6. CPU Scheduling

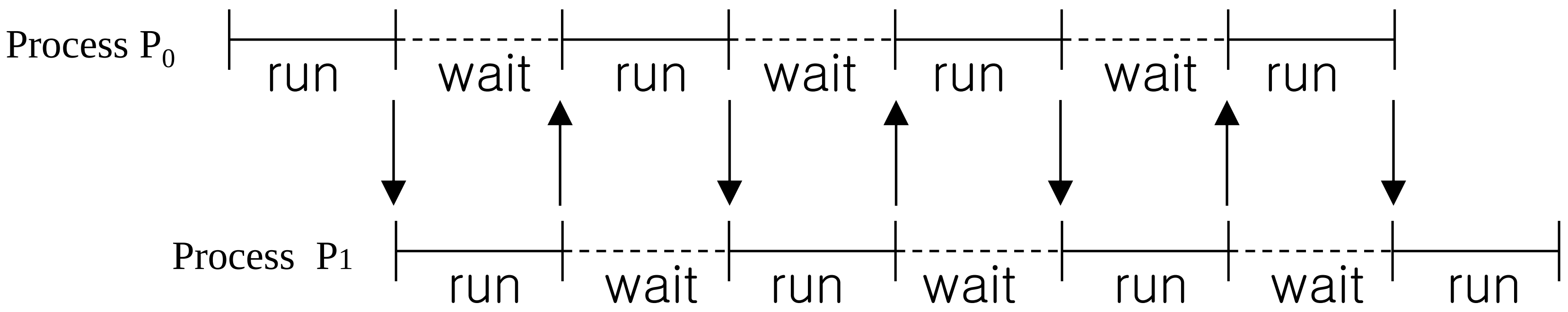
ECE30021/ITP30002 Operating Systems

Agenda

- Basic concepts
- Scheduling algorithms
- Multiple-processor scheduling
- Thread scheduling
- Operation system examples
- Algorithm evaluation

Basic Concepts

- Motivation: maximum CPU utilization obtained with multiprogramming and multitasking
  - Resources (including CPU) are shared among processes

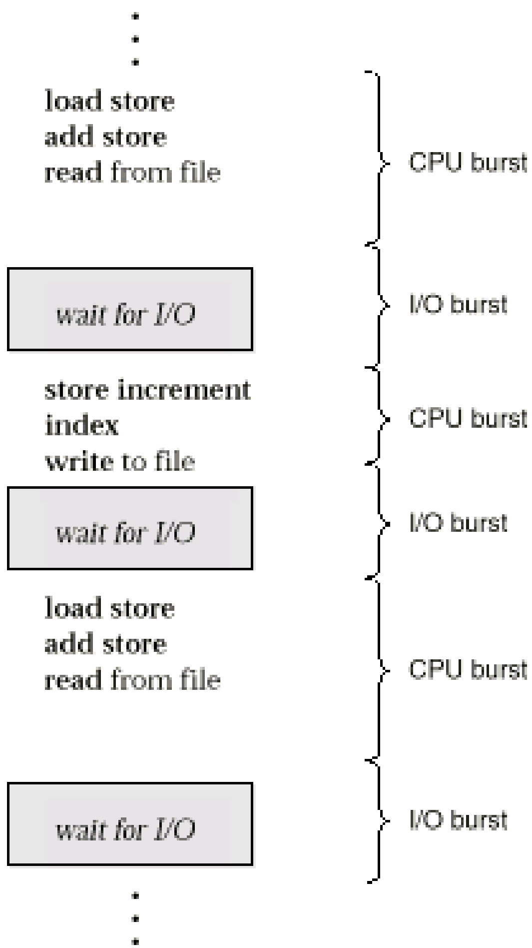


Multiprogramming

CPU–I/O Burst Cycle

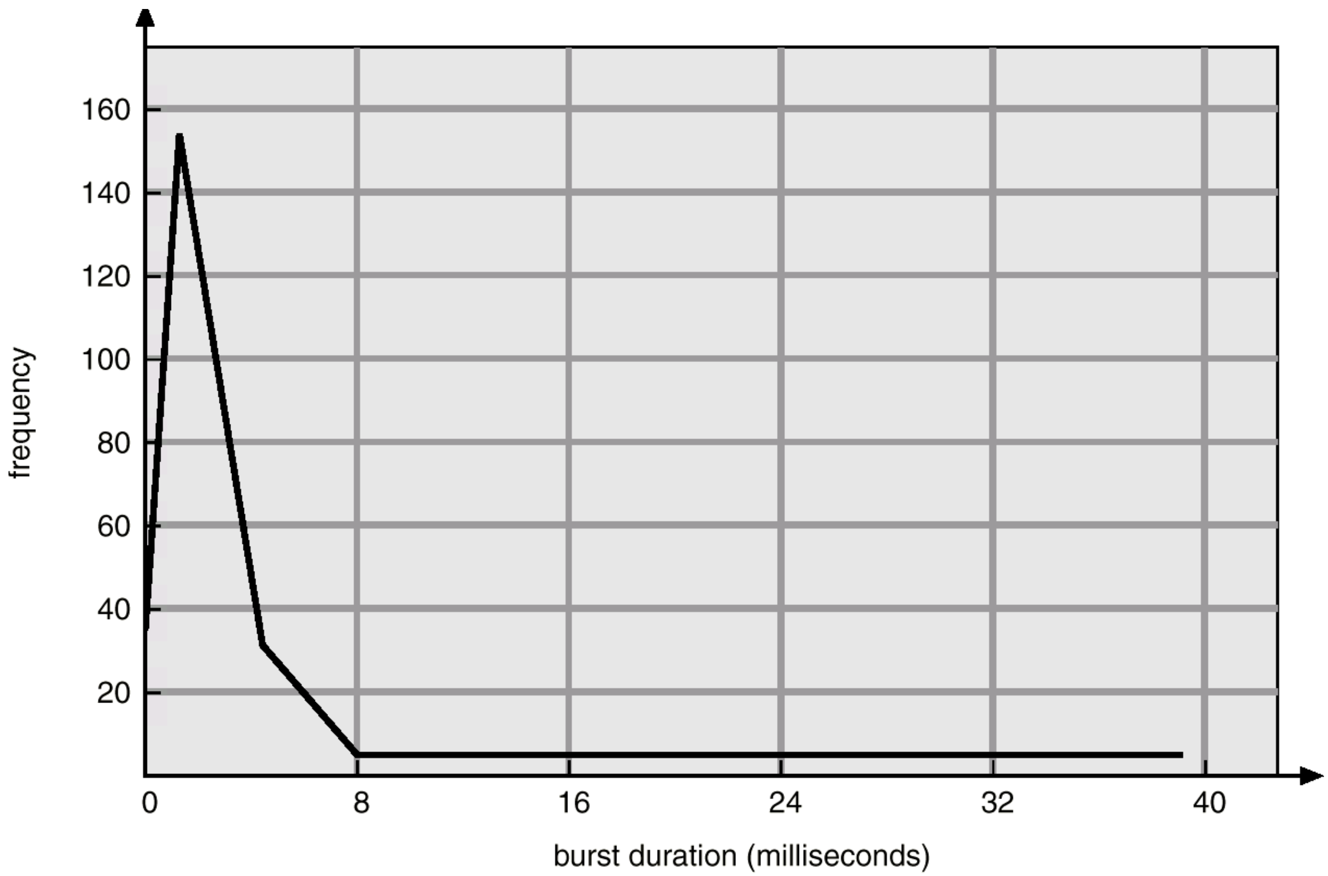
- Process execution consists of cycle of CPU execution and I/O wait
  - First/last burst: CPU burst

- Types of processes
  - I/O-bound process
    - Consists of many short CPU bursts
  - CPU-bound process
    - Consists of a few long CPU bursts



Histogram of CPU-burst Durations

- Exponential or hyper exponential



CPU Scheduler

- CPU scheduler (= short term scheduler)
  - Selects a process from ready queue and allocate CPU to it

- Implementation of ready queue
  - FIFO, priority queue, tree, unordered linked list
    - Each process is represented by PCB

Preemptive Scheduling

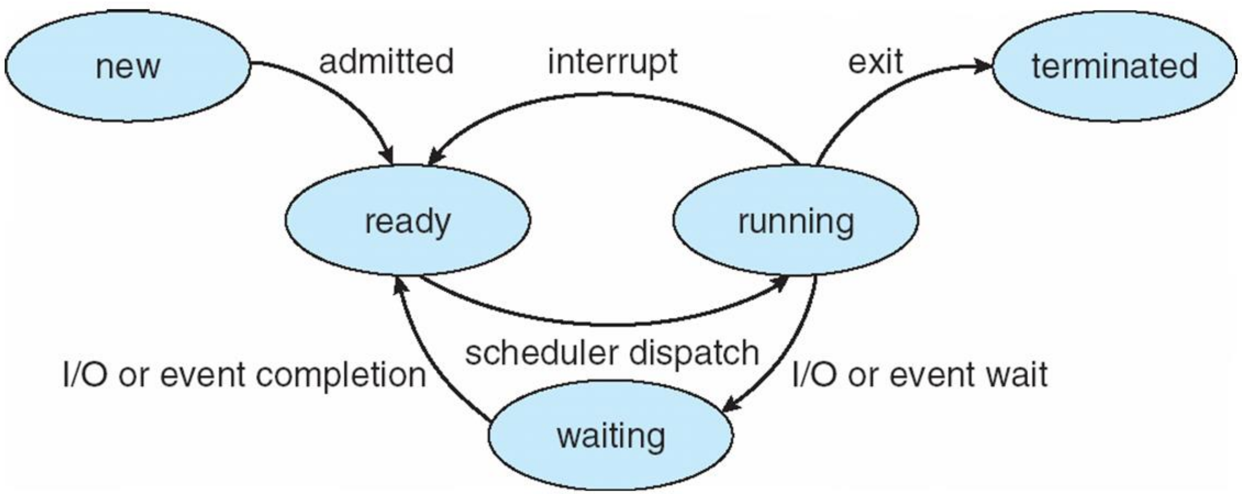
- Preemptive scheduling: scheduling may occur while a process is running.

  Ex) interrupt, process with higher priority

When Scheduling Occurs?

1. A process switches from running state to waiting state
2. A process switches from running state to ready state
   ex) an interrupt occurs
3. A process switches from waiting state to ready state
   ex) completion of I/O
4. A process terminates

-> 1 and 4 are inevitable, but 2 and 3 are optional

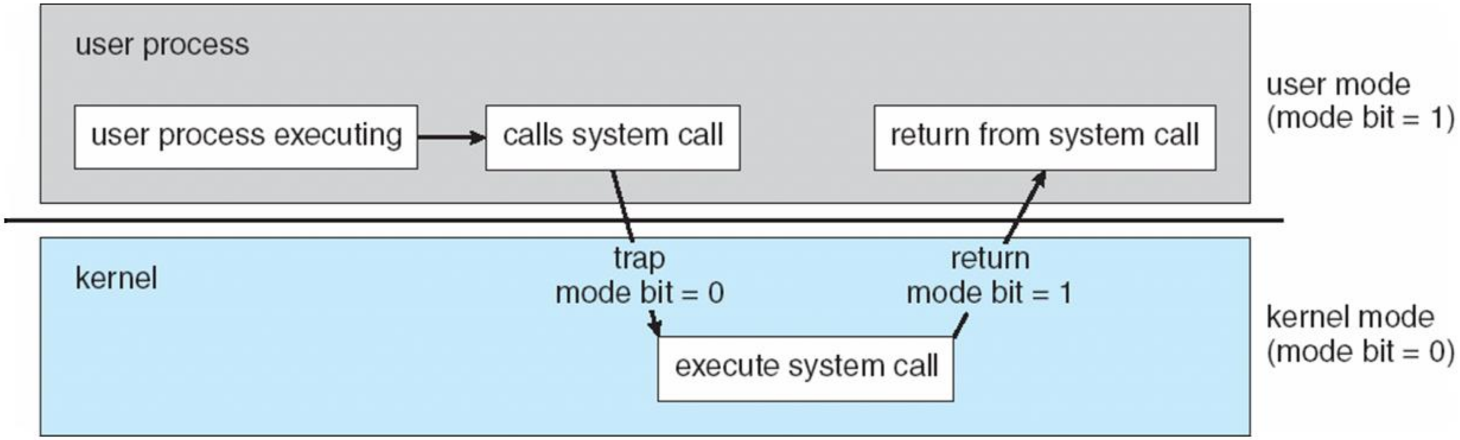


Preemptive Scheduling

- Non-preemptive (or cooperative) scheduling
  - Scheduling can occur at 1, 4 only
  - Running process is not interrupted.
- Preemptive scheduling
  - Scheduling can occurs at 1, 2, 3, 4
  - Scheduling may occur while a process is running
  - Requires H/W support and shared data handling

Preemption of OS Kernel

- Preemptive kernel: kernel allows preemption in system call.
  - Desirable for real-time support.



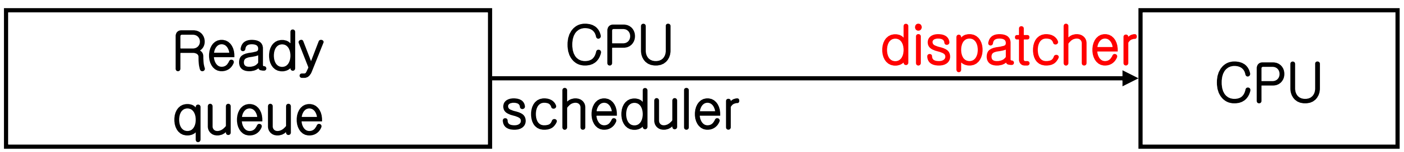
cf. System calls are non-preemptive in most OS.

- Keeps kernel structure simple

Dispatcher

- Dispatcher: a module that gives control of CPU to the process selected by short-term scheduler
  - Switching context
  - Switching to user mode
  - Jumping to proper location in user program

- Dispatch latency: time from stopping one process to starting another process



Scheduling Criteria

- CPU utilization: keep the CPU as busy as possible
- Throughput: # of processes completed per time unit
- Turnaroundtime: interval from submission of a process to its completion
- Waiting time: sum of periods spent waiting in ready queue
- Response time: time from submission of a request to first response

-> Importance of each criterion vary with systems

- Measure to optimize
  - Average / minimum / maximum value
  - Variance

Agenda

- Basic concepts
- Scheduling algorithms
- Multiple-processor scheduling
- Thread scheduling
- Operation system examples
- Algorithm evaluation

Scheduling Algorithms

- First-come, first-served (FCFS) scheduling
- Shortest-job-first (SJF) scheduling
- Priority scheduling
- Round-robin scheduling
- Multilevel queue scheduling
- Multiple feedback-queue scheduling

First-Come, First-Served Scheduling

- Process that requests CPU first, is allocated CPU first
  - Non-preemptive scheduling
  - Simplest scheduling method

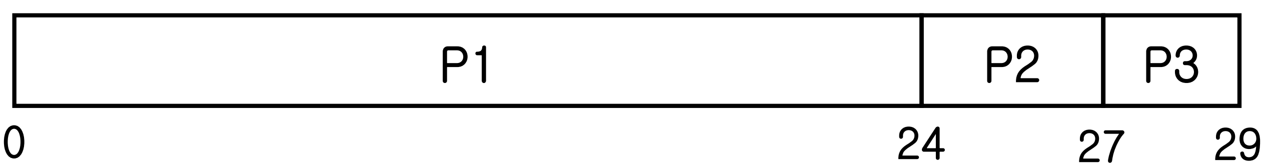
- Sometimes average waiting time is quite long
  - CPU, I/O utilities are inefficient

Ex) Three processes arrived at time 0

Process	Burst Time	Waiting time
P1	24	0
P2	3	24
P3	2	27

*time unit: msec

Average waiting time:
(0 + 24 + 27)/3 = 17



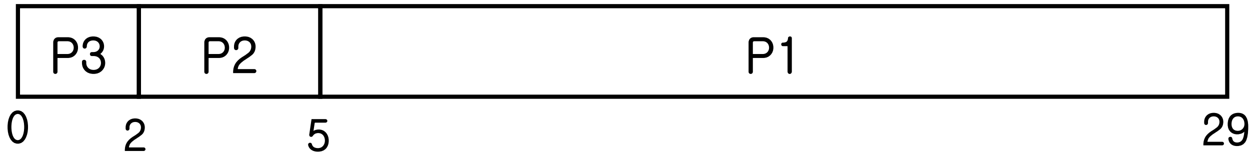
Shortest-Job-First Scheduling

- Assign to the process with the smallest next CPU burst

Process	Burst Time	Waiting time
P1	24	5
P2	3	2
P3	2	0

Average waiting time:

(5 + 2 + 0)/3 = 2.3



  - SJF algorithm is optimal in minimum waiting time
  - Problem: difficult to know **length of next CPU burst**

Shortest-Job-First Scheduling

- Predicting next CPU burst from history
  - Exponential averaging

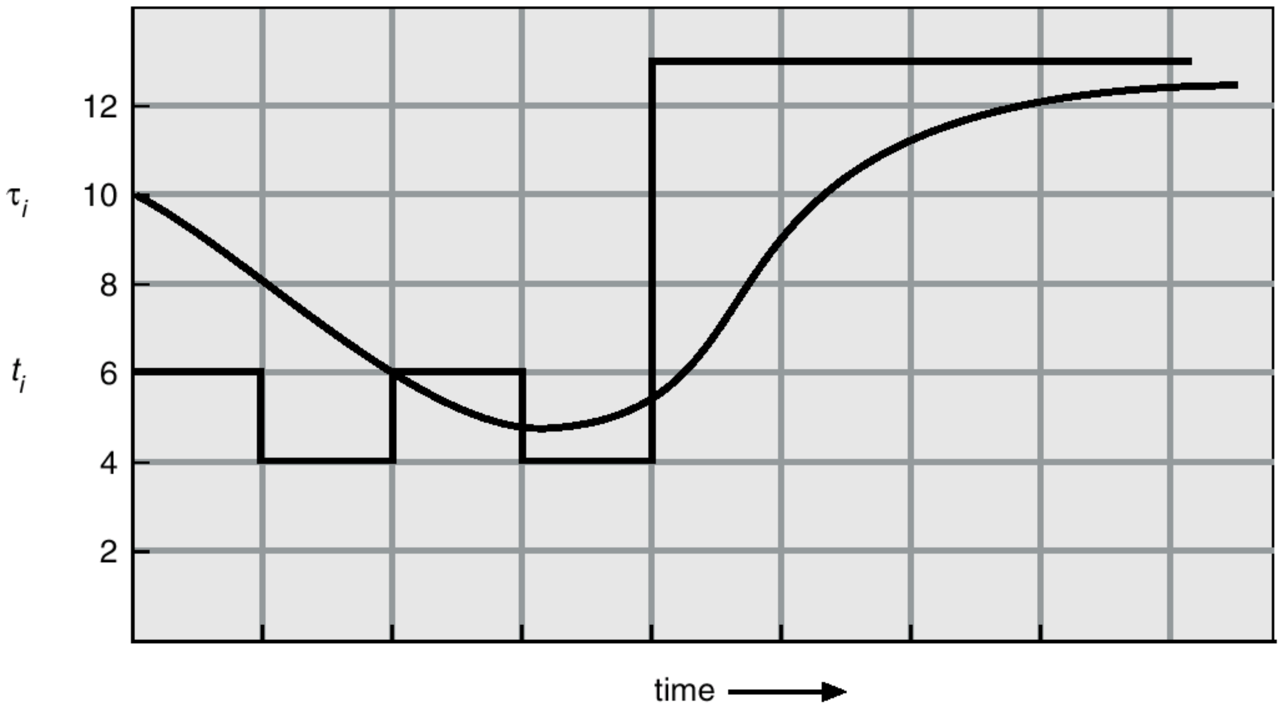
$$\tau_{n+1} = \alpha t_n + (1-\alpha)\ \tau_n$$

- t_n : actual length of n-th CPU burst
- τ_n : prediction for n-th CPU burst
- α : a coefficient between 0 and 1
  - α = 0: recent history has no effect
  - α = 1: only recent history matters
  - Usually, α = 0.5

Note: older history affects less

$$\tau_{n+1} = \alpha\ t_n + (1 - \alpha)\ \alpha\ t_{n-1} + \ldots + (1 - \alpha\ )^j\ \alpha\ t_{n-j} + \ldots + (1 - \alpha\ )^{n+1}\ \tau_0$$

Prediction of CPU Burst



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

Shortest-Job-First Scheduling

- Preemptive version of SJF scheduling
  - Shortest-remaining-time-first scheduling

Process	Arrival Time	Burst Time	Waiting Time
P1	0	7	9
P2	2	4	1
P3	4	1	0
P4	5	4	2

P1 (2)	P2 (2)	P3 (1)	P2 (2)	P4 (4)	P1 (5)

0 2 4 5 7 11 16

Average waiting time: 3

Priority Scheduling

- CPU is allocated the process with the highest priority
  - Equal-priority processes: FCFS

  Note: each process has its priority
  - In this text, lower number means higher priority

Ex)

Process	Burst Time	Priority	Waiting Time
P1	10	3	6
P2	1	1	0
P3	2	4	16
P4	1	5	18
P5	5	2	1

Average waiting time: 8.2

P2	P5	P1	P3	P4

0 1 6 16 18 19

Priority Scheduling

- Priority can be assigned internally and externally
  - Internally: determined by measurable quantity or qualities
    - Time limit, memory requirement, # of open files, ratio of I/O burst and CPU burst, …
  - Externally: importance, political factors

- Priority scheduling can be preemptive or non-preemptive.

- Major problems
  - Indefinite blocking (= starvation) of processes with lower priorities

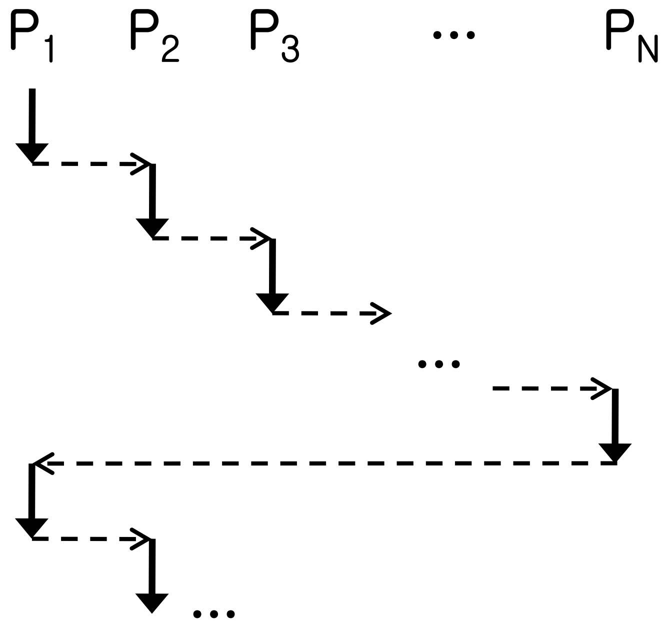
  -> Solution: aging (gradually increase priority of processes waiting for long time)

Round-Robin Scheduling

- Similar to FCFS, but it's preemptive
  - Designed for time-sharing systems
  - CPU time is divided into time quantum (or time slice)
    - A time quantum is 10~100 msec.

    Cf. switching latency: 10 μsec
  - Ready queue is treaded as **circular queue**
  - CPU scheduler goes around the ready queue and allocate CPU time up to 1 time quantum

Round-Robin Scheduling



Round-Robin Scheduling

Ex) Time quantum = 4

Process	Burst Time	Waiting Time
P1	24	6
P2	3	4
P3	3	7

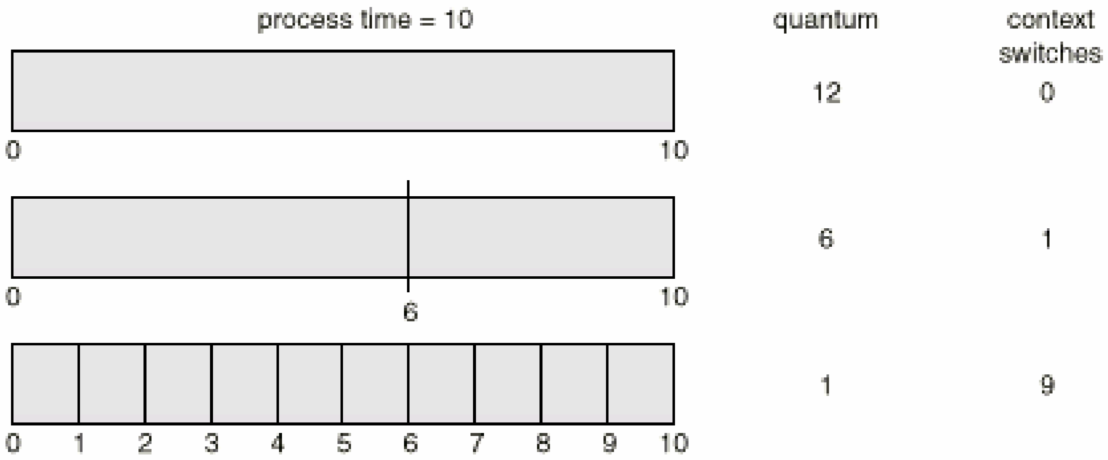
Average waiting time: 5.66

P1	P2	P3	P1	P1	P1	P1	P1

0 4 7 10 14 18 22 26 30

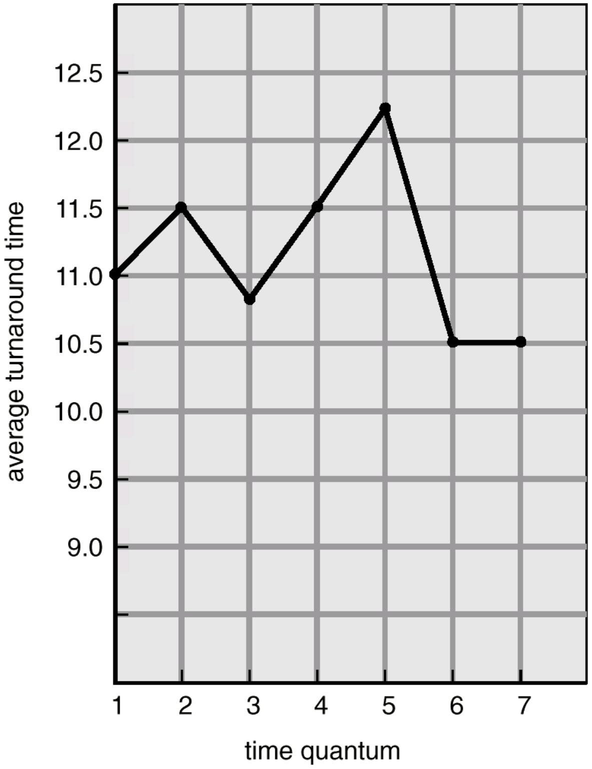
Round-Robin Scheduling

- Performance of RR scheduling heavily depends on size of time quantum.
  - Time quantum is small: processor sharing
  - Time quantum is large: FCFS
- Context switching overhead depends on size of time quantum.



Round-Robin Scheduling

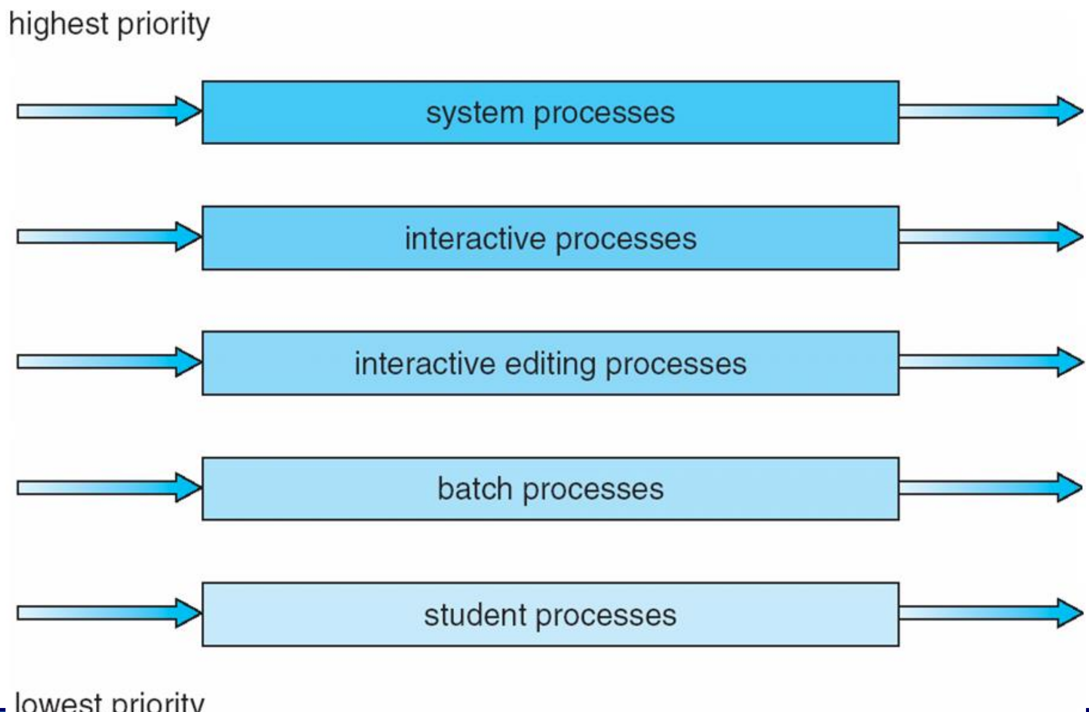
- Turnaroundtime also depends on size of time quantum
  - Average turnaround time is not proportional nor inverse-proportional to size of time quantum
  - Average turnaroundtime is improved if most processes finish their next CPU burst in a single time quantum
  - However, too long time quantum is not desirable
  - A rule of thumb: about 80% of CPU burst should be shorter than time quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue Scheduling

- Classify processes into different groups and apply different scheduling
  - Memory requirement, priority, process type, …
- Partition ready queue into several separate queues



Multilevel Queue Scheduling

- Each queue has its own scheduling algorithm

- Scheduling among queues
  - Fixed-priority preemptive scheduling
    - A process in lower priority queue can run only when all of higher priority queues all empty
  - Time-slicing among queues

    Ex) foreground queue (interactive processes): 80%

    background queue (batch processes): 20%

- Assignment of a queue to a process is permanent.

Multilevel Feedback-Queue Scheduling

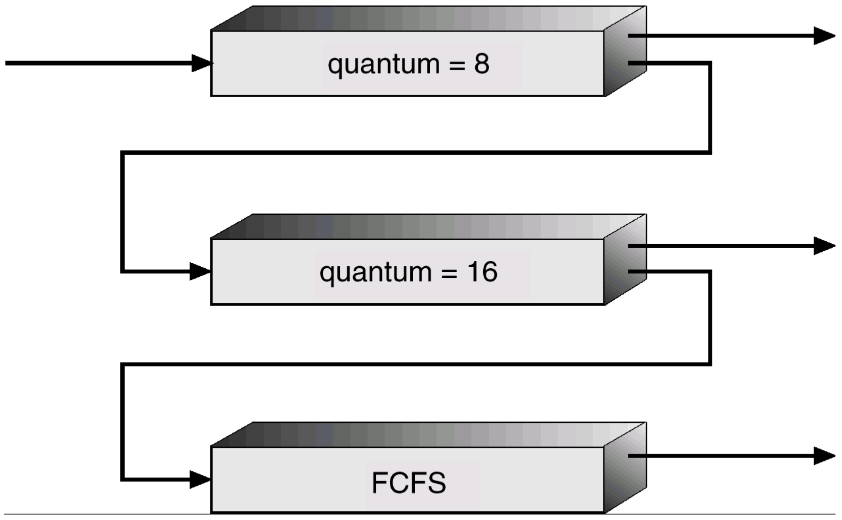
- Similar to multilevel queue scheduling, but a process can move between queues.
- Idea: separate processes according to characteristics of their CPU bursts.
  - If a process uses too much CPU time, move it to lower priority queue.
  - I/O-bound, interactive processes are in higher priority queues.

Multilevel Feedback-Queue Scheduling

Ex) Ready queue consists of three queues (0~2)

- Q_0 (time limit = 8 milliseconds)
- Q_1 (time limit = 16 milliseconds)
- Q_2 (FCFS)

➔ A new process is put in Q_0. if it exceeds time limit, it moves to lower priority queue



Multilevel Feedback-Queue Scheduling

- Parameters to define a multilevel feedback-queue scheduler
  - # of queues
  - Scheduling algorithm for each queue
  - Method to determine when to upgrade a process to higher priority queue
  - Method to determine when to demote a process to lower priority queue
  - Method to determine which queue a process will enter when it needs service

➔ The most complex algorithm

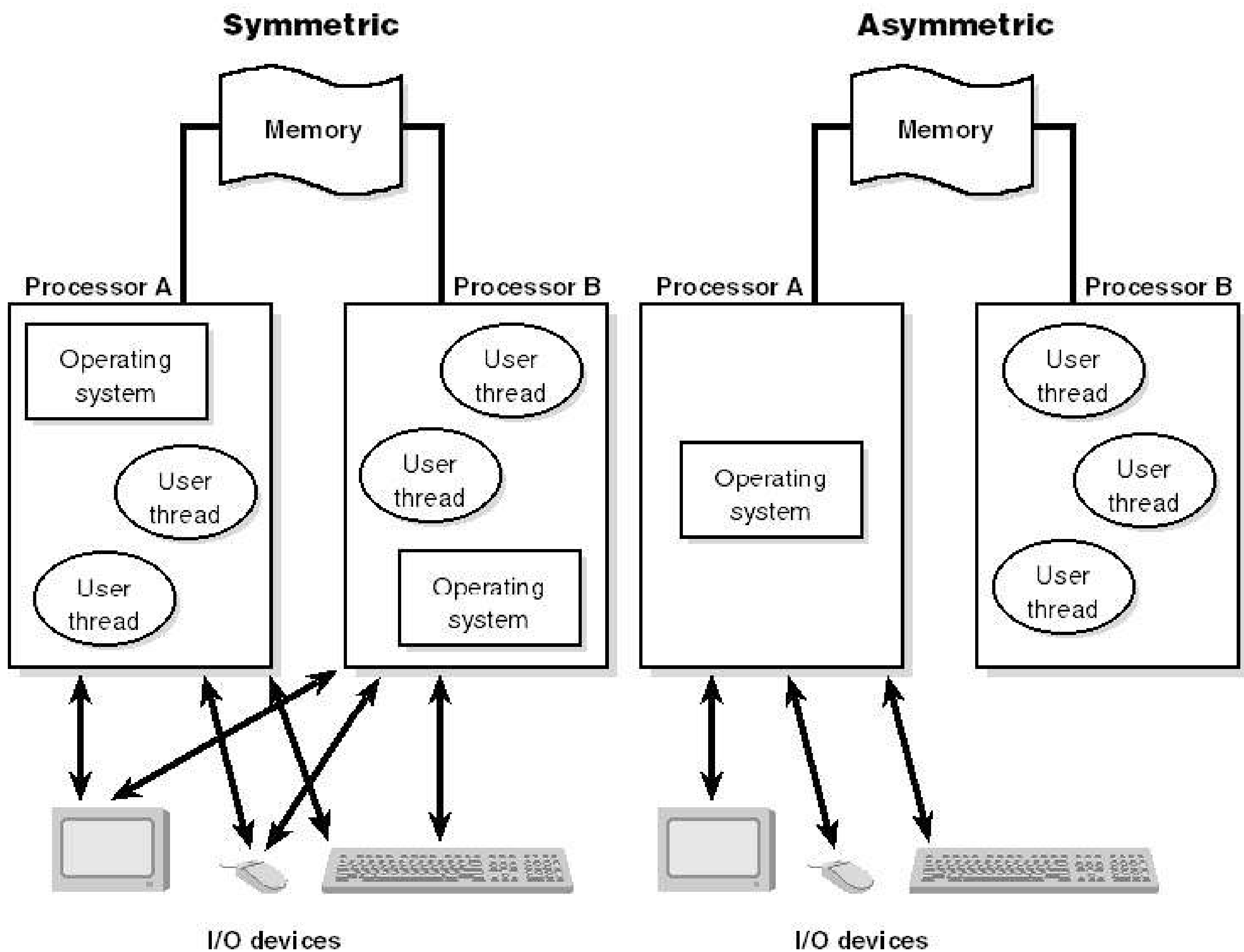
Agenda

- Basic concepts
- Scheduling algorithms
- **Multiple-processor scheduling**
- Thread scheduling
- Operation system examples
- Algorithm evaluation

Multiple-Processor Scheduling

- Multiple-processor system
  - Load sharing is possible.
  - Scheduling problem is more complex.
- There are many trials in multiple-processor scheduling, but no generally best solution
- In this text, all processors are assumed identical.
  - Any process can run on any processor

Symmetric vs. Asymmetric Multiprocessing



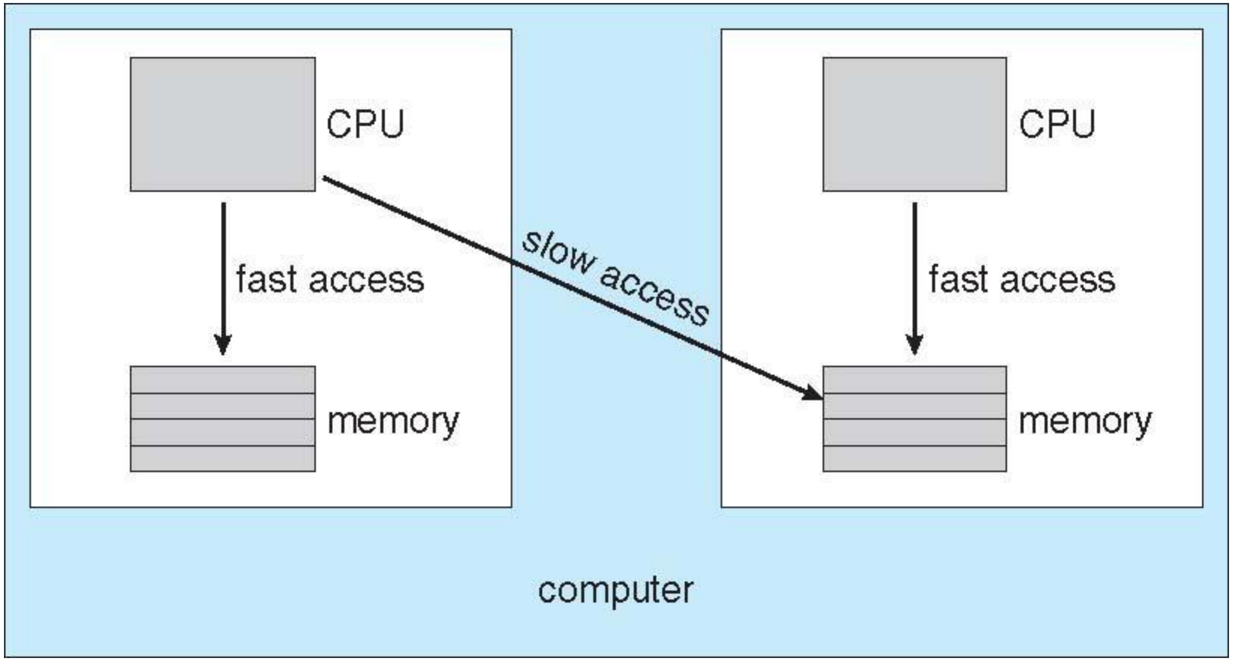
Processor Affinity

- Overhead of migration of processes from one processor to another processor
  - All contents of cache should be invalidated and re-populated

- Processor affinity: keeping a process running on the same processor to avoid migration overhead
  - **Soft affinity**: Although OS attempts to keep a process running on the same processor, a process may migrate between processors.
  - **Hard affinity**: It is possible to prevent a process from being migrated to other processor

Processor Affinity

- NUMA(Non-Uniform Memory Access) and CPU scheduling
  - A CPU has faster access to some parts of main memory than to other parts



Load Balancing

- Load balancing: attempt to keep the workload evenly distributed across all processors.
  - Necessary for system where each processor has its own ready queue (It's true for most modern OS's)

- Two general approaches
  - Push migration
    - A specific task periodically checks the load on each processor
    - If unbalance is found, move processes to idle or less-busy processors
  - Pull migration
    - An idle processor pulls a waiting task

  Note! Push and pull migration can be implemented in parallel.

  -> Linux, ULE scheduler for FreeBSD

- Load balancing can counteract the benefit of processor affinity

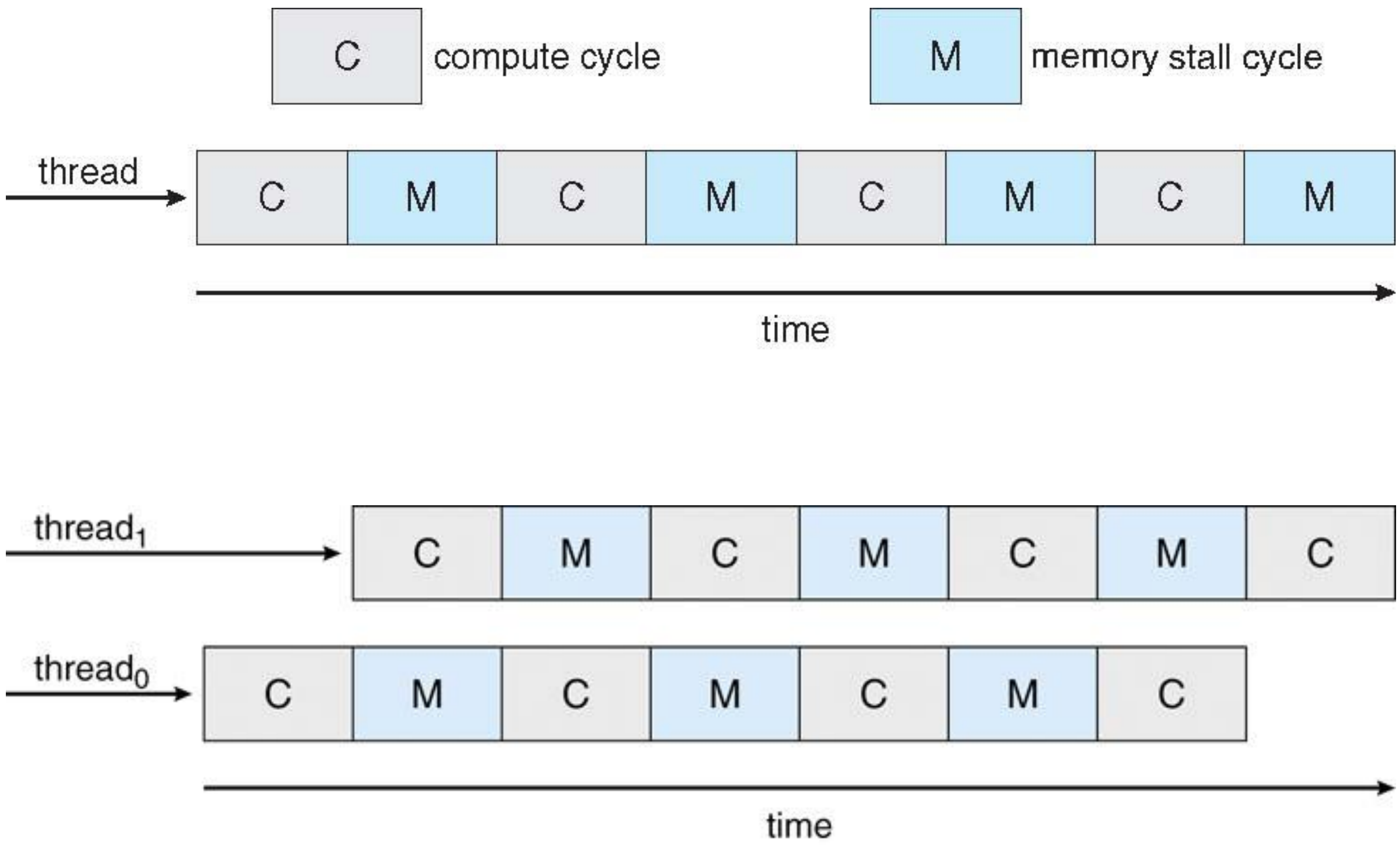
Multi-core Processors

- Multi-core processor: multiple processor cores on the same physical chip
  - Recent trend
  - Faster and consume less power than systems in which each processors has its own physical chip

- Scheduling issues on multi-core processor
  - Memory stall: for various reasons, memory access spends significant amount time waiting for the data to become available.

    Ex) accessing data not in cache
  - Remedy: multithreaded processor cores
    - Two or more hardware threads are assigned to each core

Multi-Threaded Processor Cores



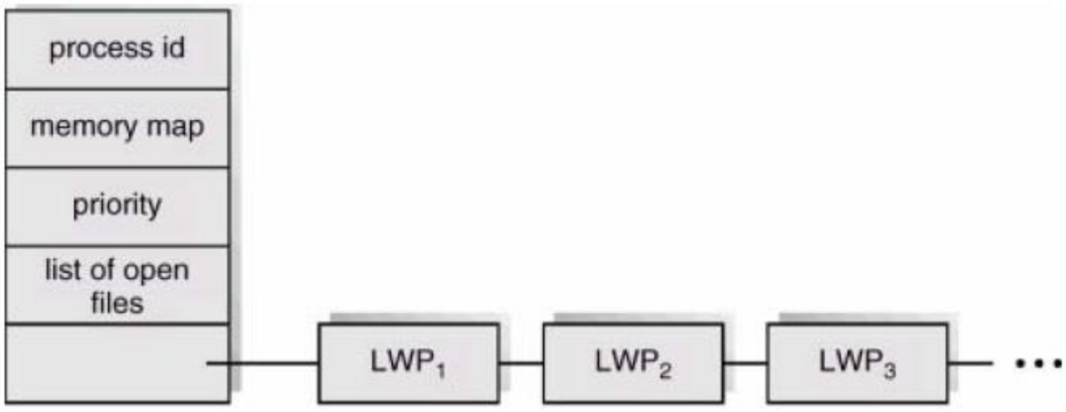
Agenda

- Basic concepts
- Scheduling algorithms
- Multiple-processor scheduling
- <u>Thread scheduling</u>
- Operation system examples
- Algorithm evaluation

Thread Scheduling

- Threads
  - User thread -> supported by thread library
    - Scheduled indirectly through LWP
  - Kernel thread -> supported by OS kernel

- Actually, it is not processes but kernel threads that are being scheduled by OS

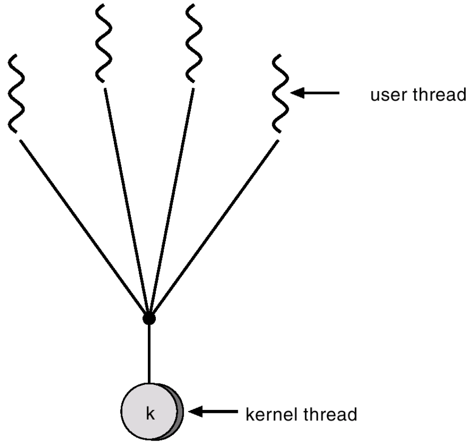


Solaris process

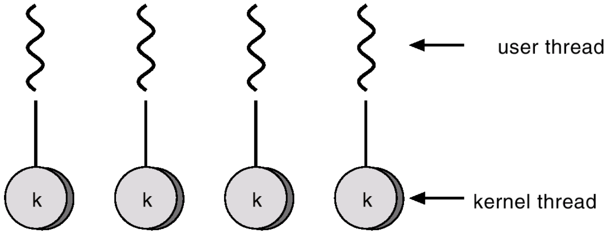
Contention Scope

- Process-contention scope (PCS)
  - Competition for LWP among user threads
    - Many-to-one model or many-to-many model
  - Priority based

- System-contention scope (SCS)
  - Competition for CPU among kernel threads



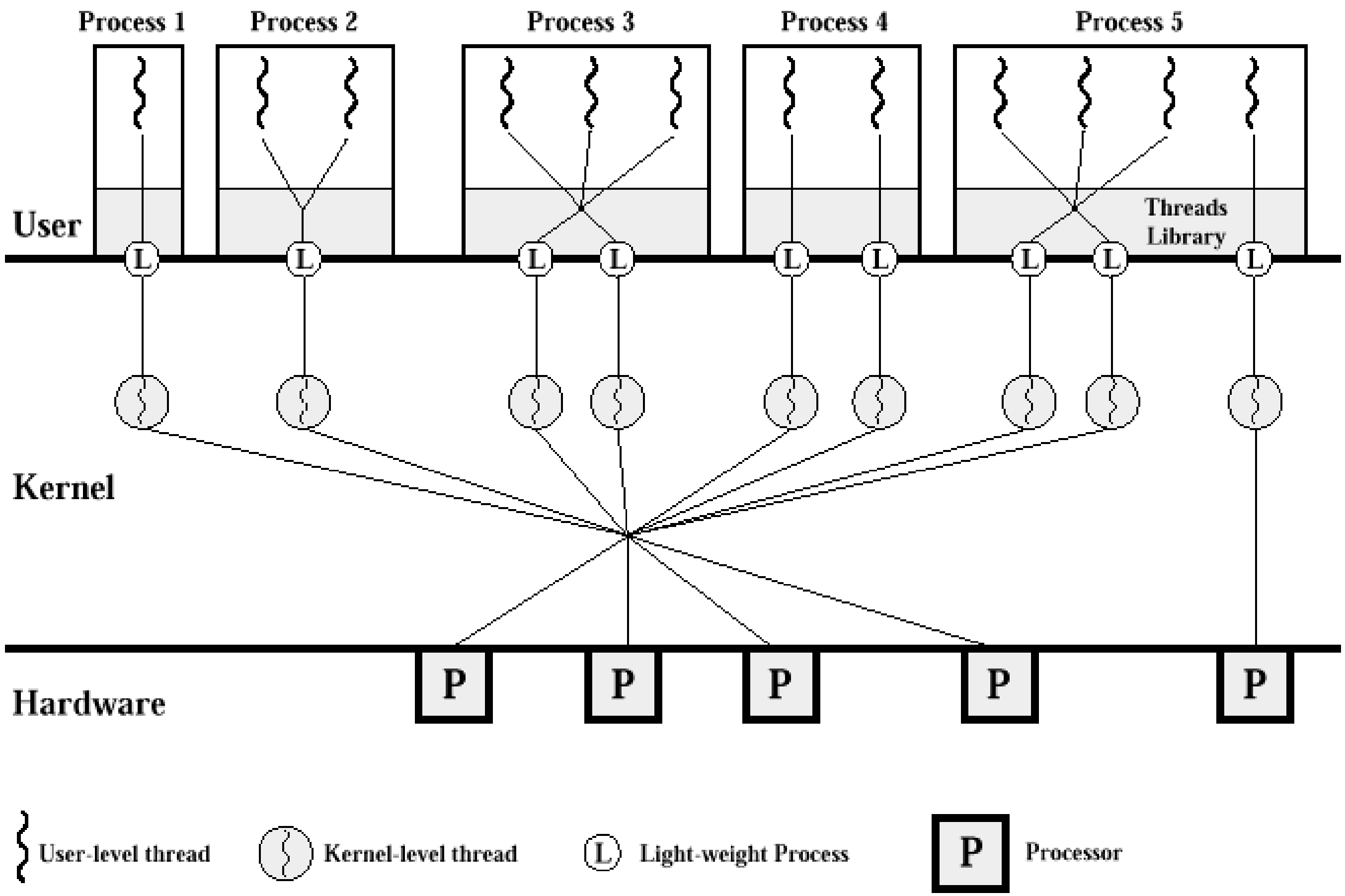
< many-to-one >



< one-to-one >

Scheduler Activation and LWP

- Connection between user / kernel threads through LWP



Pthread Scheduling

- In thread creation with Pthreads, we can specify PCS or SCS
  - PCS(PTHREAD_SCOPE_PROCESS):
    - In many-to-one, only PCS is possible
  - SCS(PTHREAD_SCOPE_SYSTEM):
    - In one-to-one model, only SCS is possible

  Note: On certain system, only certain values are allowed
    - Linux, MacOS X

- Related functions
  - pthread_attr_setscope(pthread_attr_t *attr, int scope)
  - pthread_attr_getscope(pthread_attr_t *attr, int *scope)

Pthread Scheduling

Example)

```
pthread_t tid;
pthread_attr_t attr;
int scope;

pthread_attr_init(&attr);

pthread_attr_getscope(&attr, &scope);
// scope is either PTHREAD_SCOPE_PROCESS
// or PTHREAD_SCOPE_SYSTEM

pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM)

pthread_create(&tid, &attr, runner, NULL);
```

Agenda

- Basic concepts
- Scheduling algorithms
- Multiple-processor scheduling
- Thread scheduling
- <u>Operation system examples</u>
- Algorithm evaluation

Operating System Examples

- Solaris
- Windows XP
- Linux

Windows Scheduling

- Priority-based, preemptive scheduling
  - *Dispatcher* handles scheduling
    - Dispatcher ensures the highest priority thread will always run
  - 32-level priority scheme
    - Variable class (1~15)
    - Real-time class (16~32)
    - Memory management (0)
  - If no ready thread is found, dispatcher runs *idle thread*

Windows Scheduling

- Priority classes
  - A process belongs to one of 6 levels of priority class
  - Each thread belongs to one of 7 levels of relative priority

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Windows Scheduling

- Altering priority
  - Priority class of a process: SetPriorityClass()
  - Base priority of a thread: SetThreadPriority()
- Adjusting priority for variable-priority class
  - Lower priority of a thread that expired entire time quantum
    - Never lowered below the base priority
  - Boost priority of a thread that released from waiting
    - Amount of increment vary with what it was waiting
- Scheduling of foreground / background processes
  - Time quantum for the foreground process is increased by some factor (typically, 3)

Linux Scheduling

- History
  - Two problems in Linux prior to version 2.5
    - Not adequate support for SMP system
    - Not scale well as # of tasks increases
  - Overhauled scheduler in Linux version 2.5
    - Scheduling complexity is O(1)
    - Increased support for SMP
    - Poor response time interactive processes
  - Linux 2.6
    - Completely Fair Scheduler (CFS)

Linux CFS Scheduler

- Based on scheduling classes
  - Each class is assigned a specific priority.
    - Default scheduling class
    - Real-time scheduling class
    - If necessary, new scheduling classes can be added
  - Different scheduling algorithms for different scheduling classes
  - Scheduler selects highest-priority task belonging to highest-priority class
- Relative priority
  - 'Nice value' from -20 to +19
    - Higher value represents lower priority ('nice' to others)

Linux CFS Scheduler

- CFS scheduler assigns a proportion of CPU processing time, rather than discrete values of time slices, to each task
  - The portion is calculated from nice value.
  - Proportions of CPU time are allocated from the value of targeted latency.
    - Targeted latency: interval of time during which every runnable task should run at least once
    - Targeted latency can increase if the number of active tasks in the system grows beyond a certain threshold

Linux CFS Scheduler

  - Per task variable vruntime(virtual run time) measures how long each task has run
  - vruntime of each task is associated with a decay factor based on the priority of a task

- Higher-priority task can preempt a lower-priority task.

Agenda

- Basic concepts
- Scheduling algorithms
- Multiple-processor scheduling
- Thread scheduling
- Operation system examples
- <u>Algorithm evaluation</u>

Algorithm Evaluation

- First of all, evaluation criteria should be determined

  Ex)

  1. Maximum response time is 1 second
  2. Satisfying 1, maximize CPU utilization
  3. Maximize throughput such that turnaround time is linearly proportional to total execution time

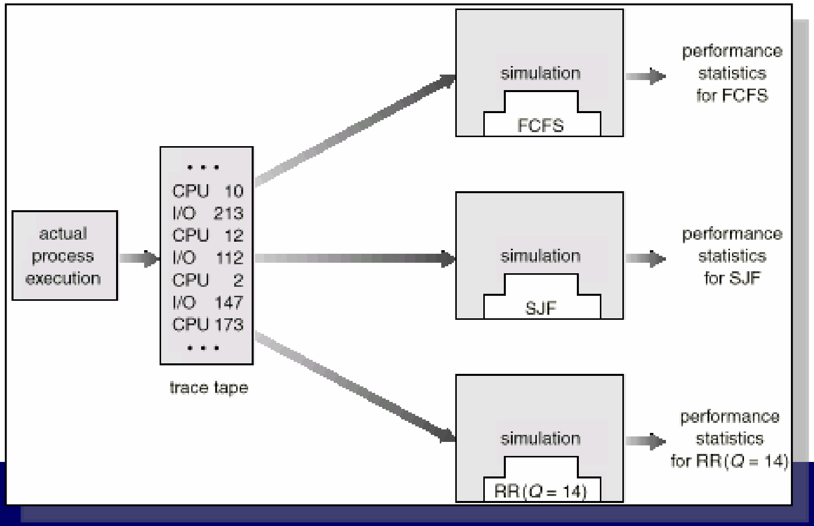
- Evaluation methods
  - Deterministic modeling
  - Queueing models
  - Simulations

Algorithm Evaluation

- Deterministic modeling
  - Evaluation for a particular predetermined workload
- Queueing models
  - Evaluation based on probabilistic distribution of…
    - CPU and I/O burst
    - Arrival time of CPU burst
  - Description of system
    - Network of servers, each of which has a ready queue
    - Given arrival rates and service rates, performance can be computed probabilistically

Algorithm Evaluation

- Simulations, using a programming model of computer system
  - Generator of process, CPU burst time, arrival, departure, …
  - Usually, the generator randomly generates the simulated events based of some distribution
  - Limitation: simulated situation isn't exactly same with the real situation
    - Especially, distribution gives no information about order of events
    - Remedy: using a record of real system. (trace tape)



Algorithm Evaluation

- **Implementation: the only way for accurate evaluation**
  - Performance depend not only on scheduling algorithm and OS support, but also user's interaction

  - Environment will change

    Ex) If short process is given higher priority, user may break a large process into several smaller processes.

  - Some OS's provides flexibility to alter scheduling scheme
    - Performance turning for specific (set of) applications
    - API's to modify priority of a process or thread