Scheduling

Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Sistemas Operativos, primer cuatrimestre de 2016

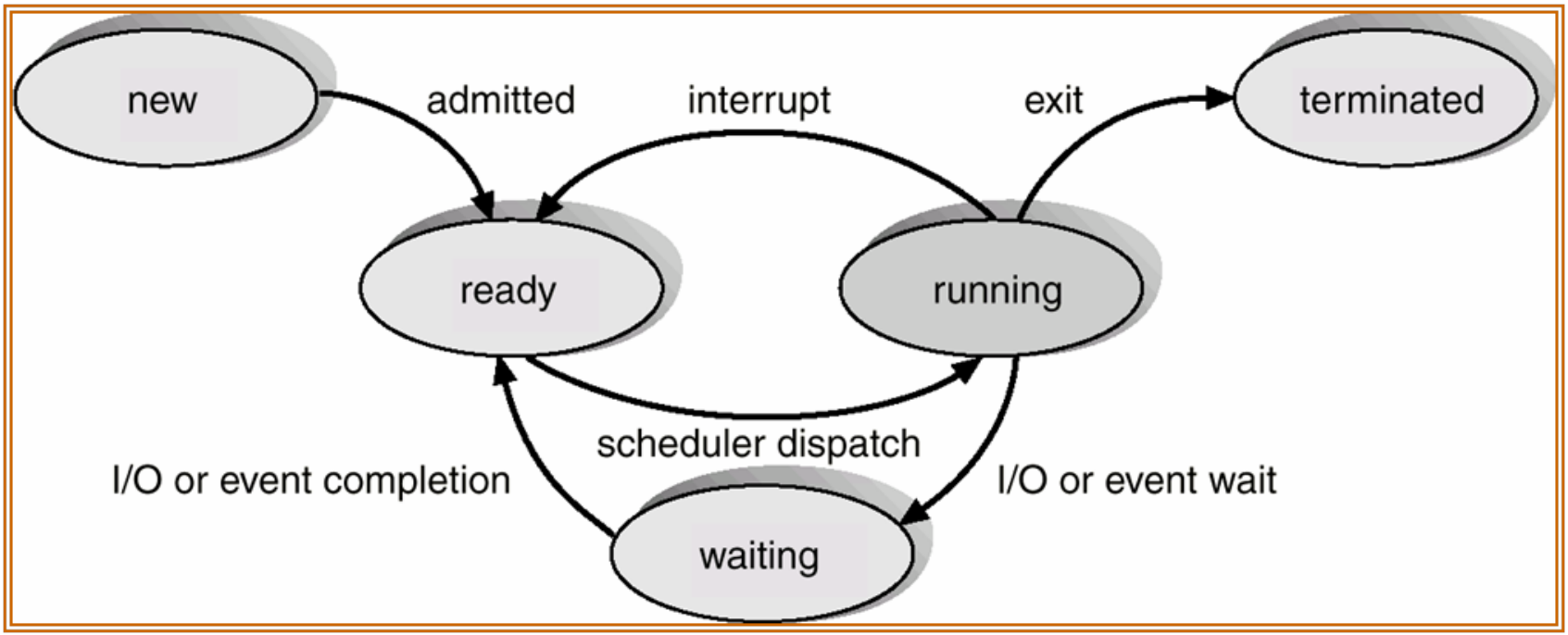
(2) Estado de situación...

- Ya vimos:
  - El concepto de proceso en detalle.
  - Sus diferentes actividades.
  - Qué es una system call.
  - Introdujimos IPC.
- Ahora nos toca:
  - Vamos a poner la lupa en el scheduler.

(3) Proceso

- *Programa: estático*
  - Texto escrito en algún lenguaje de programación.
  - Ese programa eventualmente se compila en código objeto, lo que también es un programa escrito en lenguaje de máquina.
- *Proceso: dinámico* 
  - Unidad de *ejecución*
    Cada proceso tiene un *identificador numérico único*, el *pid*.
  - Unidad de *scheduling*
  - Un proceso tiene *estado*

(4) Proceso: estado



(5) Proceso: cambios de estado

El SO ejerce las siguientes acciones sobre los procesos:

- Admite (o no) un proceso
- Otorga la CPU a un proceso
- Desaloja un proceso de la CPU

  Preemption

  Proceso es interrumpido por algún evento

  Blocking

  Proceso se bloquea en el acceso a un recurso

(6) Proceso: control de admisión

- ¿Qué es?

  Decidir si un proceso nuevo puede ser *admitido*

- ¿Para qué?

  Para evitar *sobrecargar* el sistema

- Carga (*load*):

  Cantidad de procesos activos

  Importa la carga promedio más que la carga puntual 

  Linux: `calc_load()` y `uptime` (`http://goo.gl/BCKcbp`)

```
$ uptime 11:33 up 1 day, 20:25, 2 users, load
averages: 2.98 2.83 2.69
```

(7) Proceso: control de admisión

- ¿Cómo se hace?
  - Se aplican políticas de admisión
- Umbral (*threshold*) o *Multi-Programming Level* (MPL)
  - Por debajo se admite
  - Por encima se rechaza o se pone en una cola especial
- Calidad de servicio o *Quality of Service* (QoS)
  - Prioridades: críticos vs no críticos
  - Recursos requeridos
  - Negociación

(8) Proceso: control de admisión

- ¿Cuál es el umbral óptimo?
- Rendimiento (*throughput*)

  Procesos terminados por unidad de tiempo
- Curva típica de carga vs rendimiento

- Umbral estático: constante pre-calculada
- Umbral dinámico calculado usando teoría de control optimal

(9) Proceso: control de admisión (artículos)

- J. Stankovic. Admission Control, Reservation, and Reflection in Operating Systems. 1998. `http://goo.gl/uJiOXD`
- R. Rajkumar, K. Juvva, A. Molano, S. Oikawa. Resource kernels: a resource-centric approach to real-time and multimedia systems. 1998. `http://goo.gl/nmMei1`
- J. Stankovic, R. Rajkumar. Real-Time Operating Systems. 2004. `http://goo.gl/yhtRLG`
- M. Welsh, D. Culler, E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. 2001. `http://goo.gl/AnHyWf`
- V. Gupta, M. Harchol-Balter. Self-Adaptive Admission Control Policies for Resource-Sharing Systems. 2009. `http://goo.gl/4o36YL`

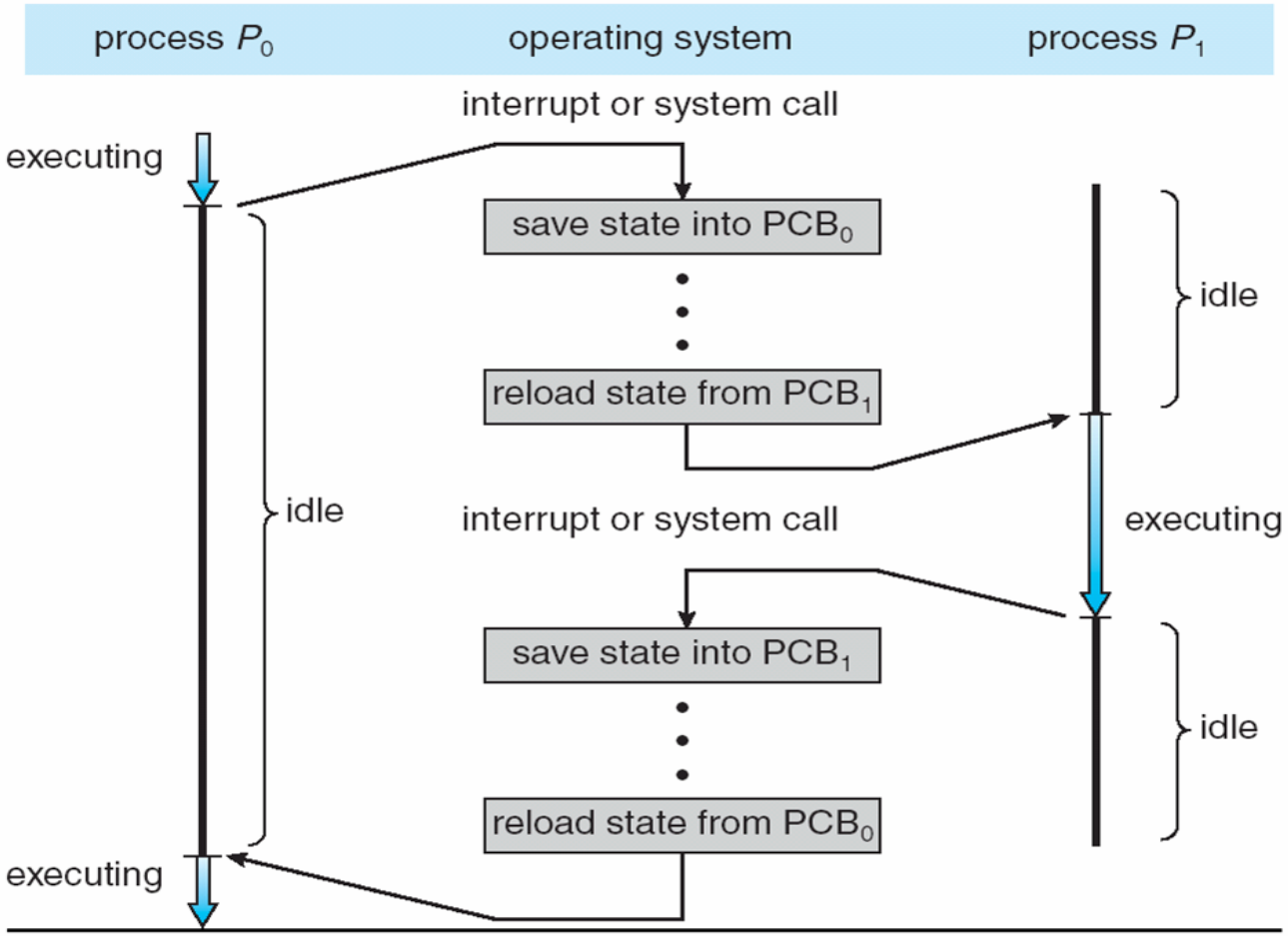
(10) Ejecución fuera del kernel

- ¿Por cuánto tiempo lo dejamos ejecutar?
  - Hasta que termina: Es lo mejor para el proceso, pero no para el sistema en su conjunto. Además, podría no terminar.
  - *time-sharing*: un "ratito" (*quantum*). 
    Curiosidad: Bemer. Origins of timesharing. `http://goo.gl/83Ptc4`
- Los SO modernos hacen *preemption*: cuando se acaba el quantum, le toca el turno a *otro* proceso.
- Surgen dos preguntas:
  - Quién y cómo decide a quién le toca
    → *scheduling*
  - Qué significa hacer que se ejecute otro proceso
    → *context switch*

(11) Cambio de contexto (context switch)

- Para el proceso desalojado el SO debe:
  - Guardar el PCB.
  - Guardar los registros.
- Para el proceso asignado el SO debe:
  - Cargar el PCB.
  - Cargar los registros.
- El tiempo utilizado en cambios de contexto es tiempo muerto, no se está haciendo nada productivo.
- Dos consecuencias de esto:
  - *Duración*: Impacto en la arquitectura del HW
  - *Cantidad*: Quantum apropiado para minimizar los cambios 
- Implementación: colgarse de la interrupción del clock.

(12) Cambio de contexto (esquema)



(13) Cambio de contexto (artículos)

- Costo
  - Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. `http://goo.gl/GqGvKt`
  - Francis M. David, et al. 2007. Context switch overheads for Linux on ARM platforms. `http://goo.gl/pj9Dwj`
- Prevención
  - Jaaskelainen, et al. Reducing context switch overhead with compiler-assisted threading. 2008. `http://goo.gl/8th0dy`.
  - Kloukinas, Yovine. 2011. A model-based approach for multiple QoS in scheduling: from models to implementation. `http://goo.gl/4xL1JT`

(14) Scheduling

- La política de scheduling es una de las principales huellas de identidad de un SO.
- Es tan importante que algunos SO proveen más de una.
- Buena parte del esfuerzo por optimizar el rendimiento de un SO se gasta en la política de scheduling.

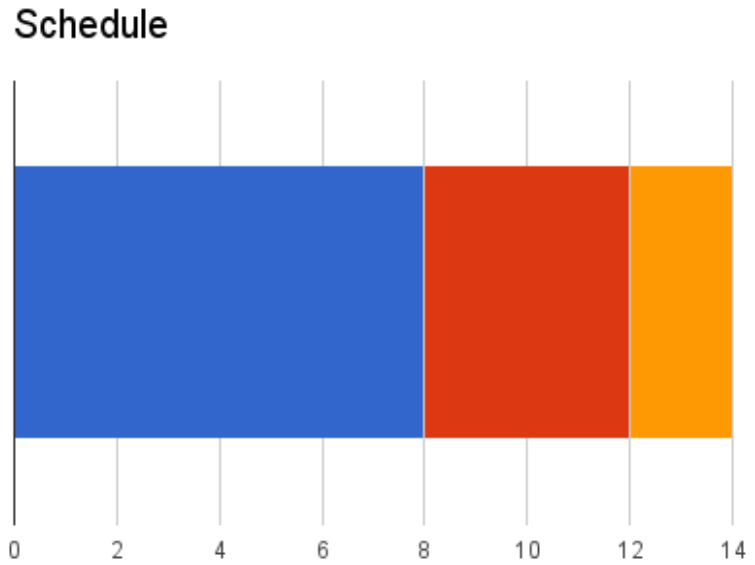
(15) La fábula del Restaurant Zen So-So

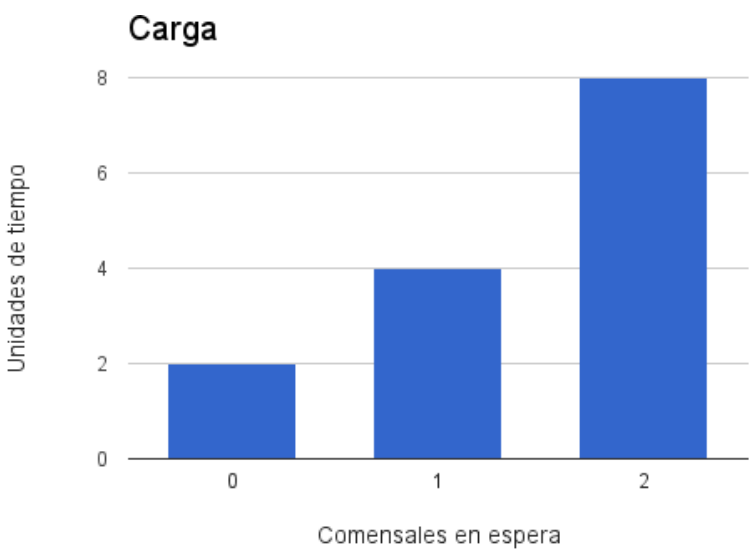
- Ambiente Zen: 1 mesa y 1 silla
- 3 combos de Sushi:
    1. Sakura: 2 piezas de Sushi
    2. Samurai: 4 piezas de Sushi
    3. Godzilla: 8 piezas de Sushi
- Velocidad de ingesta: 1 unidad de tiempo por pieza de Sushi
- Llegan 3 comensales simultáneamente:
    - A Pide un Sakura
    - B Pide un Samurai
    - C Pide un Godzilla
- ¿En qué orden conviene atenderlos?

(16) La fábula del Restaurant Zen So-So

- Comer hasta terminar (o, *eat to completion*)
- Longest Job First

	Latencia	Espera	Compleción	Ratio (E/C)
A	12	12	14	0.85
B	8	8	12	0.66
C	0	0	8	0
AVG	6.66	11.33	11.33	0.5

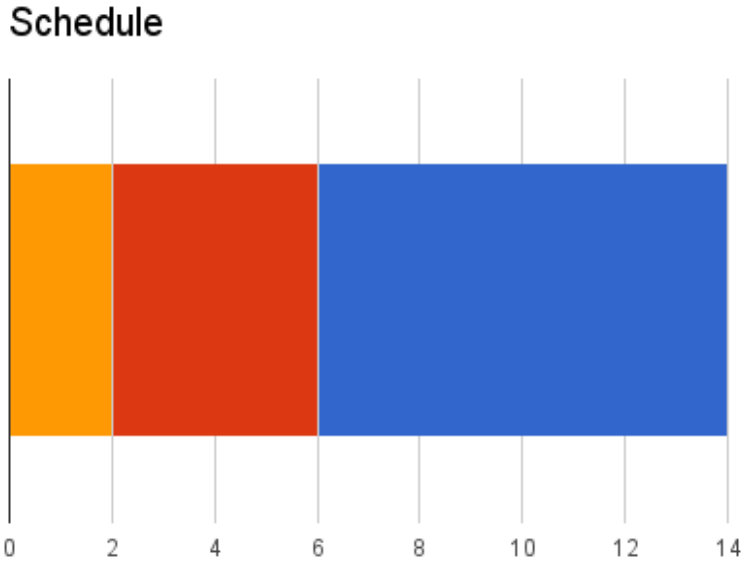


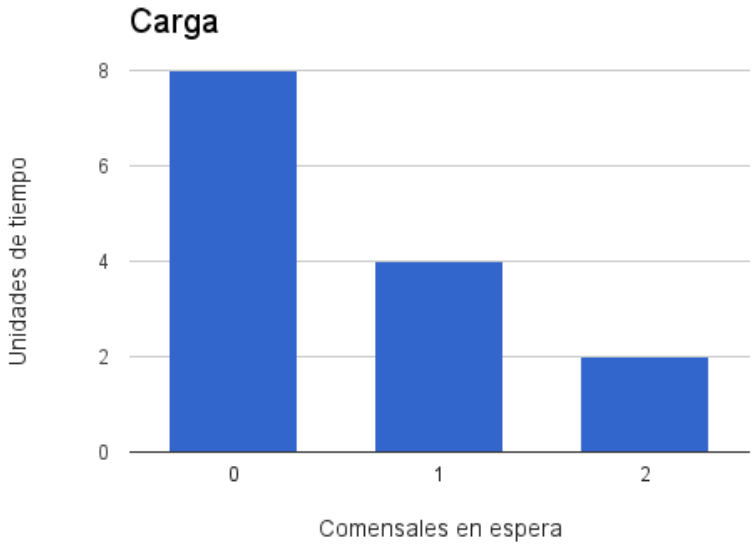


(17) La fábula del Restaurant Zen So-So

- Comer hasta terminar (o, *eat to completion*)
- Shortest Job First

	Latencia	Espera	Compleción	Ratio (E/C)
A	0	0	2	0
B	2	2	6	0.33
C	6	6	14	0.4
AVG	2.66	7.33	7.33	0.25

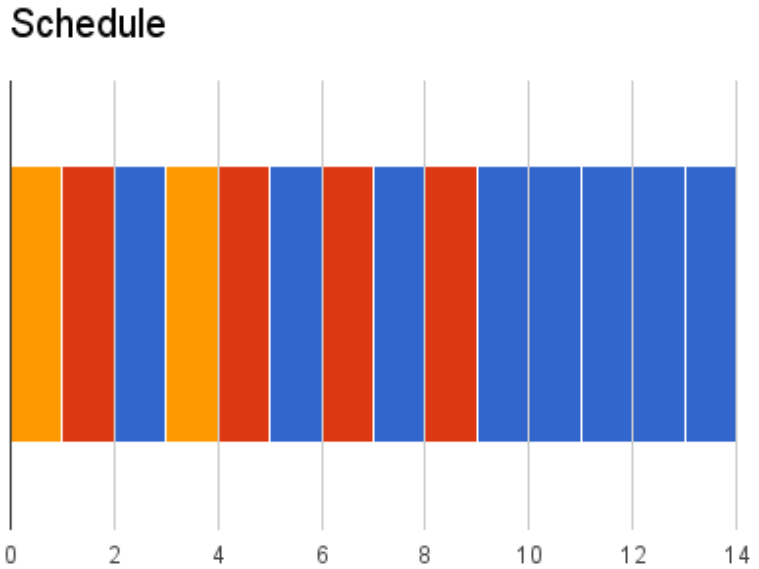


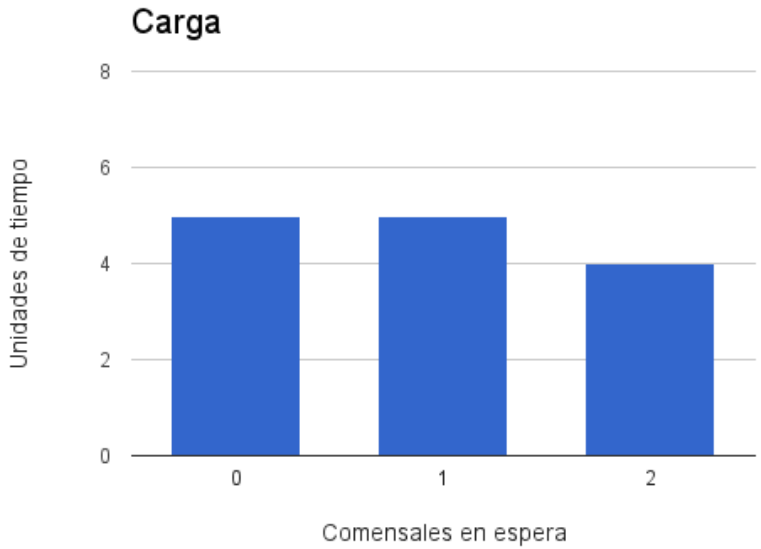


(18) La fábula del Restaurant Zen So-So

- Comer a intervalos (o, *preemptive eating*)
- Round-robin, 1 Sushi por vez (*quantum*)

	Latencia	Espera	Compleción	Ratio (E/C)
A	0	2	4	0.5
B	1	5	9	0.55
C	2	6	14	0.4
AVG	1	4.33	9	0.48





(19) Objetivos de la política de scheduling

- Qué optimizar: 
  - Ecuanimidad o Justicia (*fairness*): cada proceso recibe una dosis "justa" de CPU (para alguna definición de justicia).
  - Eficiencia: tratar de que la CPU esté ocupada todo el tiempo.
  - Carga del sistema: minimizar la cantidad de procesos listos que están esperando CPU.
  - Tiempo de espera: minimizar el tiempo que un proceso está en estado "listo".
  - Latencia: minimizar el tiempo requerido para que un proceso empiece a dar resultados. También llamado tiempo de respuesta para procesos interactivos.
  - Tiempo de ejecución (*completion time* o *turnaround*): minimizar el tiempo total que le toma a un proceso terminar.
  - Rendimiento (*throughput*): maximizar el número de procesos terminados por unidad de tiempo.
  - Liberación de recursos: hacer que terminen cuanto antes los procesos que tiene reservados más recursos.

(20) Objetivos de la política de scheduling (cont.)

- Muchos de estos objetivos son contradictorios.
- Si los usuarios del sistema son heterogéneos, pueden tener distintos intereses.
- Una cosa queda clara: no se puede tener ICI20yIMdHC.
- En definitiva, cada política de scheduling va a buscar maximizar una función objetivo, que va a ser una combinación de estas metas tratando de impactar lo menos posible en el resto. 

(21) Cuándo actúa el scheduler

- El scheduling puede ser *cooperativo* o *con desalojo*. 
- Si es con desalojo (también llamado scheduling apropiativo o *preemptive*), el scheduler se vale de la interrupción del clock para decidir si el proceso actual debe seguir ejecutándose o le toca a otro.
- Recordemos: el clock interrumpe 50 ó 60 veces/seg.
- Si bien suele ser deseable, el scheduling con desalojo:
  - Requiere un clock con interrupciones (podría no estar disponible en procesadores embebidos).
  - No le da garantías de continuidad a los procesos (podría ser un problema en SO de tiempo real).
- Cuando tenemos multitarea cooperativa,
  - El scheduler analiza la situación cuando el kernel toma control (en los syscalls).
  - Especialmente cuando el proceso hace E/S.
  - A veces se proveen llamadas explícitas para permitir que se ejecuten otros procesos.
- En realidad, los schedulers con desalojo combinan ambos enfoques.

(22) El procesador como una sala de espera

- Un enfoque posible es FIFO, también conocido como *FCFS* (*First Came, First Served*).
- El problema es que supone que todos los procesos son iguales.
- Si llega un "megaproceso" que requiere mucha CPU, tapona a todos los demás.
- Entonces, agreguémosle prioridades al modelo. Como en una sala de espera.
- Posible problema: *inanición* (*starvation*). Los procesos de mayor prioridad demoran infinitamente a los de menor prioridad, que nunca se ejecutan. 
- Una posible solución: aumentar la prioridad de los procesos a medida que van "envejeciendo".
- Cualquier esquema de prioridades fijas corre riesgo de inanición. 

(23) Round robin

- La idea es darle un quantum a cada proceso, e ir alternando entre ellos.
- ¿Cuánto dura el quantum?
  - Si es muy largo, en SO interactivos podría parecer que el sistema no responde.
  - Si es muy corto, el tiempo de scheduling+context switch se vuelve una proporción importante del quantum. Por ende, el sistema pasa un porcentaje alto de su tiempo haciendo "mantenimiento" en lugar de trabajo de verdad.
- Se lo suele combinar con prioridades.
  - Que pueden estar dadas por el tipo de usuario (administrativas) o pueden ser "decididas" por el propio proceso. Esto último no suele funcionar.
  - Que van decreciendo a medida que los procesos reciben su quantum, para evitar inanición de los otros.
- Además, los procesos que hacen E/S suelen recibir crédito extra, por ser buenos compañeros.

(24) Múltiples colas

- Colas con 1, 2, 4, 8 quanta c/u.
- A la hora de elegir un proceso la prioridad la tiene siempre la cola con menos quanta.
- Cuando a un proceso no le alcanza su cuota de CPU es pasado a la cola siguiente, lo que disminuye su prioridad, pero le asigna más tiempo de CPU en el próximo turno.
- Los procesos de máxima prioridad, los interactivos en gral, van a la cola de máxima prioridad.
- Se puede hacer que cuando un proceso termina de hacer E/S vuelva a la cola de máxima prioridad, porque se supone que va a volver a hacerse interactivo.
- La idea general es minimizar el tiempo de respuesta para los procesos interactivos, suponiendo que los cómputos largos son menos sensibles a demoras.

(25) Trabajo más corto primero

- También llamada *SJF* (*Shortest Job First*).
- Está ideada para sistemas donde predominan los trabajos batch. Está orientada a maximizar el throughput.
- En esos casos, muchas veces se puede predecir la duración del trabajo o al menos clasificarlo (por ejemplo: menos de 10', menos de 30', menos de 60', más de 60').
- Si conozco las duraciones de antemano, es óptimo (en cuanto a la latencia promedio).
- Otra alternativa es no pensar en la duración total, sino más bien en cuánto tiempo necesita hasta hacer E/S de nuevo.
- El problema real es cómo saber cuánta CPU va a necesitar un proceso.
- Una alternativa es usar la info del pasado para predecir.
- Puede salir mal si los procesos tienen comportamiento irregular.

(26) Scheduling para RT

- Los sistemas de tiempo real son aquellos en donde las tareas tiene fechas de finalización (*deadlines*) estrictas.
- En general se usan en entornos críticos: si un deadline no se cumple, algo malo pasa.
- Scheduling en RT es un problema en sí mismo. Apenas vamos a mencionarlo.
- Una política posible consiste en correr el proceso más cercano a perder su deadline.

(27) Scheduling en SMP

- Scheduling en SMP es también un problema bastante distinto.
- El problema es el caché, que es de vital importancia para el rendimiento de los programas.
- Si la política de scheduling hace pasar un proceso a otro procesador, éste llega con el caché vacío, tardando mucho más de lo que tardaría si se hubiese ejecutado en el mismo procesador que antes.
- Por eso se utiliza el concepto de *afinidad al procesador*: tratar de usar el mismo procesador, aunque se tarde un poco más en obtenerlo.
- Si esto se respeta a rajatabla, *afinidad dura*. Si simplemente es un intento, *afinidad blanda*.

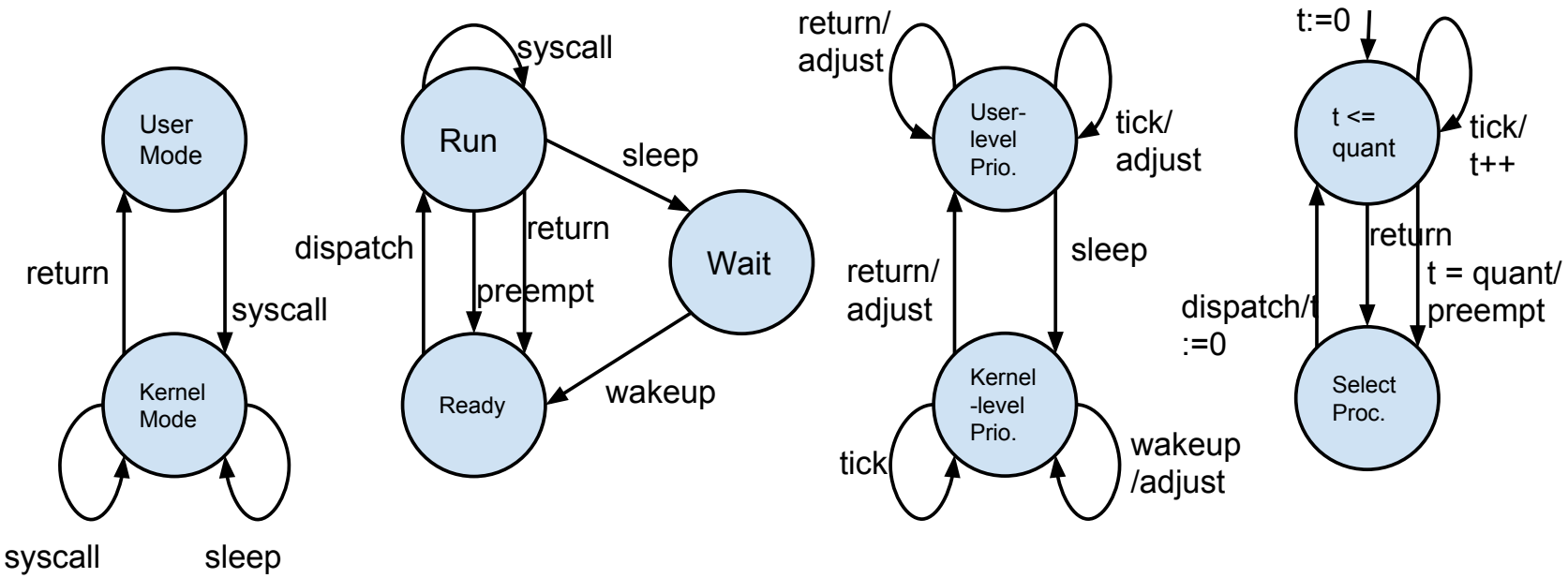
(28) En la práctica...

- Muchas de las consideraciones que planteamos tienen a su vez, bemoles. Por ejemplo:
  - ¿El scheduling debe ser justo entre procesos o usuarios? Un usuario puede tener varios procesos...
  - ¿Qué pasa con los procesos hijos?
  - Si un proceso requiere mucha CPU, ¿debo priorizarlo o matarlo? Tal vez tenga usuarios hostiles que estén tratando de abusar el sistema...
- Elegir un buen algoritmo de scheduling que funcione en la práctica es muy difícil.
- Suele requerir prueba/error/corrección, y muchas veces deben ajustarse a medida que cambian los patrones de uso.
- A veces se arman modelos matemáticos basados en teoría de colas.
- Otras, se prueban con patrones de carga tomados de sistemas concretos o benchmarks estandarizados.

(29) En la práctica... (cont.)

- Si bien cada proceso es único, algunas cosas se pueden saber:
  - Si un proceso abre una terminal, muy probablemente esté por convertirse en interactivo.
  - En algunos casos se puede usar análisis estático para ver si cierto comportamiento se va a repetir, si el proceso no tiene pensado terminar, etc.
  - Etc.
- Al cóctel le faltan aún ingredientes...
  - Usos específicos: ejemplo: motores de BD, cómputo científico, micro-/macro- benchmarking.
  - Threads (por ahora pueden pensarlo como procesos dentro de procesos).
  - Virtualización.

(30) Modelo “desmenuzado” del scheduler



return = retorno a modo usuario después de un syscall
sleep = syscall del kernel que pone un proceso en estado bloqueado
tick = interrupción del reloj

M. J. Bach. The Design of the UNIX® Operating System. Prentice Hall, 1986.

(31) Tarea

- Leer los detalles del libro.
- En especial, seguir los ejemplos numéricos para entender los algoritmos.

(32) Dónde estamos

- Vimos
  - Todas las consideraciones, contradictorias a veces, que deben hacerse a la hora de elegir un algoritmo de scheduling.
  - Analizamos los algoritmos más comunes.
- En la práctica:
  - Vamos a comparar los algoritmos de scheduling con datos de prueba.
- Próxima teórica:
  - Sincronización entre procesos.