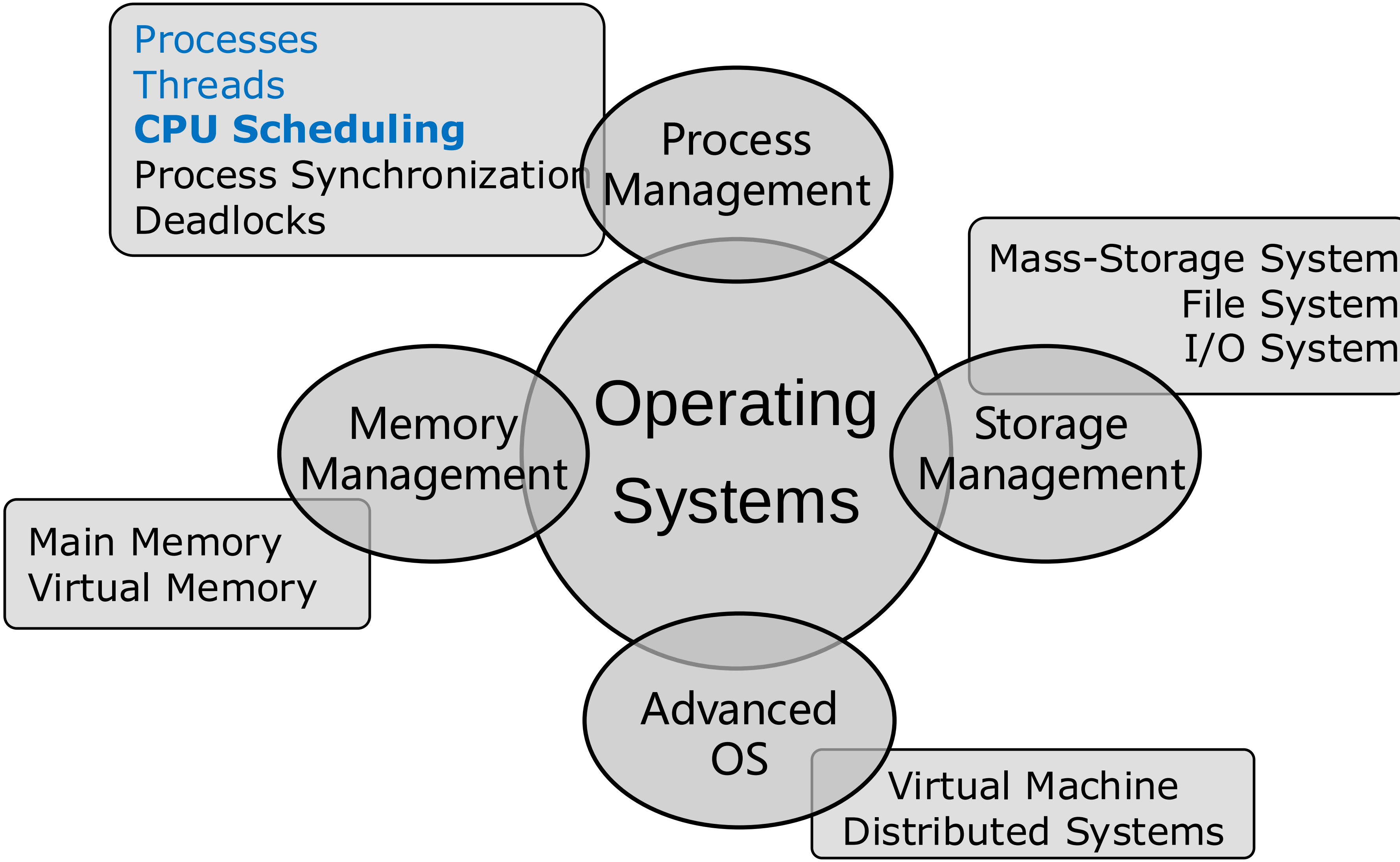
CPU Scheduling

Shengzhong Liu
Department of Computer Science and Engineering
Shanghai Jiao Tong University

Operating System Topics



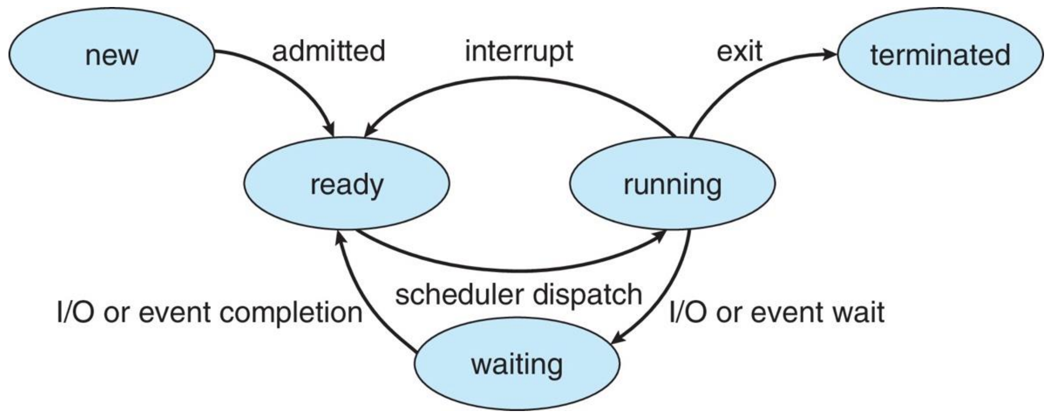
Outline

- Basic Concepts
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Algorithm Evaluation: Deterministic Modeling

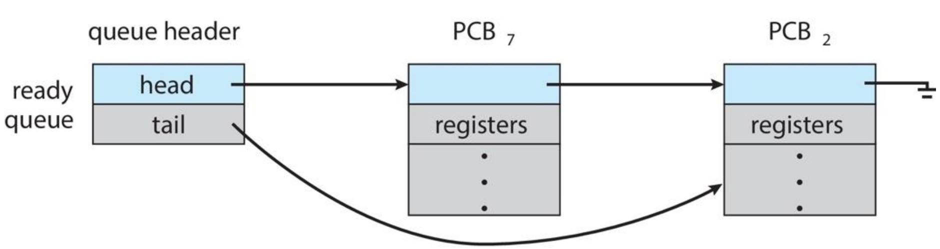
Basic Concept

Process State and Queue

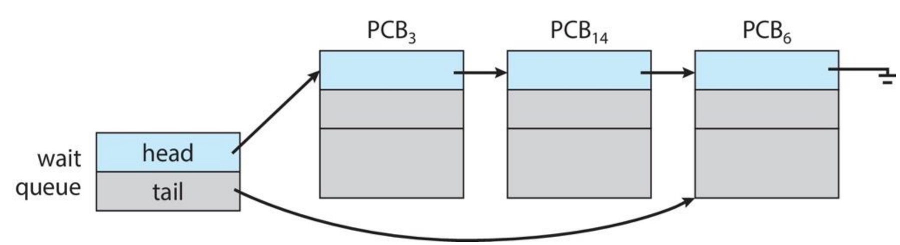
- Process state transition



- Ready queue and wait queue
  - **Ready queue** – processes wait to execute
  - **Wait queue** – processes waiting for an event (i.e., I/O)



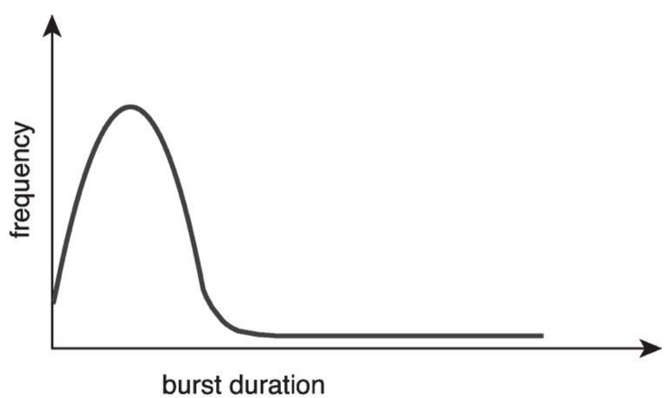
Ready Queue



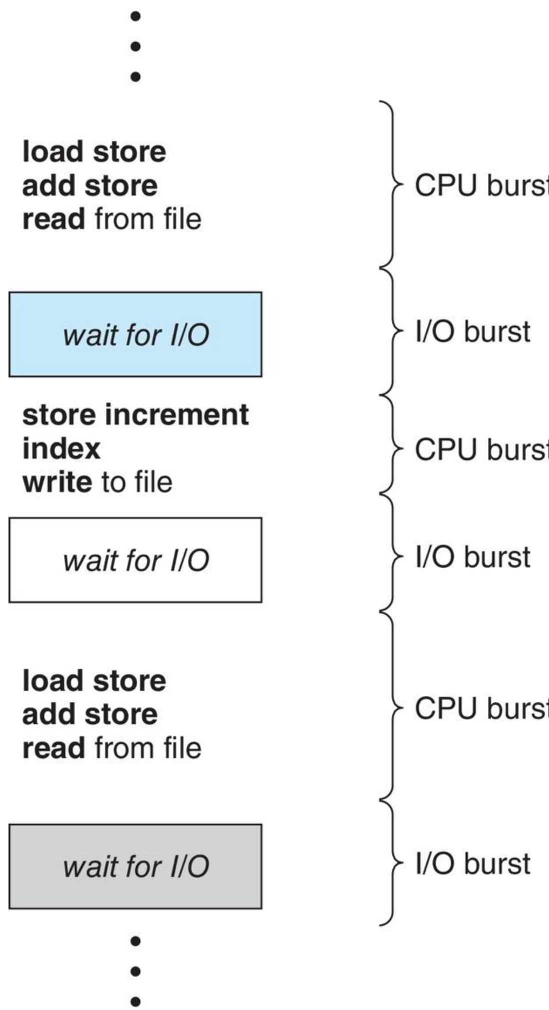
Wait Queue

Basic Concepts

- Maximize CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle
  - Process execution consists of an alternating sequence of **CPU bursts** and **I/O bursts**
- CPU burst distribution is of main concern



Histogram of CPU-burst durations



CPU Scheduler

- Selects from the processes in ready queue, and allocates the CPU to one
  - Queue may be ordered in various ways
- 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 state
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**, scheduling under 2 and 3 is **preemptive**
  - **Nonpreemptive Scheduling**: Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state
  - **Preemptive Scheduling**: The CPU is allocated to the process for a limited amount of time and then taken away

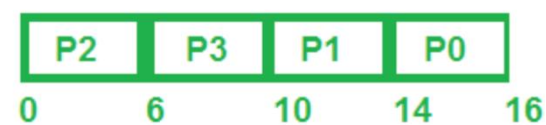
Preemptive and Nonpreemptive Scheduling

- **Preemptive scheduling** vs. **Non-preemptive scheduling**

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4

Preemptive Scheduling

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4



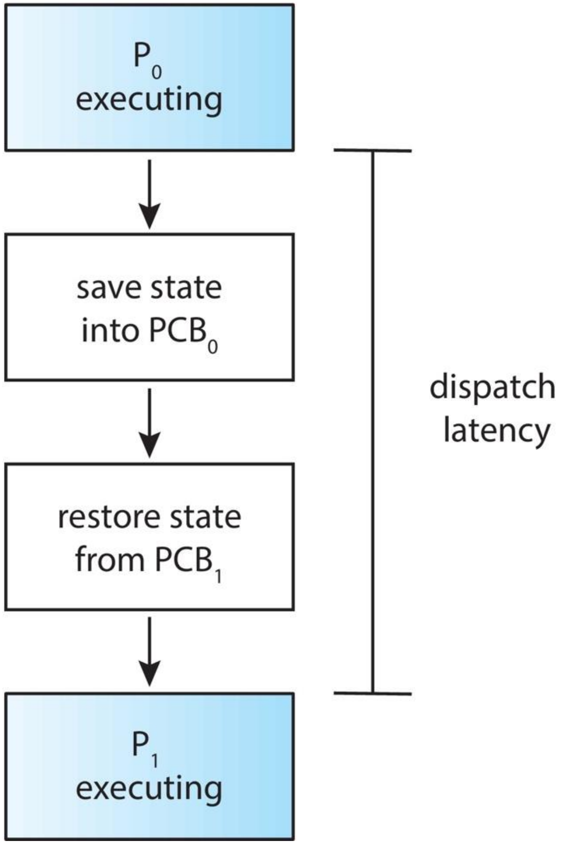
Non-Preemtive Scheduling

	Nonpreemptive	Preemptive
CPU Utilization	Lower	Higher
Response Time	Longer	Shorter
Complexity	Lower	Higher
Priority Management	Static	Allows dynamic priority
Fairness	Worse	Better

Dispatcher (分发器)

- 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 / Metrics

- **CPU utilization** – keep the CPU as busy as possible
  - Max CPU utilization
- **Throughput** – # of processes that complete execution per time unit
  - Max throughput
- **Turnaround time** – amount of time to execute a particular process.
  - The interval from the time of submission of a process to the time of completion is the turnaround time.
  - Min turnaround time
- **Waiting time** – amount of time a process has been waiting in the ready queue
  - Min waiting time
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced
  - Min response time

Scheduling Algorithms

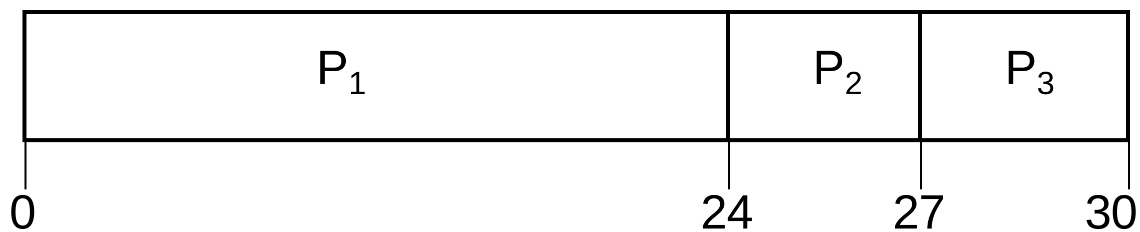
CPU Scheduling Algorithms

- First-Come, First-Served Scheduling (**FCFS**)
- Shortest-Job-First Scheduling (**SJF**)
- Priority Scheduling (**PS**)
- Round-Robin Scheduling (**RR**)
- Multilevel Queue Scheduling (**MQS**)
- Multilevel Feedback Queue Scheduling (**MFQS**)

1. First-Come, First-Served (FCFS) Scheduling

Process	CPU Burst	*Arrival* Time
P_1	24	0
P_2	3	1
P_3	3	2

- The **Gantt Chart** for the schedule is:



- **Waiting time** for P_1 = 0; P_2 = 24-1=23; P_3 = 27-2=25

  Average waiting time: (0 + 23 + 25)/3 = **16**

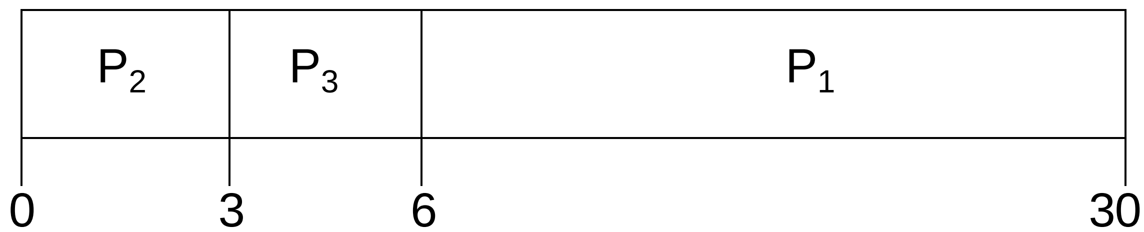
- **Turnaround time** for P_1 = 24; P_2 = 27-1=26; P_3 = 30-2=28

  Average turnaround time: (24 + 26 + 28)/3 = 26

FCFS Scheduling (Cont.)

Process	CPU Burst	*Arrival* Time
P_1	24	2
P_2	3	0
P_3	3	1

- The **Gantt chart** for the schedule is:

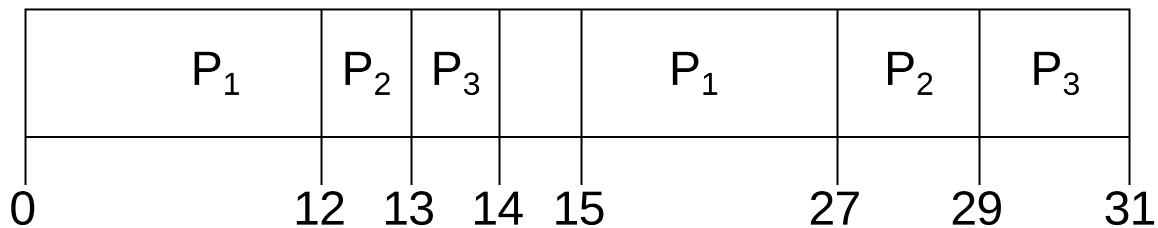


- Waiting time for $P_1 = 4$; $P_2 = 0$; $P_3 = 2$
- Average waiting time: $(4 + 0 + 2)/3 =$ **2**
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

FCFS Scheduling (Cont.)

Process	CPU Burst	I/O Burst	CPU Burst	*Arrival* Time
P_1	12	3	12	0
P_2	1	2	2	1
P_3	1	2	2	2

- The Gantt Chart for the schedule is:



- Waiting time for
  - P_1 = 15-12-3=0
  - P_2 = (12-1)+(27-13-2)=23
  - P_3 = (13-2)+(29-14-2)=24
- Turnaround time for P_1 = 27; P_2 = 29-1=28; P_3 = 31-2=29
- CPU utilization 30/31 = 96.77%

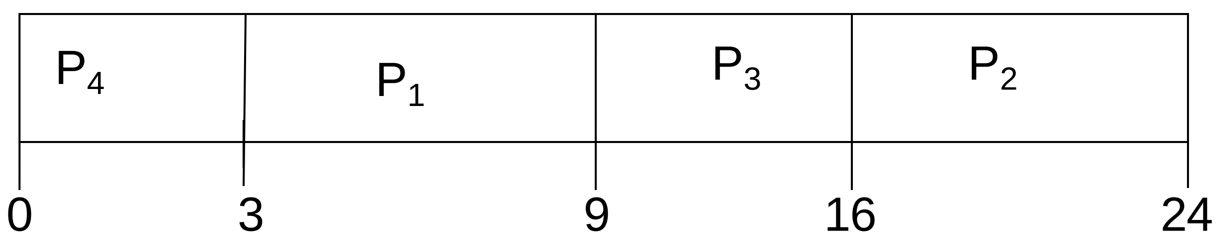
2. Shortest-Job-First (SJF) Scheduling

- 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
- Preemptive version called **shortest-remaining-time-first**

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart (assume all processes arrive at 0)



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

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using **exponential moving average**

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

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$

- Commonly, α is set to ½

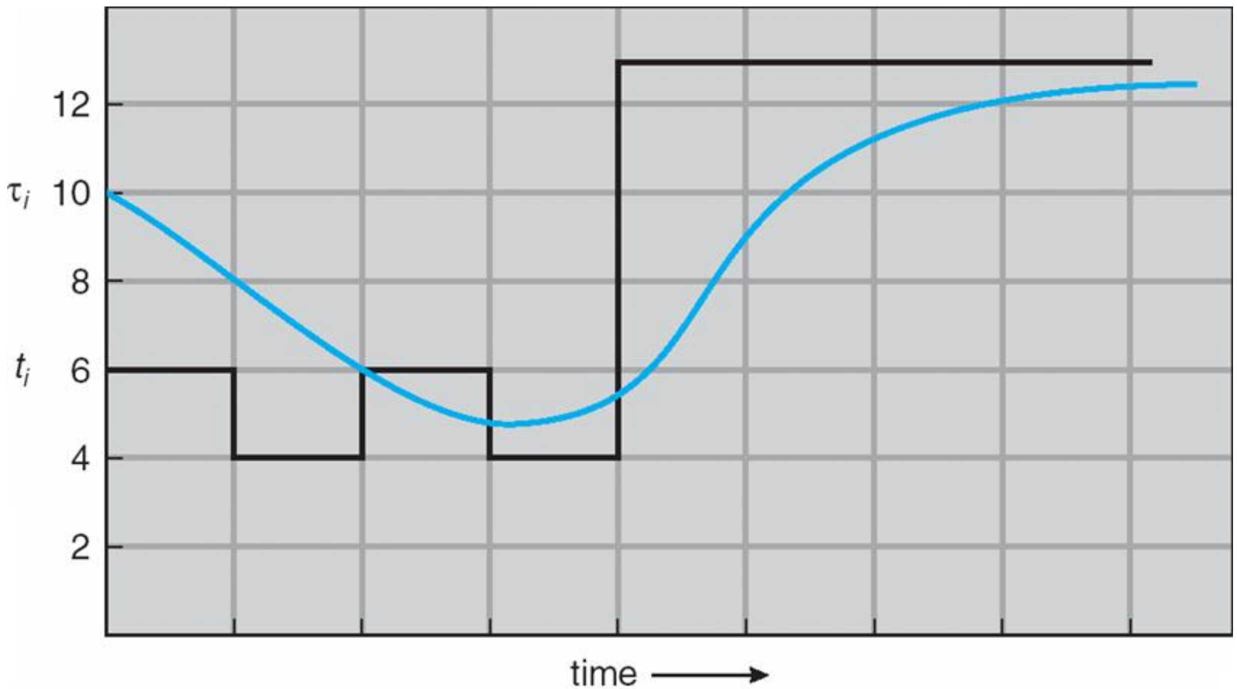
Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

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

  - Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Prediction of the Length of the Next 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 Remaining Time First Scheduling

- Preemptive version of SJF
- Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJF algorithm.
- Is SRT more "optimal" than SJF in terms of the **minimum average waiting time** for a given set of processes?

Example of Shortest-remaining-time-first

- We now add the concepts of varying arrival times and preemption to the analysis

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

- ***Preemptive*** SJF Gantt Chart

P_1	P_2	P_4	**P_1**	P_3

0 1 5 10 17 26

- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5

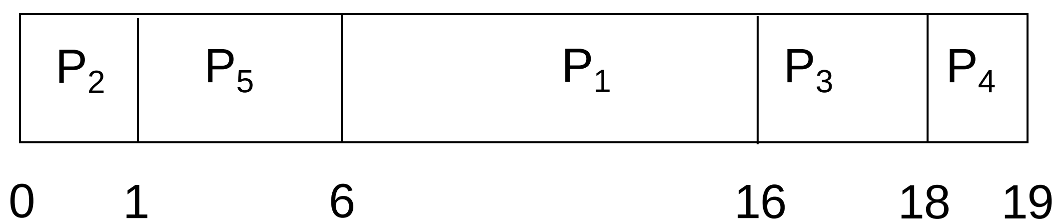
3. Priority Scheduling

- 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
  - Nonpreemptive
- SJF is priority scheduling, where priority is the inverse of predicted next CPU burst time
- Problem: **Starvation** – low priority processes may never execute
- Solution: **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

4. Round Robin Scheduling (RR)

- Round Robin (RR) is similar to FCFS scheduling, but preemption is added to switch between processes.
- Each process gets a small unit of CPU time (**time quantum** q), 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 of at most q time units at once.
  - No process waits more than $(n-1)q$ time units before next execution.
- Timer interrupts every quantum to schedule next process
- 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 Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

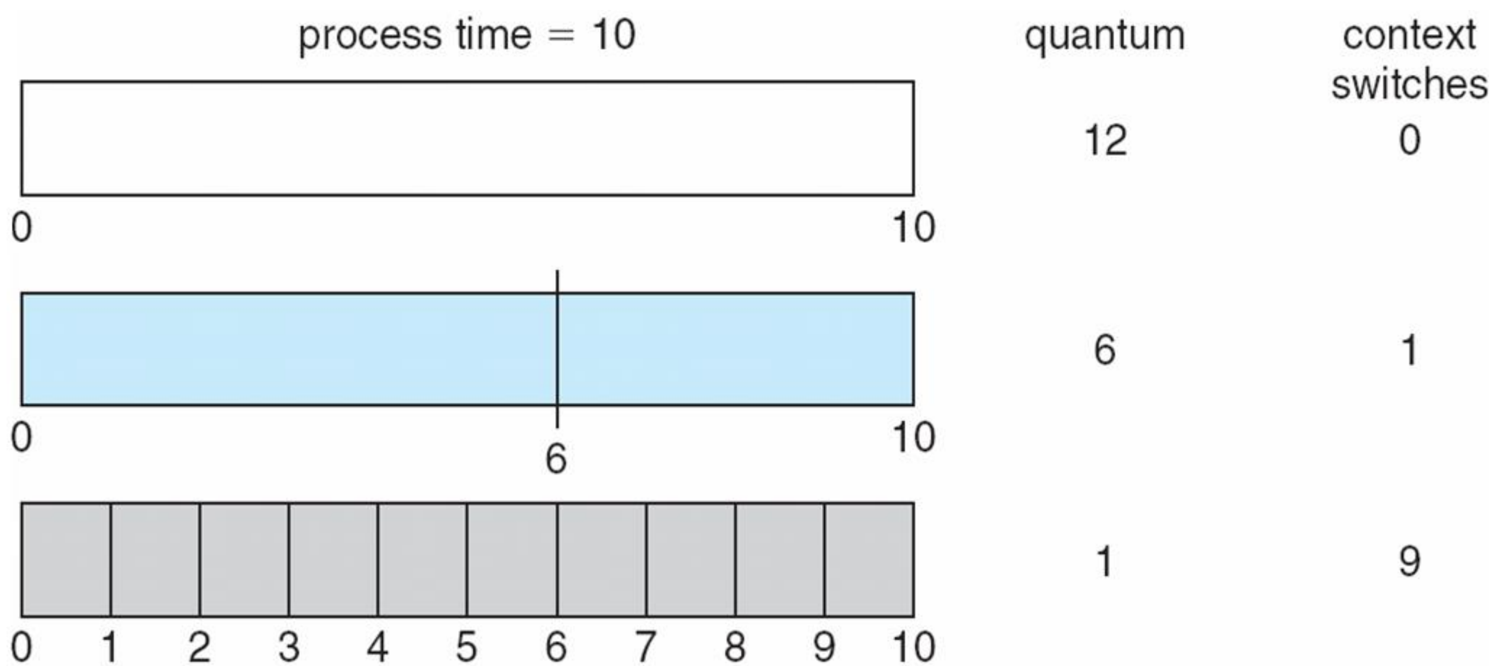
P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

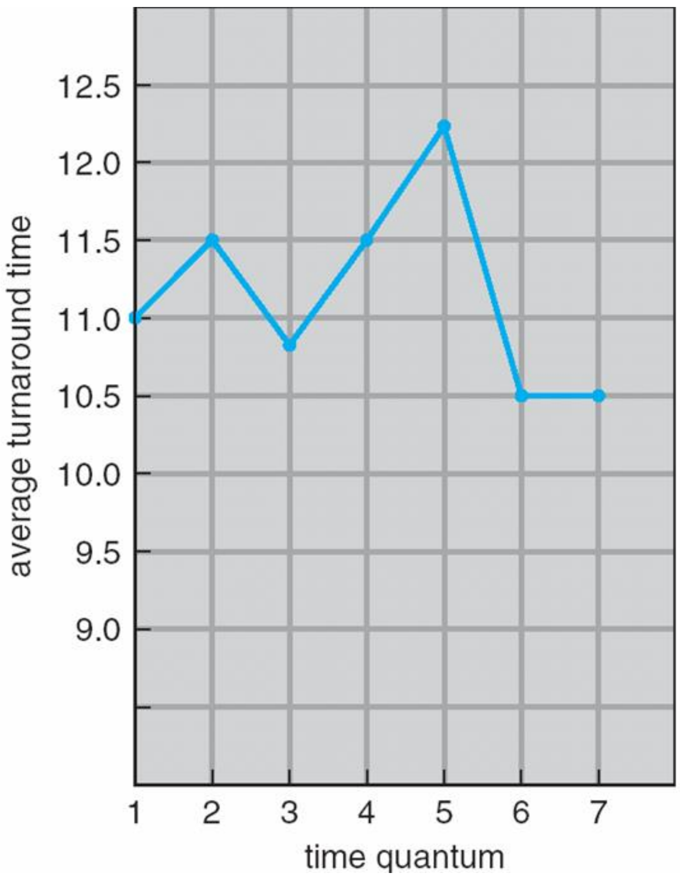
- Typically, higher average turnaround than SJF, but better *response*
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 μs
- What's the number of context switch?

Time Quantum and Context Switch Time

- Context switching



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

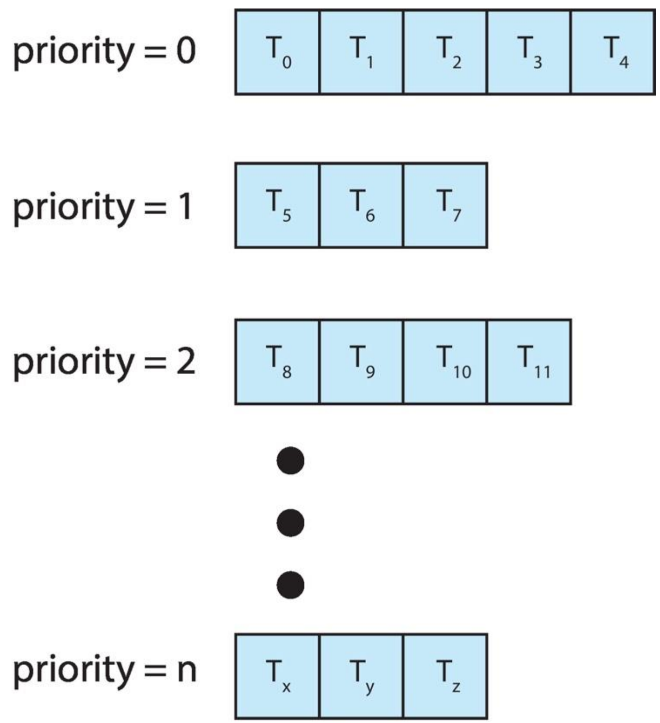
Rule of thumb:
80% of CPU bursts should be shorter than quantum

5. Multilevel Queue Scheduling

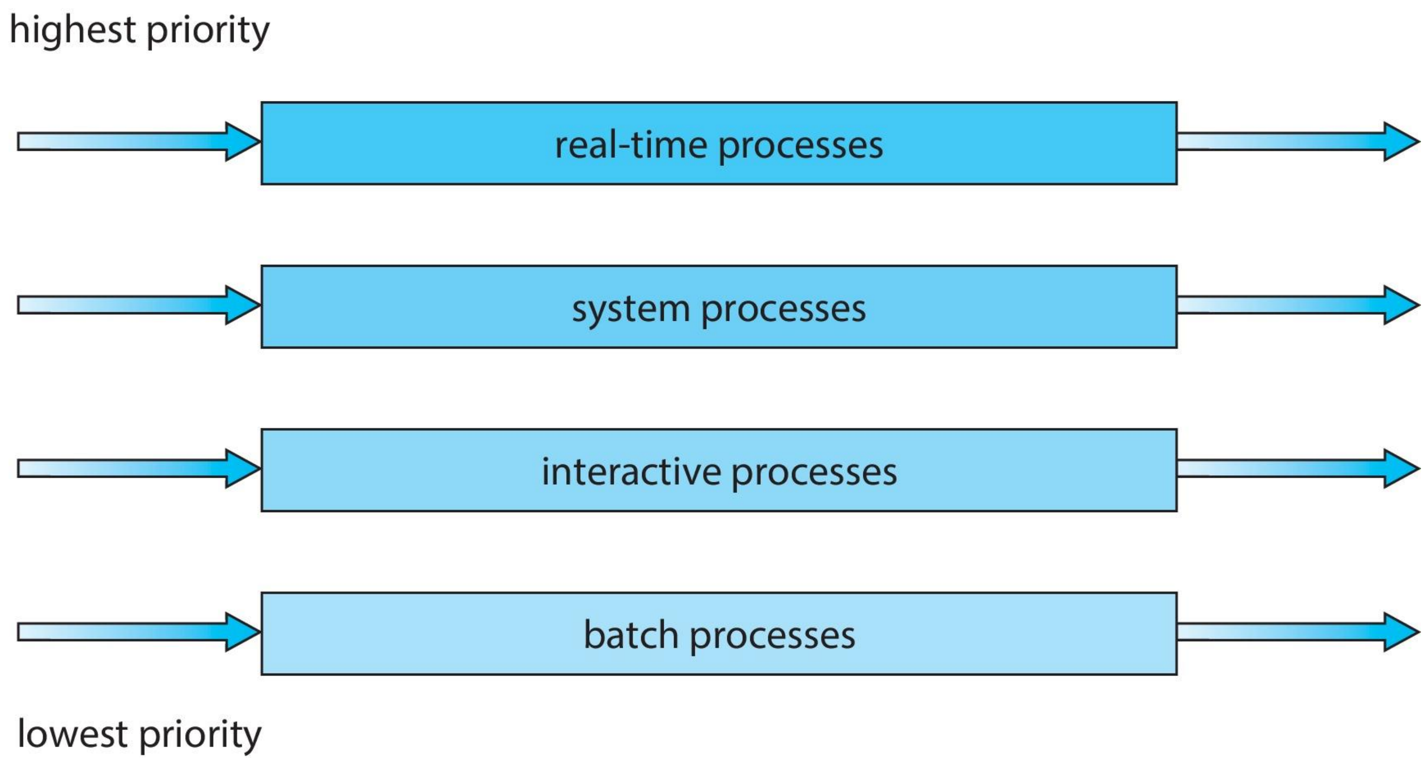
- Ready queue is partitioned into separate queues, e.g.:
  - foreground (interactive)
  - background (batch)
- Process joins a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling: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

Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



Multilevel Queue Scheduling

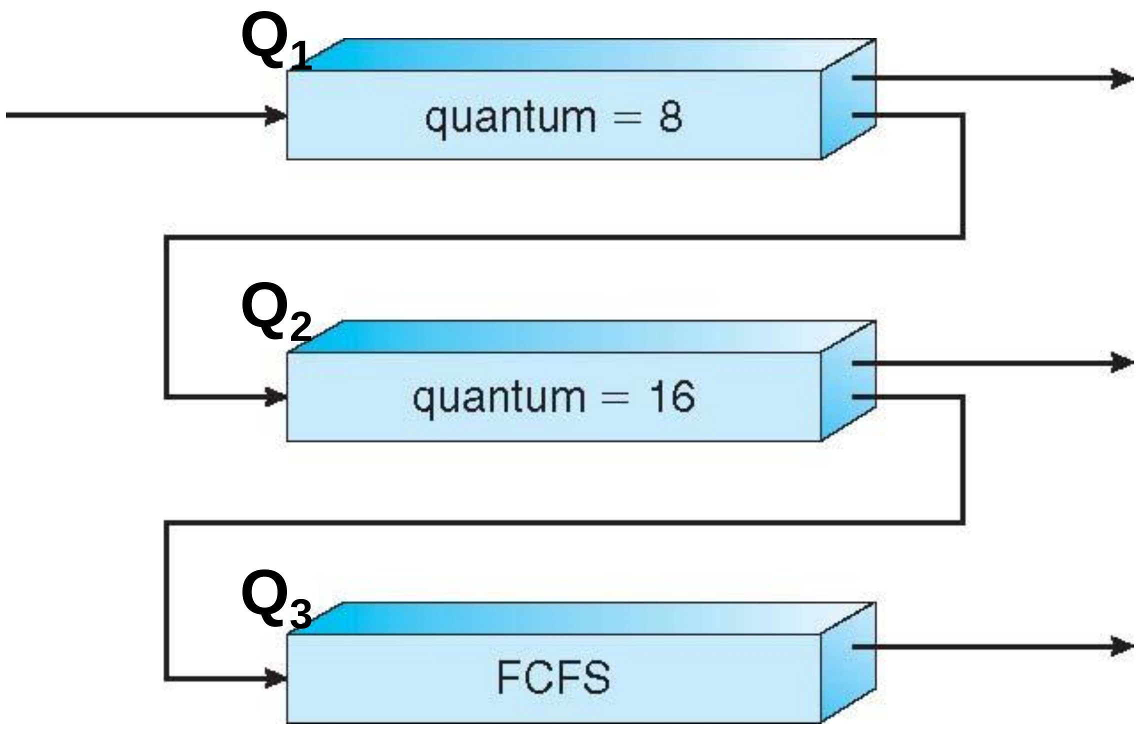


6. Multilevel Feedback Queue Scheduling

- 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
  - Queue change:
    - method used to determine when to upgrade a process
    - method used to determine when to downgrade a process
    - method used to determine which queue a process will enter when that process needs service
- Aging can be implemented using multilevel feedback queue

Example of Multilevel Feedback Queue

- Three queues:
  - Q_1 – RR with time quantum 8 milliseconds
  - Q_2 – RR with time quantum 16 milliseconds
  - Q_3 – FCFS



Multilevel Feedback Queue

- Scheduling
  - A new job enters queue Q_1 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_2
  - At Q_2 job is again served FCFS/RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_3
  - If a process does not use up its quantum in the current level, it will keep its current queuing level and be put into the end of the queue.
    - Then, it can still get the same amount of quantum (not remaining quantum) next time when it is picked.

Example of Using Multilevel Feedback Queue

Process	*Arrival* Time	Burst Time
P_1	0	36
P_2	16	20
P_3	20	12

- The Gantt chart is:

P_1	P_1	P_2	P_3	P_1	P_2	P_3	P_1
0–8	8–16	16–24	24–32	32–48	48–60	60–64	64–68

Example of Using Multilevel Feedback Queue

Process	*Arrival* Time	Burst Time
P_1	0	36
P_2	**15**	20
P_3	20	12

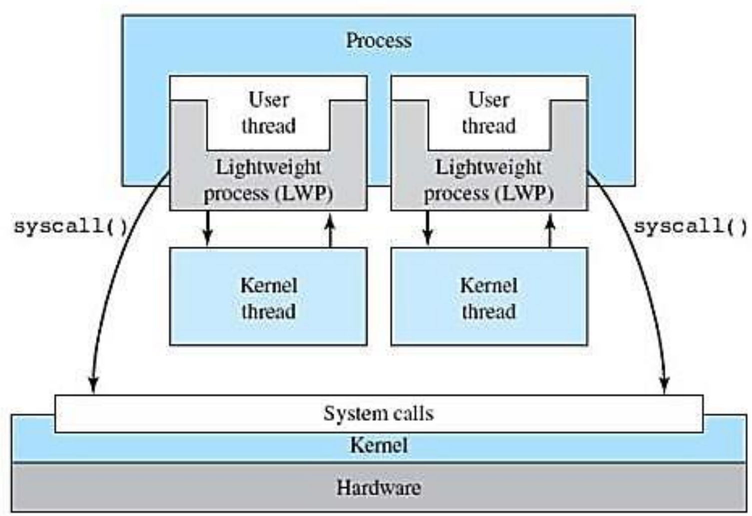
- The Gantt chart is:

P_1	P_1	P_2	P_3	P_1	P_2	P_3	P_1	
0	8	15	23	31	47	59	63	68

Thread Scheduling

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads are supported, threads are scheduled instead of processes
- In many-to-one and many-to-many models, the thread library schedules user-level threads to run on kernel-level threads
  - Known as **process-contention scope (PCS, 进程竞争范围)** since scheduling competition happens within the process
  - Typically done via priority set by the programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS, 系统竞争范围)** – competition among all threads in the system
  - One-to-one mapping use only SCS.



Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - **PTHREAD_SCOPE_PROCESS** schedules threads using PCS scheduling
    - Schedules user-level threads onto available LWPs
    - Number of LWPs is maintained by the thread library
  - **PTHREAD_SCOPE_SYSTEM** schedules threads using SCS scheduling
    - Creates and binds an LWP for each user-level thread
    - In fact, implements the one-to-one mapping
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread Scheduling API

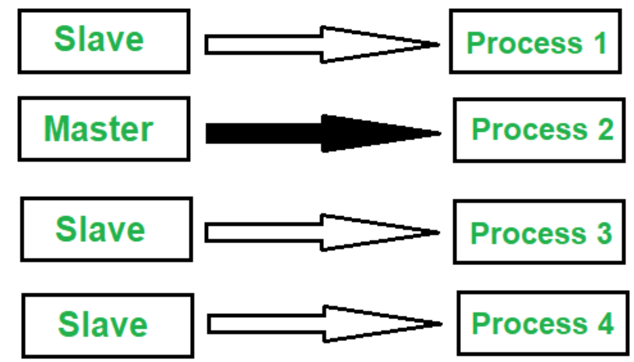
```
    /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i], &attr, runner, NULL);
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

Multi-Processor Scheduling

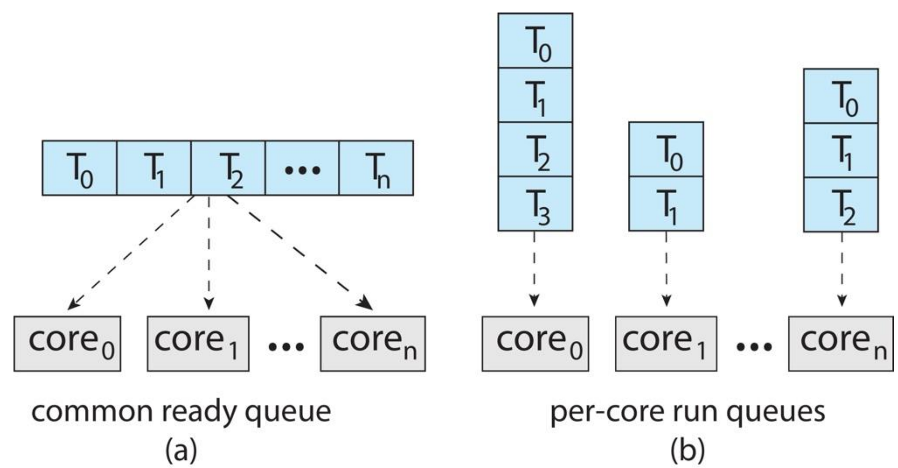
Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available
  - **Homogeneous processors** within a multiprocessor

- **Asymmetric multiprocessing**
  - One processor handles all scheduling decisions, I/O processing, and other system activities
  - Only one processor accesses the system data structures, alleviating the need for data sharing
  - All other processors just execute user code.

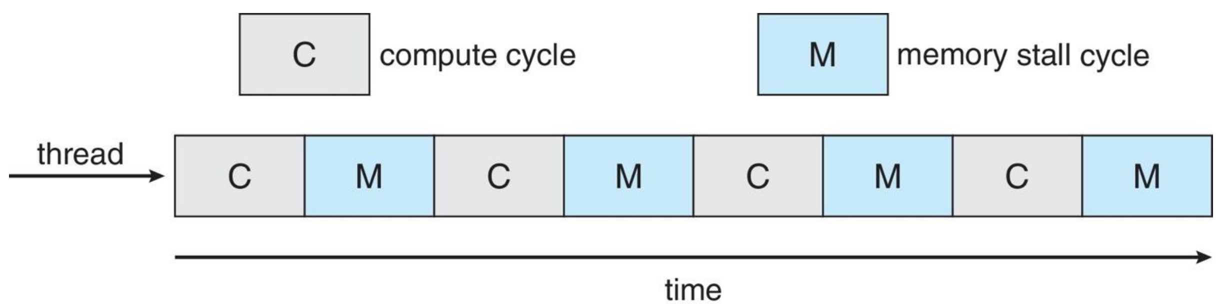


- **Symmetric multiprocessing (SMP)**
  - Each processor is self-scheduling,
  - (a) All processes are in common ready queue,
  - (b) Each processor has its own private queue of ready processes



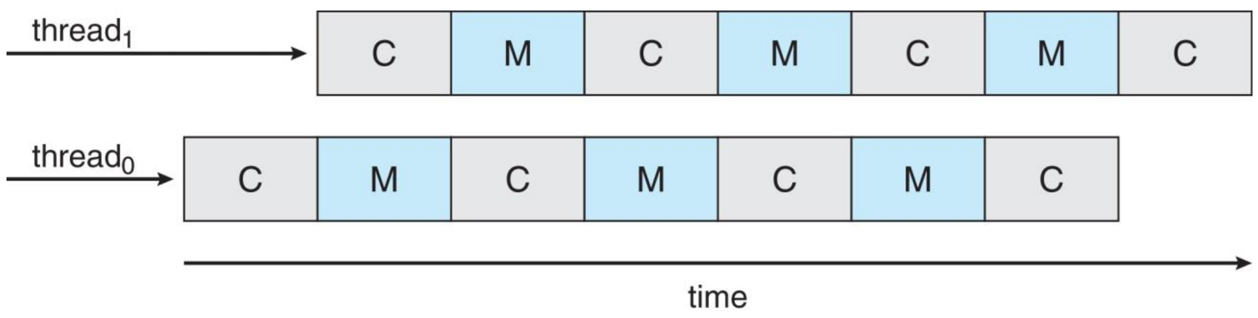
Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
  - Faster and consumes less power
- **Memory stall**:
  - When a processor accesses memory, it spends a significant amount of time waiting for the data to become available
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens



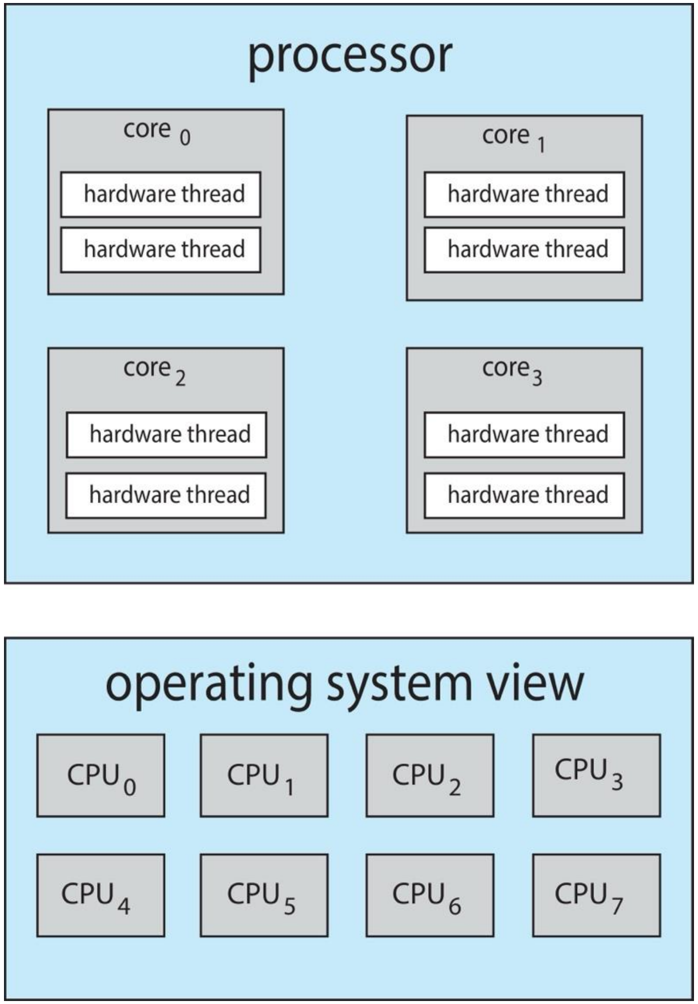
Multithreaded Multicore System

- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!
- Figure



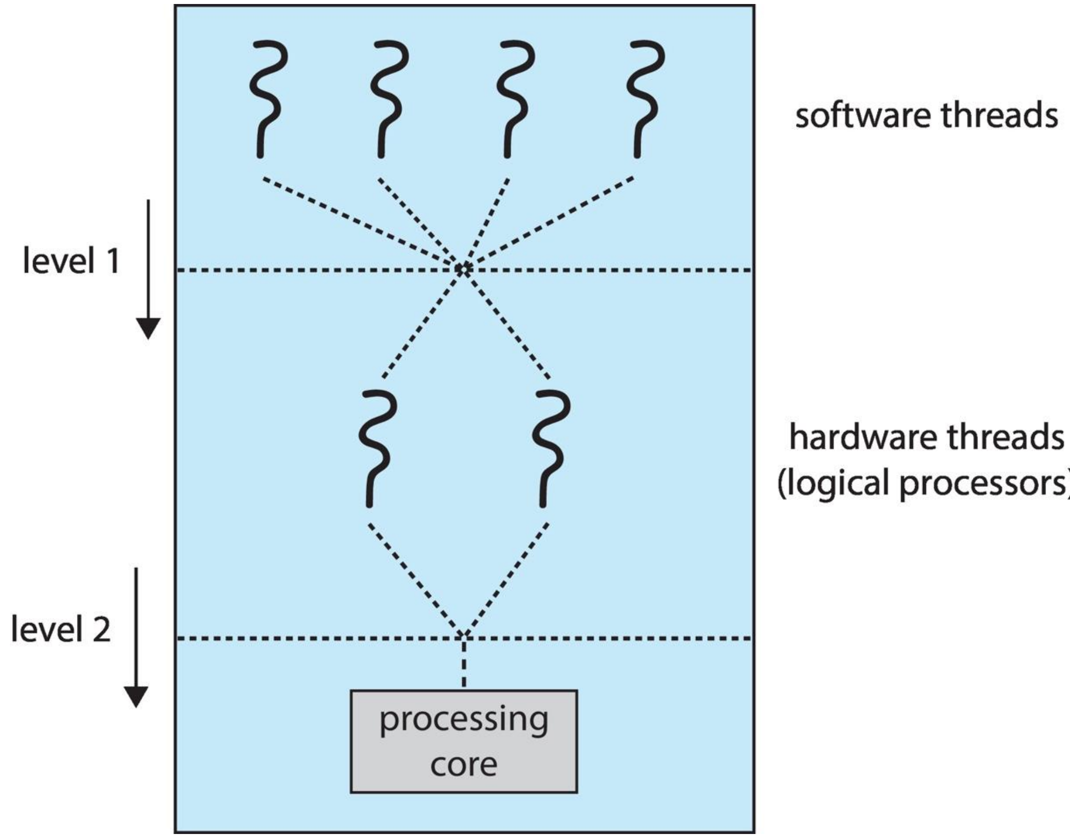
Multithreaded Multicore System

- **Chip-multithreading** (CMT) assigns each core multiple hardware threads.
  - Intel refers to this as **hyperthreading**.
- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.



Multithreaded Multicore System

- Two levels of scheduling:
  1. The operating system decides which **software thread** to run on which **hardware thread**
  2. Each core decides which **hardware thread** to run on the **physical core**

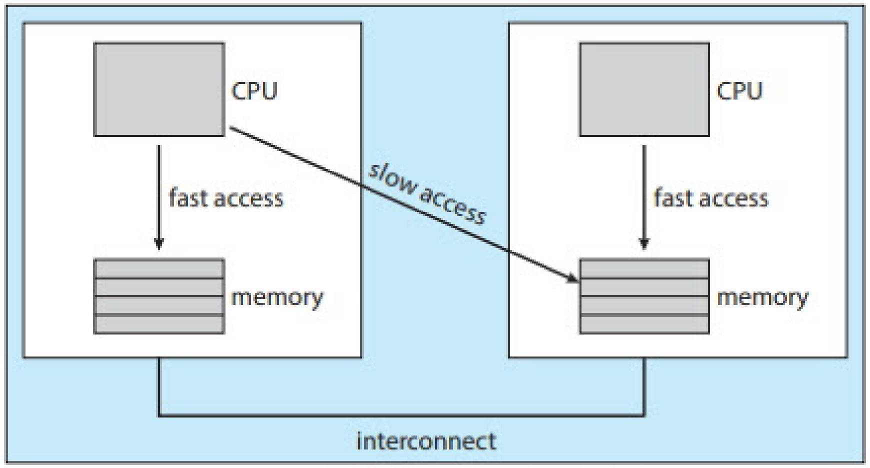


Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- Two types of migration:
  - **Push migration** – periodic task checks load on each processor, and if found overloading, pushes task from overloaded CPU to other CPUs
  - **Pull migration** – idle processors pulls waiting task from busy processor

Multiple-Processor Scheduling – Processor Affinity

- **Processor affinity:**
  - When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
  - We refer to this as a thread having affinity for a processor
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads
  - The thread loses the contents in the cache of the previous processor.
- Two types:
  - **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
  - **Hard affinity** – allows a process to specify a set of processors it may run on.



Real-Time CPU Scheduling

Real-Time CPU Scheduling

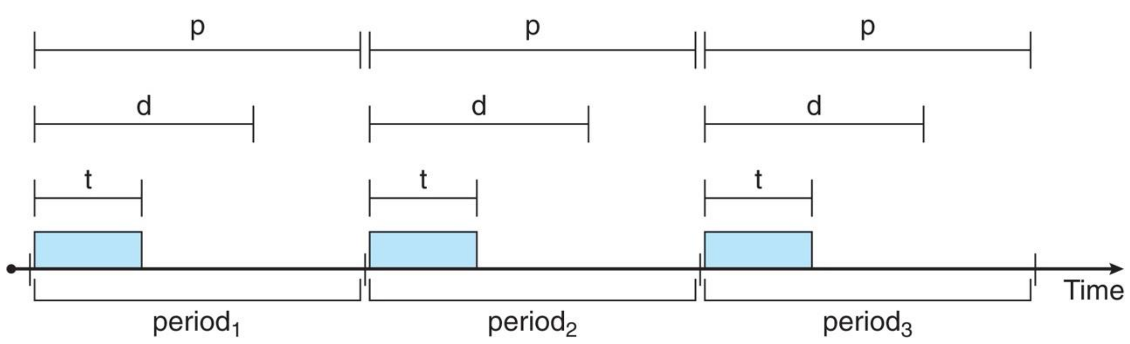
- Real-tine scheduling allocates CPU resources to tasks based on their timing constraints, ensuring they meet strict deadlines for predictable and deterministic execution.

- Can present obvious challenges

- Two types:
  - **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
  - **Hard real-time systems** – Task must be serviced by its deadline

Priority-based Scheduling

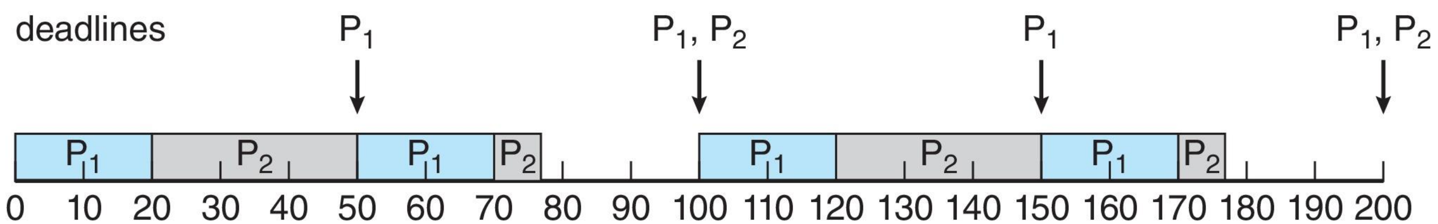
- For real-time scheduling, scheduler must support **preemptive**, **priority-based** scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d*, period *p*
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is $1/p$



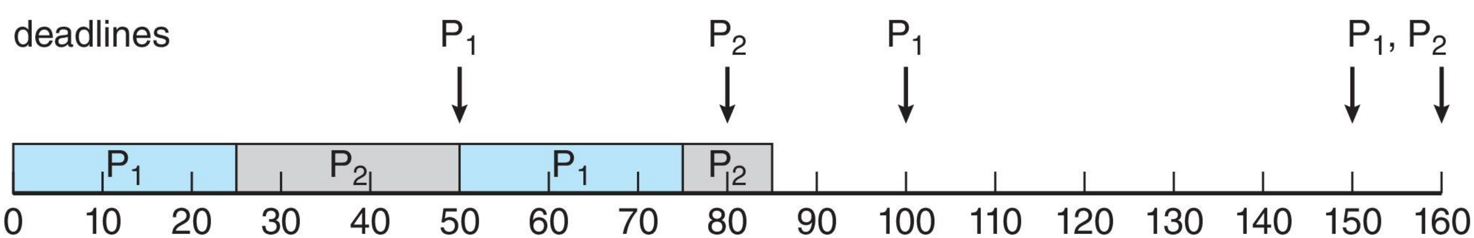
SHANGHAI JIAO TONG UNIVERSITY

RT Scheduling: Rate Monotonic

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2.



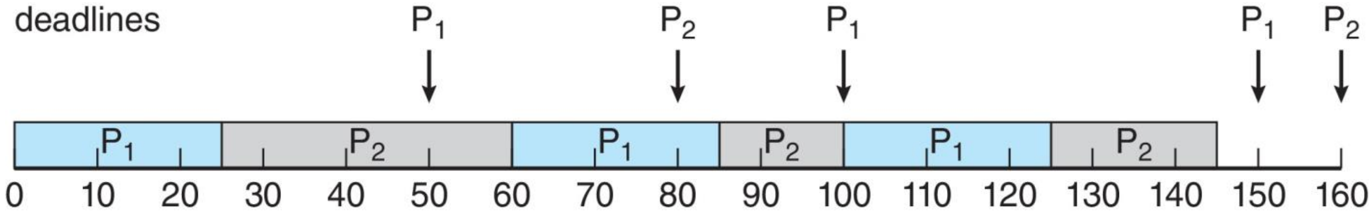
- Process P_2 misses finishing its deadline at time 80



RT Scheduling: Earliest Deadline First (EDF)

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority

- Figure



Algorithm Evaluation

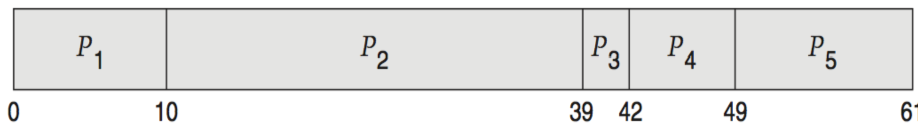
Algorithm Evaluation

- How to select a CPU-scheduling algorithm for an OS?
  - Determine criteria, then evaluate algorithms
- Analytical evaluation:
  - Use the given algorithm and the system workload to produce a formula or number to evaluate the performance of the algorithm for the workload.
- **Deterministic modeling**, one type of analytic evaluation:
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

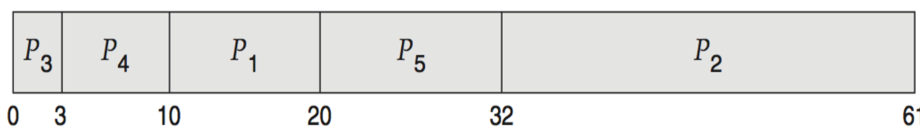
Deterministic Evaluation

- For each algorithm, calculate minimum **average waiting time**
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCFS is **28ms**:

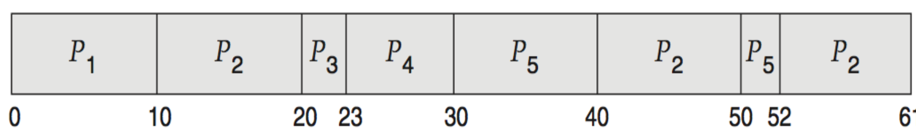


Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

  - Non-preemptive SFJ is **13ms**:



  - RR is **23ms**:



Summary

- CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it.
- Scheduling algorithms may be either preemptive or nonpreemptive. Most modern operating systems are preemptive.
- Scheduling algorithm criteria: (1) CPU utilization, (2) throughput, (3) turnaround time, (4) waiting time, and (5) response time.
- Introduced scheduling algorithms: FCFS, SJF, RR, priority scheduling, multilevel queue scheduling, multilevel feedback queue scheduling
- Multicore processors place one or more CPUs on the same physical chip, and each CPU may have more than one hardware thread.
- Real-time scheduling provides timing guarantees for real-time tasks

Homework

- Reading
  - Chapter 5

- Homework
  - Check Canvas for Homework 1 (later today), due on March 12 at 23:59.
  - Write your homework with the given LaTex template.