

Chapter 5: CPU Scheduling





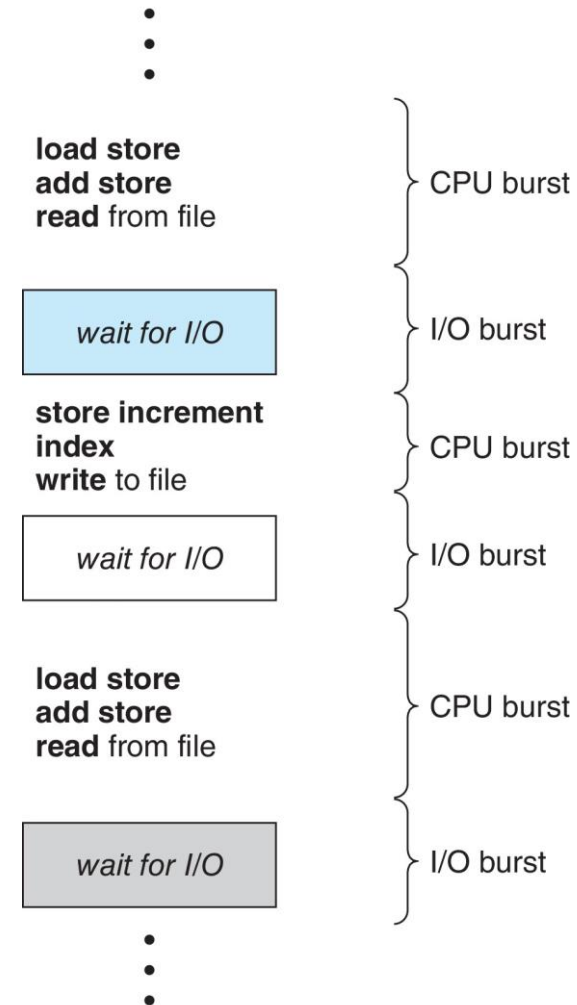
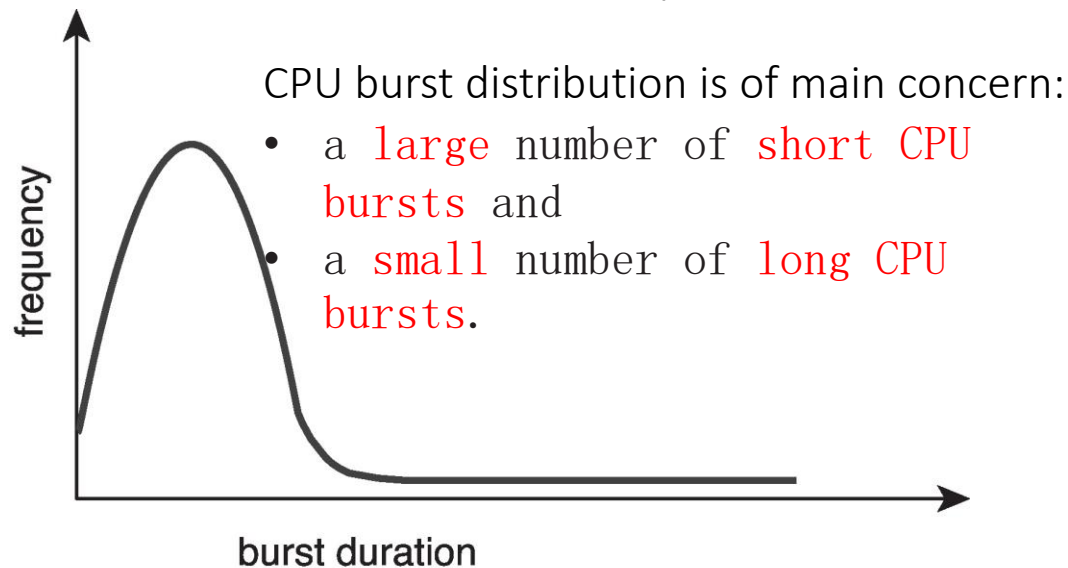
Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples



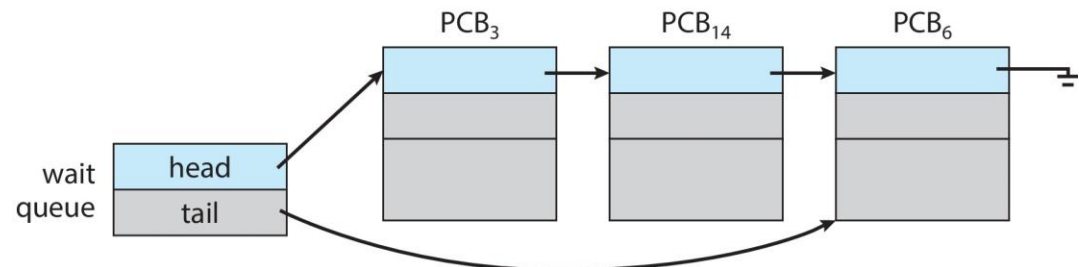
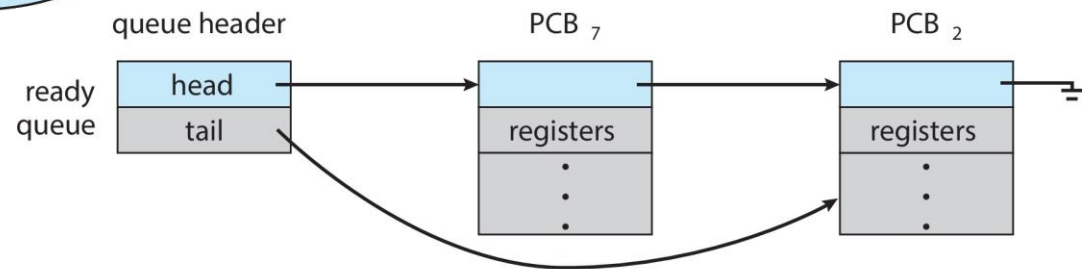
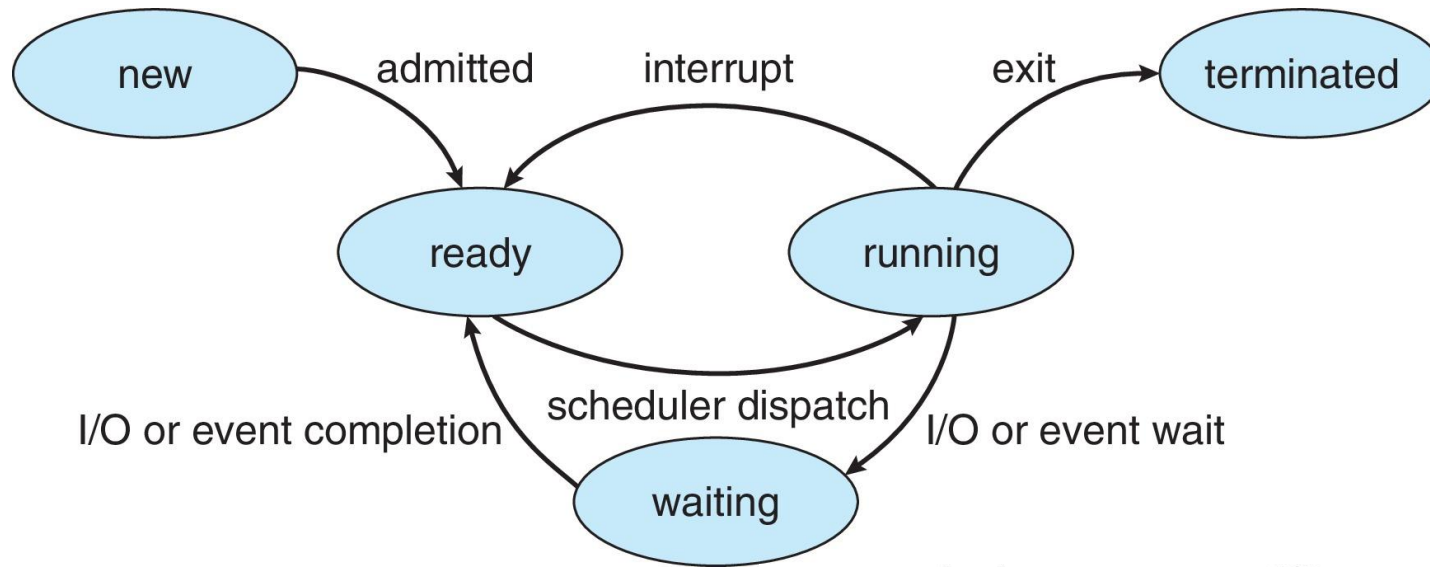
Basic Concepts

- Purpose of multiprogramming:
maximum CPU utilization
- CPU-I/O Burst Cycle
 - Process execution consists of a **cycle** of CPU execution and I/O wait
 - CPU burst followed by I/O burst





Process State Transition





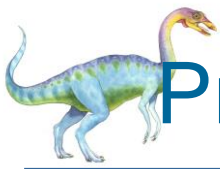
CPU Scheduler

- The CPU scheduler (CPU 调度程序) selects a process from the processes in ready queue, and allocates the CPU to it
 - Ready queue may be ordered in various ways
- CPU scheduling decisions may take place when a process
 1. switches from running to waiting state (non-preemptive 自愿离开CPU)
 - ▶ Example: the process does an I/O system call.
 2. switches from running to ready state (preemptive 强占)
 - ▶ Example: there is a clock interrupt.
 3. switches from waiting to ready (preemptive)
 - ▶ Example: there is a hard disk controller interrupt because the I/O is finished.
 4. terminates (non-preemptive 自愿离开CPU)



CPU Scheduler

- Scheduling under 1 and 4 is **non-preemptive** (非强占的, decided by the process itself)
- All other scheduling is **pre-emptive** (强占的, decided by the hardware and kernel)
- Preemptive scheduling can result in **race conditions** (will introduced in chapter 6) when data are shared among several processes
- Some considerations in pre-emptive scheduling
 1. Access to shared data
 2. Preemption issue while CPU is in kernel mode
 3. How to handle interrupts during crucial OS activities



Preemptive vs. Nonpreemptive Scheduling

- When a process is pre-empted,
 - It is moved from its current processor
 - However, it still remains in memory and in ready queue
- Why preemptive scheduling is used ?
 - Improve response times
 - Create interactive environments (real-time)
- Non-preemptive scheduling
 - Process runs until completion or until they yield control of a processor
 - Disadvantage
 - ▶ Unimportant processes can block important ones indefinitely



Scheduling Criteria

- maximize
 - CPU utilization – keep the CPU as busy as possible
 - Throughput – number of processes that complete their execution per time unit
 - Increase throughput as high as possible
- minimize
 - Response time – amount of time it takes from when a request was submitted until **the first response is produced, not output** (for time-sharing environment)
 - Waiting time – total amount of time a process has been waiting in the ready queue
 - Turnaround time – amount of time to execute a particular process (from start to end of process, including waiting time)
 - Turnaround time = Waiting time + time for all CPU bursts

How to calculate these criteria?



Scheduling Algorithms

1. First-Come, First-Served (FCFS)
2. Shortest-Job-First (SJF)
3. Priority Scheduling (PS)
4. Round-Robin (RR)
5. Multilevel Queue Scheduling (MQS)
6. Multilevel Feedback Queue Scheduling (MFQS)



First- Come, First-Served (FCFS) Scheduling

- Suppose that the processes arrive in the ready queue at time $t = 0$ in the following order: P_1, P_2, P_3
- Burst time for each process is

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

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



- Waiting time: $P_1 = 0; P_2 = 24; P_3 = 27$
- Average **waiting** time: $(0 + 24 + 27) / 3 = 17$
- Average **turnaround** time = $(24+27+30)/3 = 27$



FCFS Scheduling (Cont.)

- Suppose the order is changed to this:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is then:



- Waiting time: $P_1 = 6; P_2 = 0; P_3 = 3$
- Average **waiting time**: $(6 + 0 + 3) / 3 = 3$
- Average **turnaround time** = $(30 + 3 + 6) / 3 = 13$
- Much better than previous case
- **Convoy effect**(护送效应)[1] - **short process behind long process**
 - Consider one **CPU-bound** (long CPU burst, short I/O burst) and many **I/O-bound** (long I/O burst, short CPU burst) processes

[1] This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.



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**
 - Could ask the user



Shortest-Job-First (SJF) Scheduling

Process

Burst Time

P_1

6

P_2

8

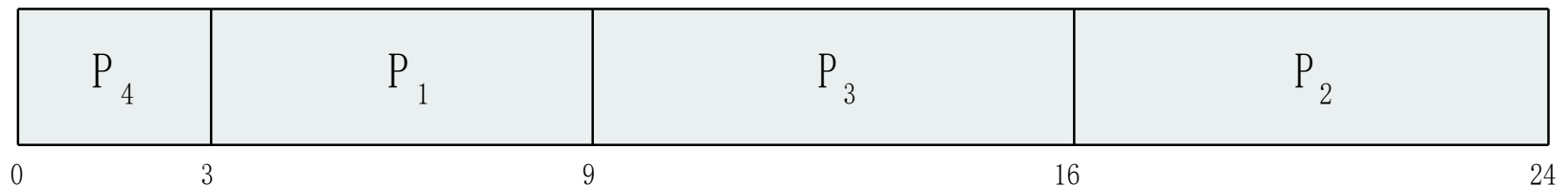
P_3

7

P_4

3

■ SJF scheduling chart



■ Average **waiting time** = $(3 + 16 + 9 + 0) / 4 = 7$

■ Average **turnaround time** = $(9 + 24 + 16 + 3) / 4 = 13$

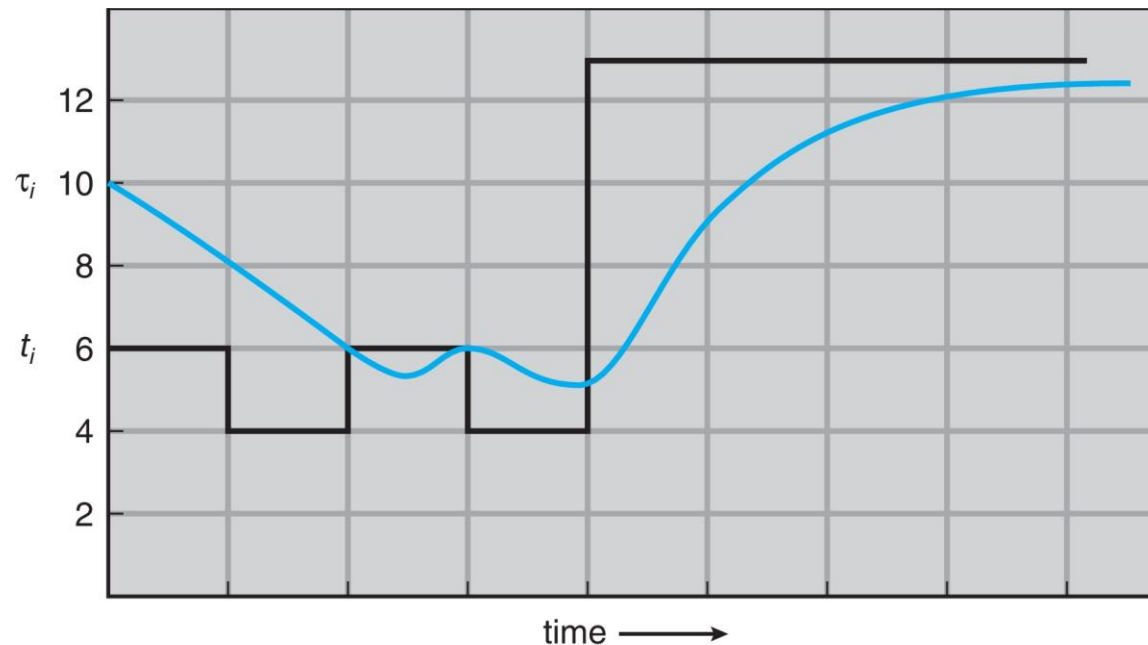


Determining Length of Next CPU Burst

- Actually the length of next CPU burst can only be estimated
 - Next burst length should be similar to the previous one (use the past to predict the future).
 - Then pick process with **shortest predicted** next CPU burst
- Use the length of previous CPU bursts, with **exponential averaging**
 1. t_n = actual length of n^{th} CPU burst
 2. τ_{n+1} = predicted value for the next CPU burst
 3. $\alpha, 0 \leq \alpha \leq 1$ (commonly, α set to $\frac{1}{2}$)
 4. Define : $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$



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	...

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n = \frac{1}{2} (t_n + \tau_n)$$

(Assume α set to $\frac{1}{2}$)



Examples of Exponential Averaging

■ $\alpha = 0$

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

- $\tau_{n+1} = \tau_n = \dots = \tau_0$
- History does not count: always use the same guess regardless of what the process actually does.

■ $\alpha = 1$

- $\tau_{n+1} = t_n$
- Only the actual last CPU burst counts

■ In general, if we expand the formula, we get:

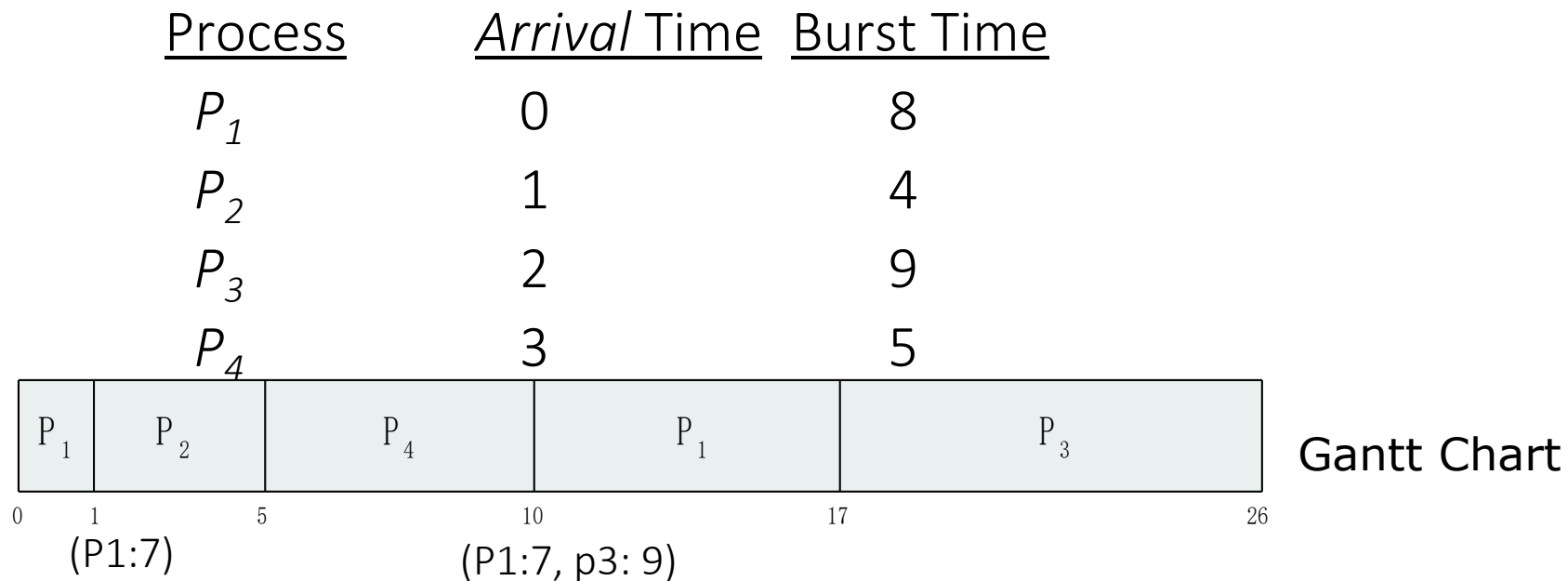
$$\begin{aligned}\tau_{n+1} = & \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots \\ & + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ & + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

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



Example of Shortest-remaining-time-first

- The preemptive version of SJF is also called shortest-remaining-time-first
- Now we add the concepts of varying arrival times and preemption to the analysis



- Average **waiting time** = $[(10-1)+(1-1)+(17-2)+(5-3)] / 4 = 26 / 4 = 6.5$
- Average **turnaround time** = $((17-0)+(5-1)+(26-2)+(10-3))/4 = 13$



Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

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

- *Preemptive* SJF (shortest-remaining-time-first) Gantt Chart

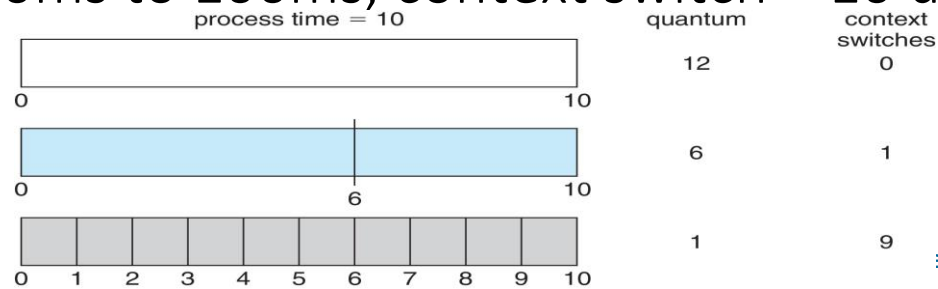
Question: how if time for P_4 is 1

- Average waiting time = ???
- Average turnaround time = ???



Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum 定额 q), usually 10-100 milliseconds.
- After q has elapsed, the process is preempted by a clock interrupt and added to the end of the ready queue.
 - Timer interrupts every quantum q to schedule next process
- If there are n processes in the ready queue and the time quantum is q . No process waits more than $(n-1)*q$.
- Performance
 - q too large \Rightarrow FCFS
 - q too small \Rightarrow too much time is spent on context switch
 - ▶ q should be large compared to context switch time
 - ▶ q usually 10ms to 100ms, context switch < 10 usec (微秒)

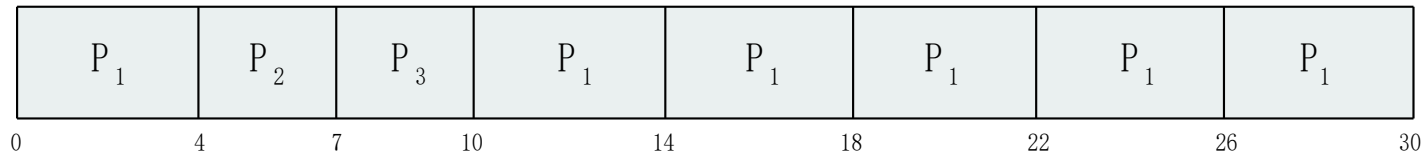




Example of RR with Time Quantum = 4

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

■ The Gantt chart is:

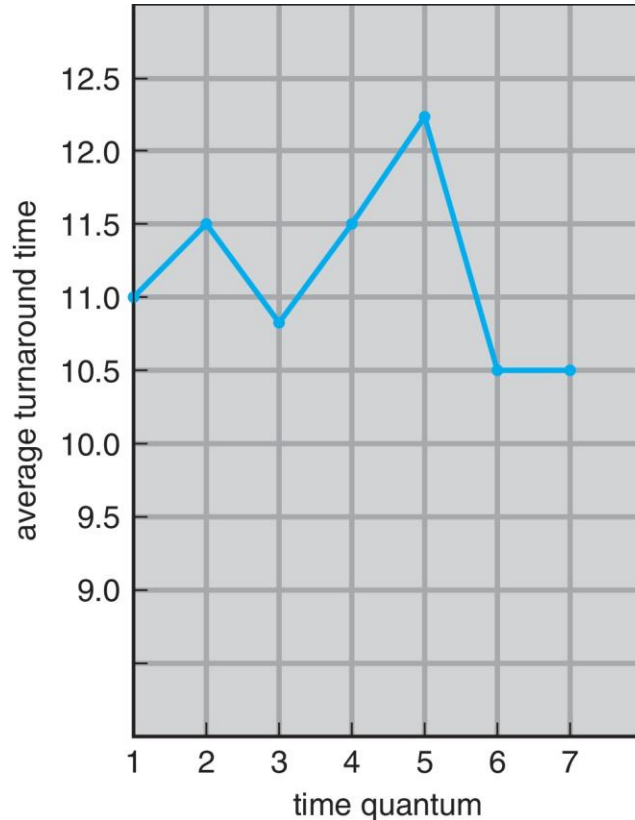


- Typically, higher average turnaround than SJF, but better *response*
- Average **waiting time** = $(6+4+7)/3 = 5.67$
- Average **turnaround time** = $(30+7+10)/3 = 15.7$

Question: how if time for q is 25?



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

General rule: 80% of CPU bursts should be shorter than q , that way most processes can finish their current CPU burst without being interrupted.

$q=6$, Average turnaround time = $(6+9+10+17)/4 = 10.5$

$q=7$, Average turnaround time = $(6+9+10+17)/4 = 10.5$



Priority Scheduling

- A priority number (integer) may be associated with each process
- The CPU is allocated to the process with the highest priority
(smallest integer \equiv highest priority)
 - Two policies
 - ▶ Preemptive
 - the current process is pre-empted immediately by high priority process
 - ▶ Non-preemptive
 - the current process finishes its burst first, then scheduler chooses the process with highest priority
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time



Priority Scheduling

■ Problem

- **Starvation**: low priority processes may never execute

■ Solution

- **Aging**: as time progresses increase the priority of the process



Example of Priority Scheduling

smallest integer \equiv highest priority

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

■ Priority scheduling (not preemptive) Gantt Chart



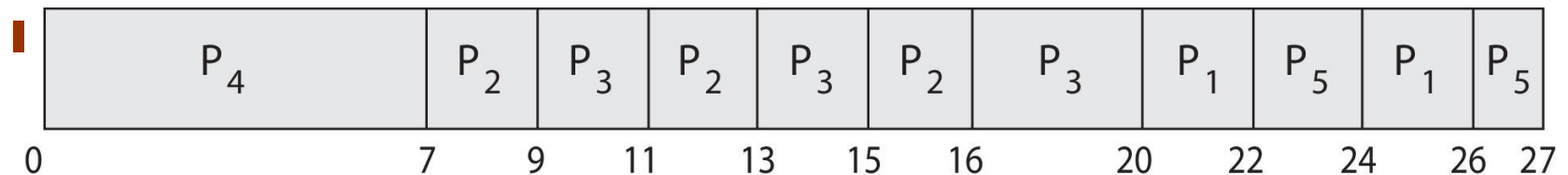
■ Average waiting time = $(6+0+16+18+1)/5 = 8.2$



Priority Scheduling w/ Round-Robin

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Run the process with the highest priority. **Processes with the same priority** run **round-robin** (in this example, assume $q=2$)



$$\text{Average waiting time} = (22+11+12+0+24)/5 = 13.8$$



Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.:
 - foreground (interactive 交互 processes)
 - background (batch 批处理 processes)
- Process permanently in a given queue (stay in that queue)
- Each queue has its own scheduling algorithm:
 - foreground – RR
 - background – FCFS
- Scheduling must be done between the queues:
 - Fixed priority scheduling
 - ▶ Each queue has a given priority
 - High priority queue is served before low priority queue
 - Possibility of starvation
 - Time slice
 - ▶ Each queue gets a certain amount of CPU time



Multilevel Queue

- With priority scheduling, for **each priority**, there is a **separate queue**
- Schedule the process in the highest-priority queue!

priority = 0

T_0	T_1	T_2	T_3	T_4
-------	-------	-------	-------	-------

priority = 1

T_5	T_6	T_7
-------	-------	-------

priority = 2

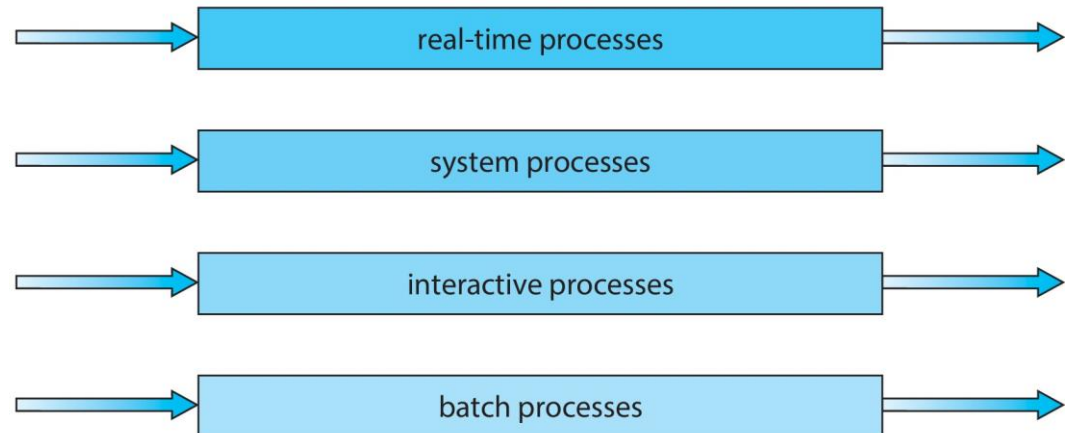
T_8	T_9	T_{10}	T_{11}
-------	-------	----------	----------



priority = n

T_x	T_y	T_z
-------	-------	-------

highest priority



lowest priority

Prioritization based upon **process type**



Multilevel Feedback Queue

- A process can **move** between the various queues;
 - aging can be considered in this way (**prevent starvation**)
 - advantage: prevent starvation
- The multilevel feedback queue scheduler
 - the most general CPU scheduling algorithm
 - defined by the following parameters:
 1. number of queues
 2. scheduling algorithms for each queue
 3. Policies on moving process between queues
 1. when to upgrade a process
 2. when to demote (降级) a process
 3. which queue a process will enter when that process needs service



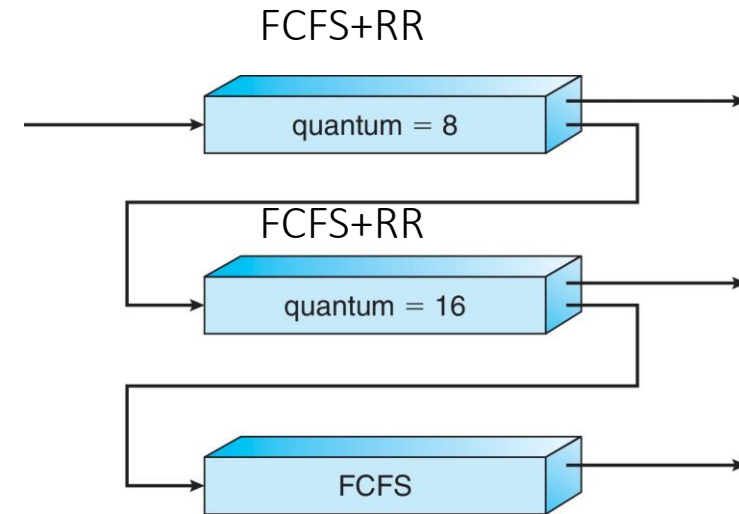
Example of Multilevel Feedback Queue

■ Three queues:

1. Q_0 – RR with time quantum 8 milliseconds
2. Q_1 – RR with time quantum 16 milliseconds
3. Q_2 – FCFS

■ Scheduling

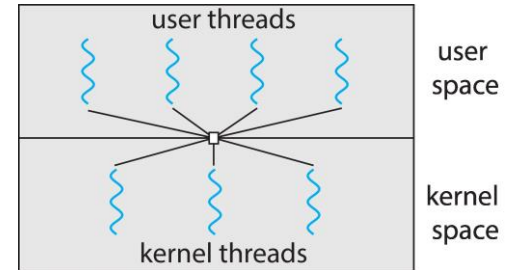
- A new job enters queue Q_0 which is served FCFS
 - ▶ When it gains CPU, job receives 8 milliseconds
 - ▶ If it does not finish in 8 milliseconds, job is moved to queue Q_1
- At Q_1 job is again served FCFS and receives 16 additional milliseconds
 - ▶ If it still does not complete, it is preempted and moved to queue Q_2 where it runs until completion but with a low priority





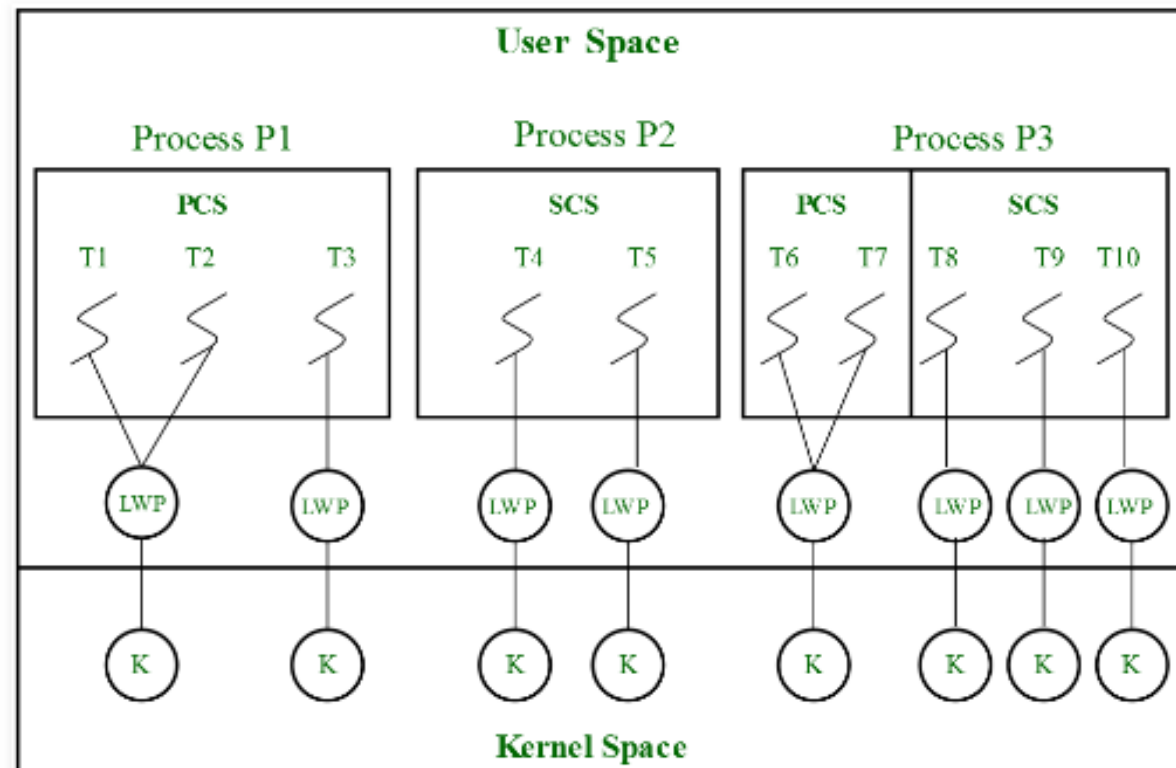
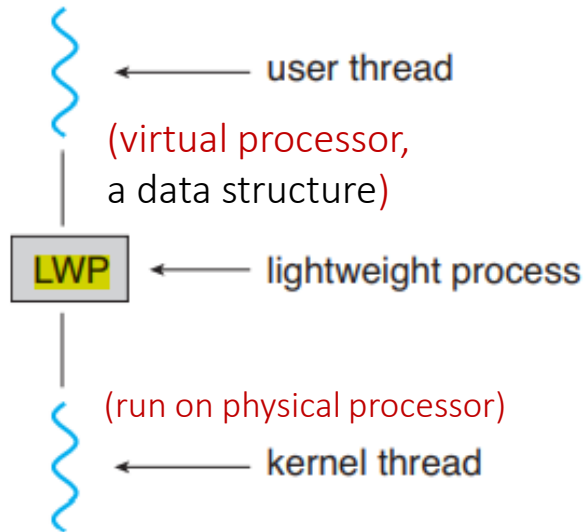
Thread Scheduling

- Distinguish between **user-level** and **kernel-level threads**
- When threads are supported by kernel,
 - threads are scheduled, not processes
- **Many-to-one** and **many-to-many** models,
 - **thread library** schedules user-level threads to run on kernel threads (LWP: light-weight process)
 - **process-contention scope (PCS)**
 - competition is between user-level threads within the same process
 - Typically priority is set by programmer
- Kernel threads are scheduled by **Kernel** onto available CPU
 - **system-contention scope (SCS)**
 - competition is among all kernel-level threads from all processes in the system





Thread Scheduling





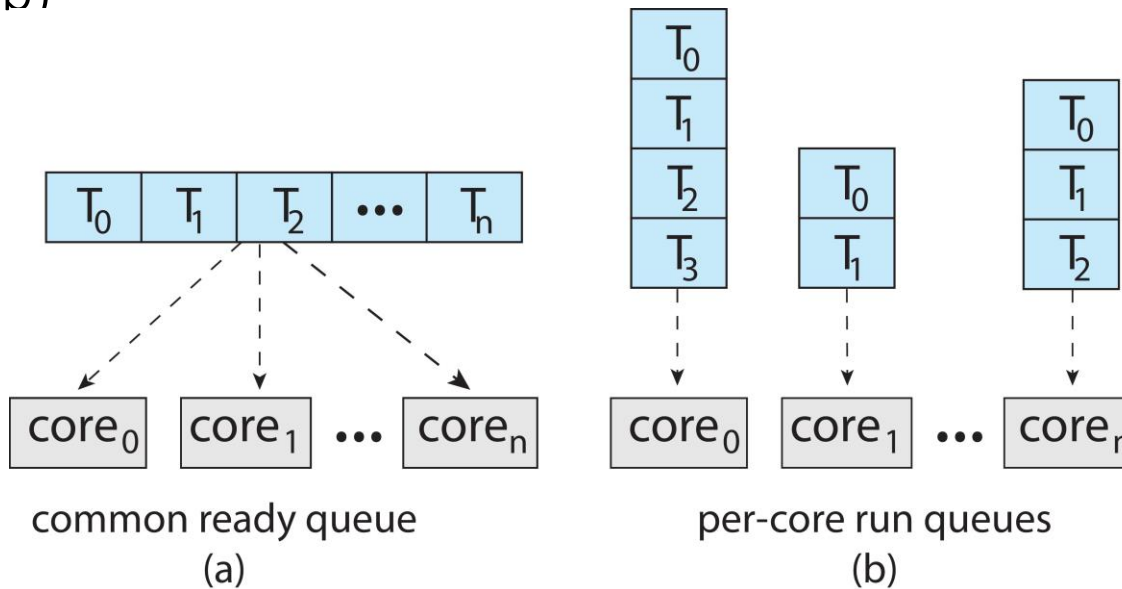
Multiple-Processor Scheduling

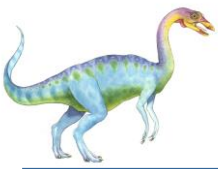
- CPU scheduling is more complex when multiple CPUs are available
- Traditionally, **Multiprocessor** means multiple processors
- The term **Multiprocessor** now applies to the following system architectures:
 - Multicore CPUs
 - Multithreaded cores
 - NUMA systems



Multiple-Processor Scheduling

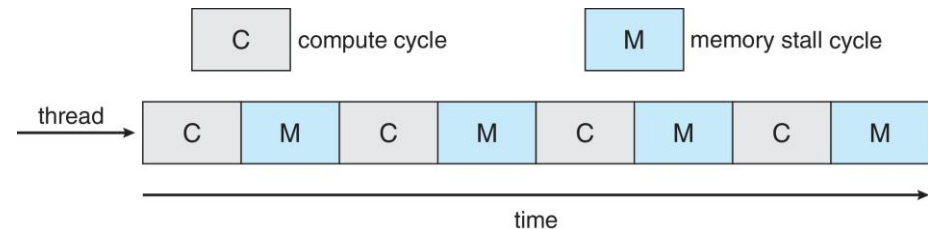
- Symmetric multiprocessing (SMP) is where each processor is self scheduling
 - Two possible strategies
 1. All threads may be in a common ready queue (a)
 2. Each processor may have its own private queue of threads (b)



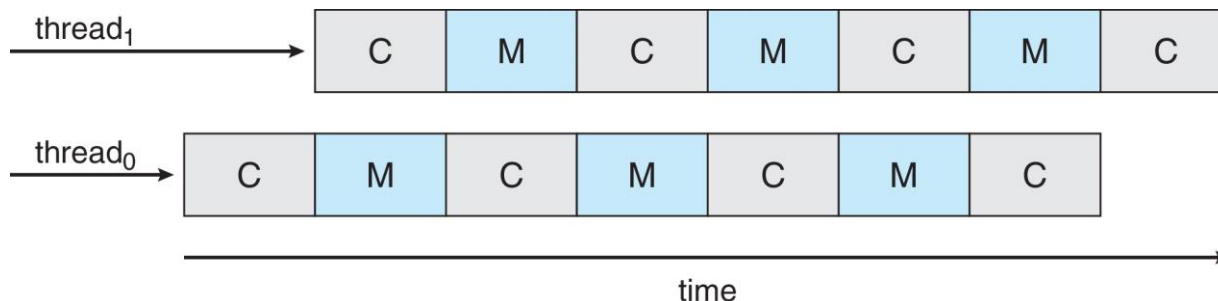


Multicore Processors

- Recent trend: multiple processor cores are on same physical chip
 - Faster and consumes less power
- Multiple threads per core also growing
 - **memory stall (延迟)** : An event that occurs when a thread is on CPU and accesses memory **content that is not in the CPU's cache**. The thread's execution stalls while **the memory content is retrieved and fetched**



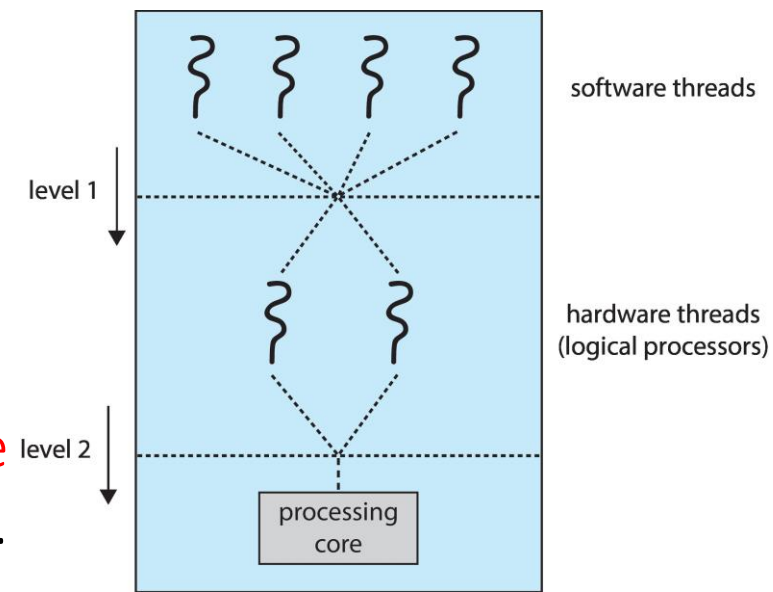
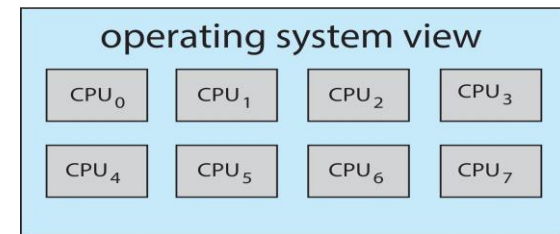
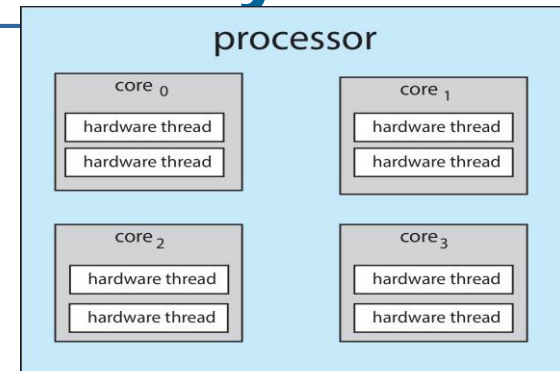
- Solution :
 - ▶ Each core has more than **one hardware threads**. If one thread has a memory stall, switch to another thread!





Multithreaded Multicore System

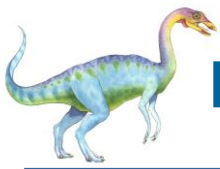
- **Chip-multithreading (CMT)** assigns each core **multiple hardware threads**. (Intel refers to this as hyperthreading.)
- On a quad-core system (4核) with 2 hardware threads per core, the operating system sees 8 logical processors.
- Two levels of scheduling:
 1. The operating system deciding which **software thread** to run on a logical CPU
 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
 - Push migration – periodic task checks load on each processor, and pushes tasks from overloaded CPU to other less loaded CPUs
 - Pull migration – idle CPUs pulls waiting tasks from busy CPU
- Push and pull migration need not be mutually exclusive
 - They are **often implemented in parallel** on load-balancing systems.



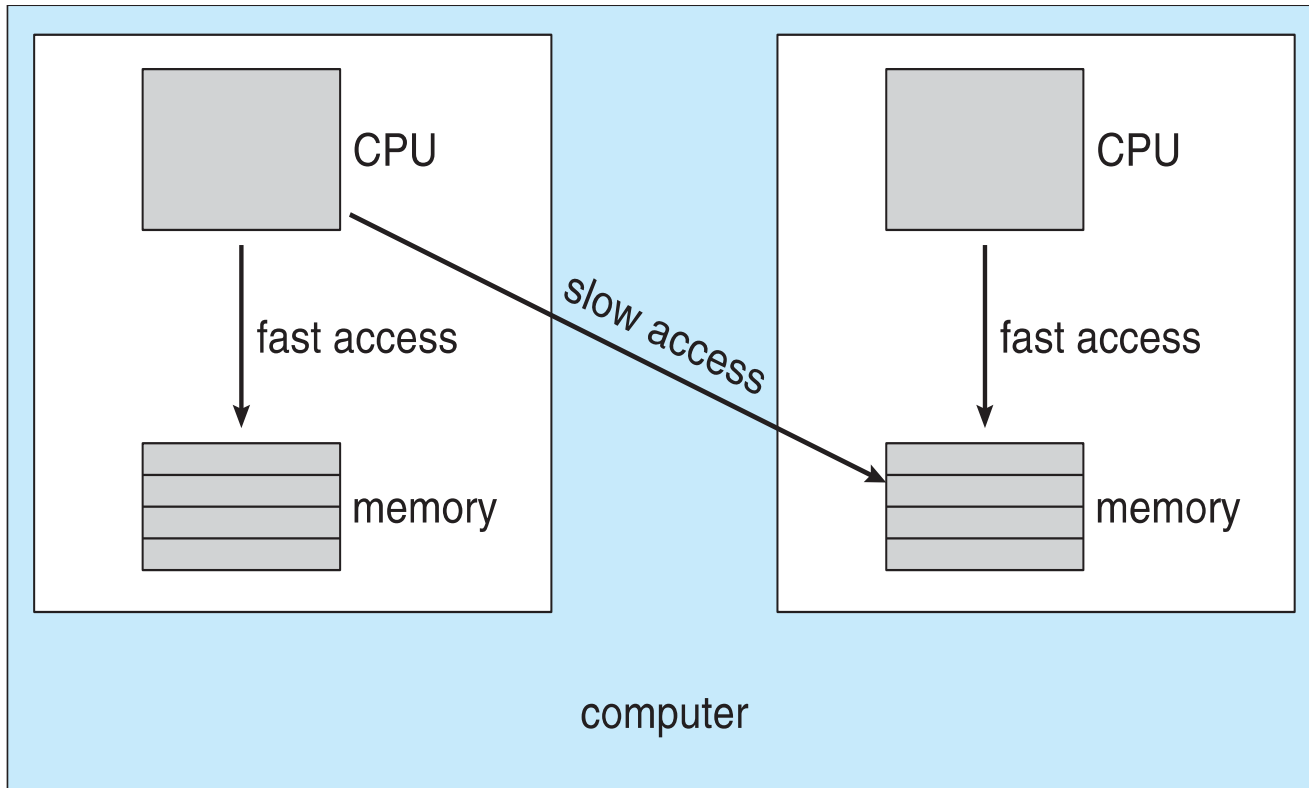
Multiple-Processor Scheduling – Processor Affinity

- Processor affinity
 - When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread, i.e., **a thread has affinity for a processor**
- **Load balancing** may **affect processor affinity**
 - a thread may be moved from one processor to another to balance loads,
 - that thread loses the contents of what it had in the cache of the processor it was moved off
- **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.
 - The kernel then never moves the process to other CPUs, even if the current CPUs have high loads.



NUMA and CPU Scheduling

- If the operating system is **NUMA-aware**, it will assign memory closest to the CPU the thread is running on.



Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own **local memory** faster than **non-local memory** (memory local to another processor or memory shared between processors).



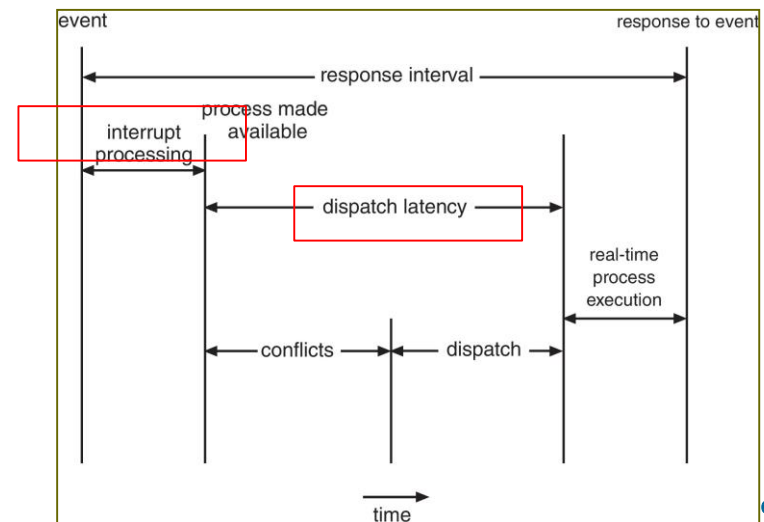
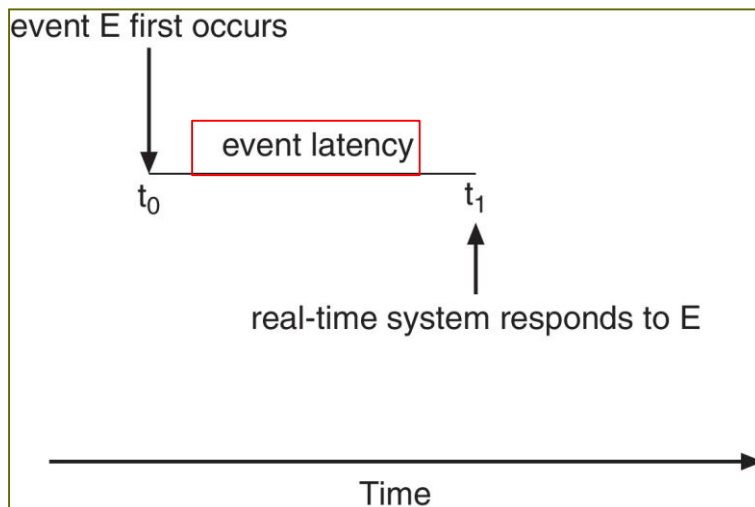
Real-Time CPU Scheduling

- Real-time CPU scheduling presents obvious challenges
 - Soft real-time systems
 - ▶ Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled (best try only)
 - Hard real-time systems
 - ▶ a task must be serviced by its deadline (guarantee)



Real-Time CPU Scheduling

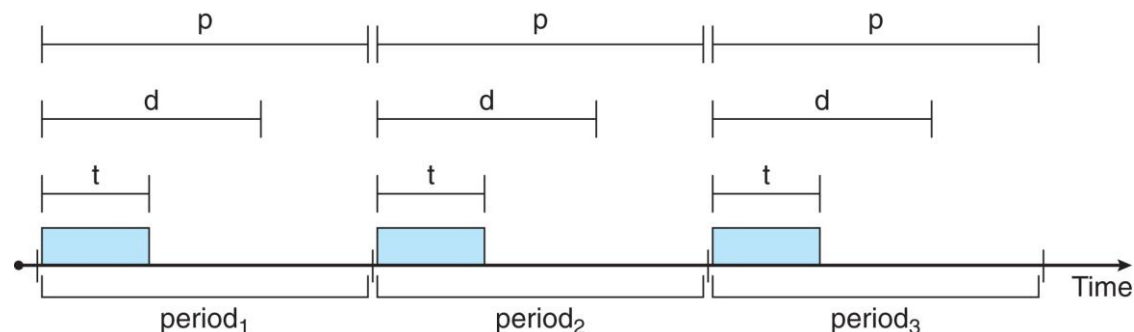
- **Event latency** – the amount of time that elapses from when an event occurs to when it is serviced.
- Two types of latencies affect performance
 1. **Interrupt latency** – time from arrival of interrupt to start of kernel **interrupt service routine (ISR)** that services interrupt
 2. **Dispatch latency**(调度延迟) – time for scheduler to take current process off CPU and switch to another





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**: **periodically** 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 $1/p$





Rate Monotonic Scheduling

- A priority is assigned based on the **inverse of its period**

Shorter periods = higher priority

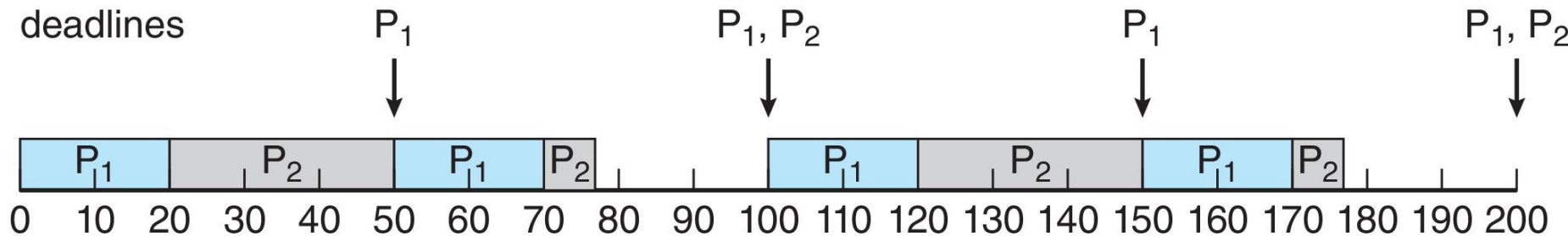
Longer periods = lower priority

- In the following example, P_1 is assigned a higher priority than P_2 .

- P_1 needs to run for 20 ms every 50 ms. $t = 20, d = p = 50$

- P_2 needs to run for 35 ms every 100 ms. $t = 35, d = p = 100$

Assume deadline $d = p$

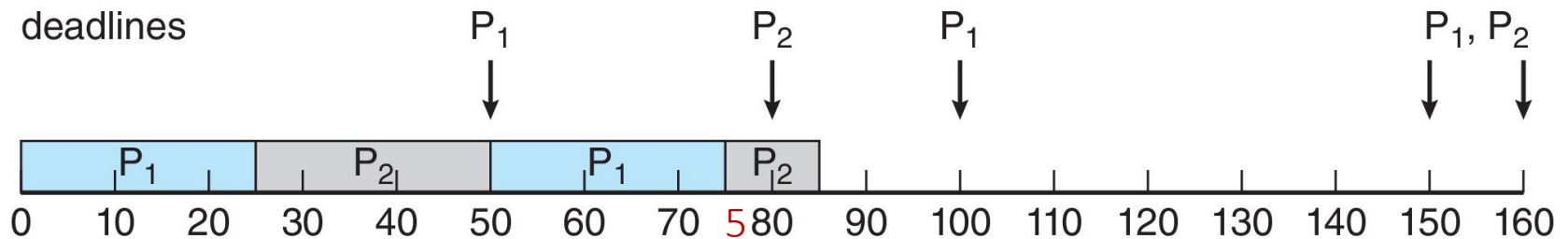




Missed Deadlines with Rate Monotonic Scheduling

■ Example:

- P_1 needs to run for 25 ms every 50 ms. $t = 25, d = p = 50$
- P_2 needs to run for 35 ms every 80 ms. $t = 35, d = p = 80$



P2:25

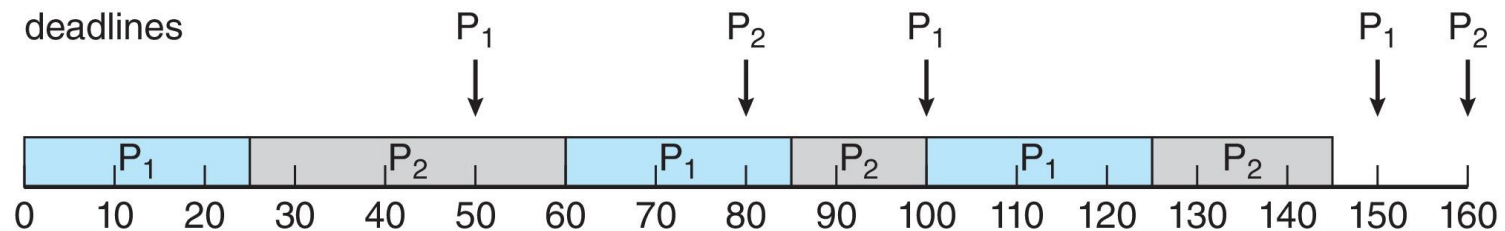
P2:needs 10

- Process P_2 misses its deadline at time 80 ms.
- Observation: if P_2 is allowed to run from 25 to 60 and P_1 then runs from 60 to 85 then both processes can meet their deadline.
- So the problem is not a lack of CPU time, the problem is that rate monotonic scheduling is not a very good algorithm.



Earliest Deadline First Scheduling (EDF)

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



- Example:
 - P_1 needs to run for 25 ms every 50 ms.
 - P_2 needs to run for 35 ms every 80 ms.
- This is the scheduling algorithm many students use when they have multiple deadlines for different homework assignments!



Proportional Share Scheduling

- T shares are allocated among all processes in the system
 - Example: $T = 20$, therefore there are 20 shares, where one share represents 5% of the CPU time
- An application receives N shares where $N < T$
 - This ensures each application will receive N / T of the total processor time
 - Example: an application receives $N = 5$ shares
 - ▶ the application then has $5 / 20 = 25\%$ of the CPU time.
 - ▶ This percentage of CPU time is available to the application whether the application uses it or not.



POSIX Real-Time Scheduling API

```
#include <pthread.h>                                     thrd-rt.c
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t_tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* get the current scheduling policy */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else {
        if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR) printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
    }
}
```



POSIX Real-Time Scheduling API (Cont.)

```
/* set the scheduling policy - FIFO, RR, or OTHER */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.\n");
/* 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 ... */
    printf("my thread ID=%u\n", *(unsigned int*)param);
    pthread_exit(0);
}
```

```
johnz@johnz-VirtualBox: ~/Desktop/OS$ ./thrd-rt
Policy: SCHED_OTHER
This is the main process.
My thread ID=1137276672.
My thread ID=1145669376.
My thread ID=1154062080.
My thread ID=1162454784.
My thread ID=1170847488.
```



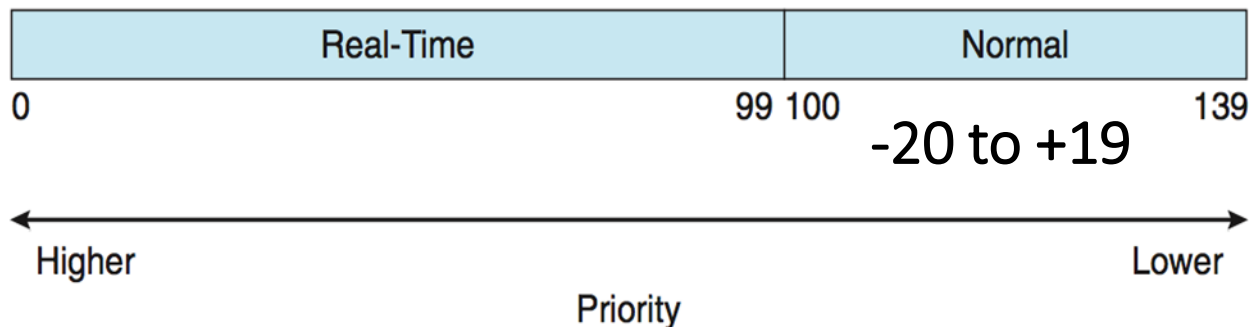
Operating System Examples

- Linux scheduling
- Windows scheduling



Linux Scheduling

- Scheduling classes
 - 2 scheduling classes are included, others can be added
 1. default
 2. real-time
 - Each process/task has specific priority
- Real-time scheduling according to POSIX.1b
 - Real-time tasks have **static priorities**
- Real-time plus normal tasks map into global priority scheme
 - Nice value of -20 maps to global priority 100
 - Nice value of +19 maps to priority 139





Linux Scheduling

■ Completely Fair Scheduler (CFS)

- Scheduler picks highest priority task in highest scheduling class
 - Quantum is not fixed
 - Calculated based on **nice value** from -20 to +19
 - » **Lower value is higher priority**

■ CFS maintains per task **virtual run time** in variable **vruntime**

- Associated with **decay factor based on priority** of task => lower priority is higher decay rate
- Normal default priority (Nice value: 0) yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with **lowest virtual run time**



Linux Scheduling

	High-priority Nice value -10	Normal-priority Nice value 0	Low-priority Nice value 10
Run-time	200 ms	200 ms	200 ms
vruntime	150 ms (decay factor: 0.75)	200 ms (decay factor: 1)	250 ms (decay factor: 1.25)



Next time:
Pick this one



Operating System Examples

- Windows scheduling
- Windows uses priority-based preemptive scheduling
 - Highest-priority thread runs next
 - ▶ Thread runs until
 1. blocks,
 2. uses time slice,
 3. preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
 - Variable class is 1-15, real-time class is 16-31
 - Priority 0 is memory-management thread
 - There is a queue for each priority
 - If no run-able thread, runs idle thread



Windows Priorities

The Windows API identifies the following six priority classes to which a process can belong:

	real-time class		Variable class				
	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

A thread within a given priority class also has a relative priority.

Variable: meaning that the priority of a thread belonging to one of these classes can change.

End of Chapter 5





Appendices

- The appendix parts are for students who are interested in knowing more about the thread programming that use Process Scope and System Scope, and algorithm evaluation.



Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
 - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
 - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- 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
void *runner(void *param);
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("Scope: PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("Scope: PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
}
```

thrd-demo2.c

gcc -o thrd-demo2 thrd-demo2.c -lpthread

pthread_t => unsigned int



Pthread Scheduling API Cont.

thrd-demo2.c

```
/* set the scheduling algorithm to PCS or SCS */
//pthread_attr_setscope(&attr, scope);

/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, &tid[i]);

printf("This is the main process\n");

/* 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 ... */
    printf("my thread ID=%d\n", *(int*)param);
    pthread_exit(0);
}
```

```
johnz@johnz-VirtualBox:~/Desktop/OS$ gcc -o thrd-demo2 thrd-demo2.c -lpthread
johnz@johnz-VirtualBox:~/Desktop/OS$ ./thrd-demo2
Scope: PTHREAD_SCOPE_SYSTEM
This is the main process.
My thread ID=1322739456.
My thread ID=1331132160.
My thread ID=1339524864.
My thread ID=1347917568.
My thread ID=1356310272.
```



Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
 1. Determine criteria
 - ▶ What is the computer used for?
 2. Evaluate algorithms
 - ▶ Find which algorithm is the best one for that kind of usage.
- Three ways to evaluate an algorithm
 1. Deterministic modeling
 2. Queueing Models
 3. Simulations



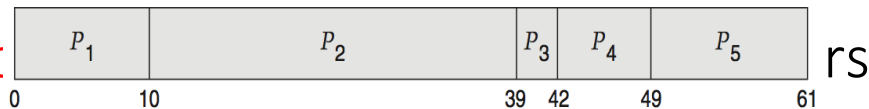
Deterministic modeling

- Deterministic modeling
 - Type of **analytic evaluation**
 - Takes a particular predetermined workload and defines the performance of each algorithm for that workload

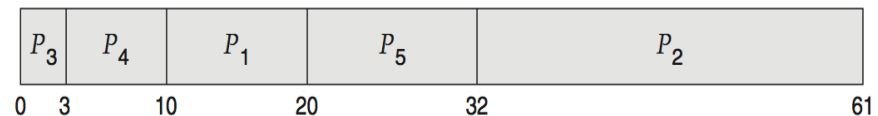
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- Consider 5 processes arriving at time 0
 - For each algorithm, calculate minimum average waiting time

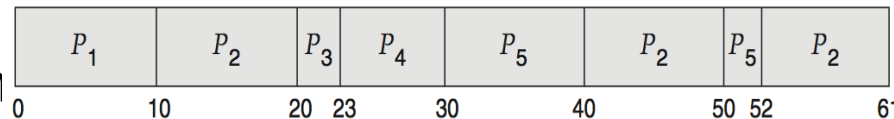
► **Simple and**
for input, **applies only**



- FCFS is 28ms:



- Non-preem

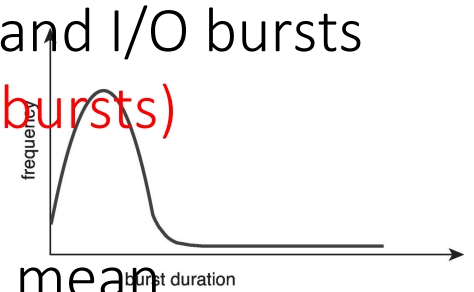


- RR is 23ms:



Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts **probabilistically (distribution of arrivals and bursts)**
 - Commonly exponential, and described by mean
 - Computes **average throughput, utilization, waiting time**, etc. according to the distribution of arrival times and CPU bursts
- Computer system is described as network of servers, each with queue of waiting processes
 - Knowing arrival rates and service rates
 - Computes utilization, average queue length, average wait time, etc.
- Queueing models has limitation
 - because the behavior of **real computers does not follow any simple ideal probability distribution**





Simulations

- Compared with Queueing models, Simulations are more accurate
 - Running simulations involves programming a model of the computer system
 - ▶ The simulator has a variable representing a clock
 - ▶ As this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler
 - Gather statistics from real computers usage indicating algorithm performance
 - Data to drive simulation can be generated in several ways
 - ▶ Uses a random-number generator that is programmed to generate processes, CPU burst times, arrivals, departures, and so on, according to probability distributions (i.e., distribution-driven, most common method)



Implementation

- Even simulations have limited accuracy
 - because trace tape used in simulation might be different from the way you use your computer
- The only completely accurate ways is
 - Just implement new scheduler and test in real systems
 - High cost, high risk, but best results
 - Environments vary
- More approaches
 - Use most flexible schedulers that can be modified or tuned per-site or per-system
 - Use APIs to modify priorities of process or thread
- However, environments can still vary