Unit 4: Scheduling and Dispatch

4.1. The Concept of Processes and Threads

Roadmap for Section 4.1.

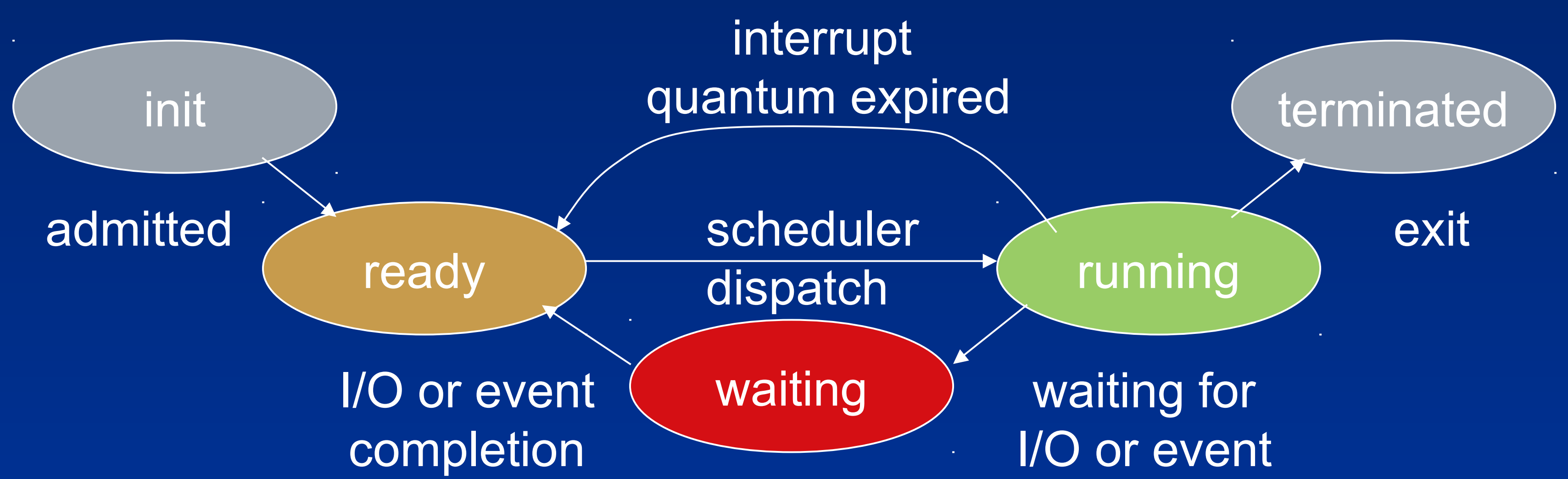
- The Process Concept
- Thread States
- Context Switches
- Approaches to CPU Scheduling
- Multithreading Models

Process Concept

- An operating system executes programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Process – a program in execution
  - Process execution must progress sequentially
- A process includes:
  - CPU state (one or multiple threads)
  - Text & data section
  - Resources such as open files, handles, sockets
- Traditionally, processes used to be units of scheduling (i.e. no threads)
  - However, like most modern operating systems, Windows schedules threads
  - Our discussion assumes thread scheduling

Thread States

- Five-state diagram for thread scheduling:
  - **init**: The thread is being created
  - **ready**: The thread is waiting to be assigned to a CPU
  - **running**: The thread's instructions are being executed
  - **waiting**: The thread is waiting for some event to occur
  - **terminated**: The thread has finished execution

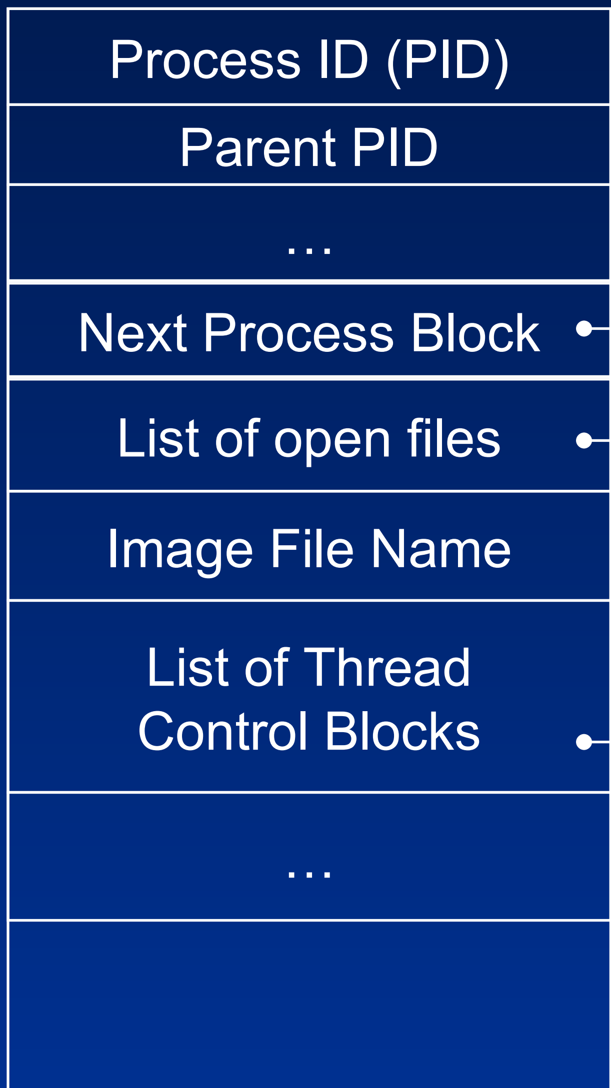


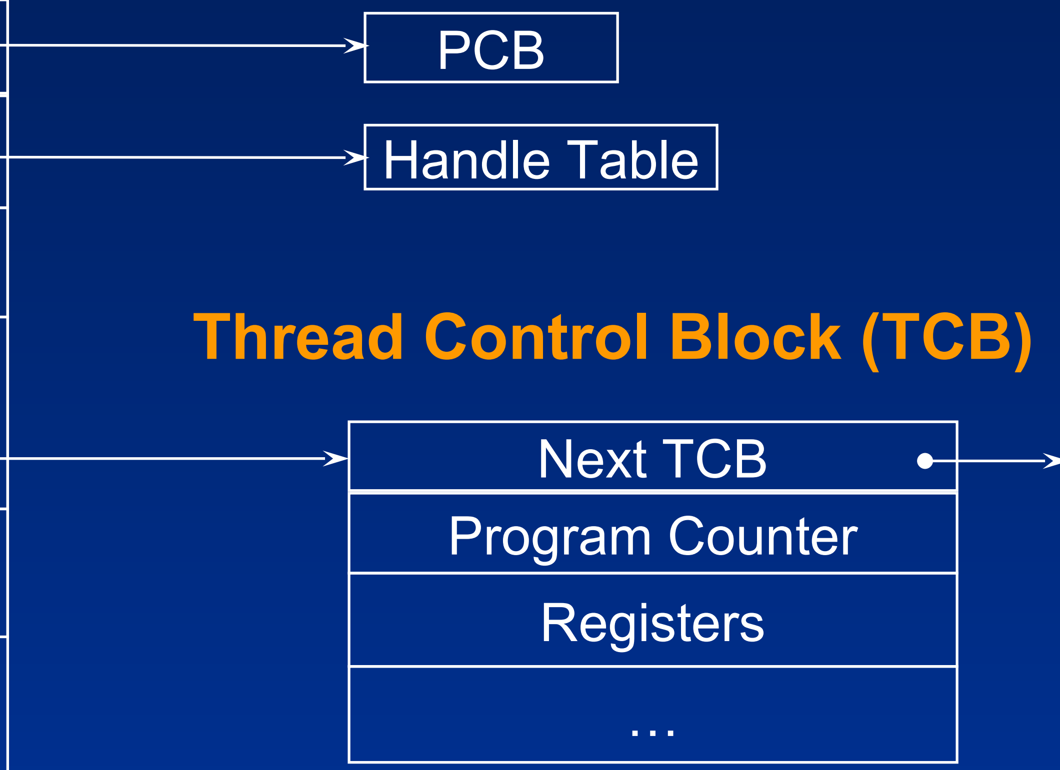
Process and Thread Control Blocks

- Information associated with each process: Process Control Block (PCB)
  - Memory management information
  - Accounting information
  - Process-global vs. thread-specific
- Information associated with each thread: Thread Control Block (TCB)
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Pending I/O information

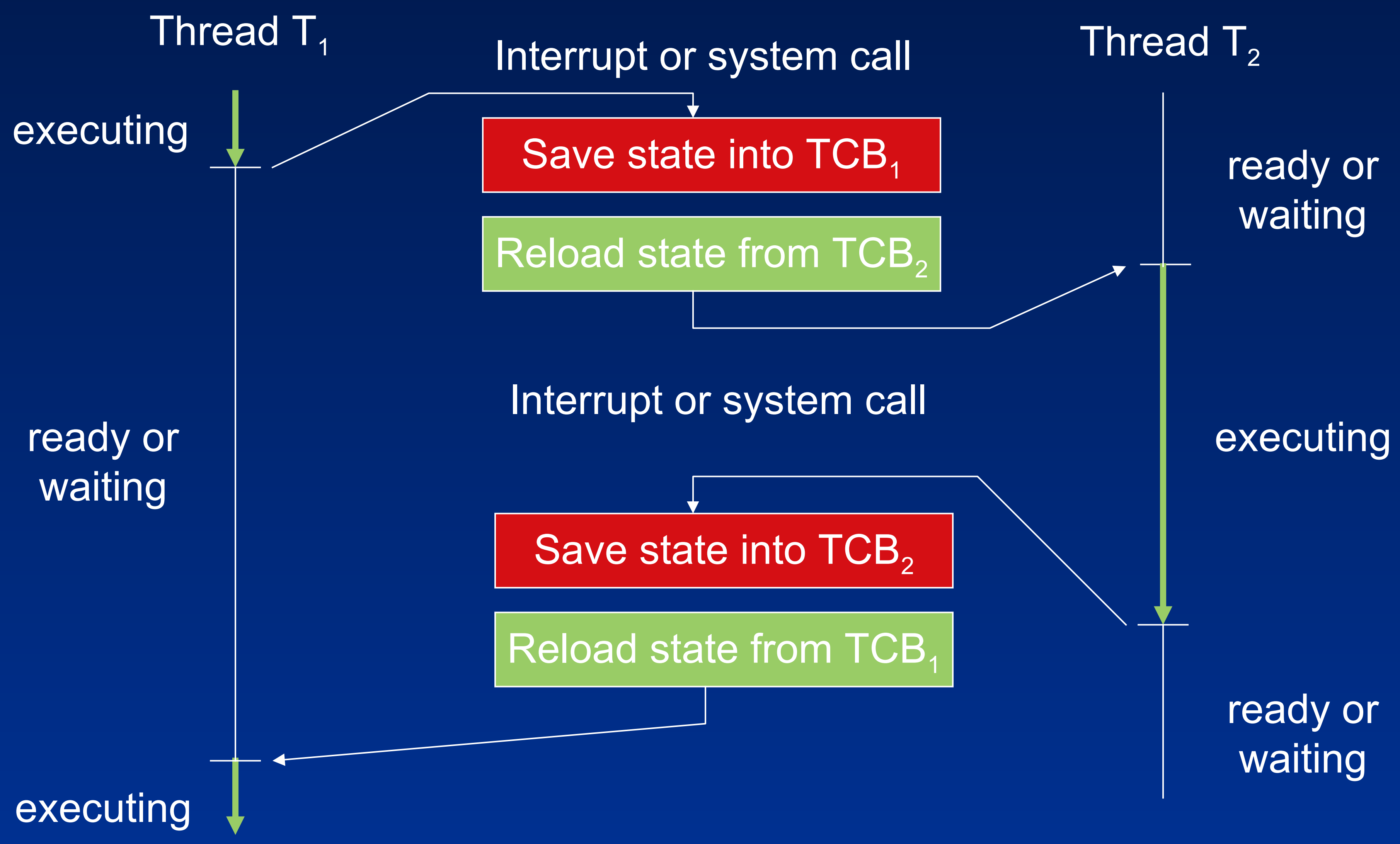
Process Control Block (PCB)

- This is an abstract view
- Windows implementation of PCB is split in multiple data structures





CPU Switch from Thread to Thread



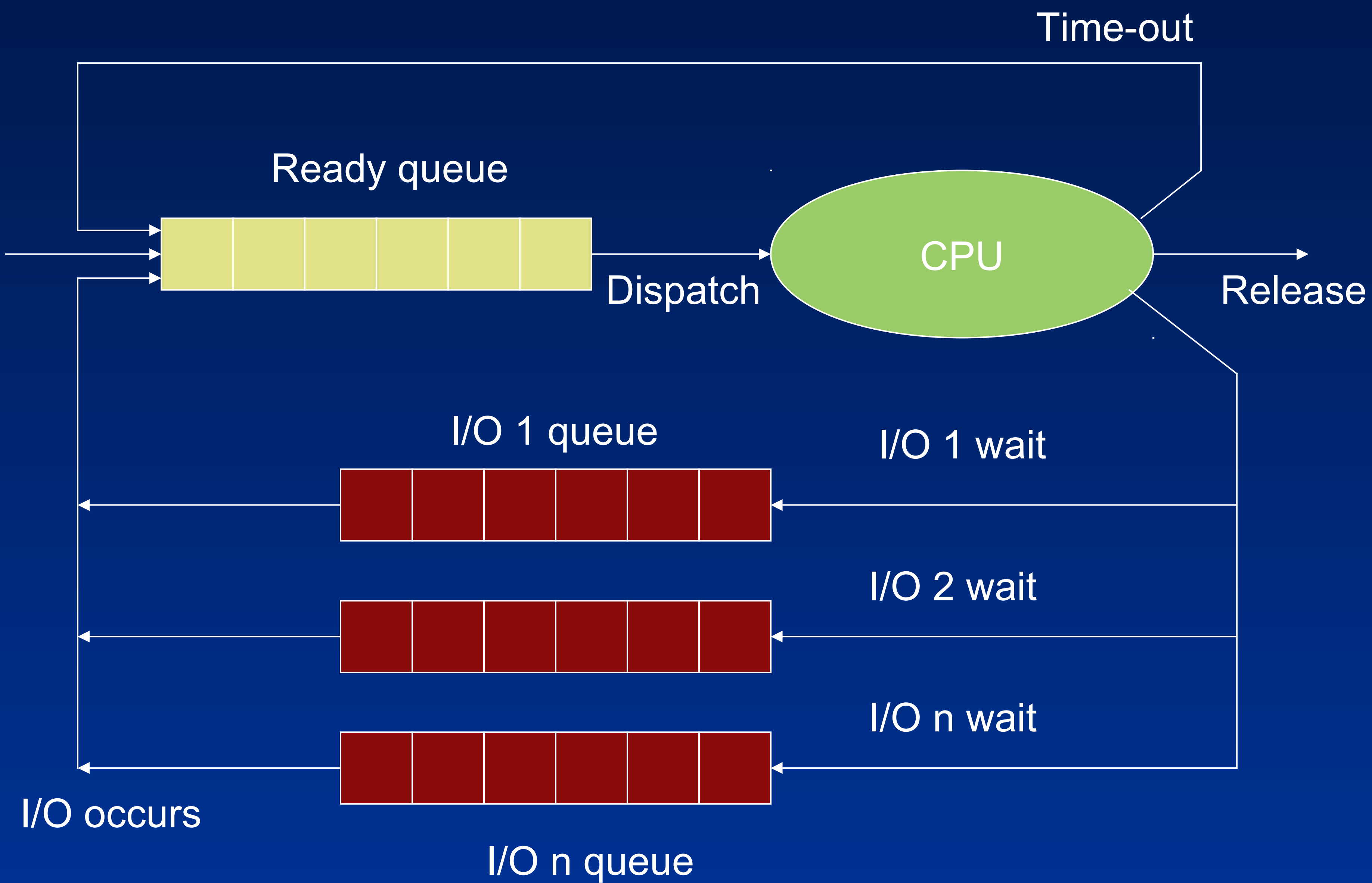
Context Switch

- When CPU switches to another thread, the system must save the state of the old thread and load the saved state for the new thread
- Context-switch time is overhead; the system does no useful work while switching
- Thread context-switching can be implemented in kernel or user mode
- Interaction with memory management (MMU) is required when switching between threads in different processes

Thread Scheduling Queues

- Ready queue
  - Maintains set of all threads ready and waiting to execute
  - There might be multiple ready queues, sorted by priorities
- Device queue
  - Maintains set of threads waiting for an I/O device
  - There might be multiple queues for different devices
- Threads migrate between the various queues

Ready Queue and I/O Device Queues



Optimization Criteria

- CPU scheduling uses heuristics to manage the tradeoffs among contradicting optimization criteria.
- Schedulers are optimized for certain workloads
  - Interactive vs. batch processing
  - I/O-intense vs. compute-intense
- Common optimization criteria:
  - Maximize CPU utilization
  - Maximize throughput
  - Minimize turnaround time
  - Minimize waiting time
  - Minimize response time

Basic Scheduling Considerations

- What invokes the scheduler?
- Which assumptions should a scheduler rely on?
- What are its optimization goals?

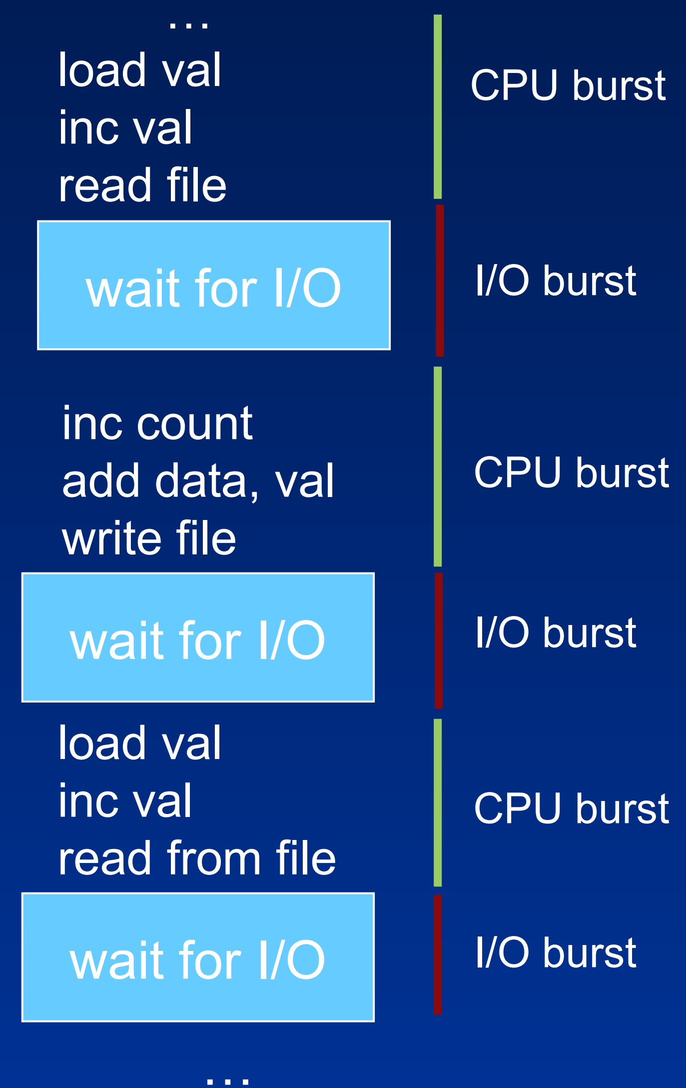
Rationale:

- Multiprogramming maximizes CPU utilization
- Thread execution experiences cycles of compute- and I/O-bursts
- Scheduler should consider CPU burst distribution

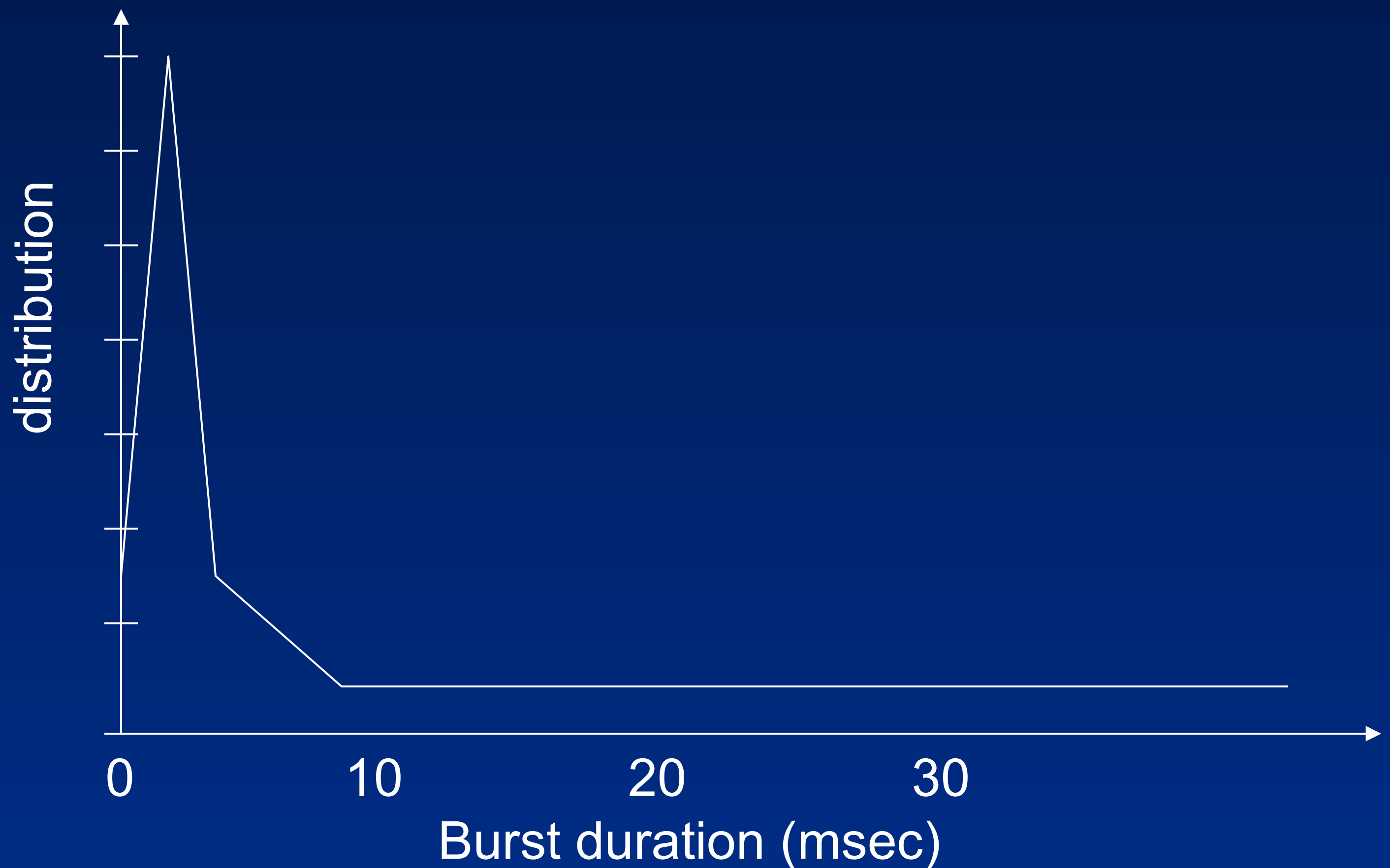
Alternating Sequence of CPU and I/O Bursts

Threads can be described as either:

- I/O-*bound* – spends more time doing I/O than computations, many short CPU bursts
- *CPU-bound* – spends more time doing computations; few very long CPU bursts



Histogram of CPU-burst Times



- Many short CPU bursts are typical
- Exact figures vary greatly by process and computer

Schedulers

- Long-term scheduler (or job scheduler)
  - Selects which processes with their threads should be brought into the ready queue
  - Takes memory management into consideration (swapped-out processes)
  - Controls degree of multiprogramming
  - Invoked infrequently, may be slow
- Short-term scheduler (or CPU scheduler)
  - Selects which threads should be executed next and allocates CPU
  - Invoked frequently, must be fast
- Windows has no dedicated long-term 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 thread 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)

Thread	Burst Time
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.)

Now 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

Thread	Burst Time
T_1	23
T_2	7
T_3	38
T_4	14

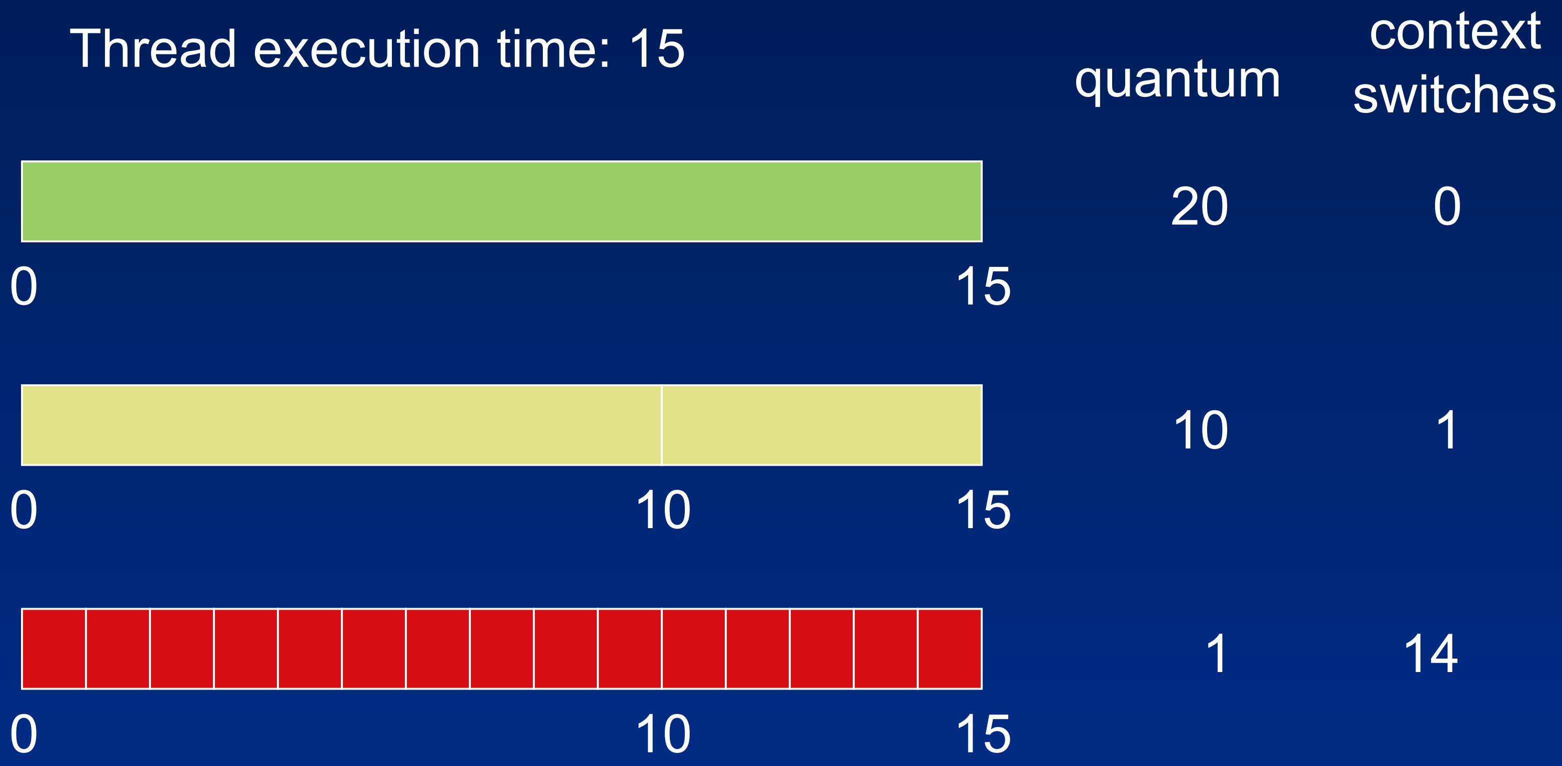
- Assuming all threads have same priority, the Gantt chart is:

T_1	T_2	T_3	T_4	T_1	T_3	T_4	T_1	T_3	T_3

0 10 17 27 37 47 57 61 64 74 82

- 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:

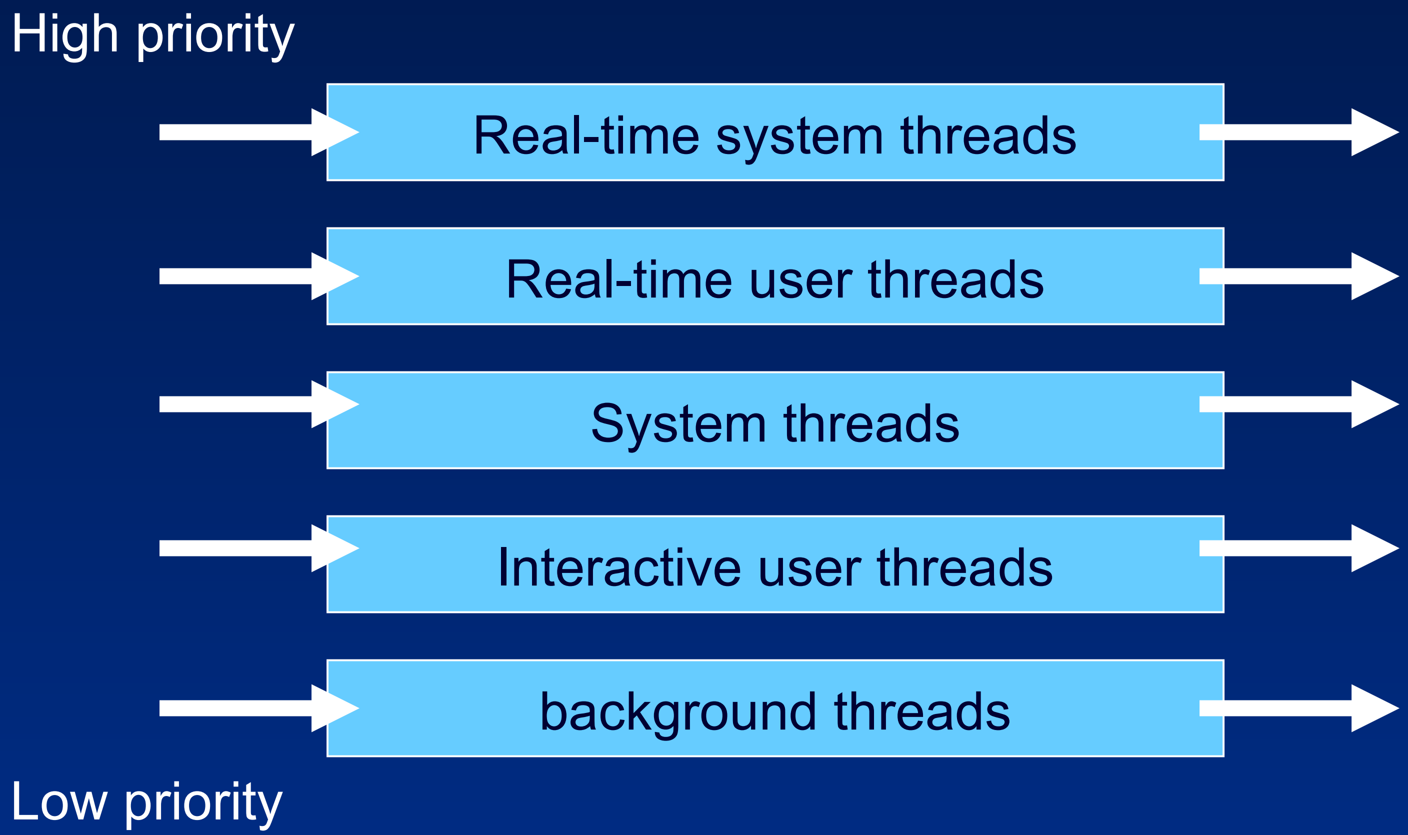
1) Decreasing priority & aging: the Unix approach
   - Decrease priority of CPU-intense threads
   - Exponential averaging of CPU usage to slowly increase priority of blocked threads

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 Scheduling

- Ready queue is partitioned into separate queues, e.g.:
  - Real-time (system, multimedia)
  - Interactive
- Queues may have different scheduling algorithm, e.g.:
  - Real-Time – RR
  - Interactive – RR + priority-elevation + quantum stretching
- Scheduling must be done between the queues – solutions:
  - 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



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

Process Creation

- Parent process creates children processes, which create other processes, forming a tree of processes
  - Processes start with one initial thread
- Resource sharing models
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent's and children' threads execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- How to set up an address space
  - Child can be a duplicate of parent
  - Child may have a new program loaded into it
- UNIX example
  - **fork()** system call creates new process
  - **exec()** system call used after a **fork** to replace the process' memory space with a new program
- Windows example
  - **CreateProcess()** system call create new process and loads new program for execution

Processes Tree on a UNIX System

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

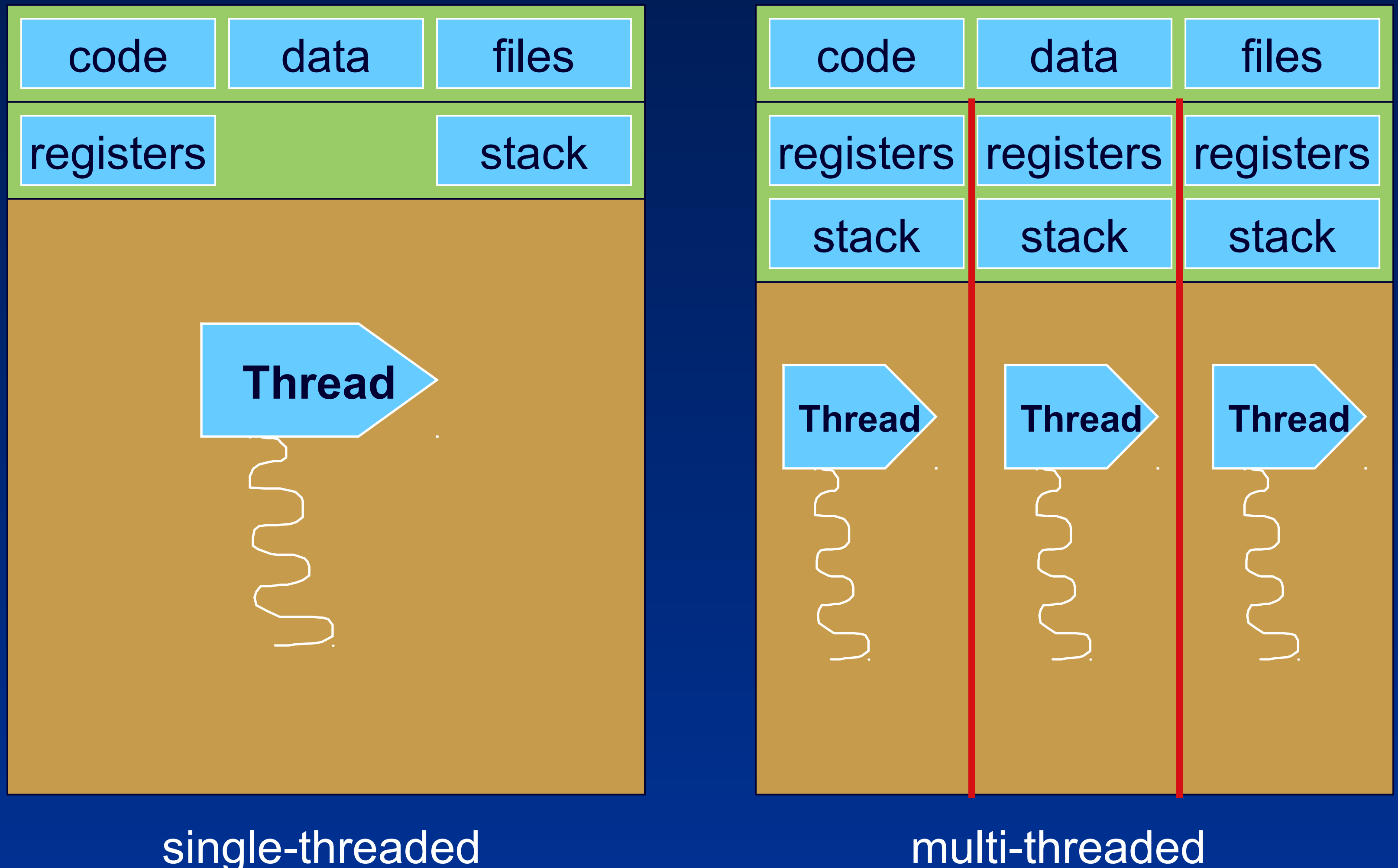
Processes Tree on a Windows System

Process	PID	CPU	Private Bytes	Working Set	Description	Company Name
System Idle Process	0	33.71	52 K	8 K		
System	4	4.83	140 K	100 K		
Interrupts	n/a	6.49	0 K	0 K	Hardware Interrupts and DPCs	
smss.exe	484		440 K	936 K	Windows Session Manager	Microsoft Corporation
Memory Compression	4116	< 0.01	316 K	100.328 K		
csrss.exe	648		1.664 K	4.768 K	Client Server Runtime Process	Microsoft Corporation
wininit.exe	732		1.284 K	6.004 K	Windows Start-Up Application	Microsoft Corporation
services.exe	804		3.976 K	7.268 K	Services and Controller app	Microsoft Corporation
lsass.exe	812	0.01	5.004 K	12.008 K	Local Security Authority Process	Microsoft Corporation
fontdrvhost.exe	992		1.288 K	3.624 K	Usermode Font Driver Host	Microsoft Corporation
csrss.exe	748	1.00	2.232 K	4.852 K	Client Server Runtime Process	Microsoft Corporation
winlogon.exe	856		2.076 K	9.084 K	Windows Logon Application	Microsoft Corporation
fontdrvhost.exe	996	10.21	2.056 K	6.168 K	Usermode Font Driver Host	Microsoft Corporation
dwm.exe	1100	3.19	47.904 K	37.772 K	Desktop Window Manager	Microsoft Corporation
explorer.exe	3028	0.83	39.352 K	69.940 K	Windows Explorer	Microsoft Corporation
MSASCuiL.exe	7252		1.800 K	9.016 K	Windows Defender notification icon	Microsoft Corporation
TOTALCMD64.EXE	7260	< 0.01	14.144 K	31.292 K	Total Commander	Ghisler Software GmbH
FoxitReaderPortable.exe	5876	0.02	37.304 K	3.936 K	Foxit Reader Portable (PortableApps.com Launcher)	PortableApps.com
FoxitReader.exe	7896	0.01	47.312 K	82.560 K	Foxit Reader 8.3	Foxit Software Inc.
notepad.exe	248		2.636 K	15.908 K	Notepad	Microsoft Corporation
LibreOfficePortable.exe	7580	0.01	38.720 K	3.628 K	LibreOffice Portable (PortableApps.com Launcher)	PortableApps.com
soffice.exe	6200		1.984 K	7.696 K	LibreOffice	The Document Foundation
soffice.bin	3924	24.85	103.448 K	155.724 K	LibreOffice	The Document Foundation
splwow64.exe	5188		4.172 K	14.844 K	Print driver host for applications	Microsoft Corporation
AmIcoSinglun64.exe	7368		1.956 K	9.248 K	Single LUN Icon Utility for VID 058F PID 6366	AlcorMicro Co., Ltd.
FirefoxPortable.exe	8004	0.01	35.340 K	8.660 K	Mozilla Firefox, Portable Edition	PortableApps.com
firefox.exe	7108	1.49	1.026.736 K	500.556 K	Firefox	Mozilla Corporation
procexp.exe	7156		3.524 K	10.584 K	Sysinternals Process Explorer	Sysinternals - www.sysinter...
procexp64.exe	744	5.06	20.224 K	40.048 K	Sysinternals Process Explorer	Sysinternals - www.sysinter...
PortableAppsPlatform.exe	4948	0.09	7.164 K	22.936 K	PortableApps.com Platform	PortableApps.com
IrfanViewPortable.exe	820	0.01	37.572 K	3.456 K	IrfanView Portable (PortableApps.com Launcher)	PortableApps.com
i_view32.exe	6640	1.78	2.588 K	11.896 K	IrfanView 32-bit	Irfan Skiljan
HControlUser.exe	7480		928 K	4.948 K	HControlUser	ASUS
ATKOSD2.exe	7540		1.552 K	8.312 K	ATKOSD2	ASUS
DMedia.exe	7584		1.104 K	5.800 K	ATK Media	ASUS
NVIDIA Web Helper.exe	424	0.08	34.536 K	1.748 K	NVIDIA Web Helper Service	Node.js
conhost.exe	3244		1.380 K	248 K	Console Window Host	Microsoft Corporation

Process Termination

- Last thread inside a process executes last statement and returns control to operating system (**exit**)
  - Parent may receive return code (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (via **kill**) – reasons:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - Parent is exiting
    - Operating system typically does not allow child to continue if its parent terminates (depending on creation flags)
    - Cascading termination inside process groups

Single and Multithreaded Processes



Benefits of Multithreading

- Higher Responsiveness
  - Dedicated threads for handling user events
- Simpler Resource Sharing
  - All threads in a process share same address space
- Utilization of Multiprocessor Architectures
  - Multiple threads may run in parallel

User Threads

- Thread management within a user-level threads library
  - Process is still unit of CPU scheduling from OS kernel perspective
- Examples
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*
  - *Fibers* on Windows

Kernel Threads

- Supported by the Kernel
  - Thread is unit of CPU scheduling
- Examples
  - Windows
  - Solaris
  - OSF/1
  - Linux - Tasks can act like threads by sharing kernel data structures

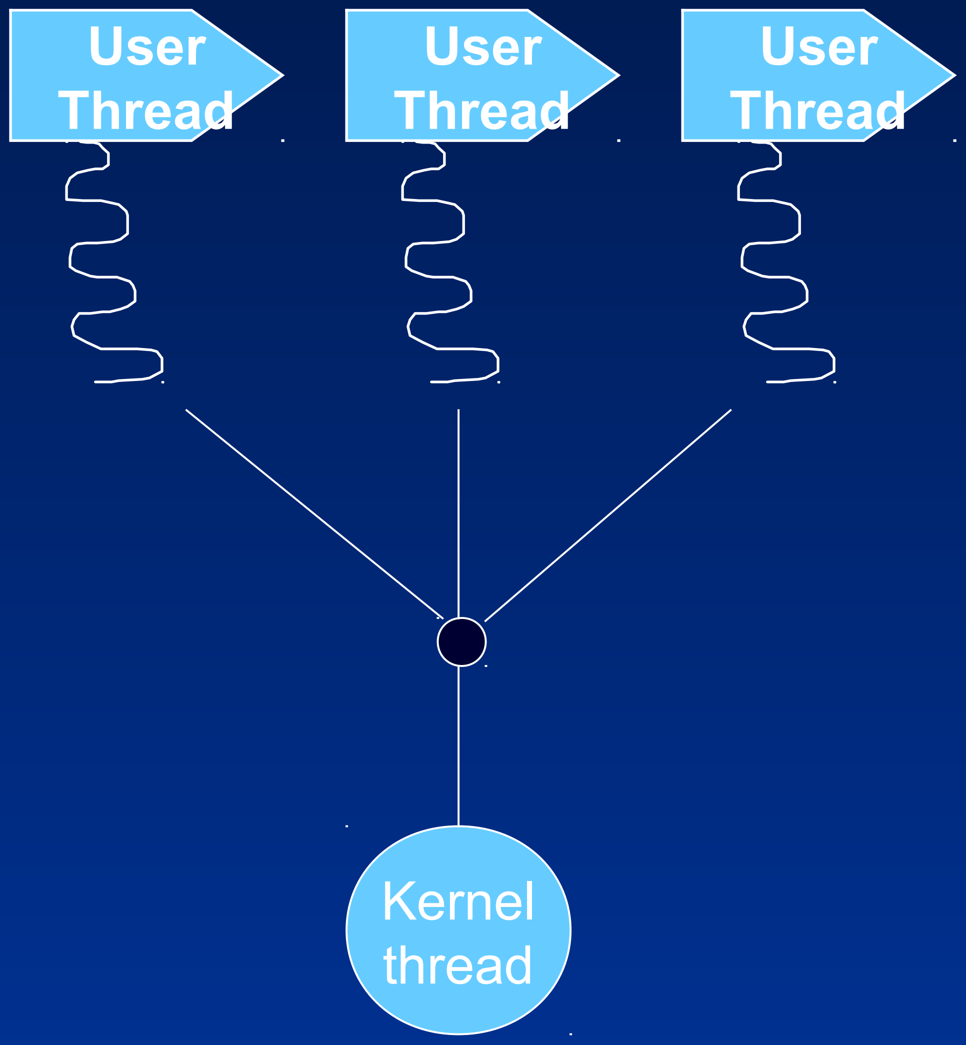
Multithreading Models

How are user-level threads mapped on kernel threads?

- Many-to-One
  - Many user-mode threads are mapped on a single kernel thread
- One-to-One
  - Each user-mode thread is represented by a separate kernel thread
- Many-to-Many
  - A set of user-mode threads is being mapped on another set of kernel threads

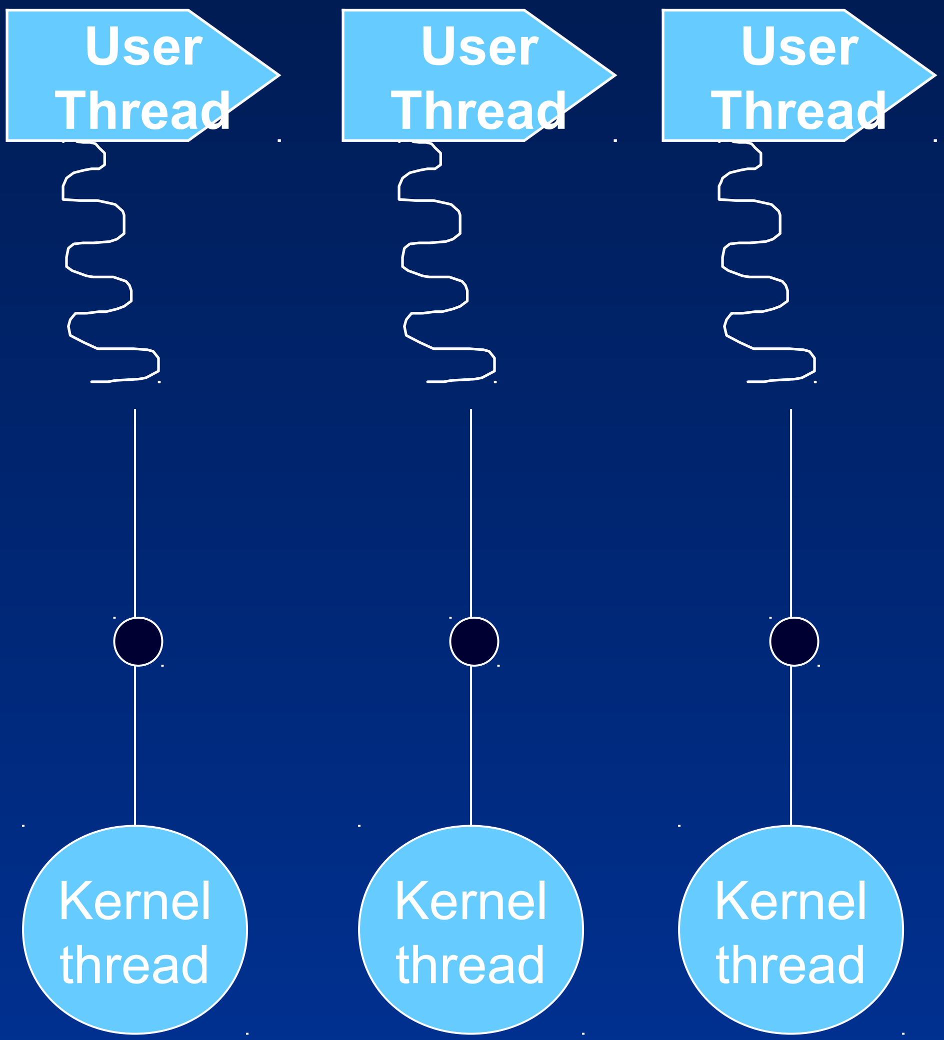
Many-to-One Model

- Many user-level threads are mapped to a single kernel thread
- Used on systems that do not support kernel threads
- Example:
  - POSIX Pthreads
  - Mach C-Threads
  - Windows Fibers



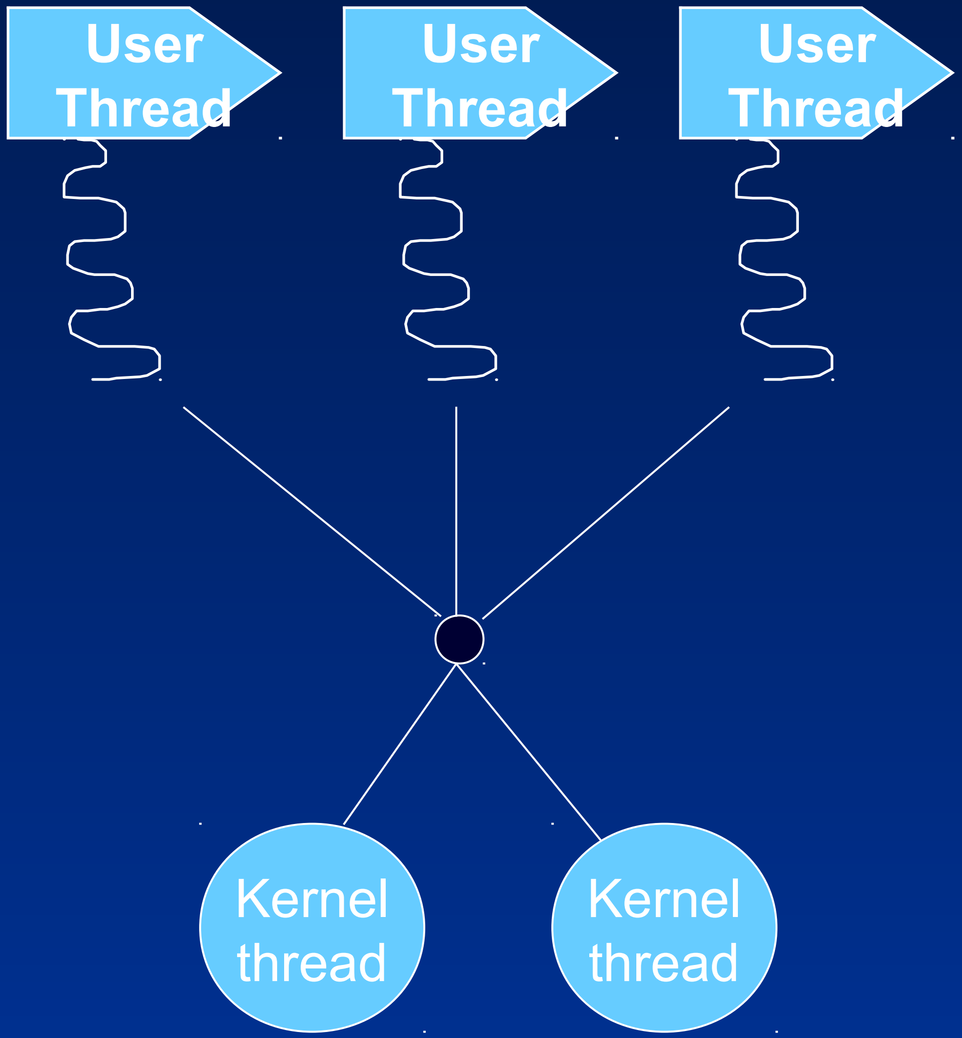
One-to-One Model

- Each user-level thread maps to kernel thread
- Examples
  - Windows threads
  - OS/2 threads



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.
- Example
  - Solaris 2



Problems with Multithreading

- Semantics of fork()/exec() or CreateProcess() system calls
- Coordinated termination
- Signal handling
- Global data, errno, error handling
- Thread specific data
- Reentrant vs. non-reentrant system calls

Pthreads

- a POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, not an implementation
- Implemented on many UNIX operating systems
- Services for Unix (SFU) implemented Pthreads on Windows

Further Reading

- Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, "*Operating System Concepts*", 9th Edition, John Wiley & Sons, 2013.
  - Chapter 3 – Processes
  - Chapter 4 – Threads
  - Chapter 6 – CPU Scheduling
- Pavel Yosifovich, Alex Ionescu, et al., "Windows Internals", 7th Edition, Microsoft Press, 2017.
  - Chapter 3 – Processes and jobs (from pp. 156)
  - Chapter 4 – Threads (from pp. 275)