Betriebssysteme

Process Management

Lehrstuhl Systemarchitektur

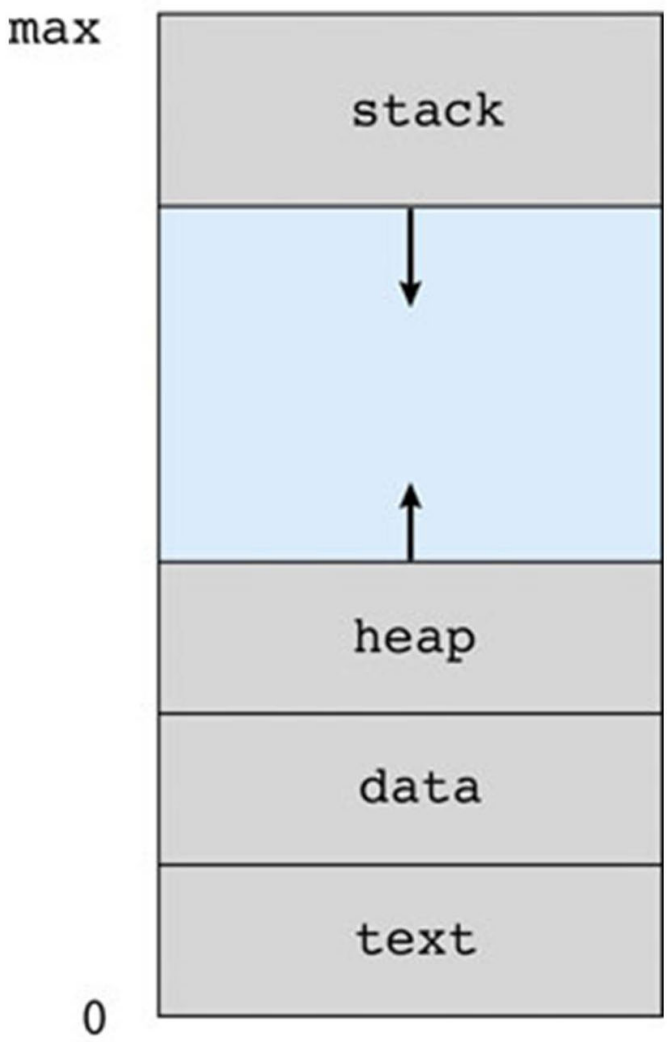
WS 2009/2010

Process Concept

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

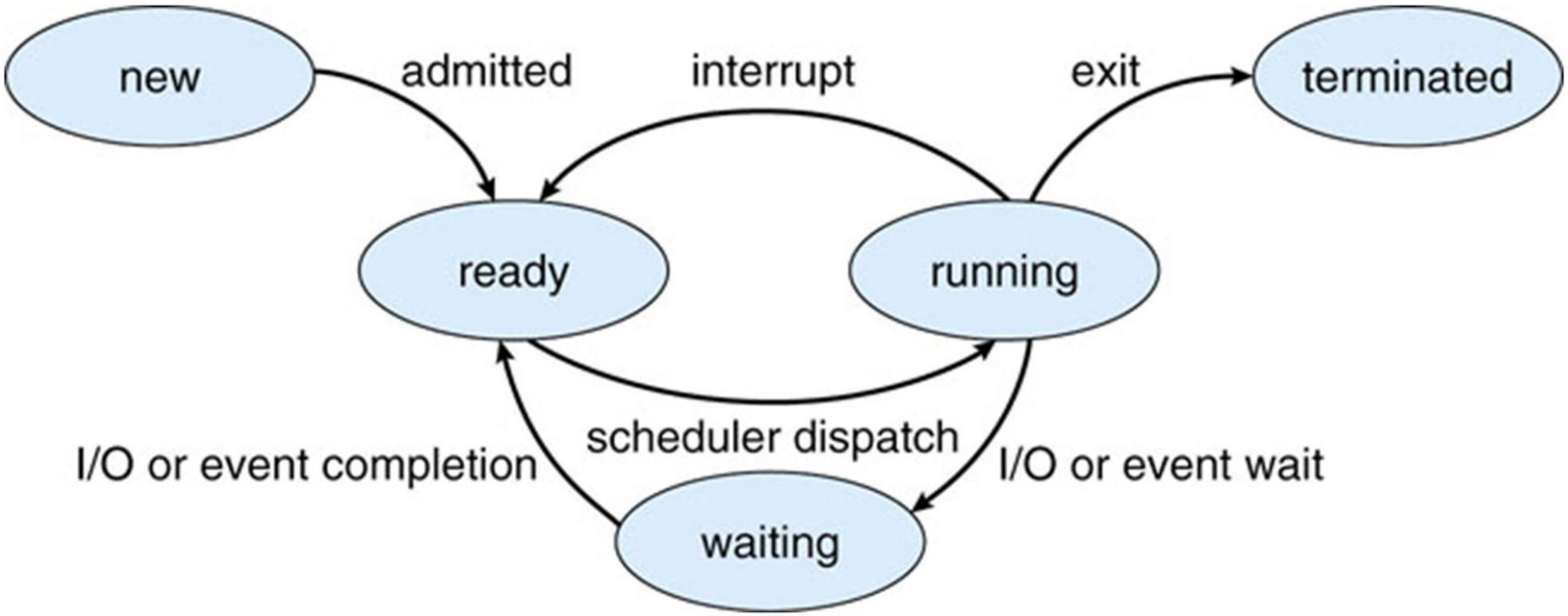
Process in Memory

- Process - a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - registers
  - code
  - data section
  - stack
  - heap



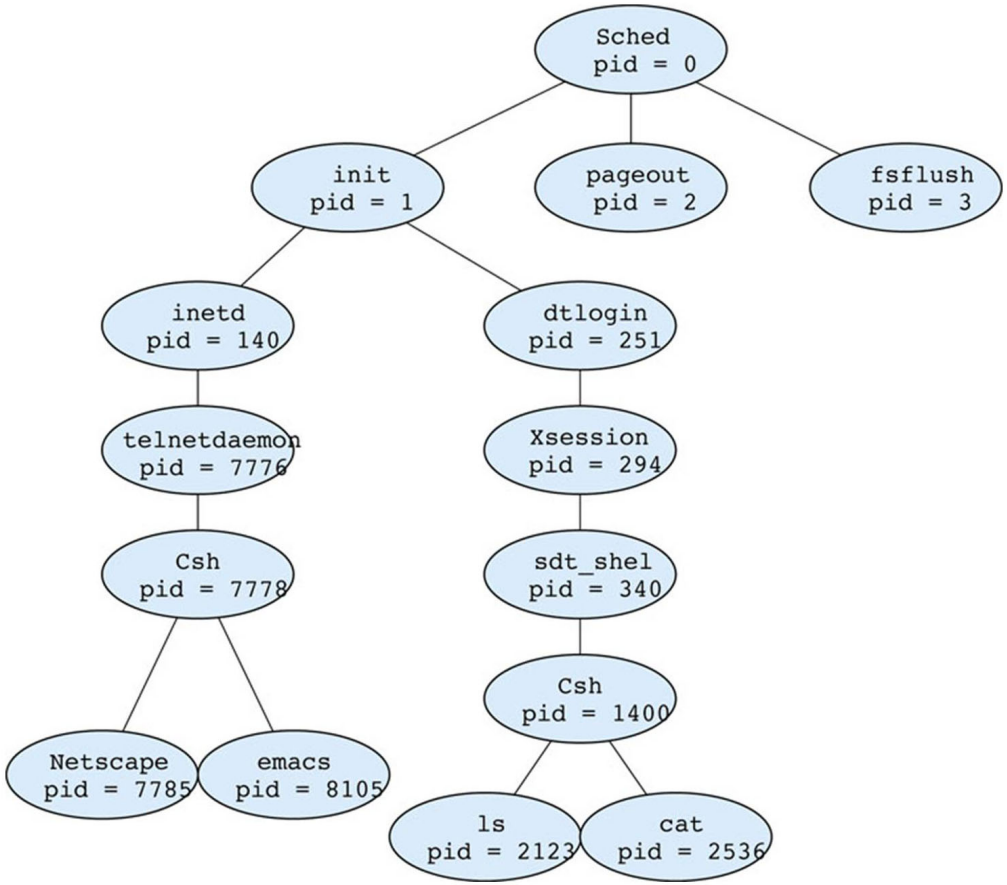
Process State

- As a process executes, it changes state
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur
  - ready: The process is waiting to be assigned to a processor
  - terminated: The process has finished execution



Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)



Process Control Block (PCB)

Information associated with each process

- Process state
- Process id
- CPU scheduling information
- Program counter
- CPU registers
- Credentials (uid, gui, ...)
- Accounting information
- Memory-management information
- I/O status information

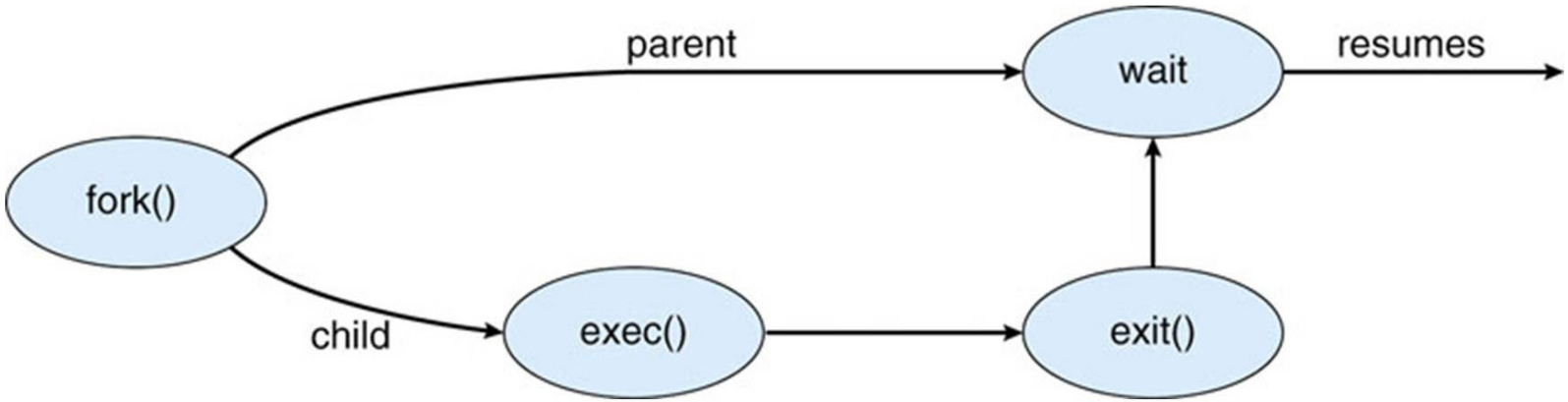
Process Creation (Cont)

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parents resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - fork system call creates new process
  - exec system call used after a fork to replace the process memory space with a new program

Process Termination

- Process executes last statement and asks the operating system to delete it (exit)
  - Output data from child to parent (via wait)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
- If parent is exiting
  - Some operating system do not allow child to continue if its parent terminates ➜All children terminated - cascading termination
  - Child continues ➜child's state is reported to next available level process in the hierarchy

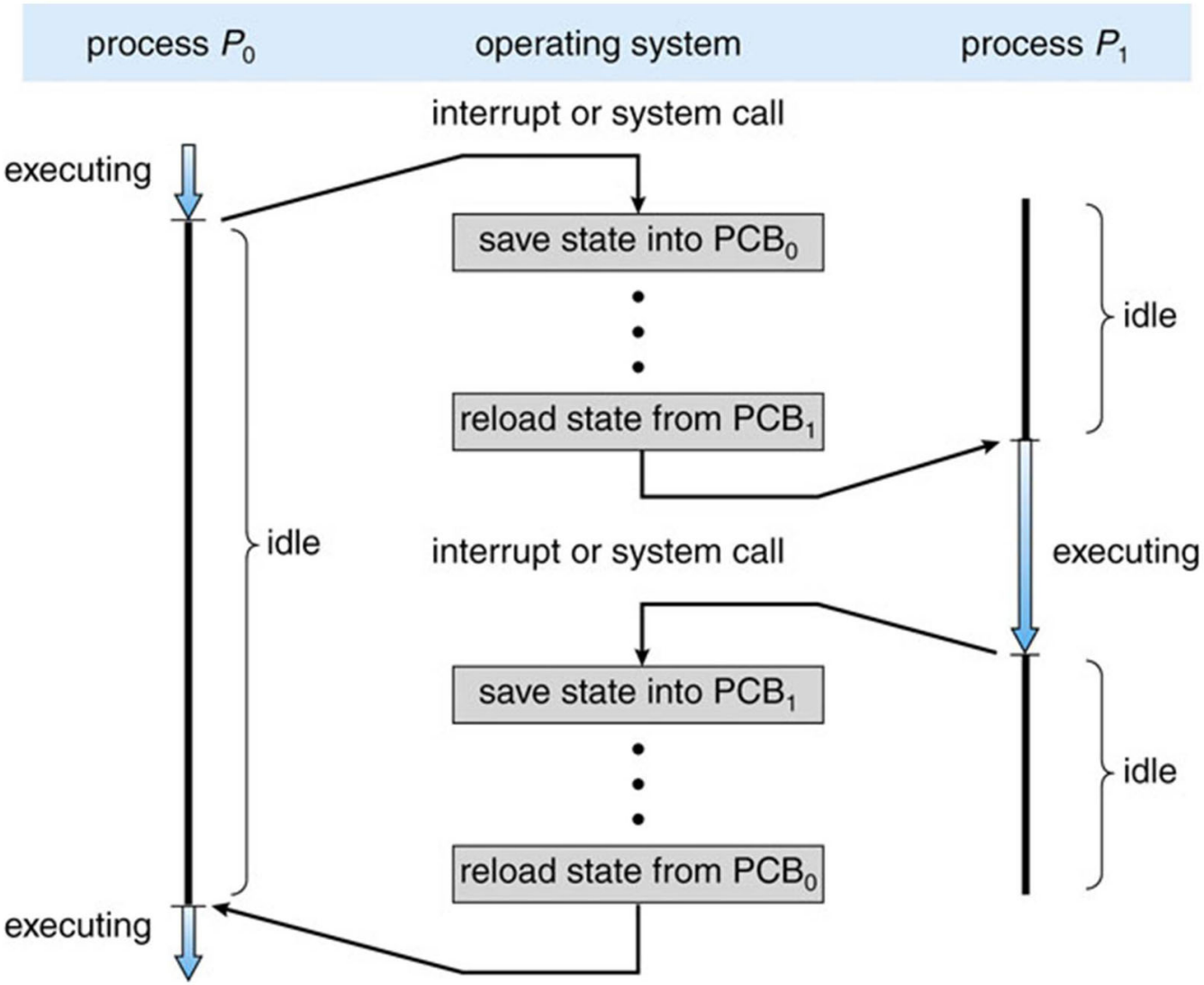
Unix Process Life Cycle



Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

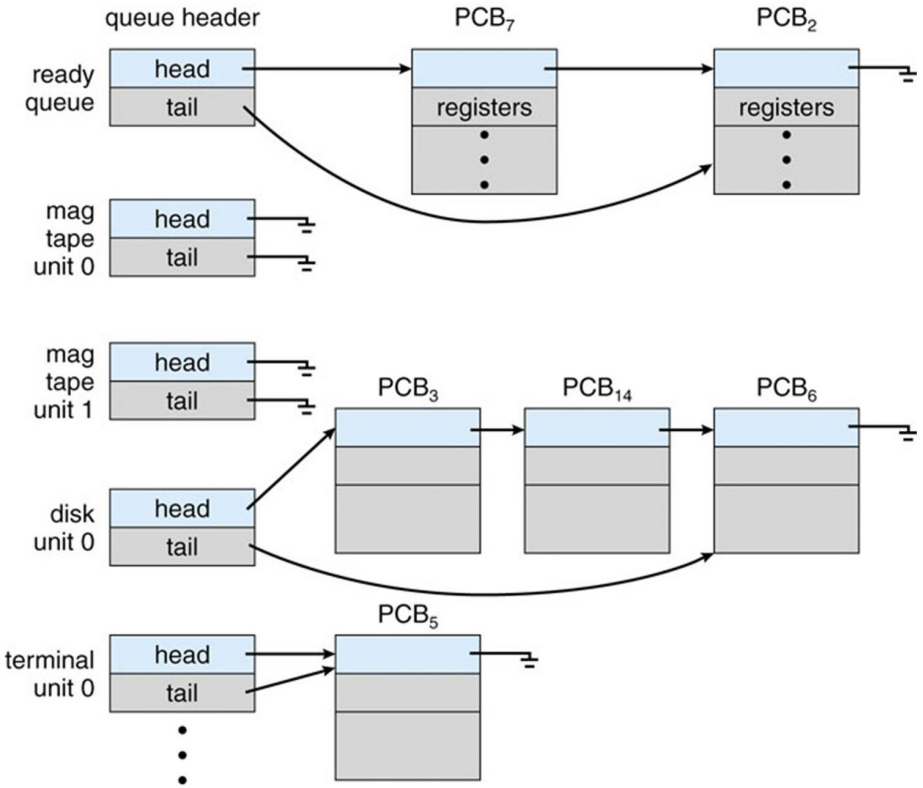
CPU Switch From Process to Process



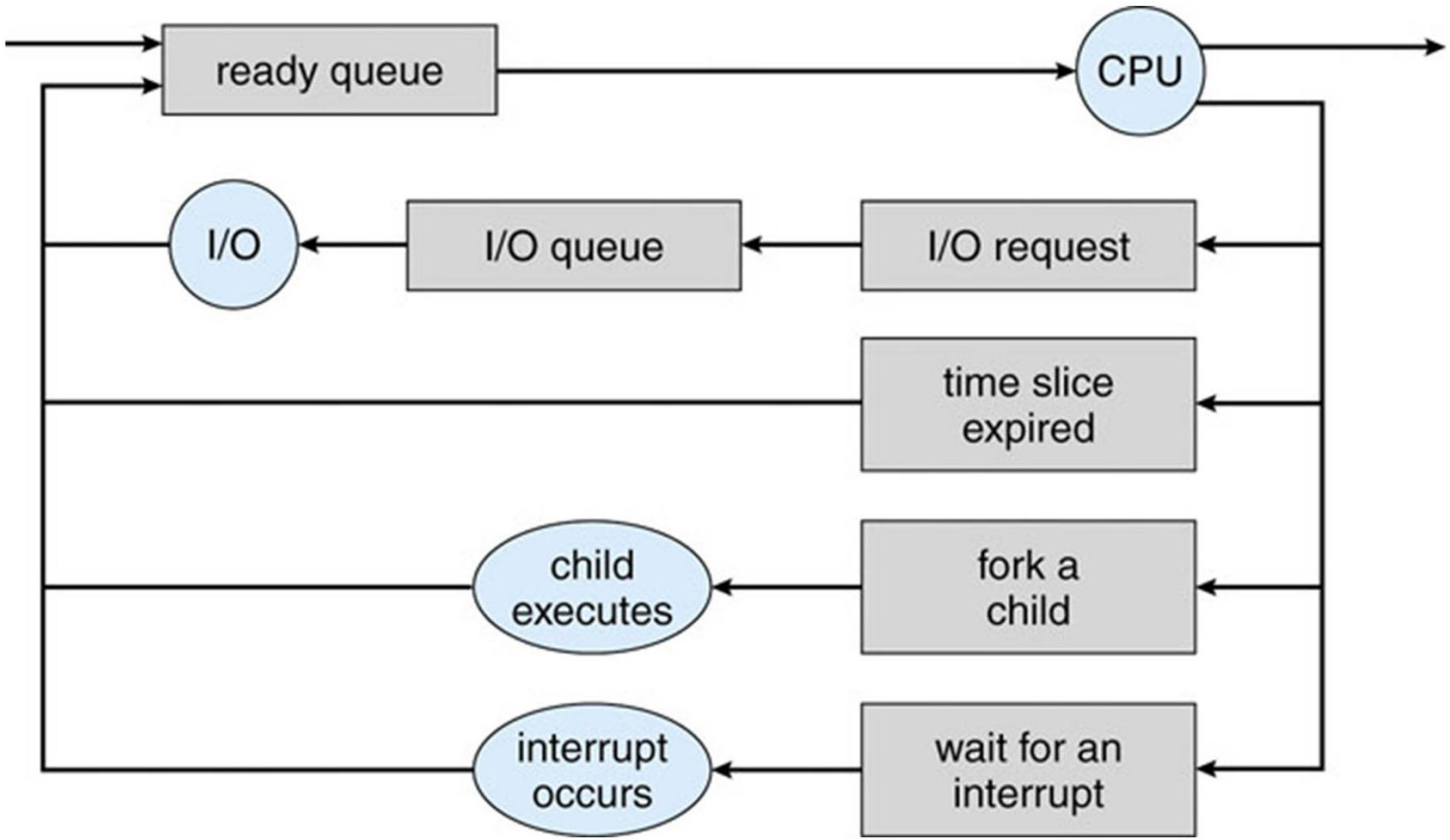
Process Scheduling Queues

- Job queue: Set of all processes in the system
- Ready queue: Set of all processes residing in main memory, ready and waiting to execute
- Device queues: Set of processes waiting for an I/O device

Processes migrate among the various queues



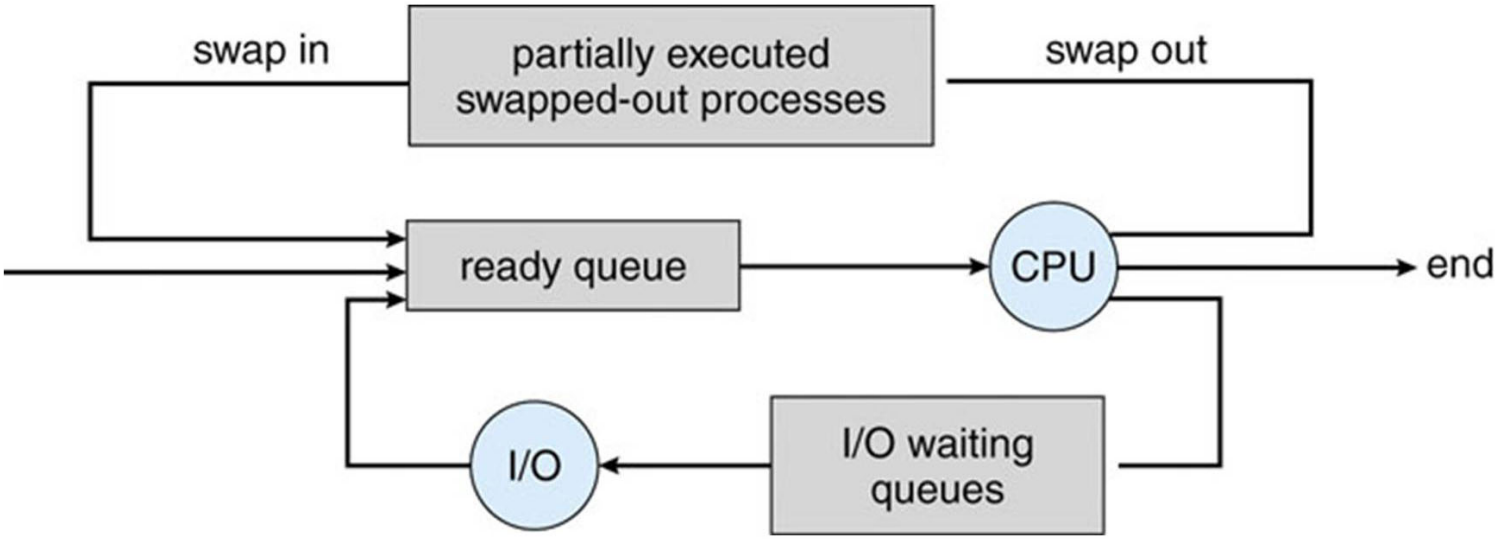
Representation of Process Scheduling



Scheduler

- Long-term scheduler (or job scheduler) - selects which processes should be brought into the ready queu
  - Long-term scheduler is invoked very infrequently (seconds, minutes) ➜(may be slow)
  - The long-term scheduler controls the *degree of multiprogramming*
- Short-term scheduler (or CPU scheduler) - selects which process should be executed next and allocates CPU
  - Short-term scheduler is invoked very frequently (milliseconds) ➜(must be fast)
- Processes can be described as either:
  - I/O-bound process - spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process - spends more time doing computations; few very long CPU bursts

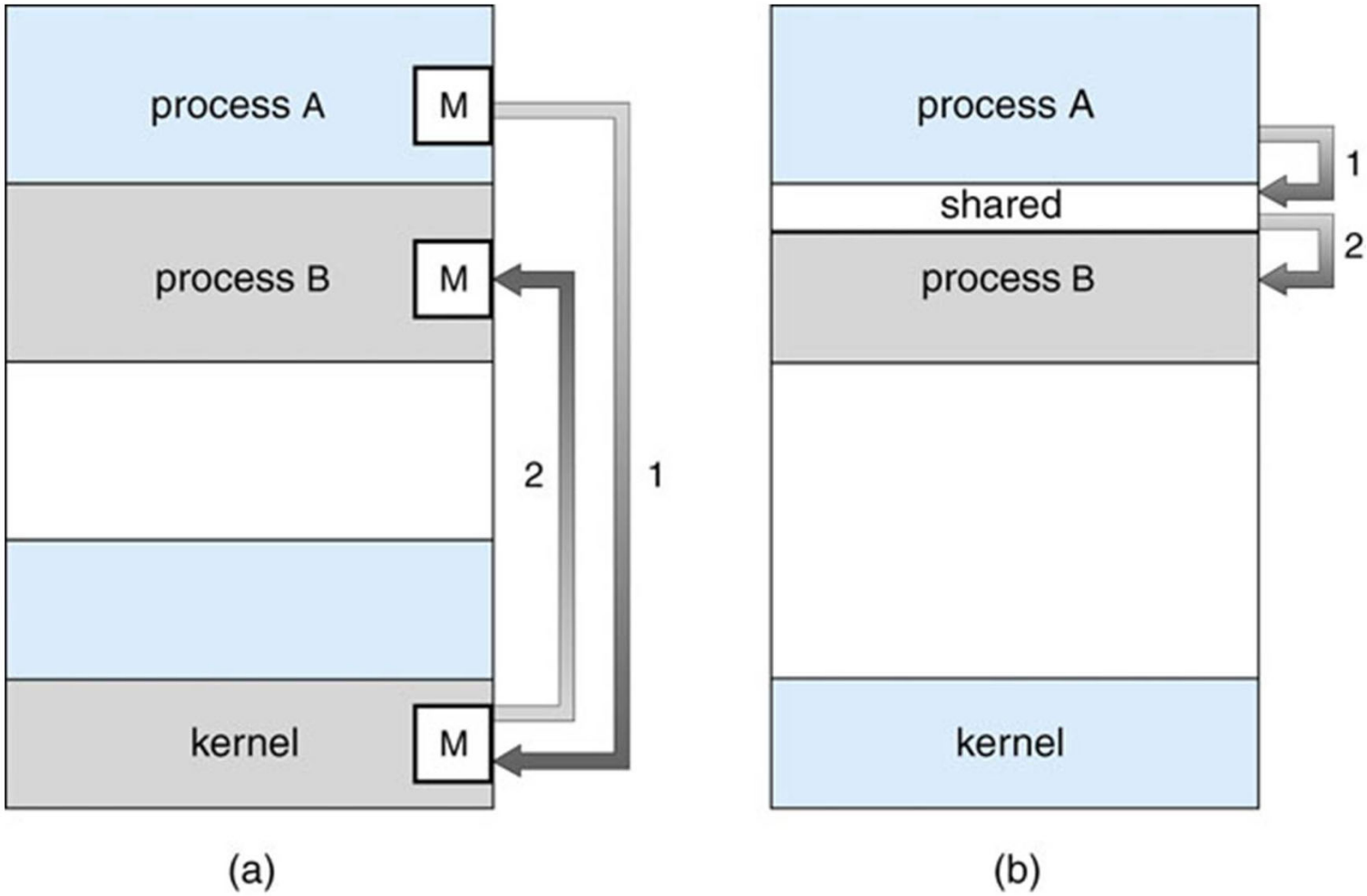
Addition of Medium Term Scheduling



Interprocess Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speed-up
  - Modularity
- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models



Interprocess Communication Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system - processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - send(message) - message size fixed or variable
  - receive(message)
- If P and Q wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Link Implementation Questions

- How are links established (direct or via mailbox)?
- Can a link be associated with more than two processes?
- Is a send/receive call blocking or non-blocking?
- What is the capacity of a link (zero, bounded, unbounded)?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
  - send(P, message) - send a message to process P
  - receive(Q, message) - receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (= ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - send(A, message) send a message to mailbox A
  - receive(A, message) receive a message from mailbox A

Indirect Communication II

- Mailbox sharing
  - P1, P2, and P3 share mailbox A
  - P1, sends; P2 and P3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Blocking send has the sender block until the message is received
  - Blocking receive has the receiver block until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send has the sender send the message and continue
  - Non-blocking receive has the receiver receive a valid message or null

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  - Zero capacity- 0 messages
    Sender must wait for receiver (rendezvous)
  - Bounded capacity - finite length of n messages
    Sender must wait if link full (see UNIX “pipe” and “named pipe”)
  - Unbounded capacity - infinite length
    Sender never waits

Examples of IPC Systems - POSIX Shared Memory

- POSIX Shared Memory
  - Process first creates shared memory segment
    ```
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
    ```
  - Process wanting access to that shared memory must attach to it
    ```
    shared_memory = (char *) shmat(segment_id, NULL, 0);
    ```
  - Now the process could write to the shared memory
    ```
    sprintf(shared_memory, "Writing to shared memory");
    ```
  - When done a process can detach the shared memory from its address space
    ```
    shmdt(shared_memory);
    ```
  - Finally, a shared-memory segment can be removed from the system
    ```
    shmctl(segment_id, IPC_RMID, NULL);
    ```

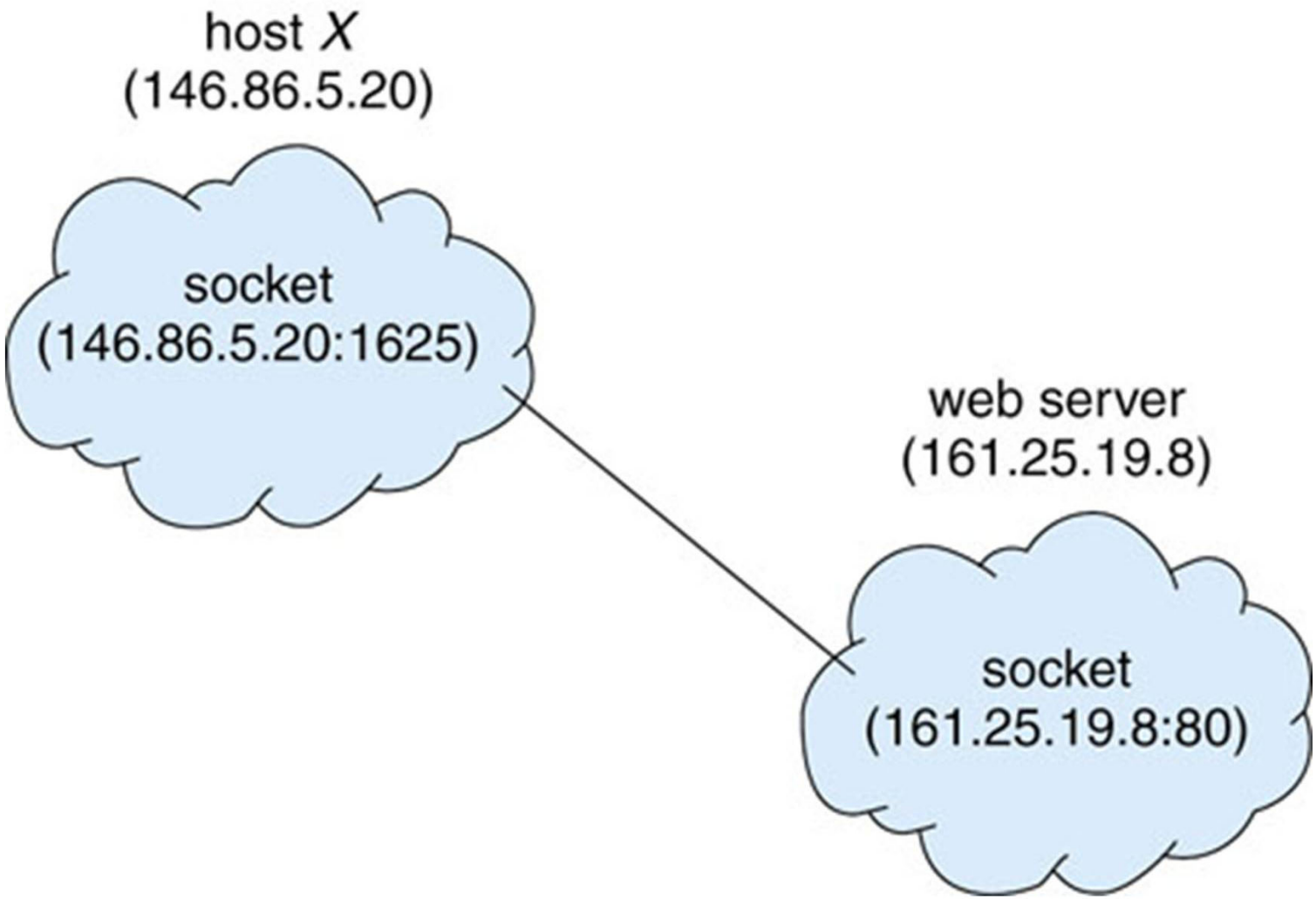
Examples of IPC Systems - Mach, OS X

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two initial mailboxes at creation - Kernel and Notify
  - Only three system calls needed for message transfer
    `msg_send()`, `msg_receive()`, `msg_rpc()`
  - Mailboxes needed for communication, created via
    `port_allocate()`
  - Maximal capacity are 8 messages
  - Just one process is owner of the port and allowd to receive messages (right can be transferred)
  - Flexible sync. options: blocking, time-out, non-blocking
  - Mailbox-Set allows to receive from multiple mailboxes
  - `port_status()` reports the number of messages in a mailbox

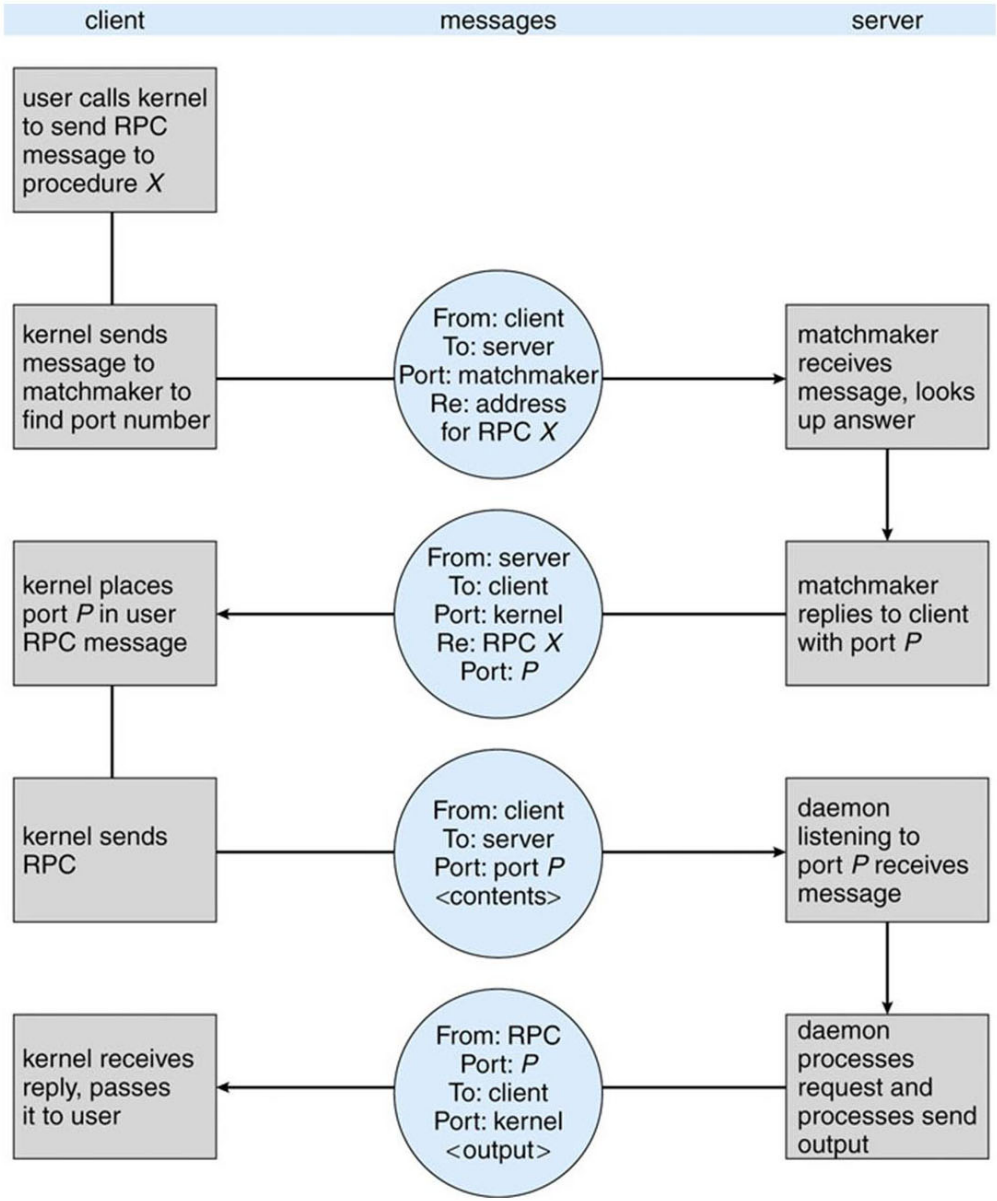
Communications in Client-Server Systems

- Sockets
  - A socket is defined as an *endpoint for communication*
  - Concatenation of IP address and port
  - The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
  - Communication consists between a pair of sockets
- Remote Procedure Calls
  - Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Stubs client-side proxy for the actual procedure on the server
  - The client-side stub locates the server and marshalls the parameters (engl. marshalling = dt. Zugbildung)
  - The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- Remote Method Invocation (Java)

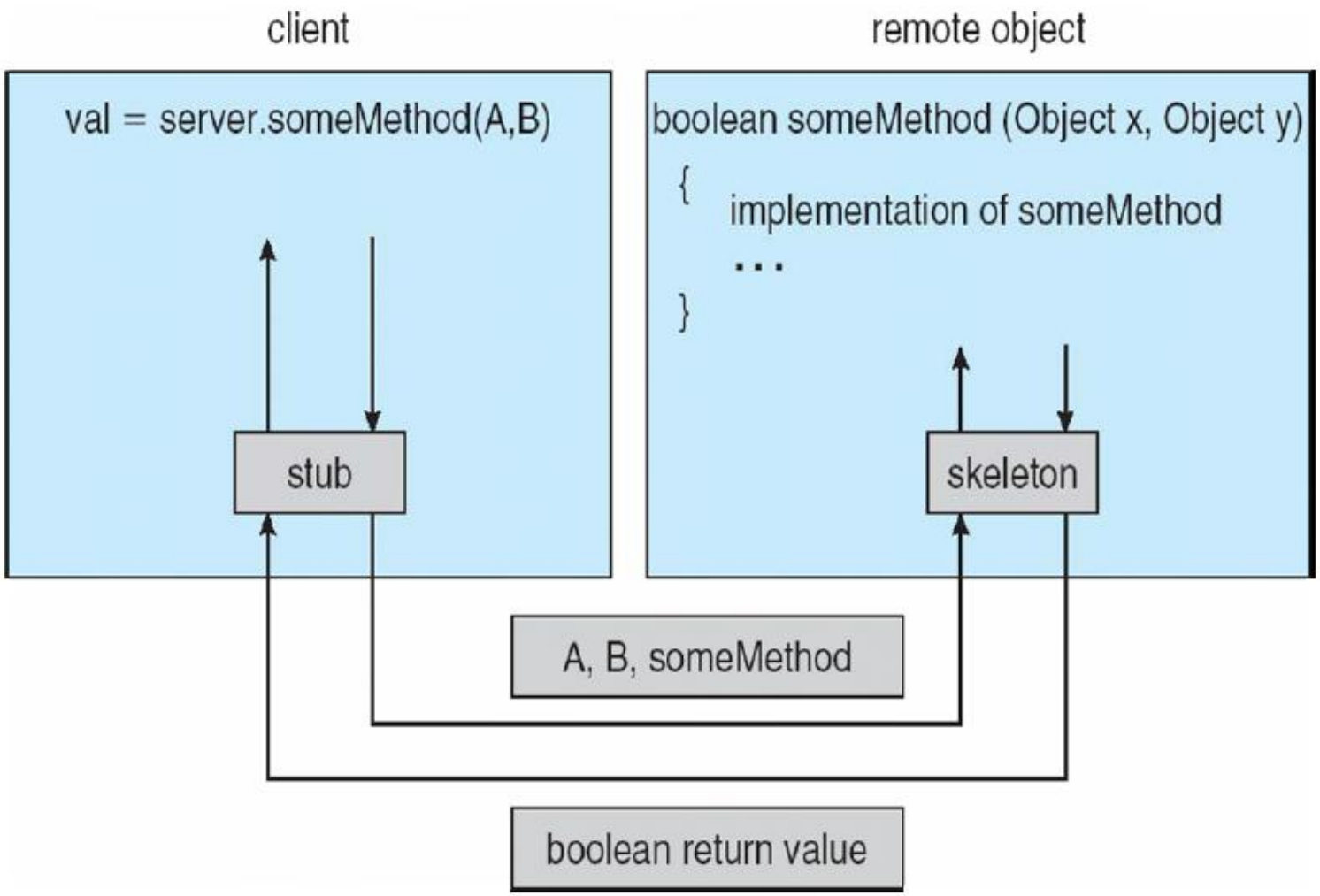
Socket Communication



Execution of RPC

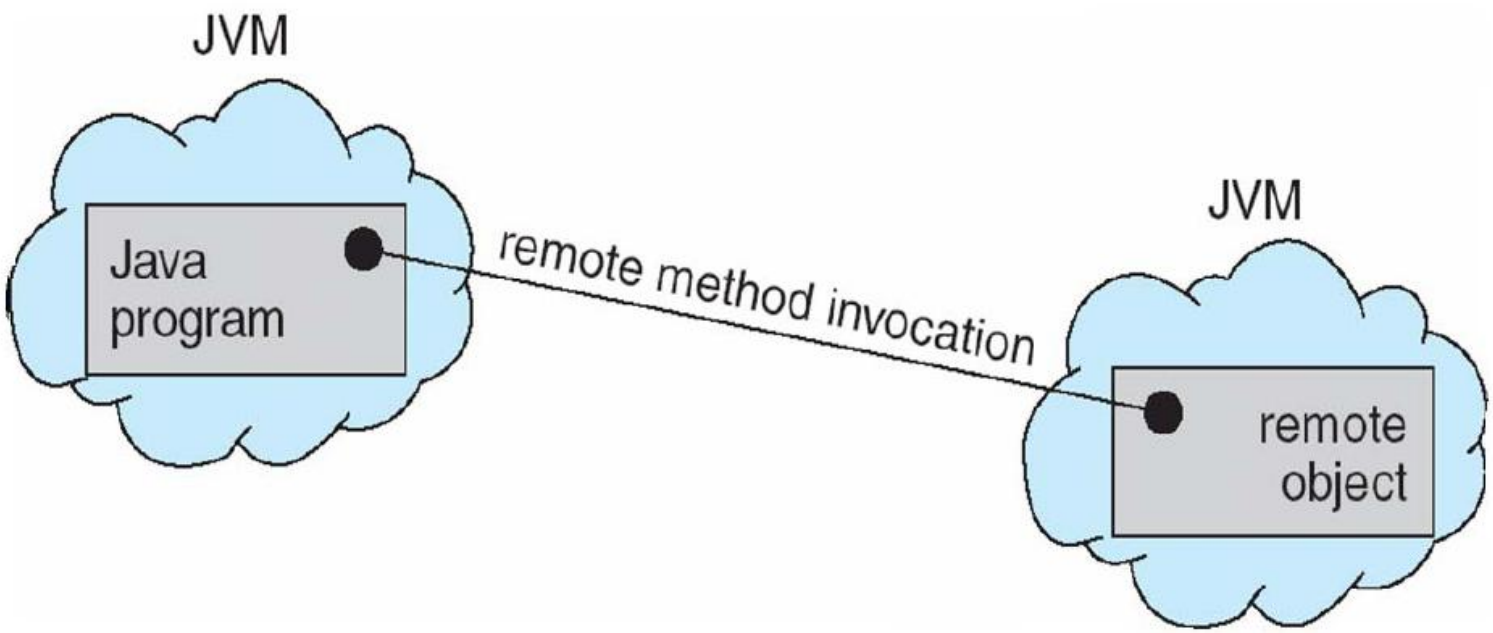


Marshalling Parameters

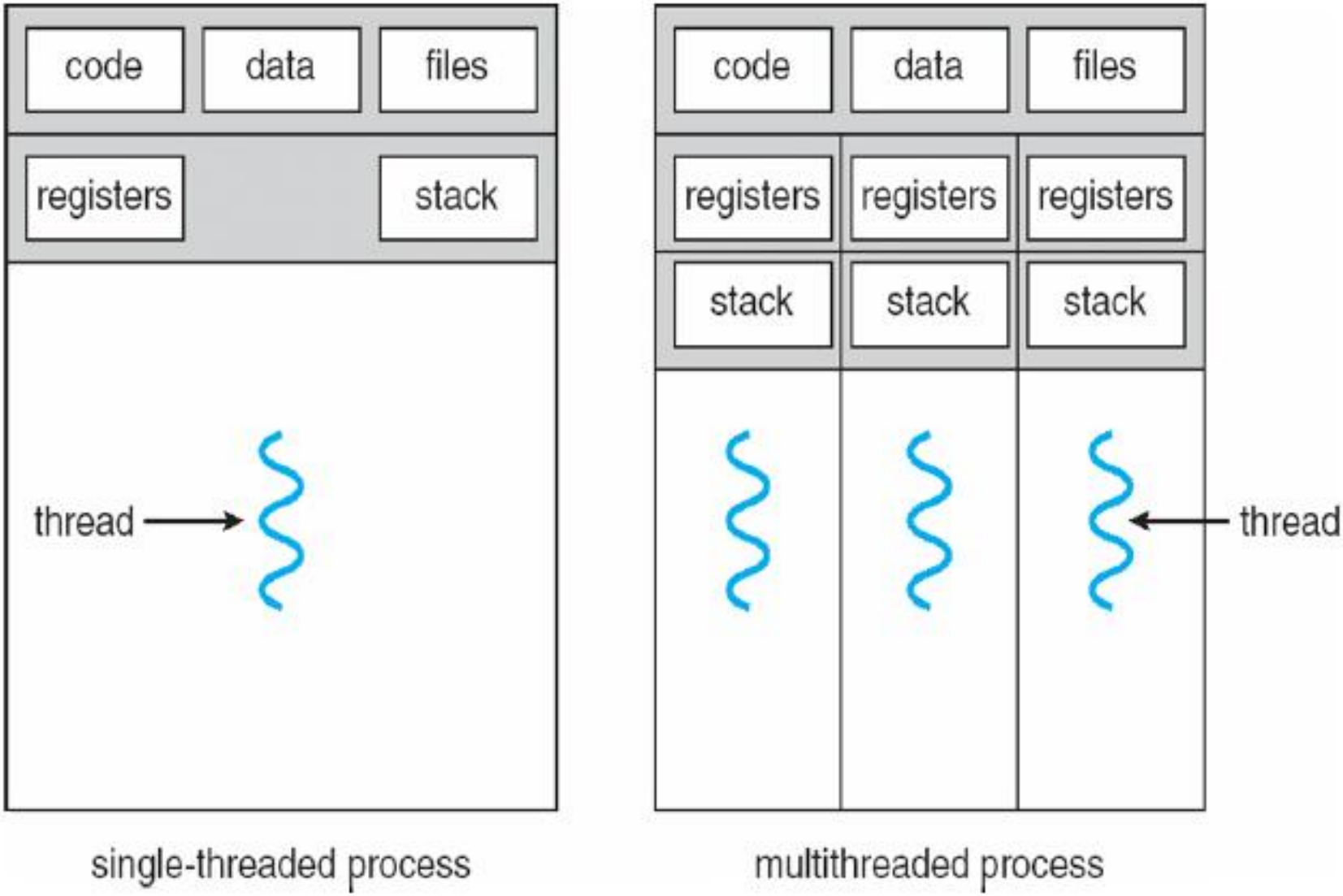


Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object



Single and Multithreaded Processes



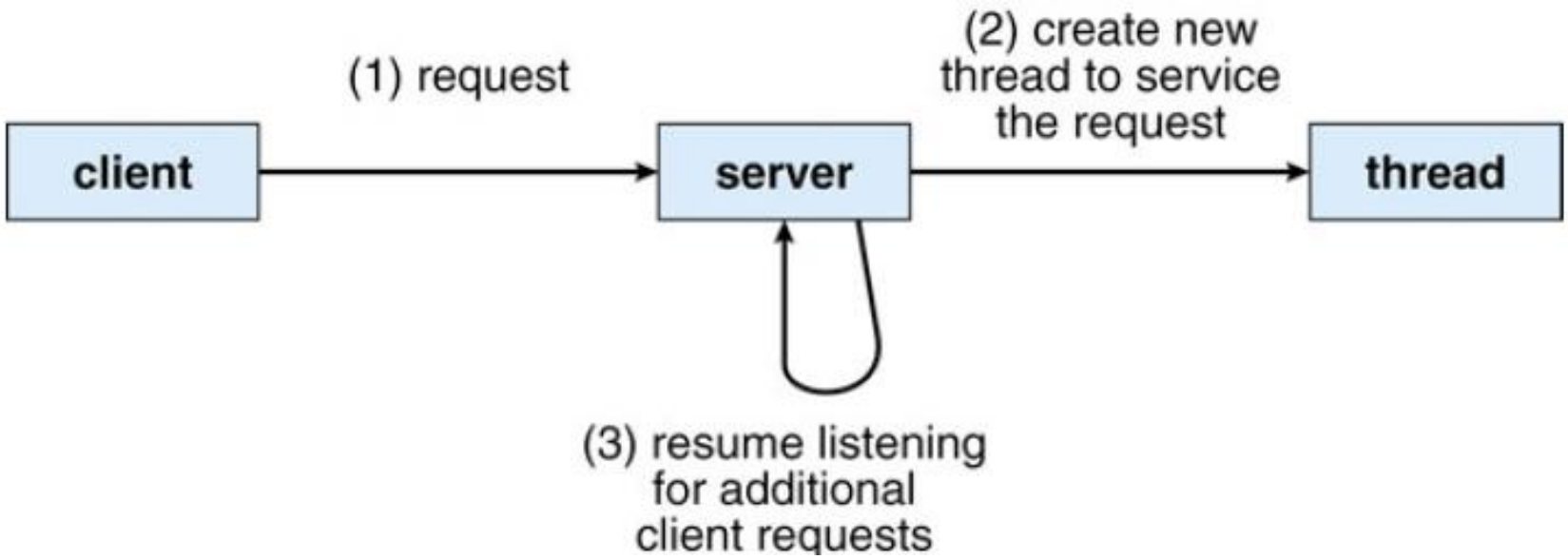
Benefits

- Responsiveness
- Resource Sharing
- Economy
- Scalability

Multicore Programming

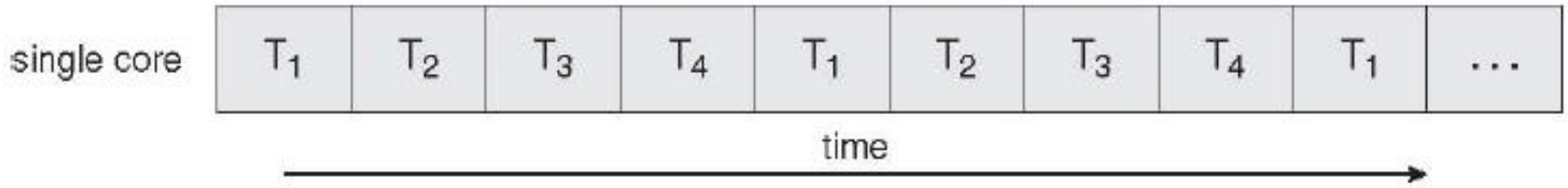
- Multicore systems putting pressure on programmers, challenges include
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

Multithreaded Server Architecture

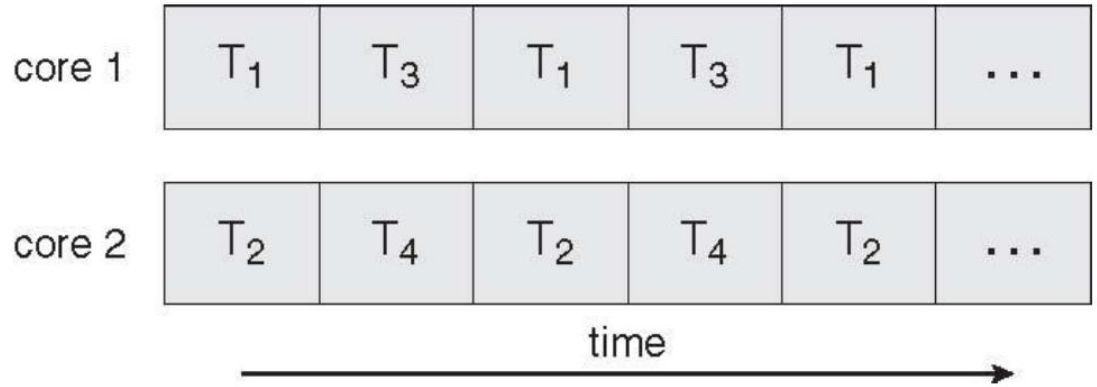


Thread Execution on Single- vs. Multi-Core Systems

- Concurrent Execution on a Single-core System



- Parallel Execution on a Multicore System



User Threads

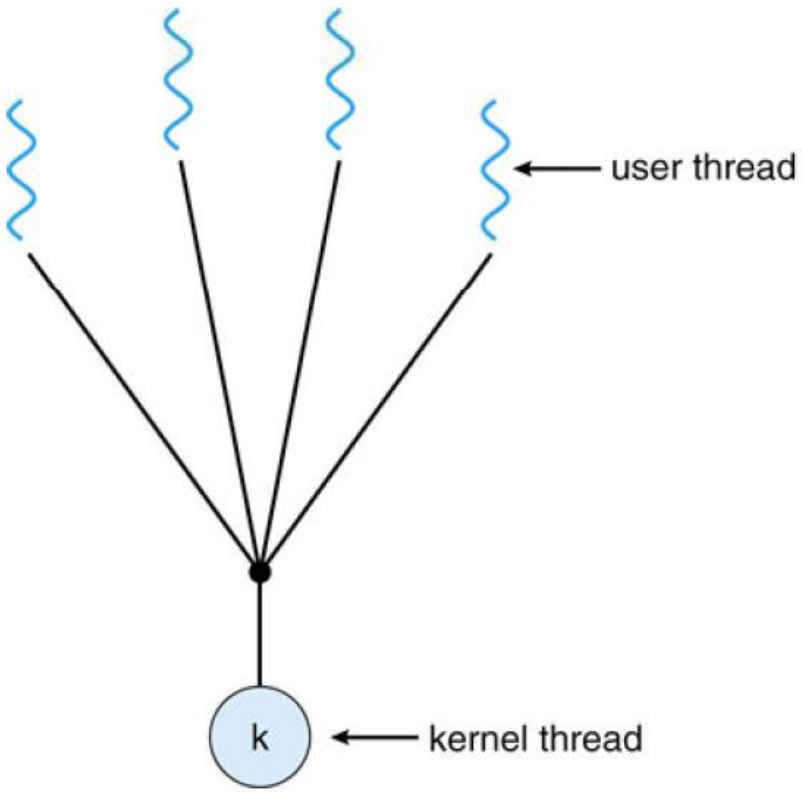
- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

- Supported by the Kernel
- Examples
  - Windows XP/Vista/W7
  - Linux
  - Solaris
  - Mac OS X

Many-to-One (M x 1) Model

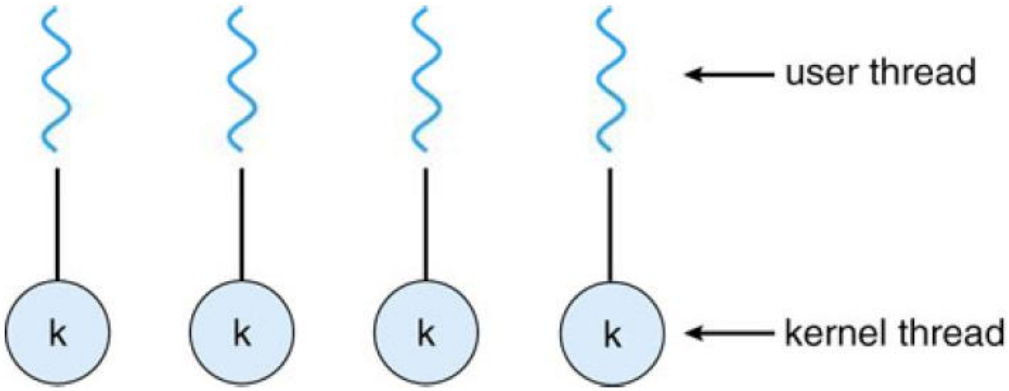
- Many user threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



- Pros:
  - Fast thread management operations (up 100 times)
  - Flexible scheduling policy
  - Saving of system resources
  - Concurrent prog. model that can be transferred to 1x1 or MxN without modification
- Cons:
  - Whole process blocks if only one user thread blocks
  - Profiling and debugging is critical

One-to-One (1 x 1)Model

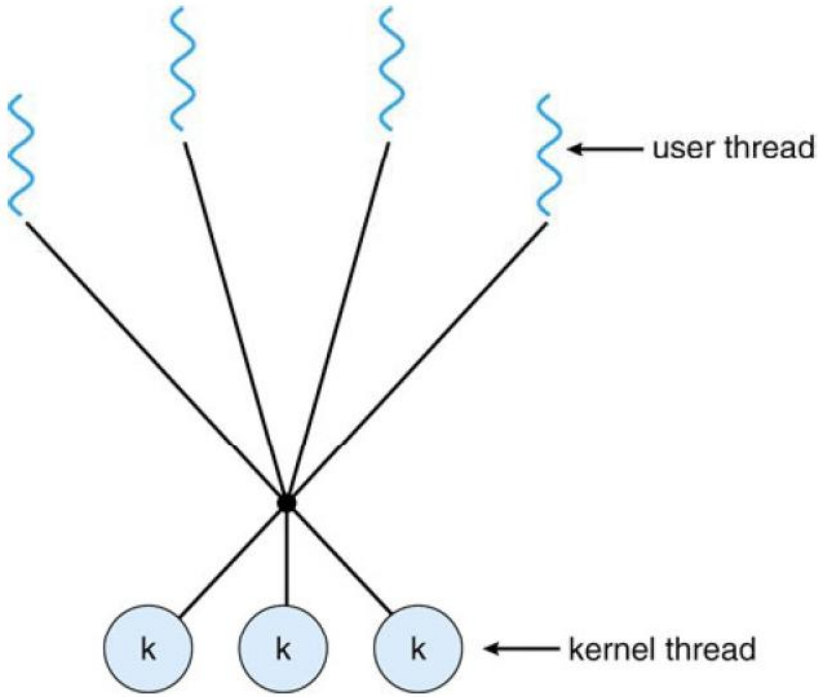
- Each user-level thread maps to kernel thread
- Examples:
  - Windows XP/Vista/W7
  - Linux
  - Solaris 9 and later



- Pros:
  - Real parallelism
  - Profiling and debugging is possible
- Cons:
  - Overhead for kernel policy
  - System memory intensive (TCB, stack)

Many-to-Many (M x N) Model

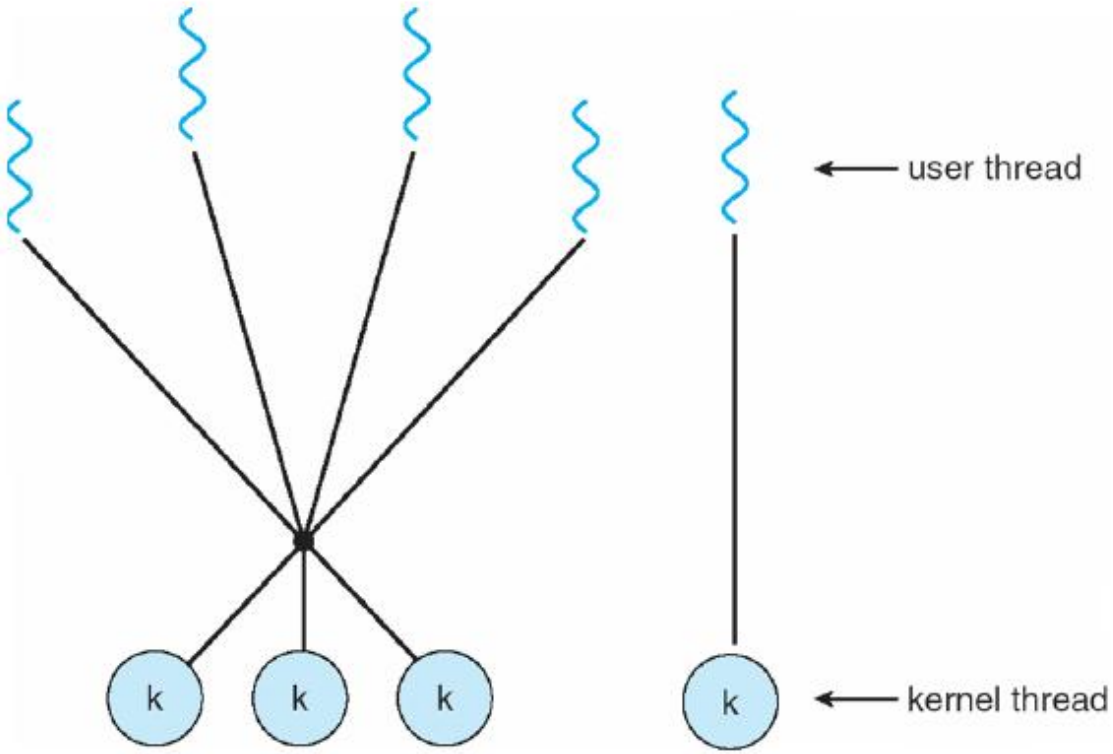
- Allows many user level threads to be mapped to many kernel threads
- OS can create a "sufficient" number of kernel threads
- Examples:
  - Solaris prior to version 9
  - Windows NT/2000 with the ThreadFiber package



- Pros:
  - Flexible scheduling policy
  - Efficient execution
  - Deadlock recovery by kernel thread creation
- Cons:
  - Two-level scheduling
  - Profiling and debugging critical

Two-Level Model

- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples:
  - HP-UX
  - Solaris 8 and earlier



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
- Pthreads
  - May be provided either as user-level or kernel-level
  - A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  - API specifies behavior of the thread library, implementation is up to development of the library
  - Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Threading Issues I

- Semantics of fork() and exec() system calls
  - Does fork() duplicate only the calling thread or all threads?
- Thread cancellation of target thread
  - Asynchronous cancellation terminates the target thread immediately
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled
- Signal Handling
  - Signals are used in UNIX systems to notify a process that a particular event has occurred
  - A signal handler is used to process signals
    - Signal is generated by particular event
    - Signal is delivered to a process
    - Signal is handled
  - Options:
    - Deliver the signal to the thread to which the signal applies
    - Deliver the signal to every thread in the process
    - Deliver the signal to certain threads in the process
    - Assign a specific thread to receive all signals for the process

Threading Issues II

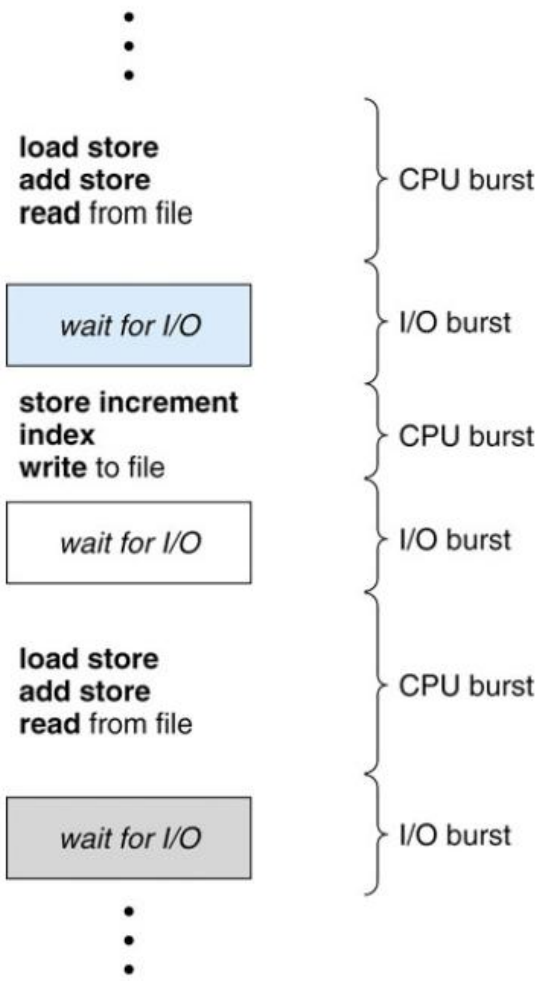
- Thread Pools
  - Create a number of threads in a pool where they await work
  - Advantages:
    - Usually slightly faster to service a request with an existing thread than create a new thread
    - Allows the number of threads in the application(s) to be bound to the size of the pool
- Thread Specific Data
  - Allows each thread to have its own copy of data
  - Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Scheduler Activations
  - Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
  - Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
  - This communication allows an application to maintain the correct number of kernel threads

Process Scheduling

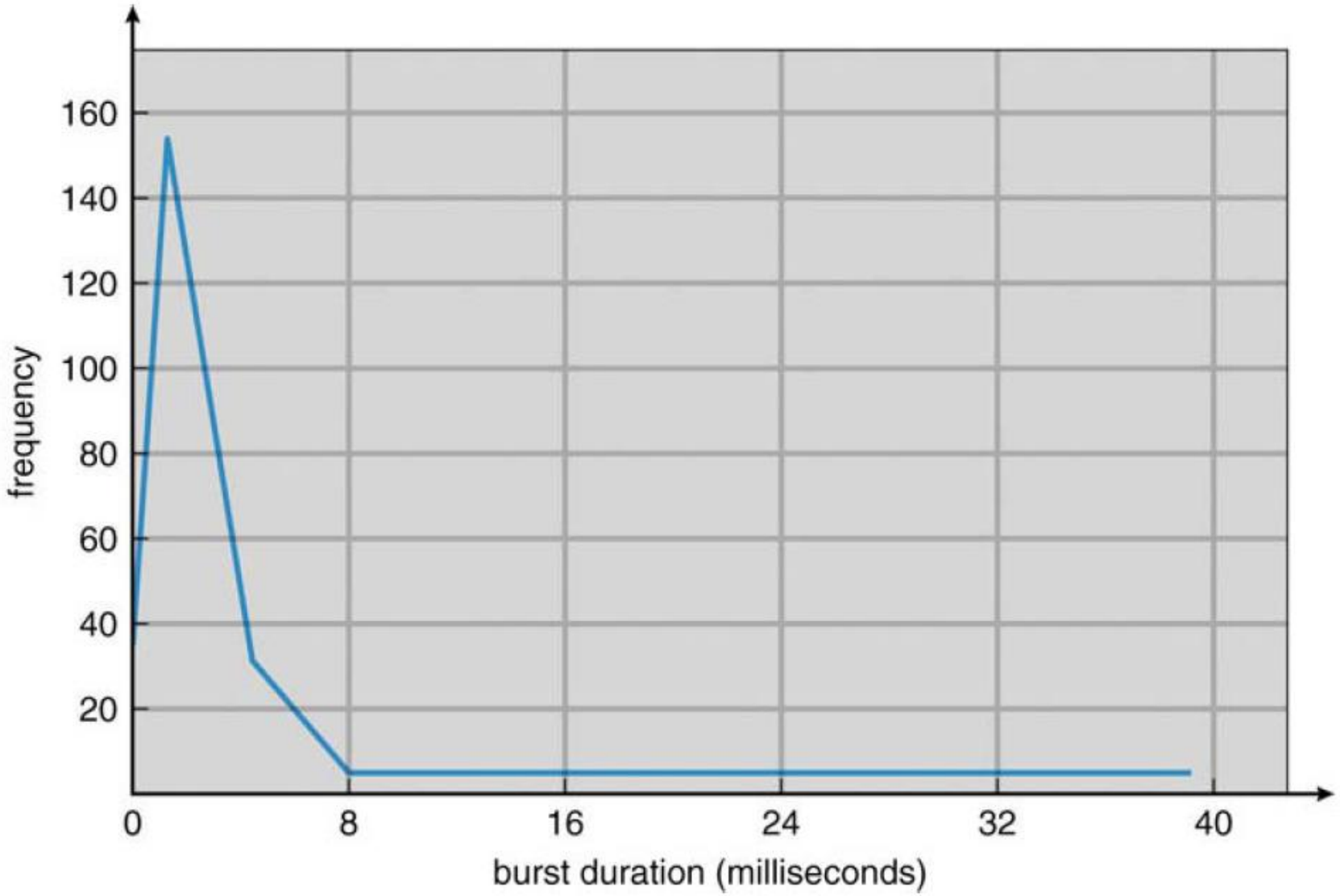
- Basic Concepts
- Scheduling Criteria
- Scheduling Policies
- Multi-Processor Scheduling
- OS Scheduler Examples
- Scheduling Evaluation

CPU - I/O Burst Cycle

- Process execution consists of a cycle of CPU execution and I/O wait

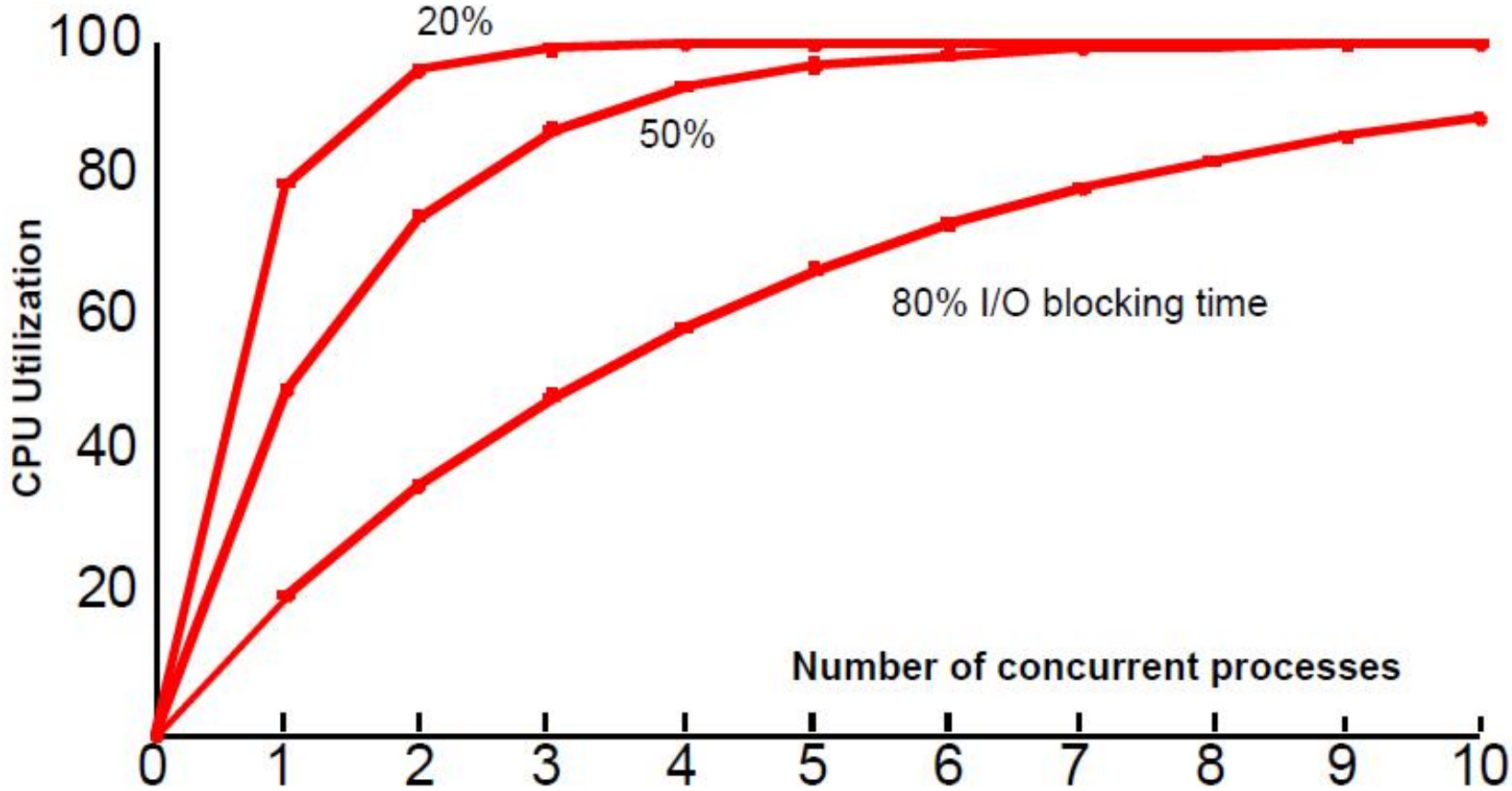


CPU Burst Distribution



The Benefit of Multiprogramming

- Maximum CPU utilization obtained with multiprogramming



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  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 process 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 process and start another running

Scheduling Criteria

- CPU utilization - keep the CPU as busy as possible
- Throughput - # of processes that complete their execution per time unit
- Turnaround time - amount of time to execute a particular process
- Waiting time - amount of time a process has been waiting in the ready queue
- Response time - amount of time it takes from when a request was submitted until the first response is produced (for time-sharing environment)

Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling I

- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3
  The Gantt Chart for the schedule is:

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

First-Come, First-Served (FCFS) Scheduling II

- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_2, P_3, P_1
  The Gantt Chart for the schedule is:

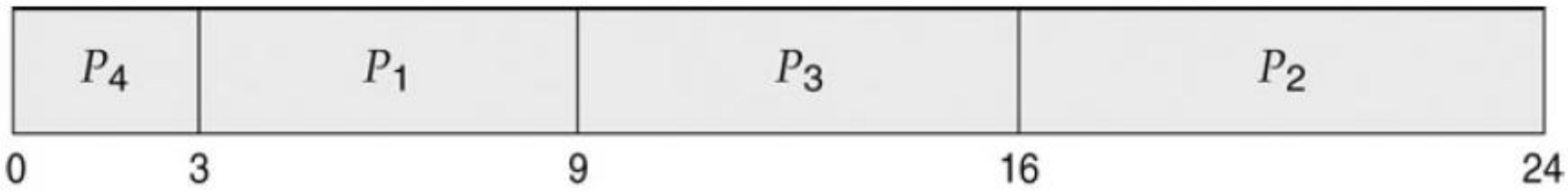
- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- *Convoy effect* - short process behind long process

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal - gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
- Example:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

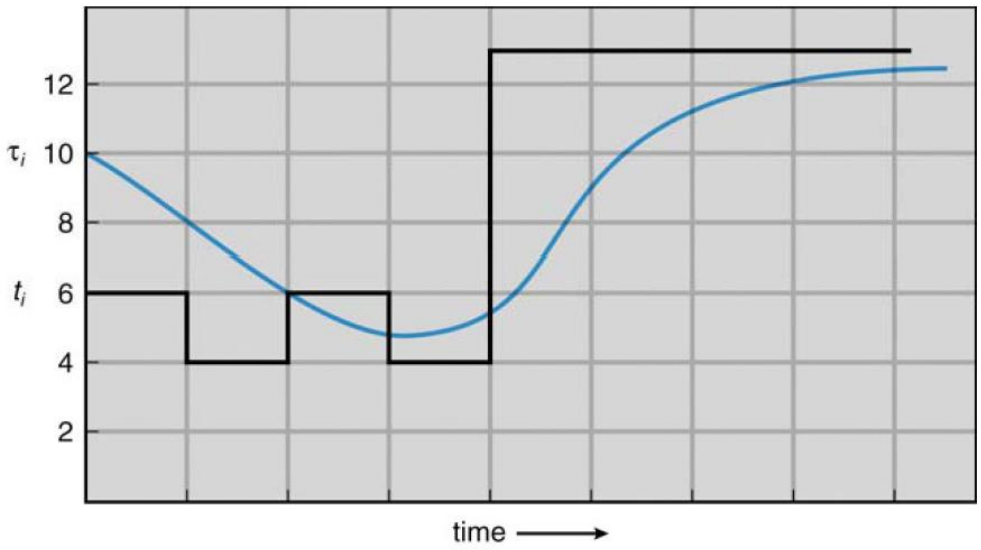
- SJF scheduling chart:



- Average waiting time: $(3 + 16 + 9 + 0)/4 = 7$

Estimating the Length of Next CPU Burst

- Can be done by using the length of previous CPU bursts, using exponential averaging
  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. $\alpha, 0 \leq \alpha \leq$
  4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
- Example: $\alpha = 0.5$



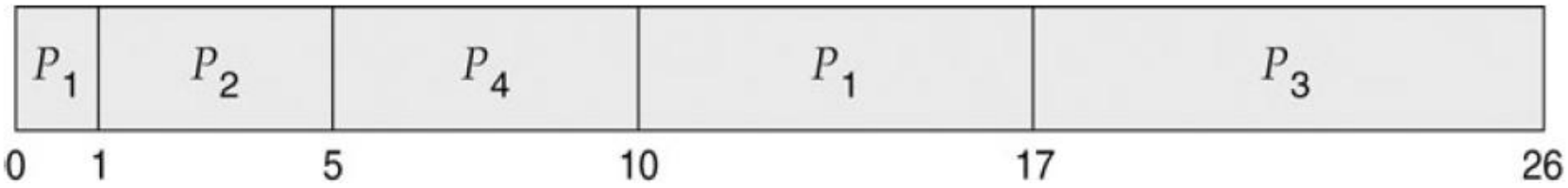
CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Preemptive Shortest-Job-First (PSJF) Scheduling

- PSJF = shortest-remaining-time-first
- Example:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- SJF scheduling chart:



- Average waiting time:
  $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 6.5$

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer $\equiv$ highest priority)
  - Preemptive
  - Non-preemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem: Starvation - low priority processes may never execute
- Solution: Aging- as time progresses increase the priority of the process

Round Robin (RR)

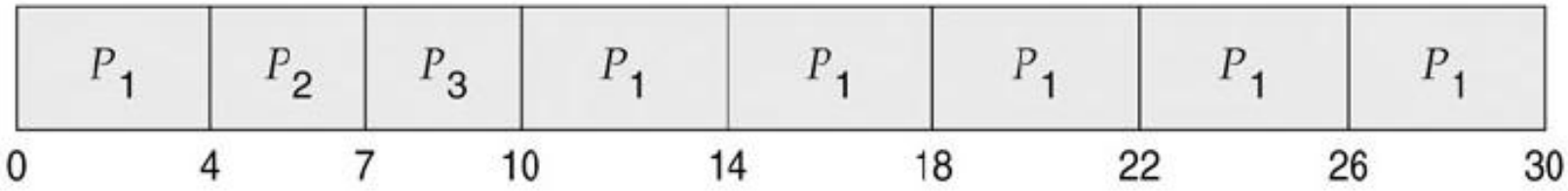
- Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once. No process waits more than (n-1)q time units.
- Overhead
  - q large $\Rightarrow$ FIFO $\Rightarrow$ low number of context switches
  - q small $\Rightarrow$ many context switches $\Rightarrow$ q must be large with respect to context switch

Round Robin II

- Example: RR with TQ 4

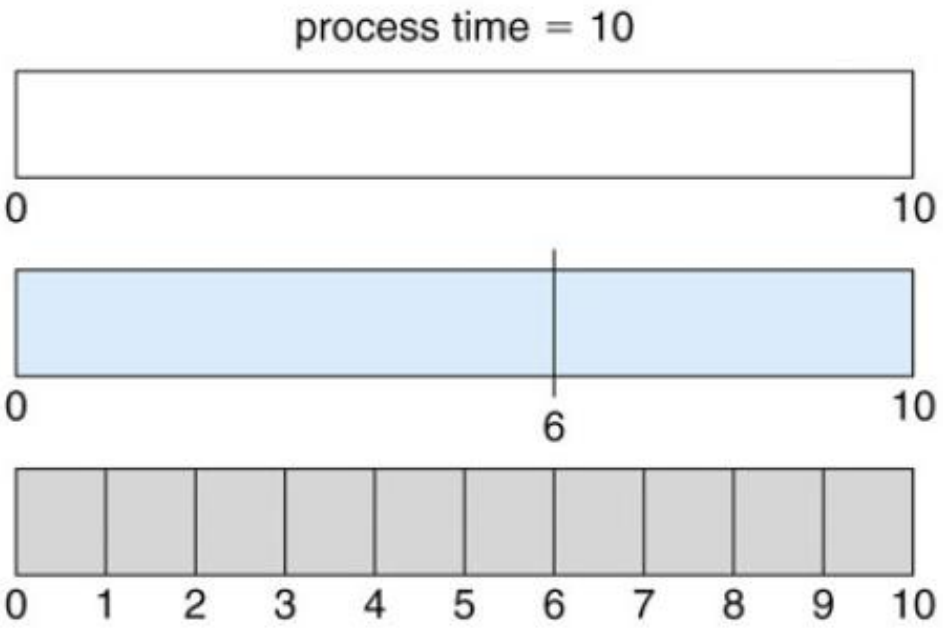
Process	Burst Time
P_1	24
P_2	3
P_3	3

- Gantt chart:



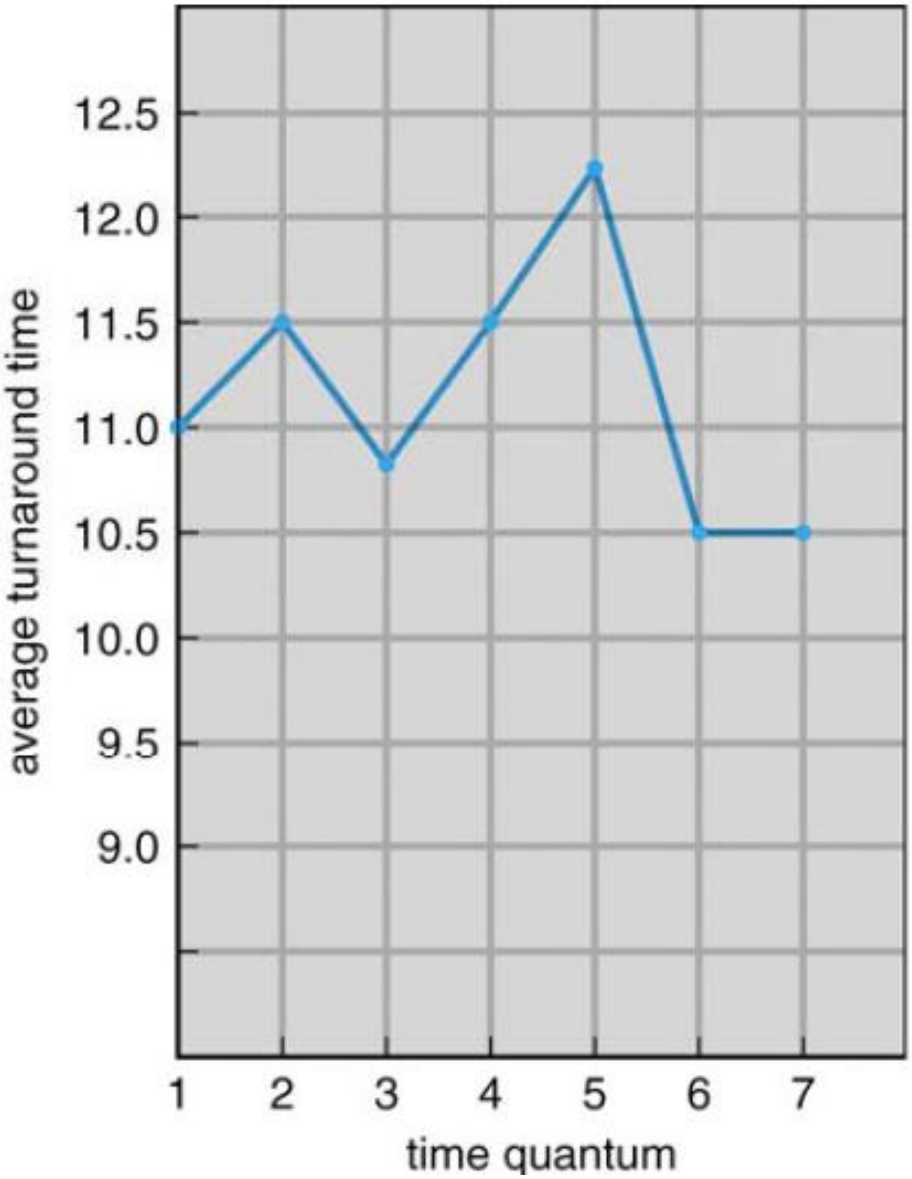
- Typically, higher average turnaround than SJF, but better response

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum

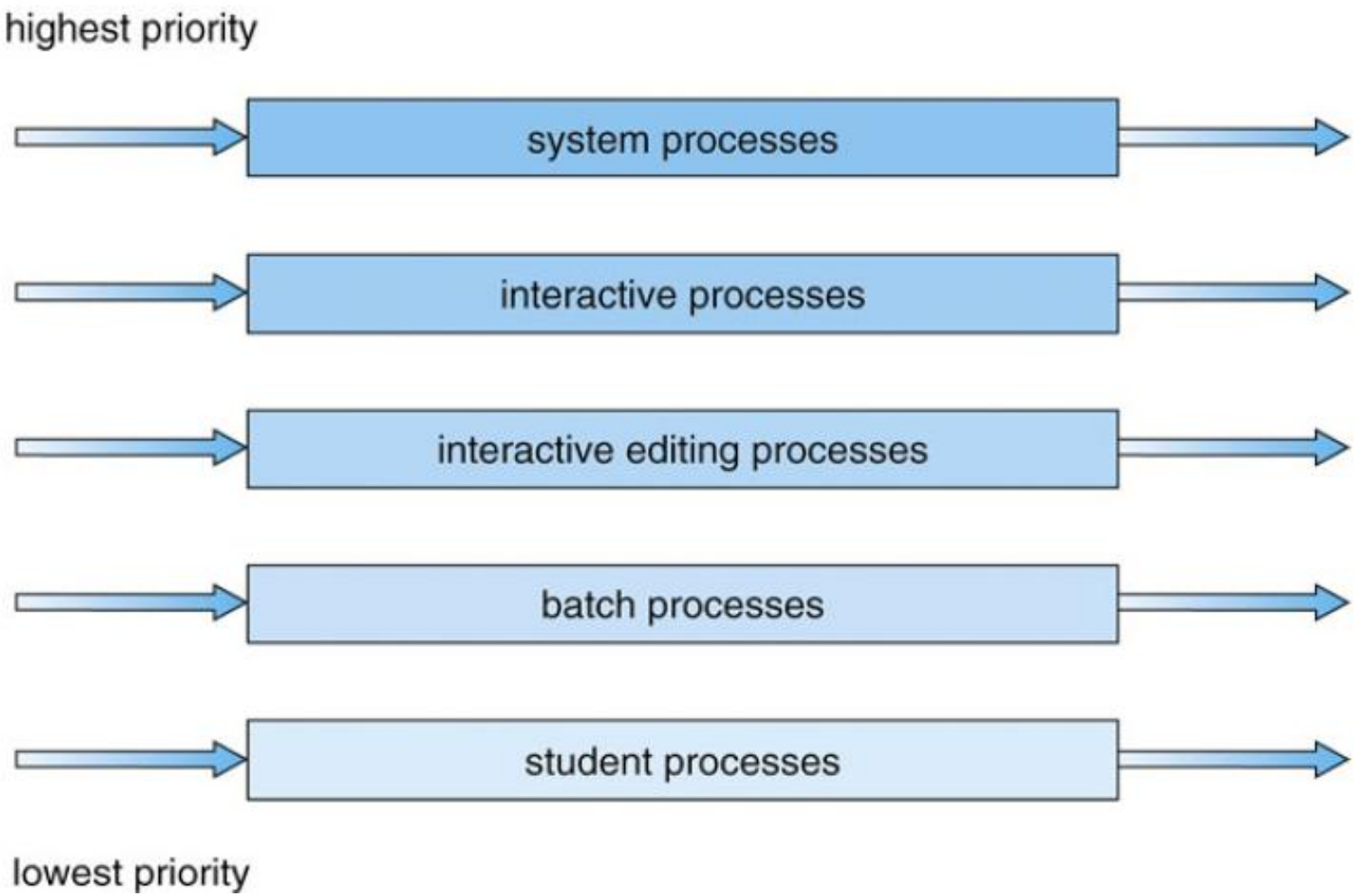


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

- Ready queue is partitioned into separate queues: (e.g., foreground=interactive) and background=batch)
- Each queue has its own scheduling algorithm
  - foreground - RR
  - background - FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background) $\Rightarrow$ Possibility of starvation
  - Time slice - each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% RR, 20% to FCFS

Multilevel Queue II



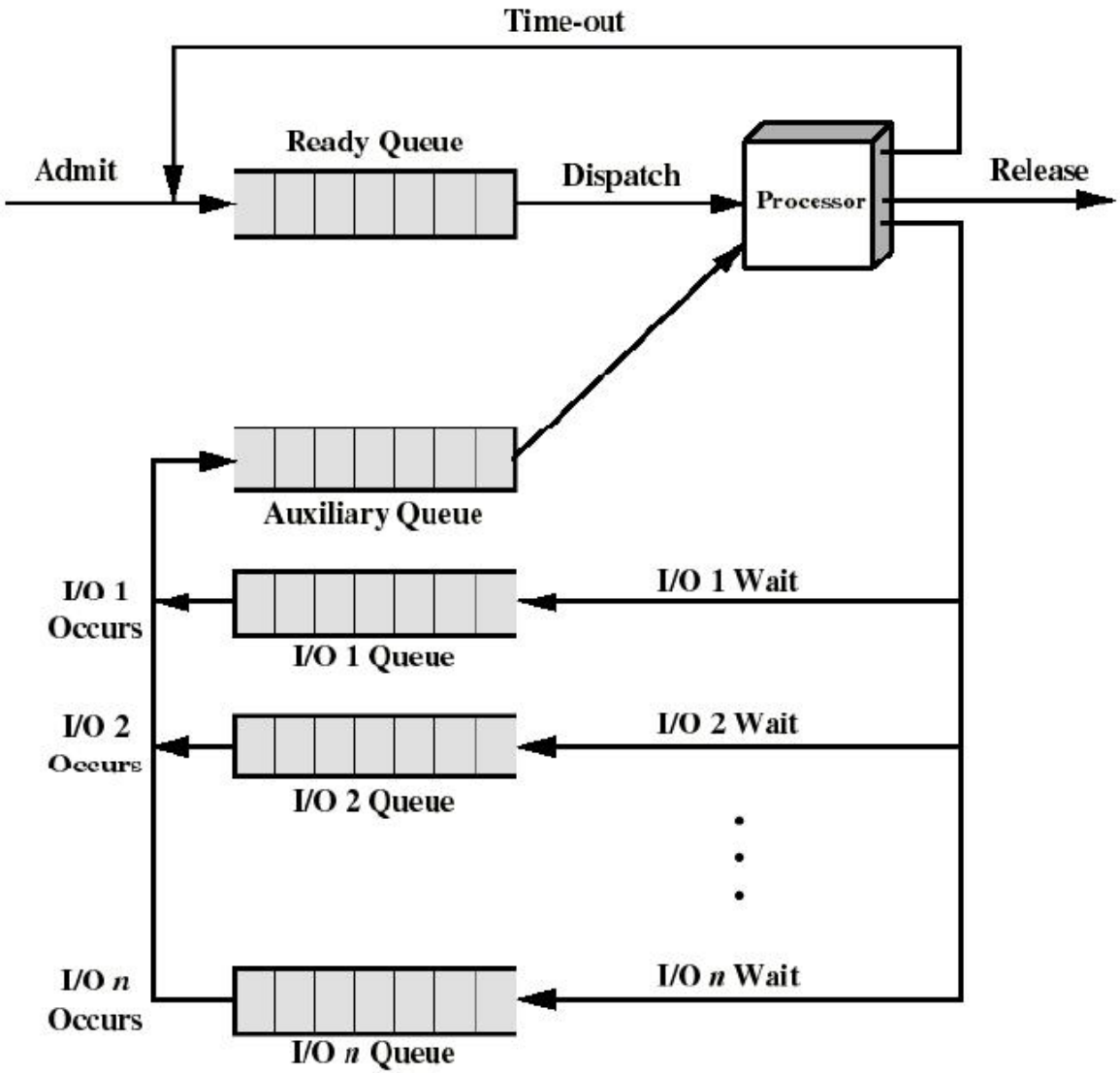
Round Robin: I/O vs. CPU-Bound Processes

- An I/O bound process uses the CPU for a time less than the time quantum and then is blocked waiting for I/O
- A CPU-bound process can run for all its time slice and is put back into the ready queue (thus getting in front of blocked processes)

➜ Virtual Round Robin

- When an I/O has completed, the blocked process is moved to an auxiliary queue which gets preference over the main ready queue
- A process dispatched from the auxiliary queue runs no longer than the basic time quantum minus the time spent running since it was selected from the ready queue

Virtual Round Robin

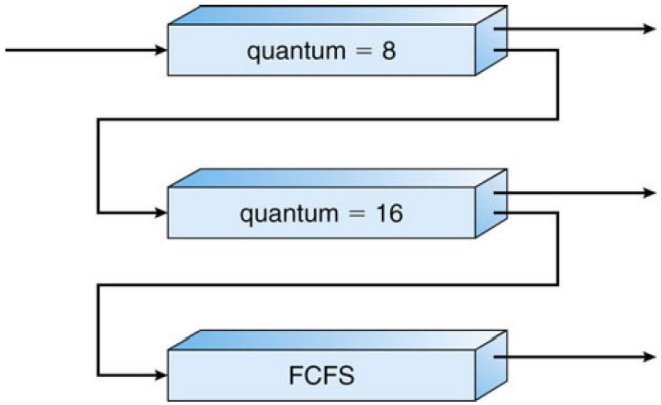


Multilevel Feedback (MLFB) Queue

- A process can move between the various queues
  - Aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Multilevel Feedback (MLFB) Queue II

- Example with three queues:



- Q_0 - RR time quantum 8 ms
- Q_1 - RR time quantum 16 ms
- Q_2 - FCFS

- Scheduling:
  - A new job enters queue Q_0 which is served RR. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served RR and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

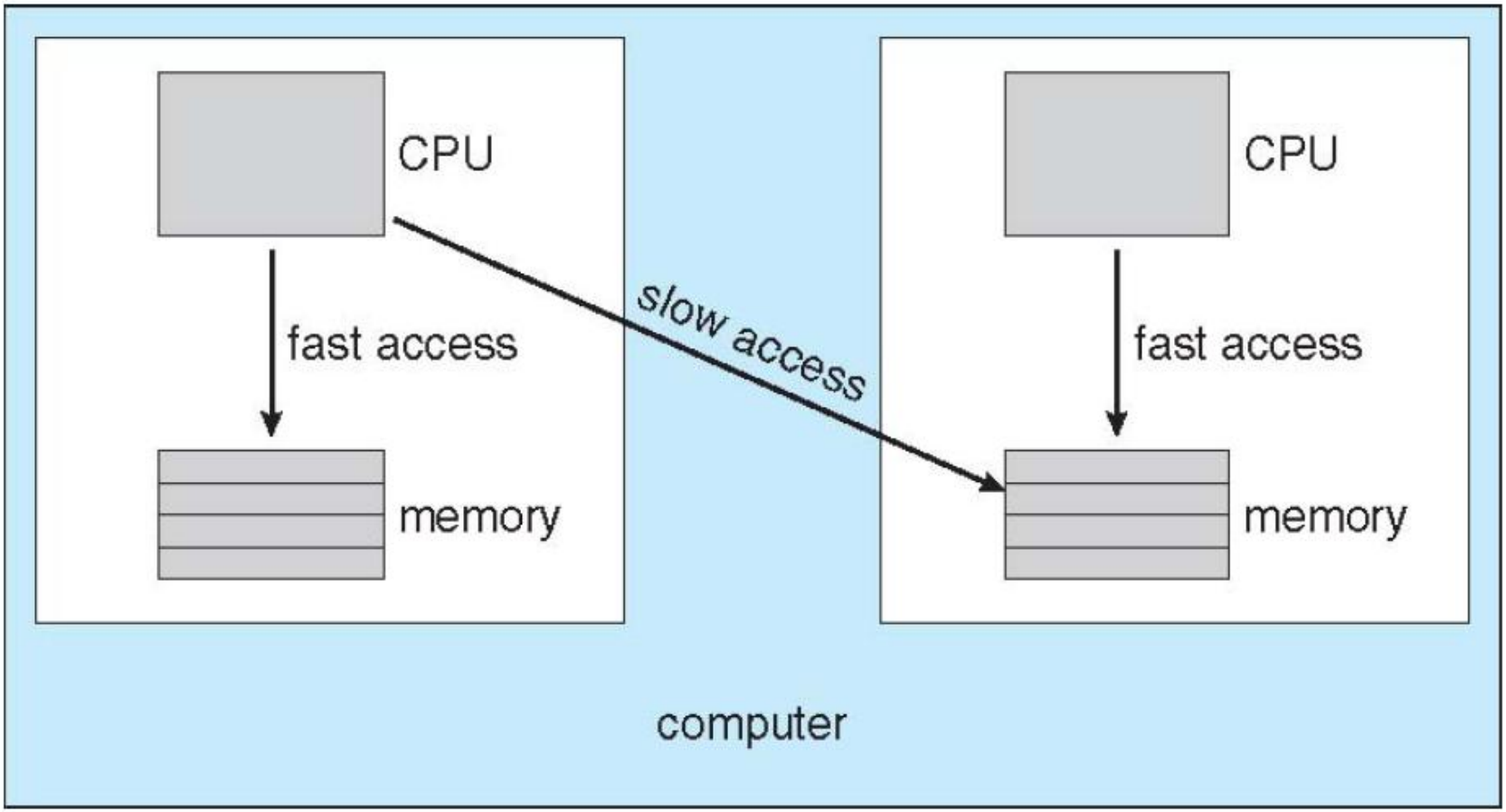
Lottery Scheduling

- Give each process some lottery tickets
- On each time slice randomly pick a ticket (lottery)
- Ticket owner gets CPU for one time slice
- Scheduling behavior is dependent on number of tickets a process owns
- How to assign tickets?
  - To approximate SRTF, short runners get more, long runners get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a process affects all others proportionally, independent of how many tickets each one possesses
  - Processes can hand over tickets to other procs, e.g. a client to a server (called ticket donation)

Multiple-Processor Scheduling

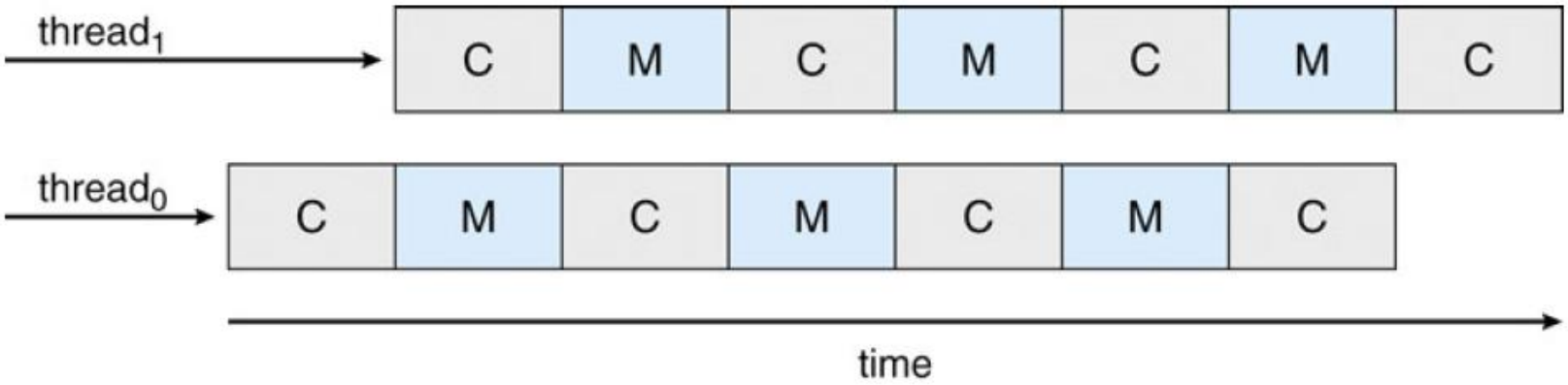
- CPU schedulingis more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor
- Asymmetric multiprocessing - only one processor accesses the system data structures, alleviating the need for data sharing
- Symmetric multiprocessing (SMP) - each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- Processor affinity - process has affinity for processor on which it is currently running
  - soft affinity
  - hard affinity

NUMA Scheduling

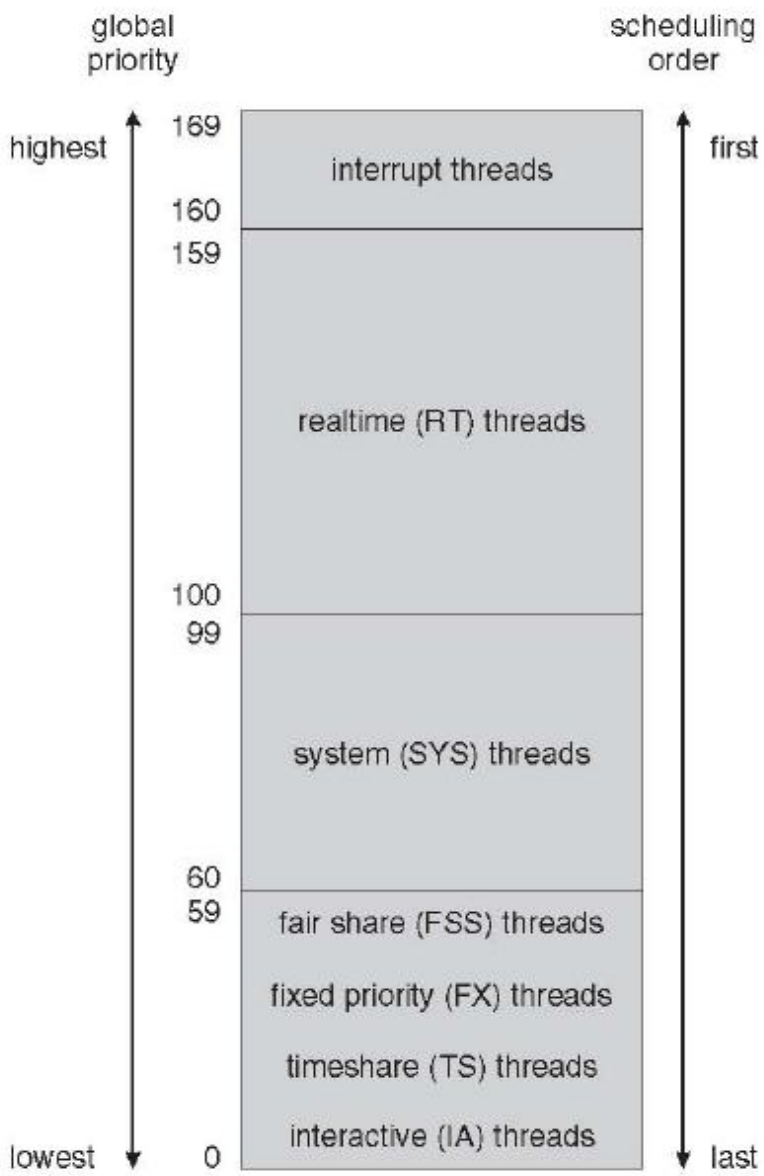


SMT und Multi-Core Scheduling

- Recent trend to place multiple processor cores on same physical chip
- Faster and consume less power
- Multiple threads per core (SMT) also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens



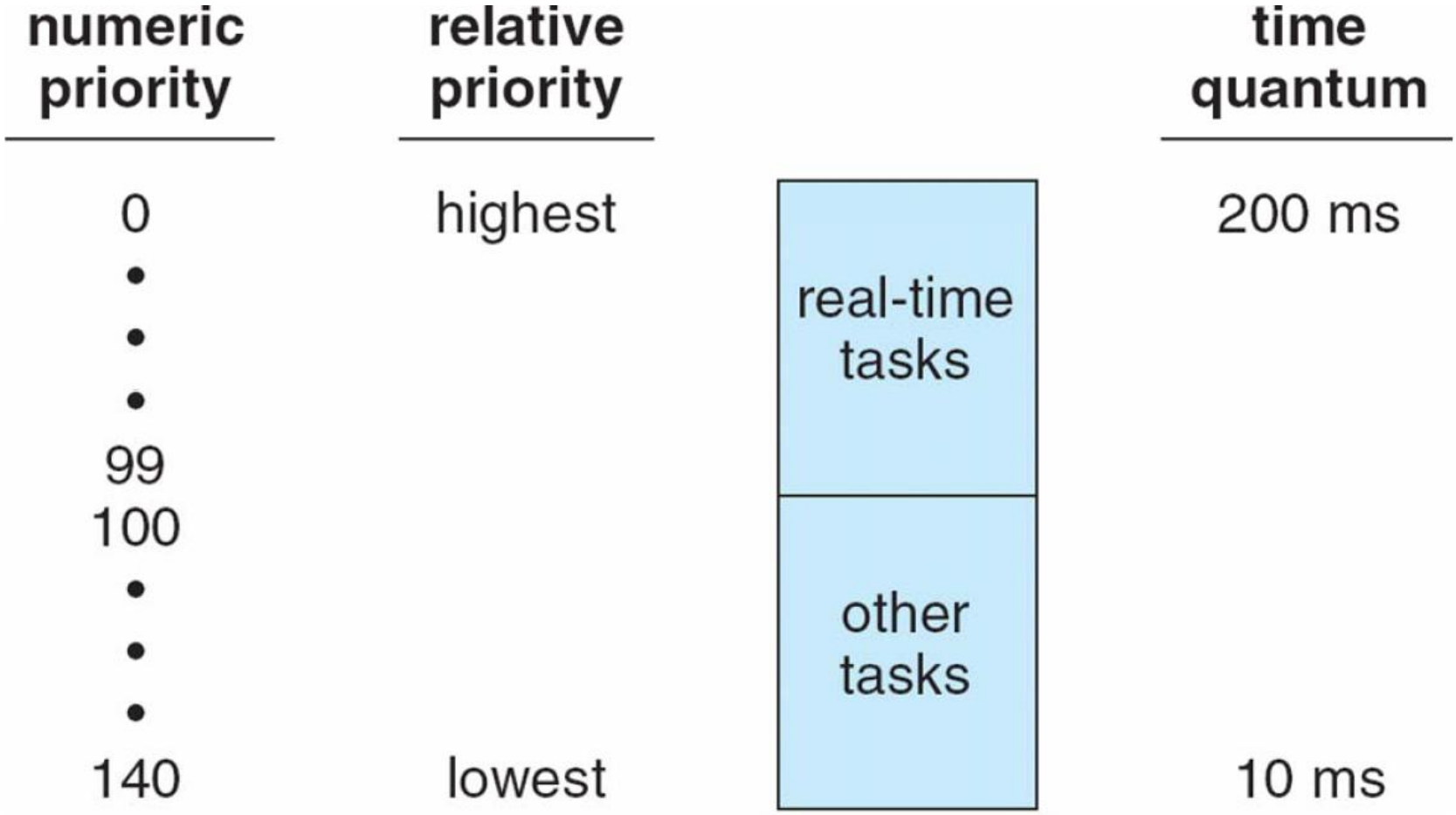
Solaris Scheduling



Solaris TS Scheduler

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Linux O(1) Scheduler



Linux O(1) Task Arrays

- see "Understanding the Linux 2.6.8.1 CPU Scheduler" by Josh Aas, Feb. 2005

Deterministic Modelling

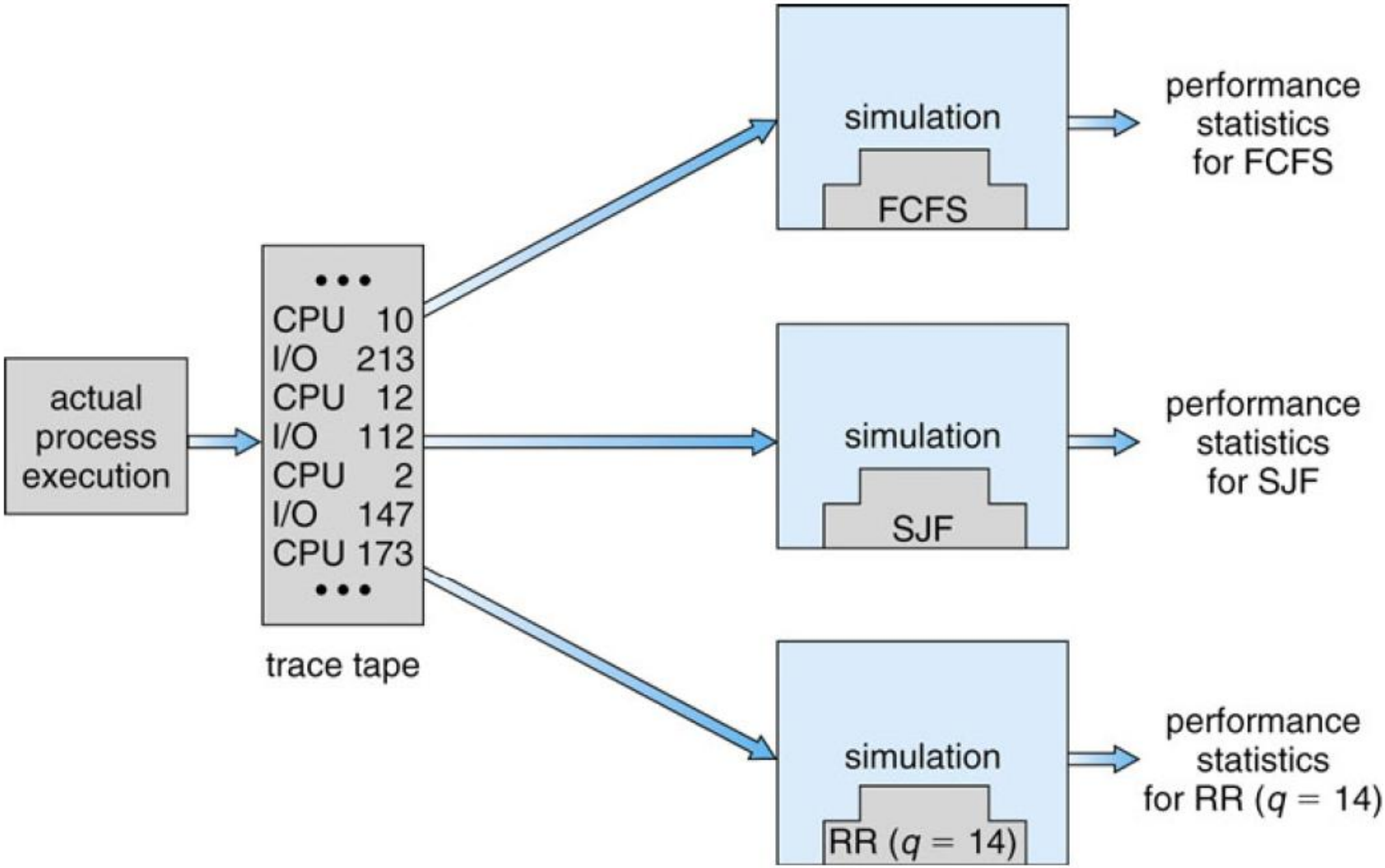
- Predetermined Workload
- Deterministic scheduling algorithm

➜ Formula or concrete numbers that evaluate the performance of a scheduling policy for that workload

Queueing Models

- Based on the foundations of Queueing Theory
- System is modelled as a network of queues
- Activities (CPU and I/O) are modelled with distribution of arrival time, execution time, ... (often unrealistic)
- Queueing-network analysis determines the average throughput, utilization, queue length, waiting time, ...
- Little's formula: $n = \lambda * W$ (valid for all policies and distributions)
  - n: average queue length
  - λ: average arrival rate
  - W: average waiting time per process in the queue

Simulations



Implementation and Measurement

- Implementation with “real” operating system sources
- Measurement with “typical” workload on “real” hardware

➜ Minor changes in the execution environment can have dramatic impact on the outcome of measurements