución.
- Sistema debe determinar si el proceso puede ser ejecutado dado el *deadline* d, período p y tiempo de ejecución t.
- En cada período el proceso debe avanzar en su ejecución una vez.



Algoritmo *Rate Monotonic Scheduling* (RMS)

- A cada proceso se le asigna una prioridad $\frac{t_i}{p_i}$
- Es un algoritmo estático, de modo que podría perder *deadlines*.



Algoritmo *Earliest Deadline First* (EDF)

- Es dinámico, eligiendo siempre el proceso con *deadline* más cercana.