

Sistemas Operativos 1.0

Procesos e Hilos

Copyright Notice

© 2000-2005 David A. Solomon and Mark Russinovich

- These materials are part of the *Windows Operating System Internals Curriculum Development Kit*, developed by David A. Solomon and Mark E. Russinovich with Andreas Polze
- Microsoft has licensed these materials from David Solomon Expert Seminars, Inc. for distribution to academic organizations solely for use in academic environments (and not for commercial use)

Agenda

- El concepto de proceso
- Estados de proceso
- Cambios de contexto
- Planificación de CPU
- Modelos de multithreading

¿Qué es un proceso?

- Un sistema operativo ejecuta programas:
 - En un sistema batch – jobs
 - En sistemas de tiempo compartido – programas de usuario o tasks
- Proceso – un programa en ejecución
 - La ejecución del proceso debe progresar secuencialmente
- Un proceso incluye:
 - Estado del CPU (uno o más threads)
 - Secciones de Text & data (mapa de memoria)
 - Recursos: archivos abiertos, handles, sockets
- En OS tradicionales, los procesos eran la unidad de planificación (i.e. no habían threads)
 - Los OS modernos planifican threads
 - Todo lo que hablaremos hace referencia a threads

Estados de un Proceso o Thread

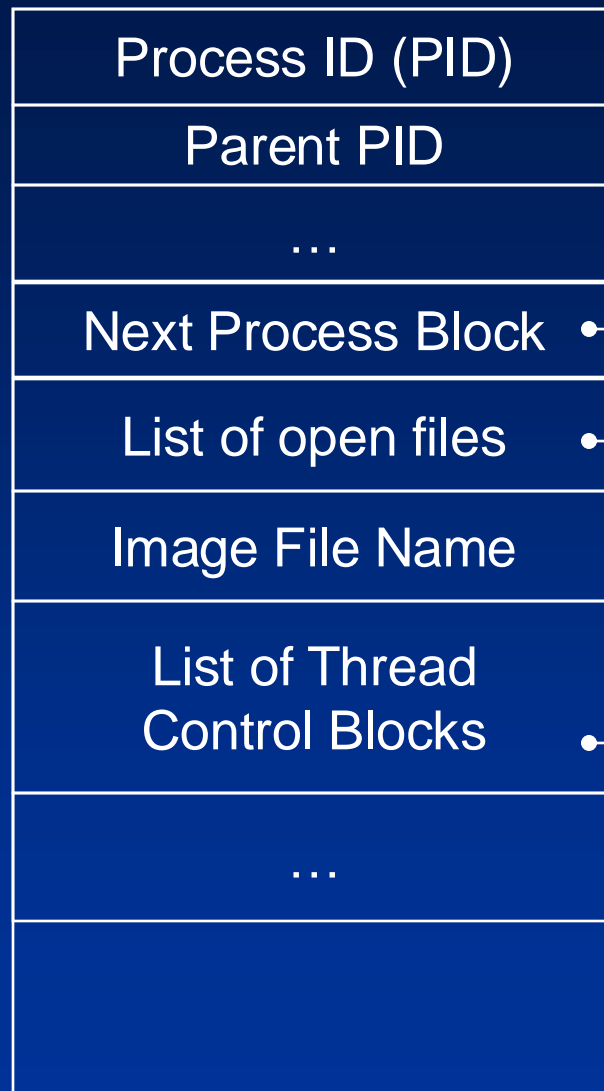
- Diagrama de 5 estados para planificación (scheduling):
 - **init**: El hilo está siendo creado
 - **ready**: El hilo está esperando ser asignado a un CPU
 - **running**: El hilo está en ejecución
 - **waiting**: El hilo está esperando un evento
 - **terminated**: El hilo ha finalizado su ejecución



Process y Thread Control Blocks

- Información asociada con cada proceso: Process Control Block (PCB)
 - Información de manejo de memoria
 - Información de contabilización
 - Proceso-global vs. thread-específico
- Información asociada a cada thread: Thread Control Block (TCB)
 - Program counter
 - Registros del CPU
 - Información de planificación del CPU
 - Información de I/O pendiente

Process Control Block (PCB)

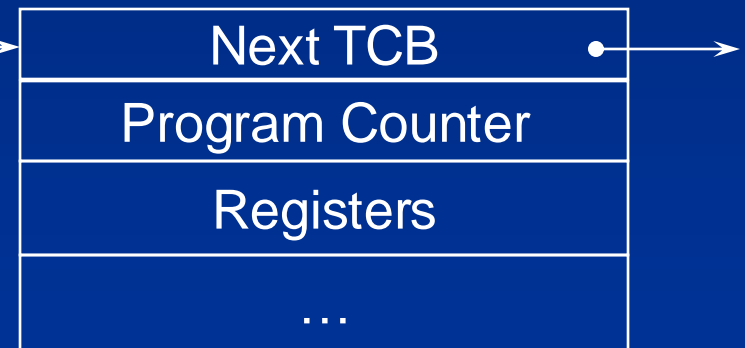


- Vista abstracta
- En Windows, se implementa en varias estructuras

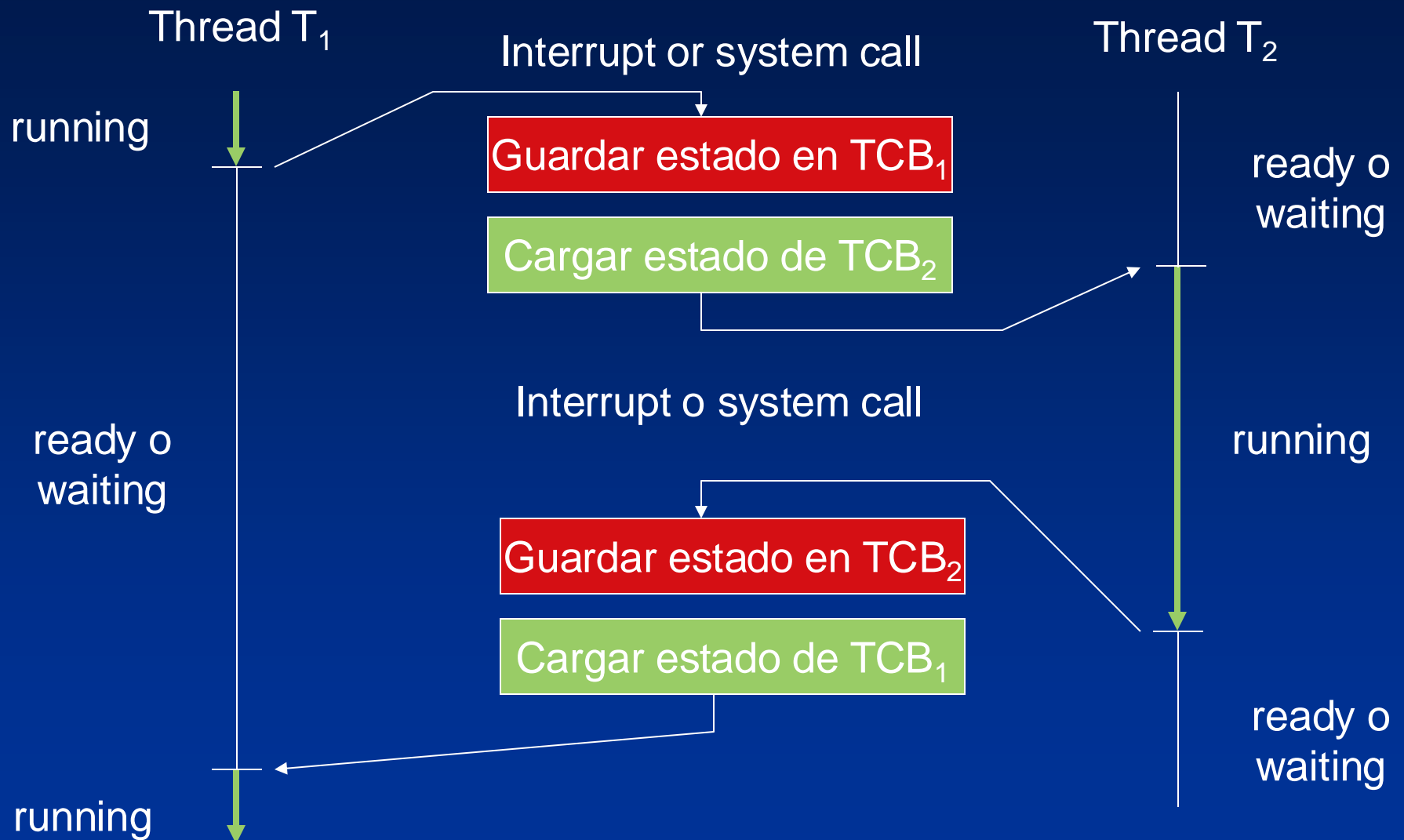
PCB

Handle Table

Thread Control Block (TCB)



Cambio de contexto de un thread a otro



Cambio de contexto

- Context-switch time is overhead; the system does no useful work while switching
- El cambio de contexto entre threads puede implementarse en modo kernel o modo usuario
- Se necesita la interacción con el memory management unit (MMU) para intercambiar entre hilos de diferentes procesos

Cambio de contexto

```
swap32:#{old,new)  swap64:#{old,new)
mov 4(%esp),%eax   # Save registers
mov 8(%esp),%ecx   push %rdi
# Save registers   push %rbp
push %ebp          push %rbx
push %ebx          push %r12
push %esi          push %r13
push %edi          push %r14
# Save old stack   push %r15
mov %esp, (%eax)    #Save old stack
# Load new stack  mov %rsp, (%rdi)
mov (%ecx), %esp    #Load new stack
# Restore regs     mov (%rsi), %rsp
pop %edi           # Restore regs
pop %esi           pop %r15
pop %ebx           pop %r14
pop %ebp           pop %r13
ret               pop %r12
                  pop %rbx
                  pop %rbp
                  pop %rdi
                  ret
```

(a)

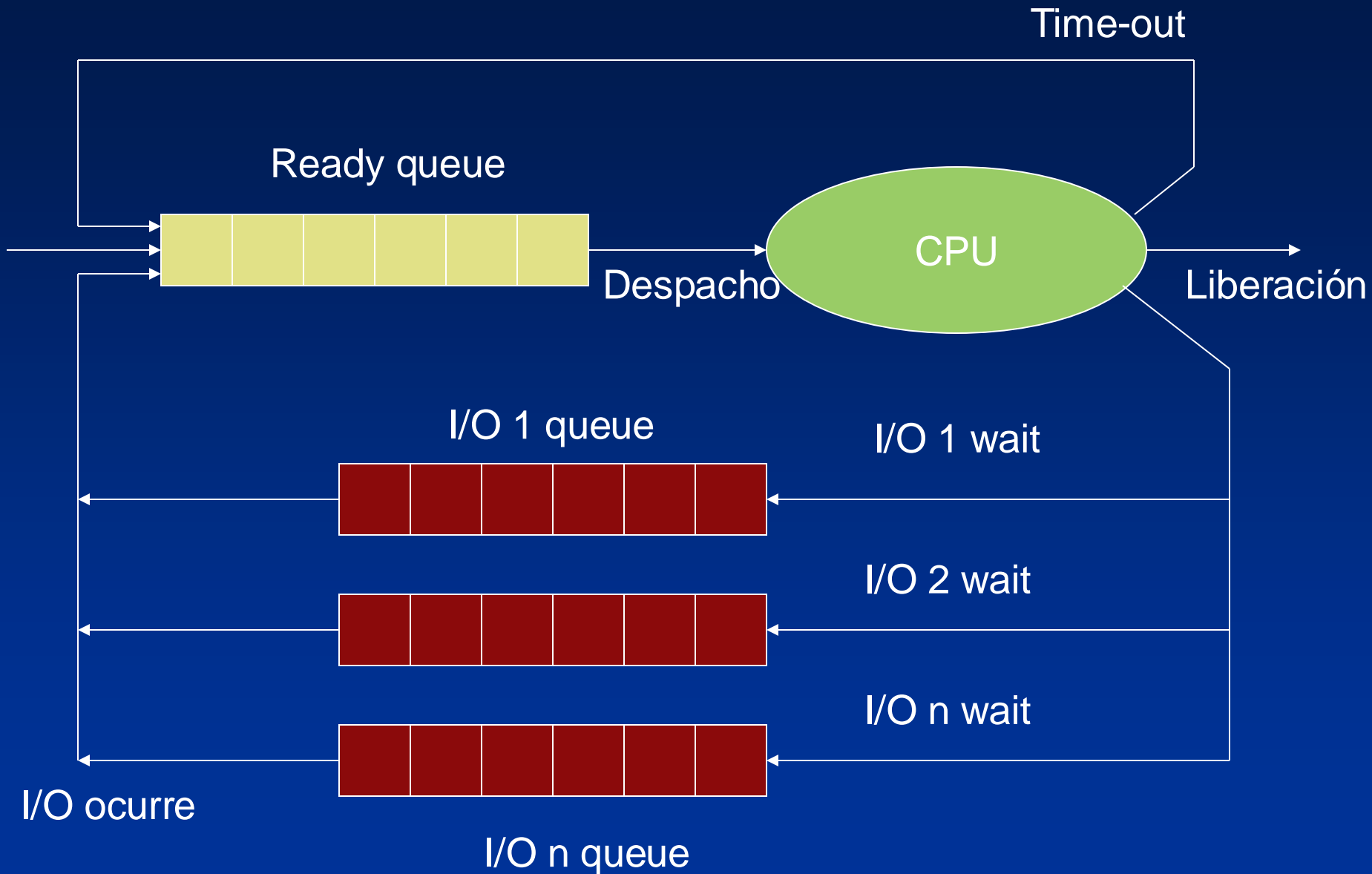
(b)

- Se muestra el cambio de contexto para arquitectura x86-32 en (a) y x86-64 en (b)

Colas de planificación de threads

- Cola de Ready (ready queue)
 - Mantiene el conjunto de todos los threads listos y esperando a ser ejecutados
 - Pueden haber multiples colas de ready, ordenadas por prioridad
- Cola de I/O
 - Mantiene el conjunto de los threads que esperan por un dispositivo de I/O
 - Pueden haber diferentes colas por los diferentes dispositivos
- Threads pueden migrar entre las diferentes colas

Ready Queue y las I/O Queues







Creación de procesos

- Procesos padre crean procesos hijo, que crean otros procesos, que forman un árbol de procesos
 - Procesos arrancan con un thread inicial
- Diversos modelos de compartir recursos
 - Padres e hijos comparten todos los recursos
 - Hijos comparten un subconjunto de los recursos del padre
 - Padres e hijos no comparten recursos
- Ejecución
 - Los threads del padre e hijo ejecutan concurrentemente
 - Padres pueden esperar hasta que el hijo termine

Creación de procesos (Cont.)

- Creación del espacio de direccionamiento (memoria)
 - Hijo puede ser duplicado del padre
 - Puede cargarse un programa diferente en el espacio del hijo
- Ejemplo de UNIX
 - **fork()** crea nuevos procesos, como una copia del padre
 - **exec()** llamado después de **fork** para reemplazar el espacio de memoria del proceso con un programa nuevo
- Ejemplo de Windows
 - **CreateProcess()** crea un nuevo proceso y carga el programa para ejecución

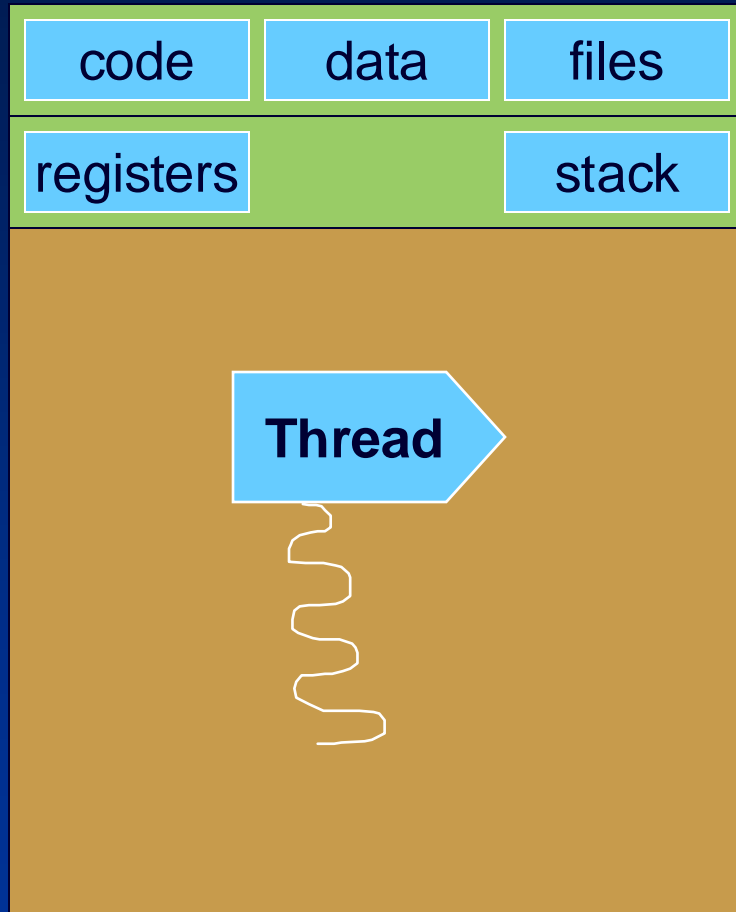
Árbol de procesos en UNIX

Prozess-ID	Prozessname	Threads	% CPU
0	kernel_task	32	1,40
1	▼ init	1	0,00
193	▼ WindowServer	2	4,40
1876	 PowerPoint	2	30,90
1874	Microsoft Word	1	15,40
358	▼  Aktivitäts-Anzeige	2	11,90
361	pmTool	1	2,90
1898	 Bildschirmfoto	3	9,90
356	▼  Terminal	4	1,90
364	▼ login	1	0,00
366	▼ tcsh	1	0,00
415	▼ su	1	0,00
416	tcsh	1	0,00

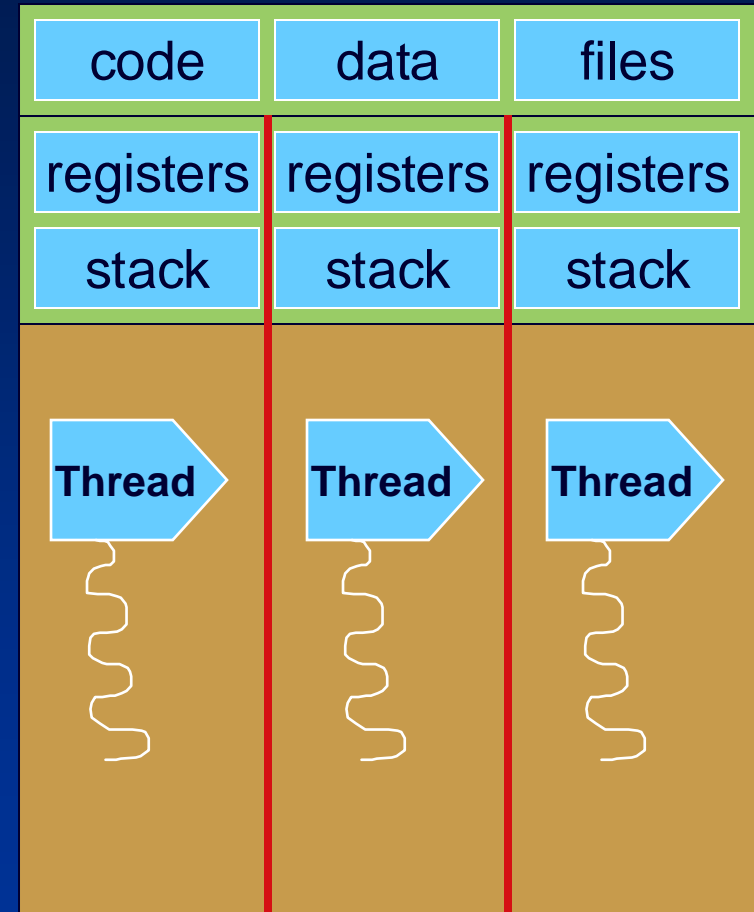
Terminación de procesos

- El último thread dentro de un proceso ejecuta la última sentencia y regresa control al OS (**exit**)
 - Padre puede recibir un código de retorno (via **wait**)
 - Los recursos del proceso son desasignados por el OS
- Padre puede terminar la ejecución de procesos hijos (**kill**)
 - El hijo puede exceder recursos asignados
 - La tarea asignada al hijo ya no se necesita
 - El padre está finalizado
 - Típicamente, el OS no permite a un hijo continuar si su padre termina (dependiendo de las banderas de creación)
 - Terminación en cascada dentro de los grupos de proceso

Procesos de uno o múltiples hilos



single-threaded



multi-threaded

Beneficios del multithreading

- Mejor respuesta
 - Threads dedicados a manejar eventos de usuario
- Compartir recursos de forma simple
 - Todos los threads en un proceso comparten el mismo espacio de direccionamiento (memoria)
- Uso de arquitecturas multiprocesadores
 - Múltiples threads pueden correr en paralelo

Threads en modo usuario

- Administración de threads dentro de una librería de modo usuario
 - Desde el punto de vista del kernel, el proceso es la unidad de planificación de CPU
- Ejemplos
 - POSIX *Pthreads*
 - Mach *C-threads*
 - Solaris *threads*
 - *Fibers* en Windows
 - *Grand Central Dispatch* en OS X

Kernel Threads

- Soportadas por el kernel
 - Thread es la unidad de CPU scheduling
- Ejemplos
 - Windows
 - Solaris
 - OSF/1
 - Linux — Tasks actúan como threads compartiendo estructuras de datos del kernel

Modelos de multithreading

Cómo se mapean los threads de usuario a threads de kernel?

- Muchos a uno

- Muchos threads de modo usuario se mapean a un solo thread de kernel

- Uno a uno

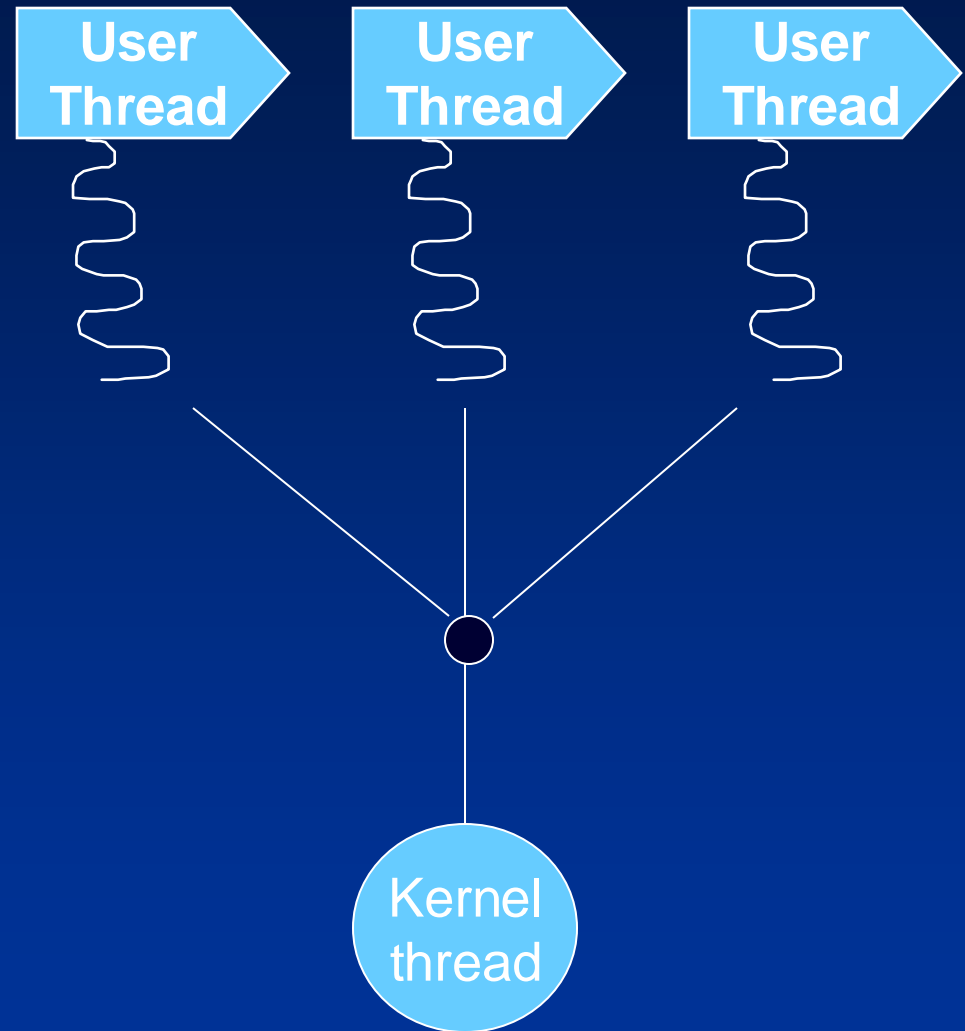
- Cada thread de modo usuario se representa por un kernel thread separado

- Muchos a muchos

- Un conjunto de threads de modo usuario se mapea a un conjunto de threads de kernel

Modelo Muchos a Uno

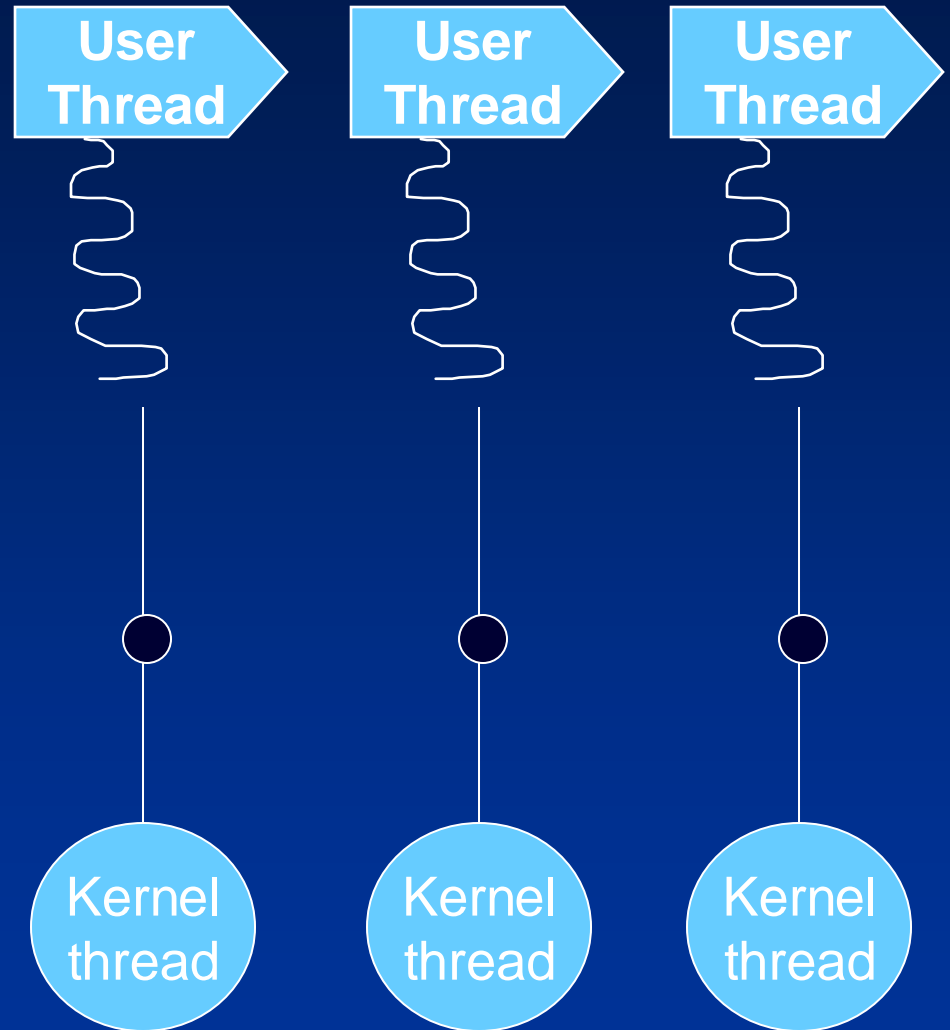
- Usado en sistemas que no soportan threads a nivel de kernel
- Ejemplos:
 - POSIX Pthreads
 - Mach C-Threads
 - Windows Fibers



Modelo Uno a Uno

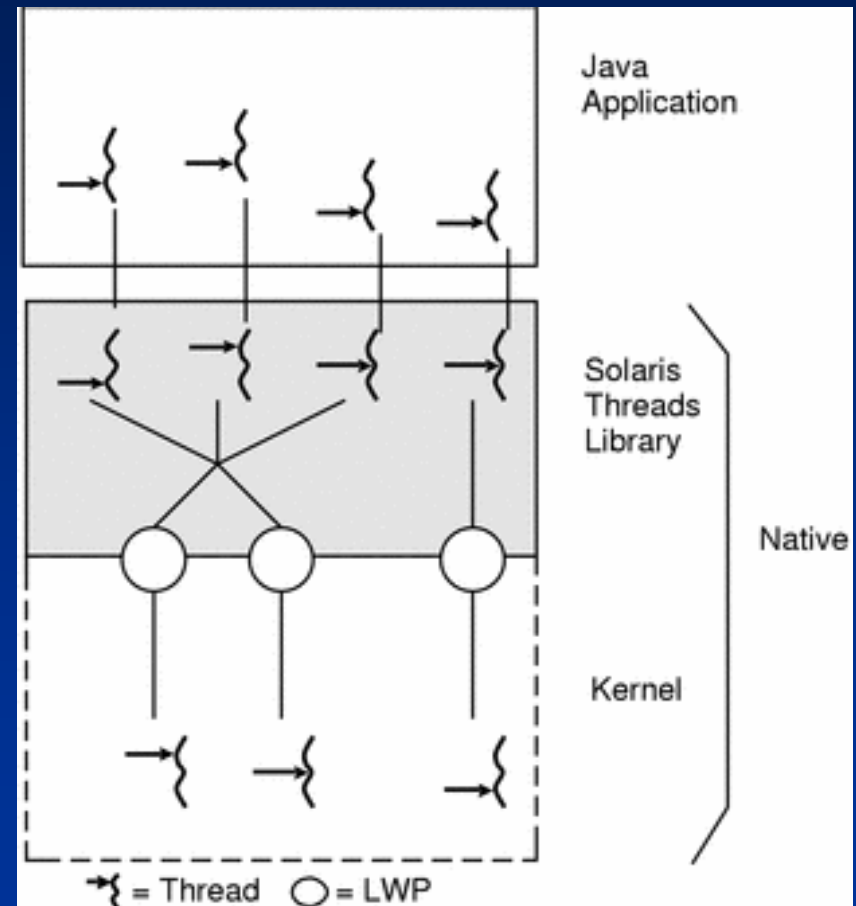
Ejemplos

- Windows threads
- OS/2 threads



Modelo Muchos a Muchos

- Permite que el OS cree un número suficiente de threads de kernel
- Ejemplo
 - Solaris 2



Problemas con el Multithreading

- Semántica de system calls como fork()/exec() or CreateProcess()
- Terminación coordinada
- Manejo de señales
- Manejo global de data, errno, errores
- Data específica al thread
- System calls Reentrantes vs. no-reentrante

Pthreads

- API estándar POSIX (IEEE 1003.1c) para creación de threads y sincronización
- API especifica el funcionamiento de la librería de threads, no cómo implementarlo
- Se ha implementado en muchos OS basados en UNIX, incluyendo OS X (en OS X se usa NSThreads y Pthreads)
- Services for Unix (SFU) / Subsystem for Unix Applications (SUA) implementa PThreads en Windows

Planificación de CPU

Criterios de optimización

- La planificación de CPU se refiere a la forma en la que los procesos son despachados desde la cola de ready hacia el estado de run
- Usa heurísticas para manejar los tradeoffs entre criterios de optimización contradictorios
- Los planificadores (schedulers) se optimizan para cargas de trabajo específicas
 - Interactiva vs. batch
 - I/O-intense vs. compute-intense

Criterios de optimización

- Criterios comunes de optimización:
 - Maximizar el uso de CPU
 - Maximizar el throughput
 - Minimizar el tiempo de turnaround
 - Minimizar el tiempo de espera
 - Minimizar el tiempo de respuesta

Consideraciones básicas de planificación

- Qué invoca el planificador?
- Qué debe asumir el planificador?
- Cuáles son sus metas de optimización?

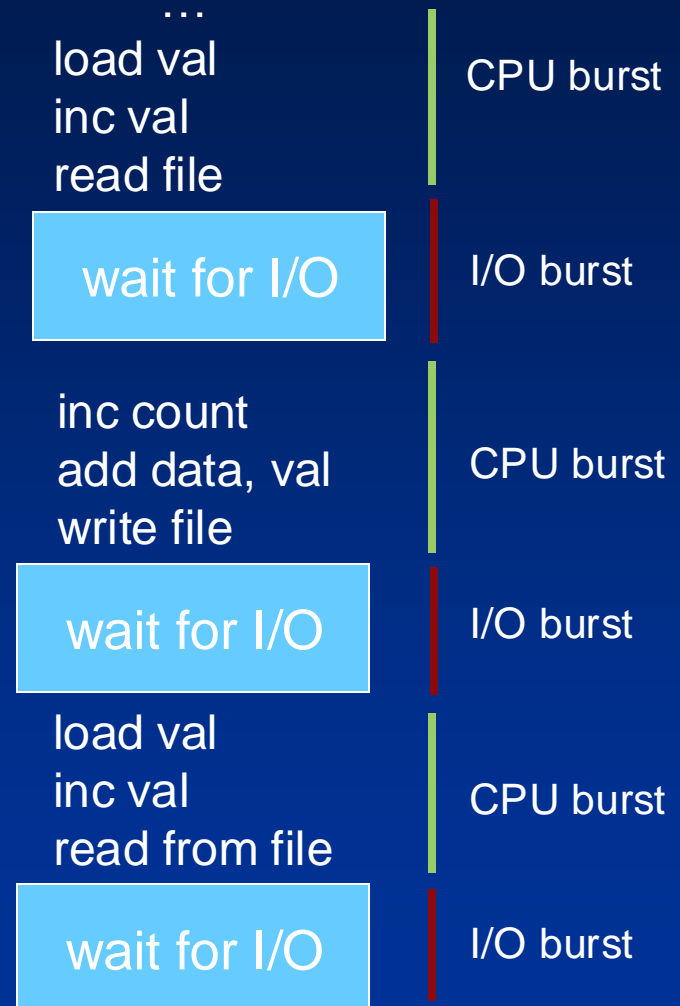
Consideraciones:

- La multiprogramación maximiza la utilización de CPU
- La ejecución de los hilos experimenta ciclos de ráfagas de uso de CPU e I/O
- Planificador debe considerar la distribución de ráfagas de CPU

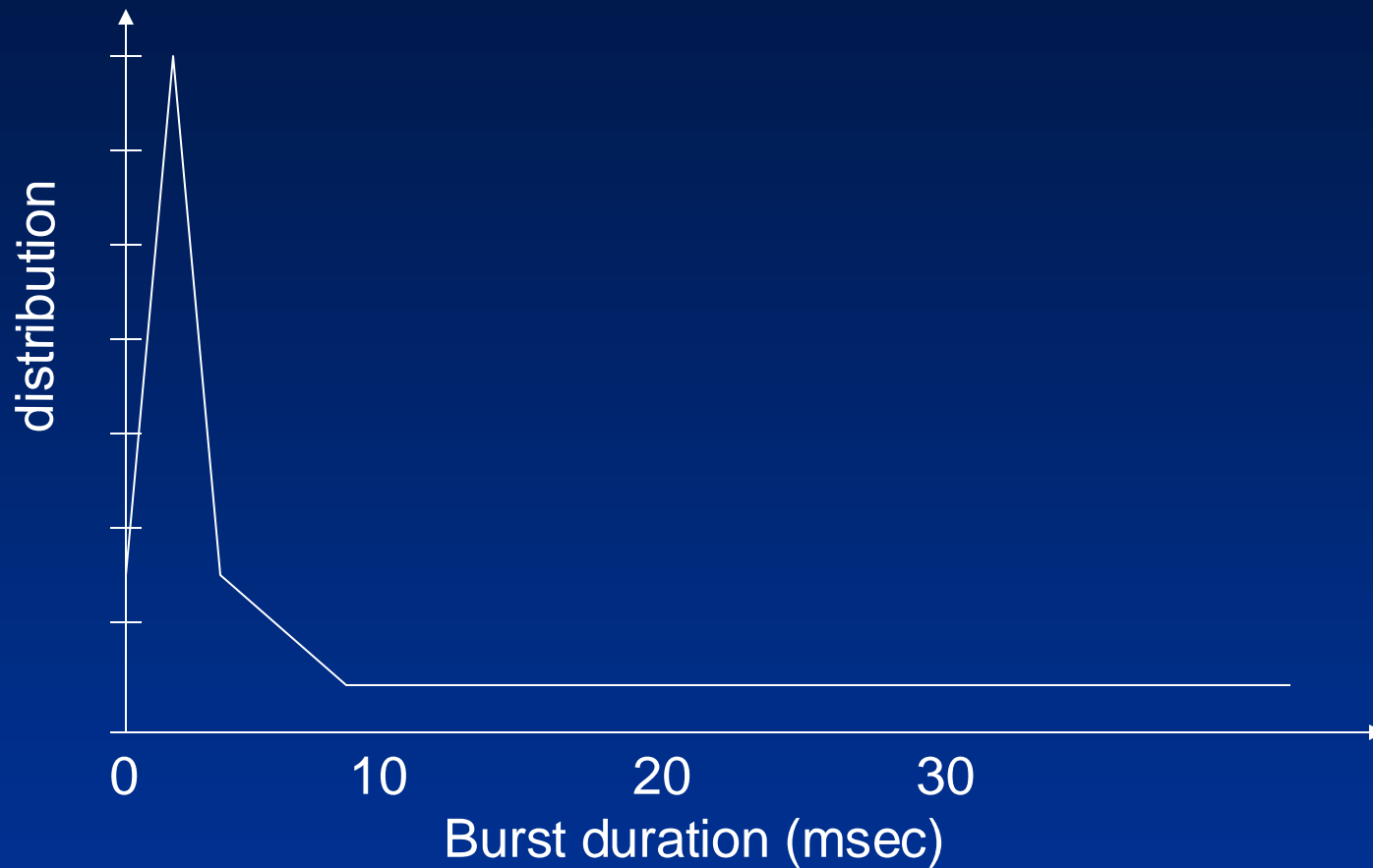
Secuencia alternativa de ciclos de uso de CPU e I/O

Threads pueden describirse como:

- *I/O-bound* – pasa más tiempo haciendo I/O que procesos computacionales, ráfagas cortas de uso de CPU
- *CPU-bound* – pasa más tiempo haciendo procesos computacionales, ráfagas largas de uso de CPU



Histograma de ráfagas de CPU



- Las ráfagas cortas de CPU son típicas
- Datos exactos varían por proceso o computador

Planificadores (schedulers)

- Planificador de largo plazo (job scheduler)
 - Selecciona qué procesos con sus hilos deben ser incorporados en la cola ready
 - Toma en consideración el manejo de memoria (procesos que han sido swapped out)
 - Controla el nivel de multiprogramación
 - No se ejecuta frecuentemente, puede ser lento
- Planificador de corto plazo (CPU scheduler)
 - Selecciona qué threads deben ser ejecutados y asigna CPU
 - Se ejecuta frecuentemente, debe ser rápido
- Windows/Linux no tienen job scheduler

CPU Scheduler

- Selects from among the threads in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a thread:
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches from waiting to ready
 4. Terminates
- Scheduling under 1 and 4 is *nonpreemptive*
- All other scheduling is *preemptive*

Dispatcher

- Dispatcher module gives control of the CPU to the thread selected by the short-term scheduler; this involves:
 - switching context
 - switching to user mode
 - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one thread and start another running.
- Windows scheduling is event-driven
 - No central dispatcher module in the kernel

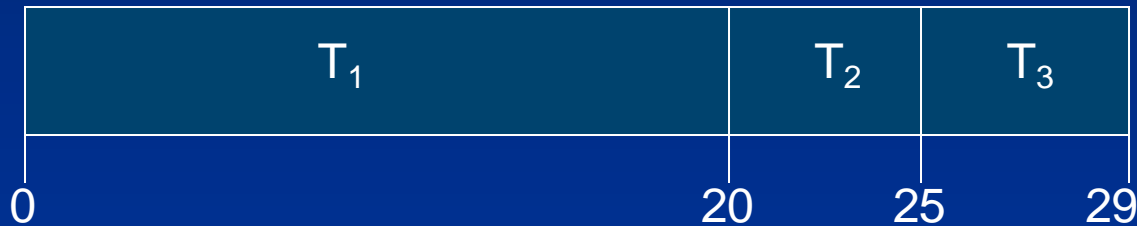
Scheduling Algorithms:

First-In, First-Out (FIFO)

- Also known as First-Come, First-Served (FCFS)

<u>Thread</u>	<u>Burst Time</u>
T_1	20
T_2	5
T_3	4

- Suppose that the threads arrive in the order: T_1 , T_2 , T_3
 - The Gantt Chart for the schedule is:



- Waiting time for $T_1 = 0$; $T_2 = 20$; $T_3 = 25$
- Average waiting time: $(0 + 20 + 25)/3 = 15$
- Convoy effect*: short thread behind long threads experience long waiting time

FIFO Scheduling (Cont.)

Suppose that the threads arrive in the order

$$T_2, T_3, T_1.$$

- The Gantt chart for the schedule is:



- Waiting time for $T_1 = 9$; $T_2 = 0$; $T_3 = 5$
- Average waiting time: $(9 + 0 + 5)/3 = 4.66$
- Much better than previous case

Scheduling Algorithms:

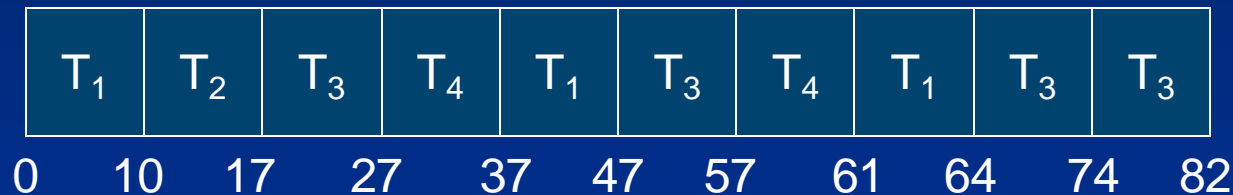
Round Robin (RR)

- Preemptive version of FIFO scheduling algorithm
 - Each thread gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
 - After this time has elapsed, the thread is preempted and added to the end of the ready queue
 - Each of n ready thread gets $1/n$ of the CPU time in chunks of at most quantum q time units at once
 - Of n ready threads, no one waits more than $(n-1)q$ time units
- Performance
 - q large \Rightarrow FIFO
 - q small $\Rightarrow q$ must be large with respect to context switch, otherwise overhead is too high

Example of RR with Quantum = 10

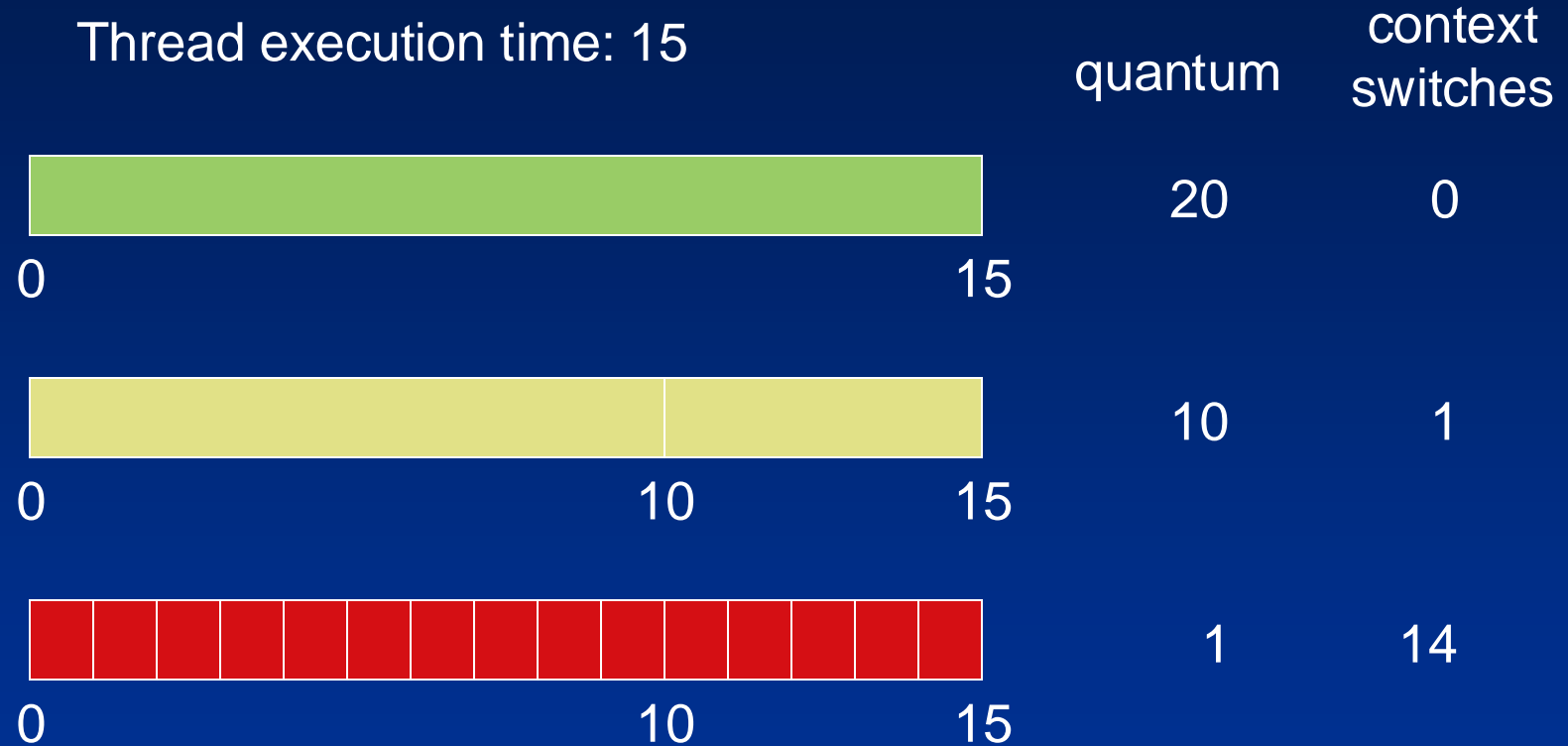
<u>Thread</u>	<u>Burst Time</u>
T_1	23
T_2	7
T_3	38
T_4	14

- Assuming all threads have same priority, the Gantt chart is:



- Round-Robin favors CPU-intense over I/O-intense threads
- Priority-elevation after I/O completion can provide a compensation
- Windows uses Round-Robin with a priority-elevation scheme

Shorter quantum yields more context switches



- Longer quantum yields shorter average turnaround times

Scheduling Algorithms:

Priority Scheduling

- A priority number (integer) is associated with each thread
- The CPU is allocated to the thread with the highest priority
 - Preemptive
 - Non-preemptive

Priority Scheduling - Starvation

Starvation is a problem:

- low priority threads may never execute

Solutions

1) Decreasing priority & aging: the Unix approach

- Decrease priority of CPU-intense threads
- Exponential averaging of CPU usage to slowly increase priority of blocked threads

2) Priority Elevation: the Windows/VMS approach

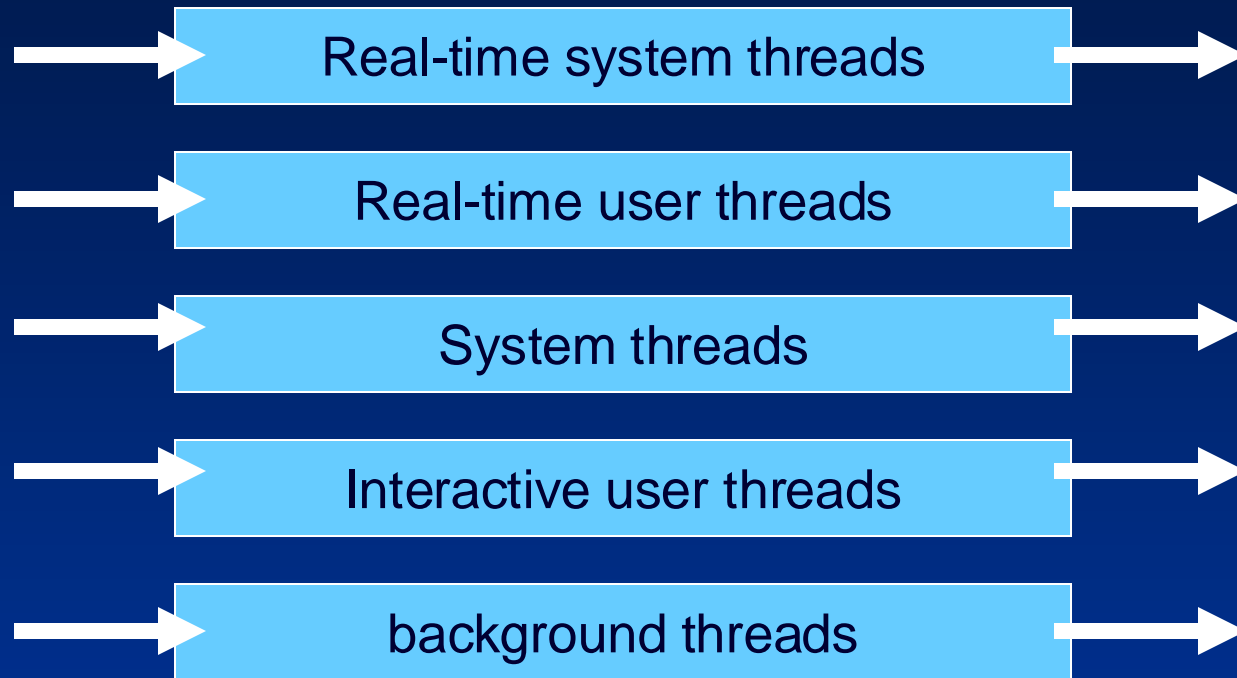
- Increase priority of a thread on I/O completion
- System gives starved threads an extra burst

Multilevel Queue

- Ready queue is partitioned into separate queues:
 - Real-time (system, multimedia)
 - Interactive
- Queues may have different scheduling algorithm,
 - Real-Time – RR
 - Interactive – RR + priority-elevation + quantum stretching
- Scheduling must be done between the queues
 - Fixed priority scheduling (i.e., serve all from real-time threads then from interactive)
 - Possibility of starvation
 - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its threads;
 - CPU reserves

Multilevel Queue Scheduling

High priority



Low priority

- Windows uses strict Round-Robin for real-time threads
- Priority-elevation can be disabled for non-RT threads

Further Reading

- Abraham Silberschatz, Peter B. Galvin, Operating System Concepts, John Wiley & Sons, 6th Ed., 2003;
 - Chapter 4 - Processes
 - Chapter 5 - Threads
 - Chapter 6 - CPU Scheduling
- Mark E. Russinovich and David A. Solomon, Microsoft Windows Internals, 4th Edition, Microsoft Press, 2004.
 - Chapter 6 - Processes, Thread, and Jobs (from pp. 289)