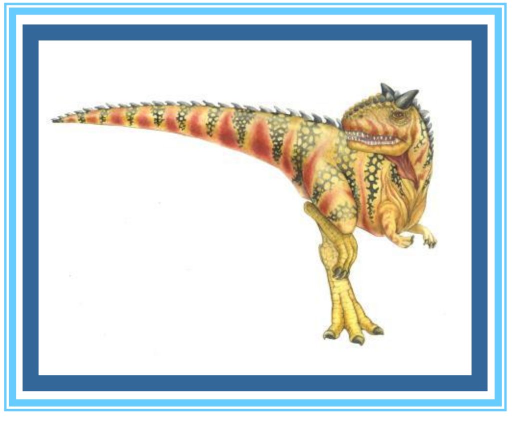
6: CPU Scheduling



Chapter 5: CPU Scheduling

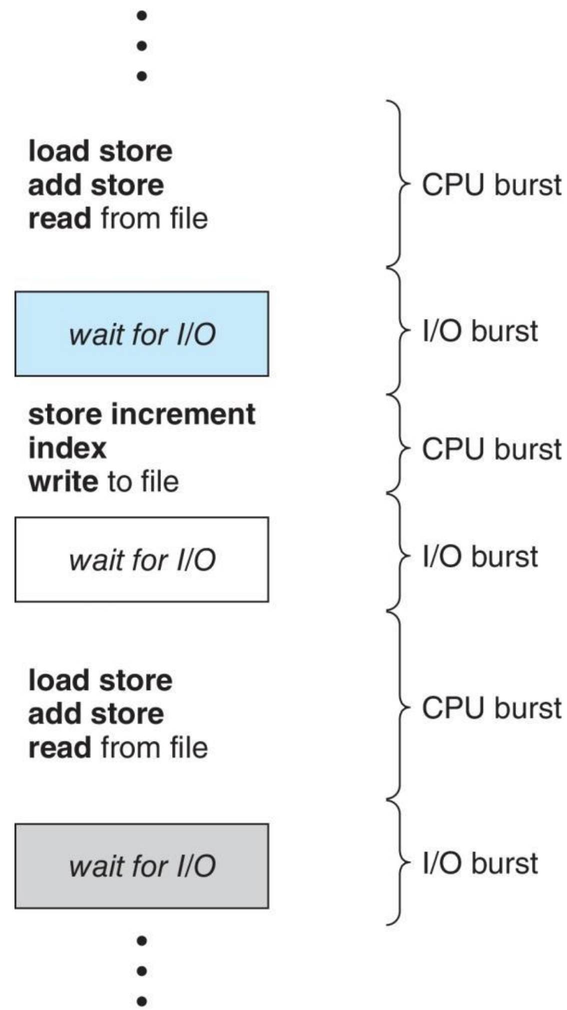
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Objectives

- Describe various CPU scheduling algorithms
  - We assume there is only 1 CPU with 1 core
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows and Linux operating systems
- Apply modeling and simulations to evaluate CPU scheduling algorithms

Basic Concepts

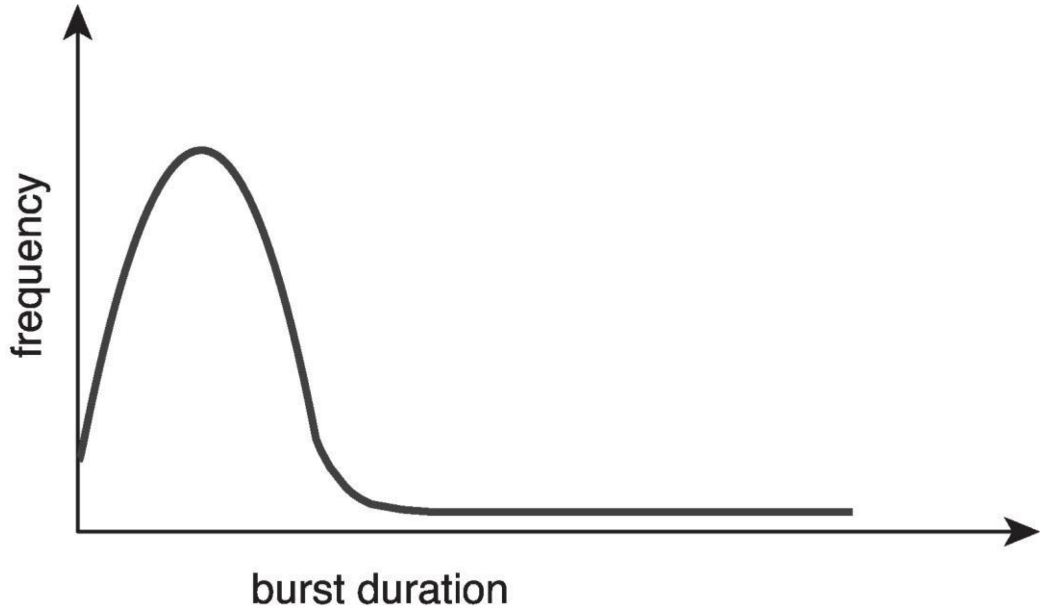
- **Maximum CPU utilization** obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



Curve of CPU-burst Times

Large number of short bursts (on typical PC)

Small number of longer bursts



The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts.

CPU Scheduler

- The **CPU scheduler** **selects from among the processes in ready queue, and allocates the CPU to one of them**
  - Ready queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
     - Example: the process does an I/O system call.
  2. Switches from running to ready state
     - Example: there is a clock interrupt.
  3. Switches from waiting to ready
     - Example: there is a hard disk controller interrupt because the I/O is finished.
  4. Terminates

CPU Scheduler

- Scheduling under 1 and 4 is **nonpreemptive** (decided by the process itself)

  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.**

- All other scheduling is **preemptive** (decided by the hardware and kernel)

  preemptive scheduling can result in **race conditions[1]** when data are shared among several processes.

  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

[1]A **race condition** is the behavior of an electronic, software, or other system where the output is dependent on the sequence or timing of other uncontrollable events. It becomes a bug when events do not happen in the order the programmer intended. Race conditions can occur in electronics systems, **especially logic circuits**, and in computer software, **especially multithreaded programs**.

Preemptive vs. Nonpreemptive Scheduling

- Preemptive processes
  - Can be removed from their current processor
  - Can lead to improved response times
  - Important for interactive environments
  - Preempted processes remain in memory
- Non-preemptive processes
  - Run until completion or until they yield control of a processor
  - Unimportant processes can block important ones indefinitely

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – number of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process (from start to end of process, including waiting time)
- **Waiting time** – total 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, not output (for time-sharing environment)
- Turnaround time = Waiting time + time for all CPU bursts

How to calculate these criteria?

Scheduling Algorithm Optimization Criteria

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

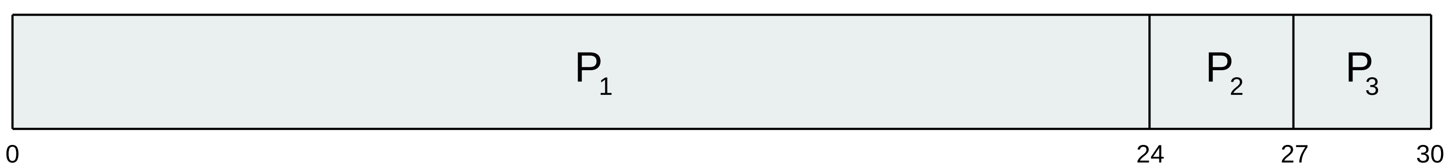
Scheduling Algorithms

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

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the ready queue at time t = 0 in the following 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
- **Average turnaround time** = (24+27+30)/3 = 27

FCFS Scheduling (Cont.)

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: $(6 + 0 + 3) / 3 = 3$
- **Average turnaround time** = (30+3+6)/3 = 13
- Much better than previous case
- **Convoy effect[1]** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

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

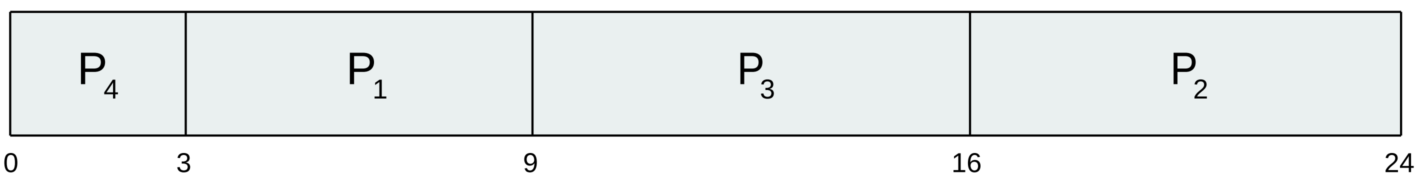
Shortest-Job-First (SJF) Scheduling

- 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

Example of SJF

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

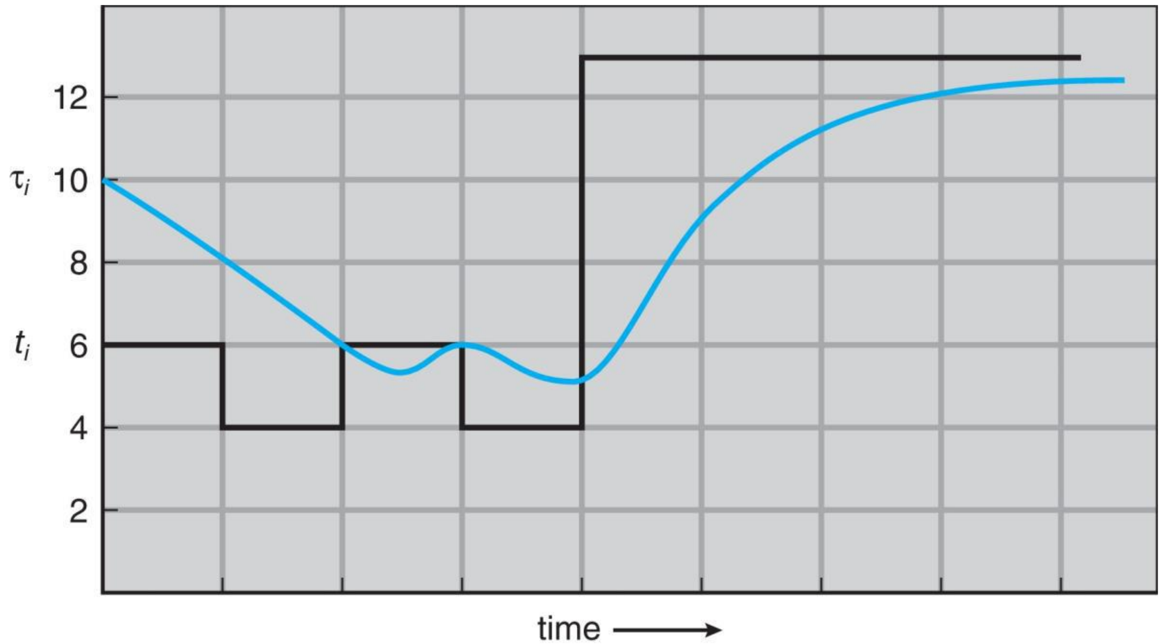
- Can only **estimate** the length – should be similar to the previous one (use the past to predict the future).
  - Then pick process with shortest predicted next CPU burst

- 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. $\tau_{n+1} =$ predicted value for the next CPU burst
  3. $\alpha, 0 \leq \alpha \leq 1$
  4. Define : $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

- Commonly, α set to ½
- Preemptive version called **shortest-remaining-time-first**

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	…

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

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n = \ldots = \tau_0$
  - 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:

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

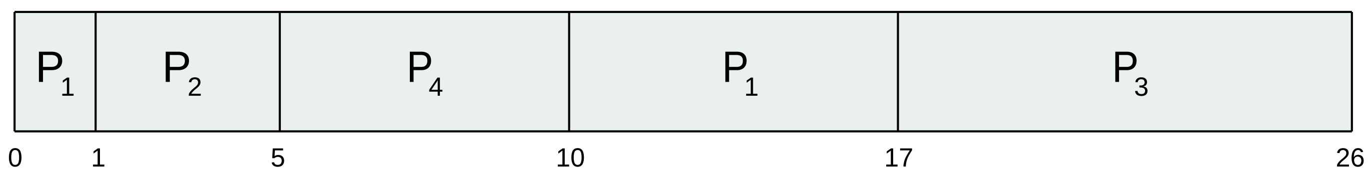
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Example of Shortest-remaining-time-first

- Now we 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



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)] / 4 = 26 / 4 = 6.5 msec
- Average turnaround time = (17+4+24+7)/4 = 13

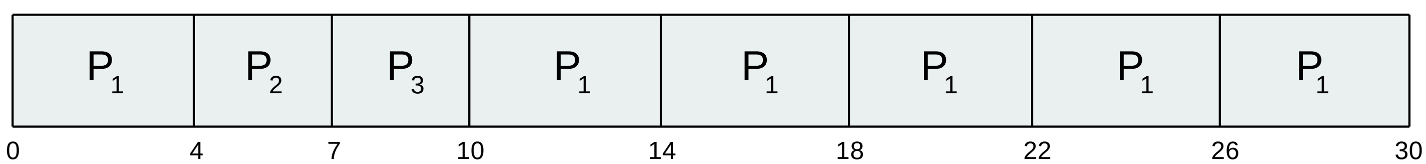
Round Robin (RR)

- 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 by **a clock interrupt** 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.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large $\Rightarrow$ FCFS
  - *q* small $\Rightarrow$ *q* must be large with respect to **context switch**, otherwise overhead is too high

Example of RR with Time Quantum = 4

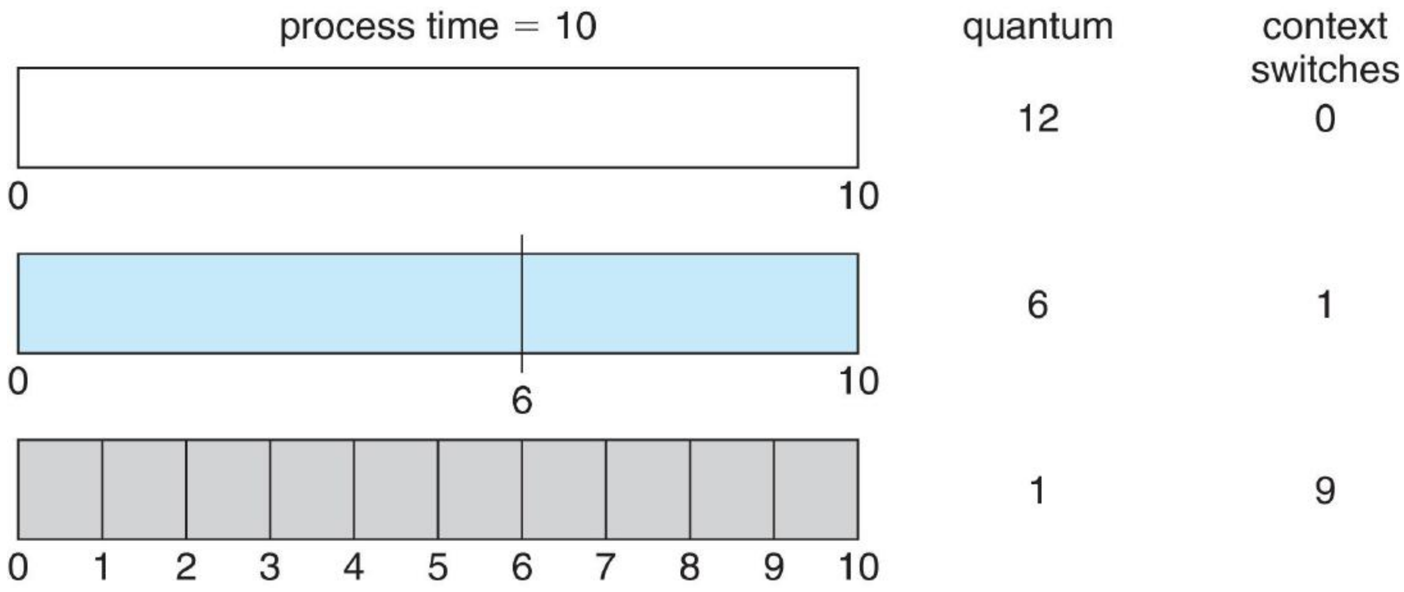
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

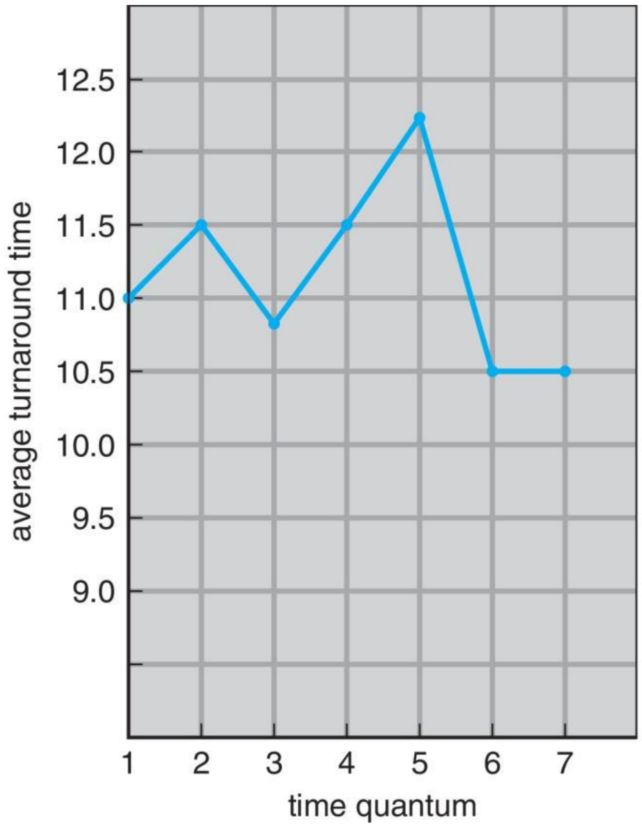


- Typically, higher average turnaround than SJF, but better ***response***
- Average turnaround time = (30+7+10)/3 = 15.7
- Average waiting time = (6+4+7)/3 = 5.67
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



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

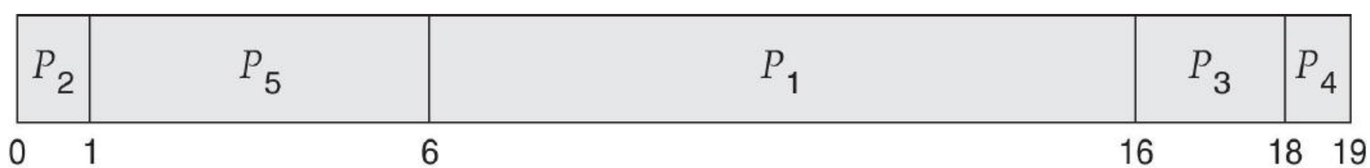
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with **the highest priority** (smallest integer ≡ 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 (not preemptive) Gantt Chart



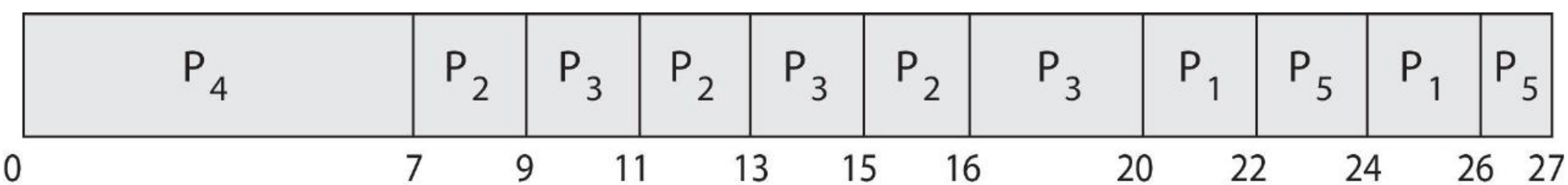
- Average waiting time = (6+0+16+18+1)/5 = 8.2 msec

Priority Scheduling w/ Round-Robin

Process	Burst Time	Priority
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**

- Gantt Chart wit 2 ms time quantum (q=2)



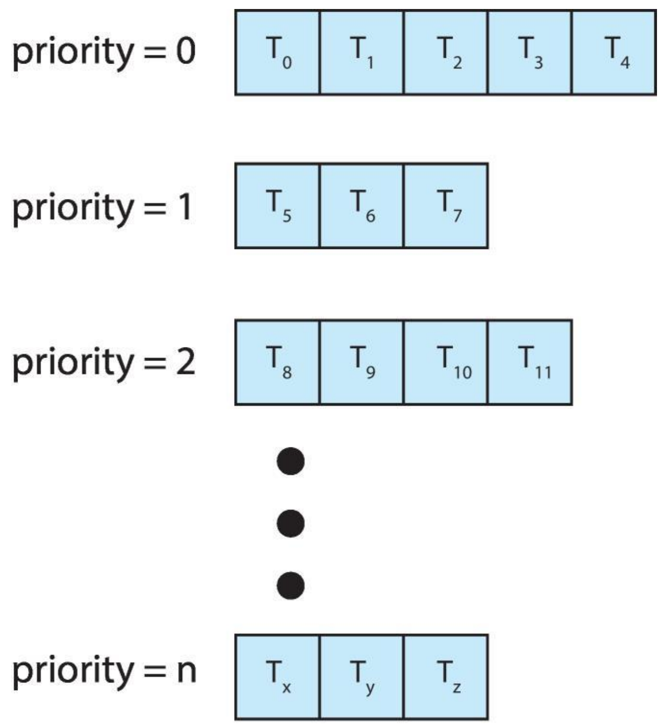
Average waiting time = (22+11+12+0+24)/5 = 13.8 msec

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive processes)
  - **background** (batch 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**; (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; i.e., 80% to foreground in RR, 20% to background in FCFS

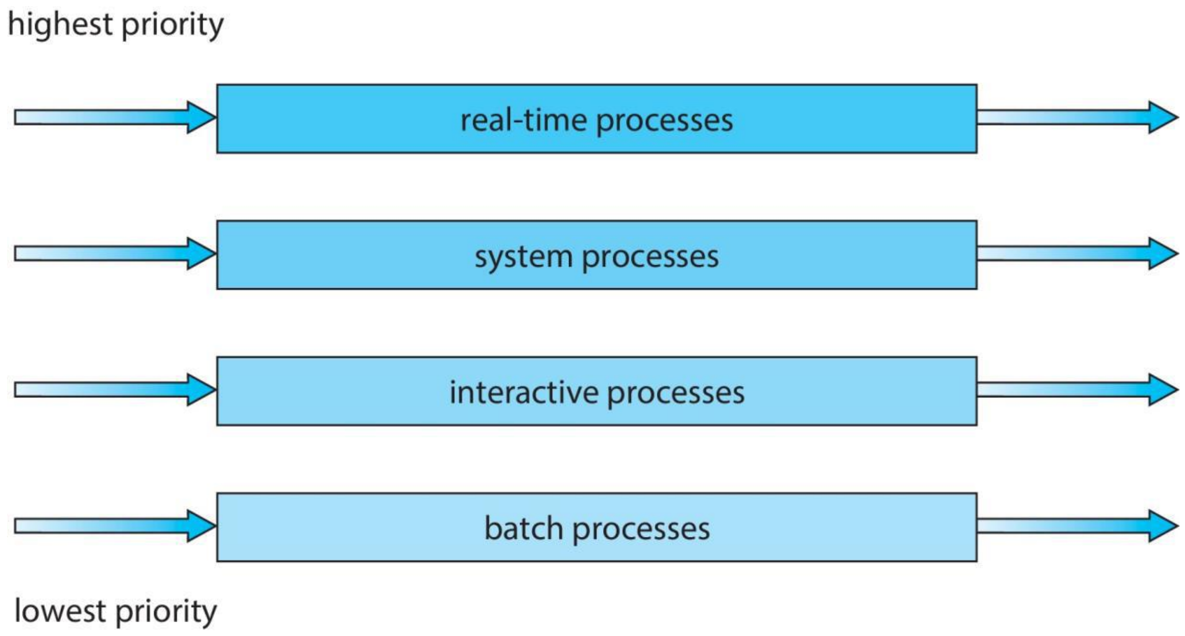
Multilevel Queue

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



Multilevel Queue

- Prioritization based upon **process type**



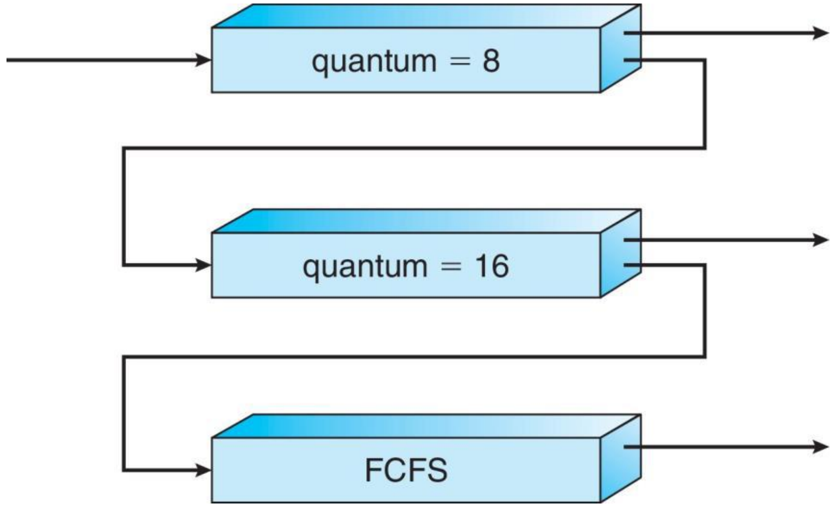
Multilevel Feedback Queue

- **A process can move between the various queues**; aging can be implemented this way(prevent starvation)
- 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

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm.

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR with time quantum 16 milliseconds
  - 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

- Distinction between user-level and kernel-level threads
- When threads supported by kernel, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on kernel threads (themselves scheduled by kernel)
  - Known as **process-contention scope (PCS)** since scheduling competition is between user-level threads within the same process
  - Typically done via priority set by programmer
- **Kernel thread** scheduled onto available CPU is **system-contention scope (SCS)** – competition among all kernel-level threads within all processes in the system

Process Contention Scope

- **Process Contention Scope** is one of the two basic ways of scheduling threads.
  - Process local scheduling (known as Process Contention Scope)
  - System global scheduling (known as System Contention Scope).
- PCS scheduling means that all of the scheduling mechanism for the thread is local to the process—the thread's library has full control over which thread will be scheduled on an LWP. This also implies the use of either the Many- to-One or Many-to-Many model.
- PCS: done by the threads library. The library chooses which thread will be put on which LWP.
- **SCS:** used by the kernel to decide which kernel-level thread to schedule onto a CPU, wherein all threads (as opposed to only user-level threads, as in the PCS) in the system compete for the CPU. This also implies the use of **One- to-One model**.

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

thrd-demo2.c

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

```
#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");
   }
```

Pthread Scheduling API Cont.

thrd-demo2.c

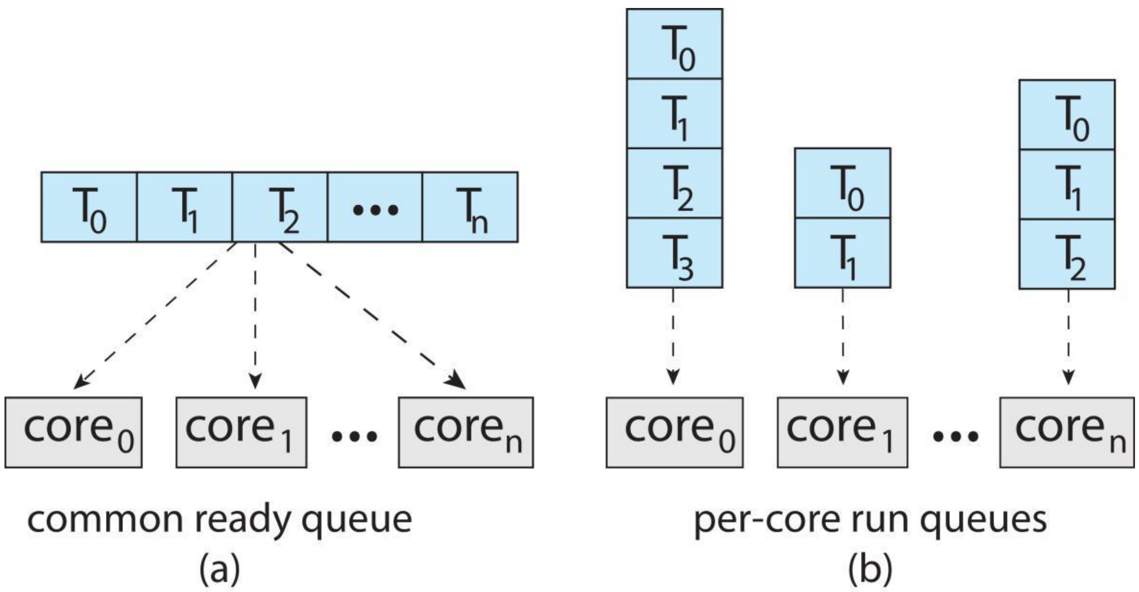
```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, scope);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,&tid[i]);

   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);
}
```

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems

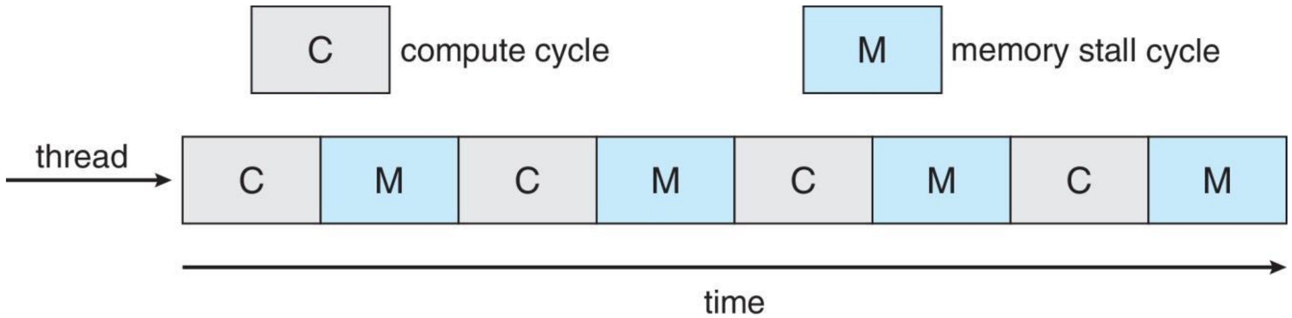
Multiple-Processor Scheduling

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



Multicore Processors

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

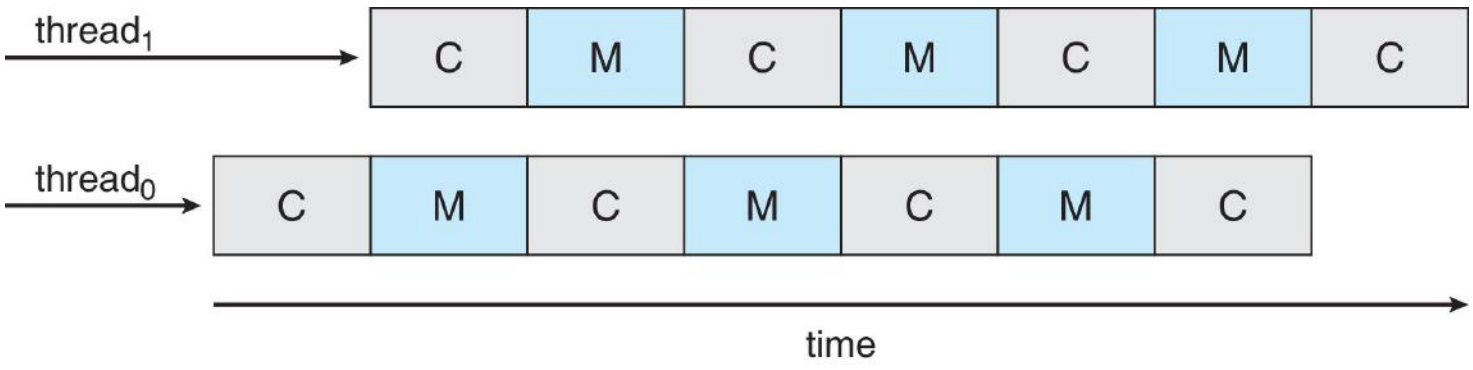


memory stall: An event that occurs when a thread is on CPU and accesses memory content that is not in the CPU's cache. The thread's execution stalls while the memory content is fetched.

Multithreaded Multicore System

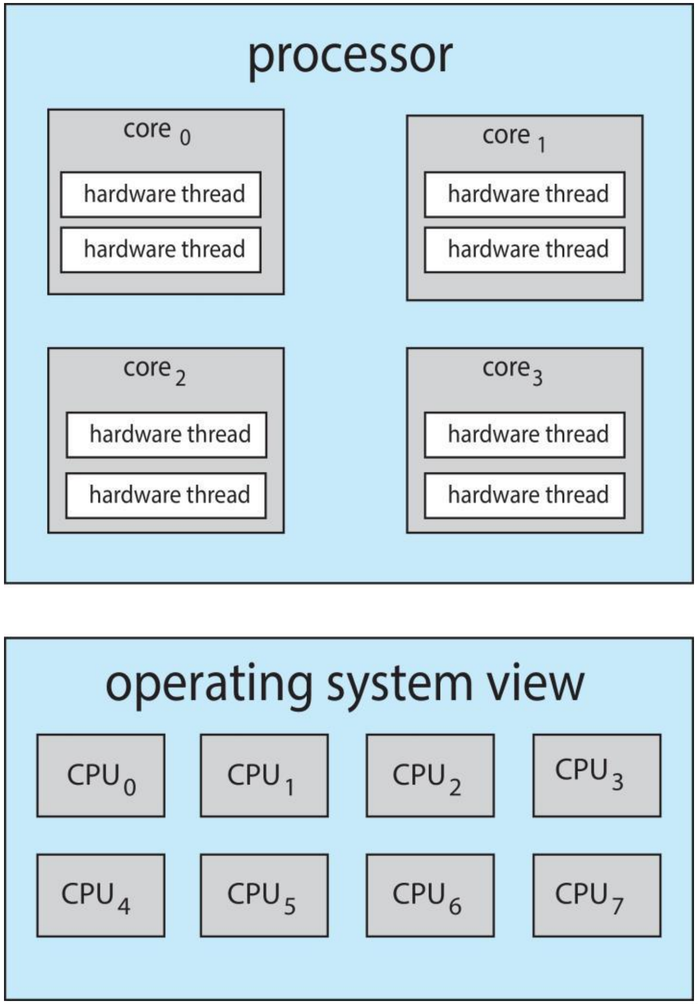
Each core has > 1 hardware threads.

If one thread has a memory stall, switch to another thread!



Multithreaded Multicore System

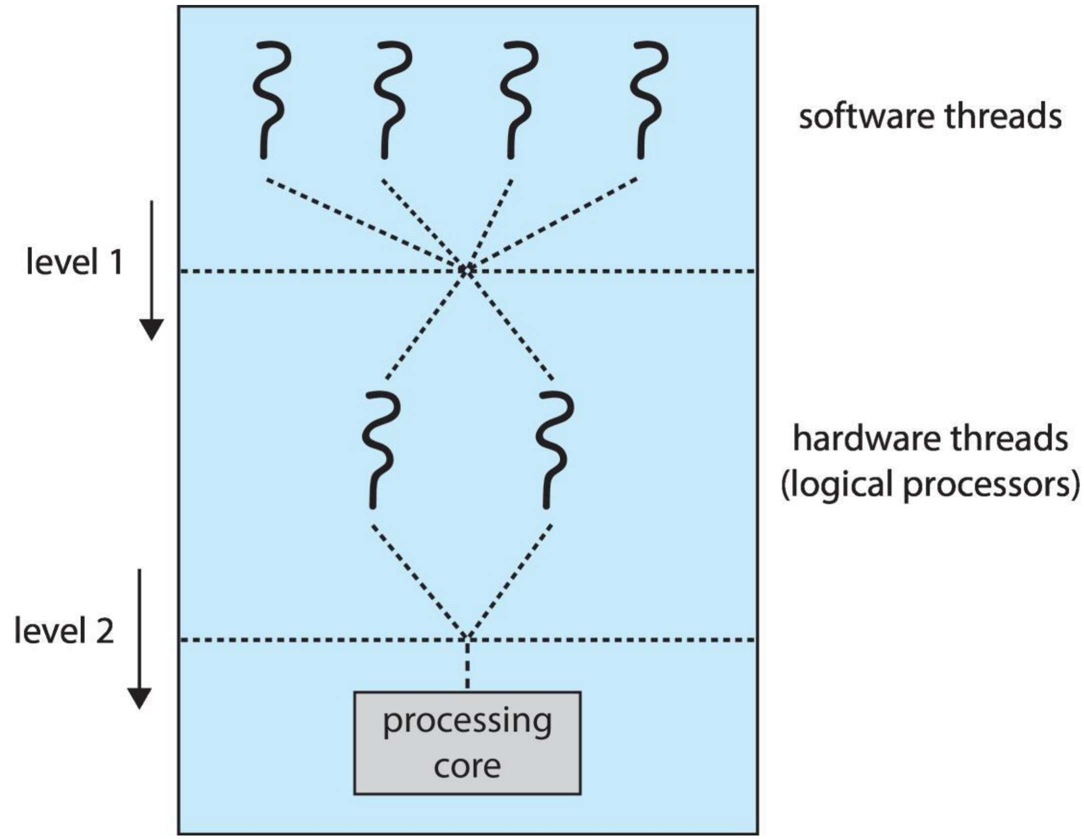
- **Chip-multithreading** (CMT) assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)
- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.



Multithreaded Multicore System

- Two levels of scheduling:

1. The operating system deciding which software thread to run on a logical CPU
2. How 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

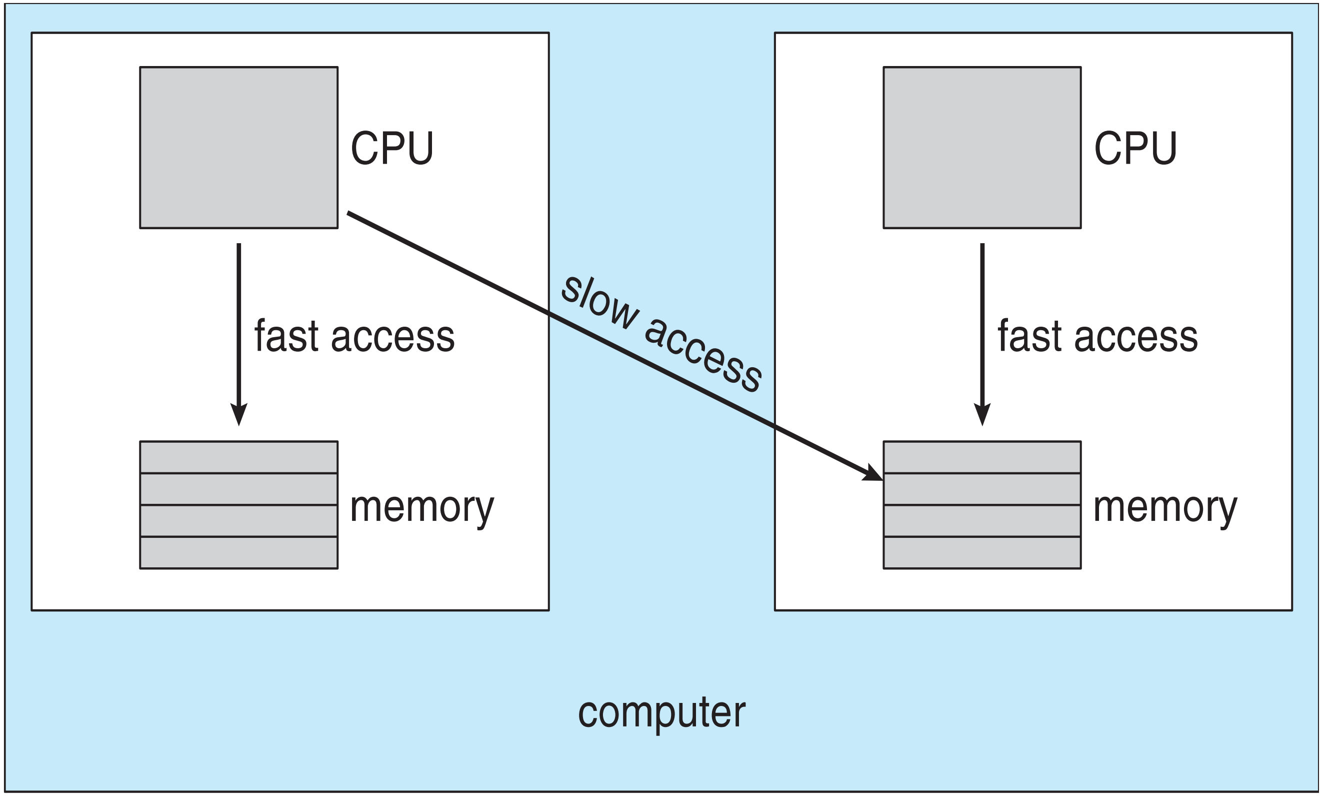
Push and pull migration need not be mutually exclusive and are in fact **often implemented in parallel** on load-balancing systems.

Multiple-Processor Scheduling – 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** (i.e. “**processor affinity**”)
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
  1. **Soft affinity** – the operating system attempts to keep a thread running on the same processor, **but no guarantees**.
  2. **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.



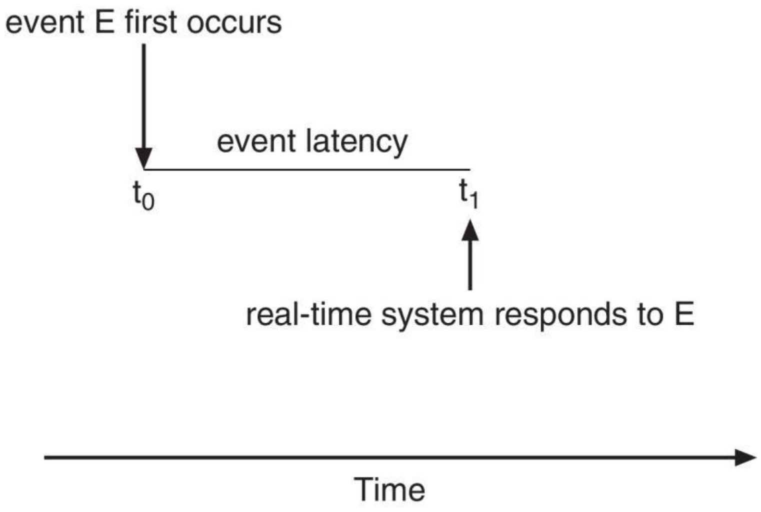
Non-uniform memory access (**NUMA**) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own **local memory** faster than **non-local memory** (memory local to another processor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

Real-Time CPU Scheduling

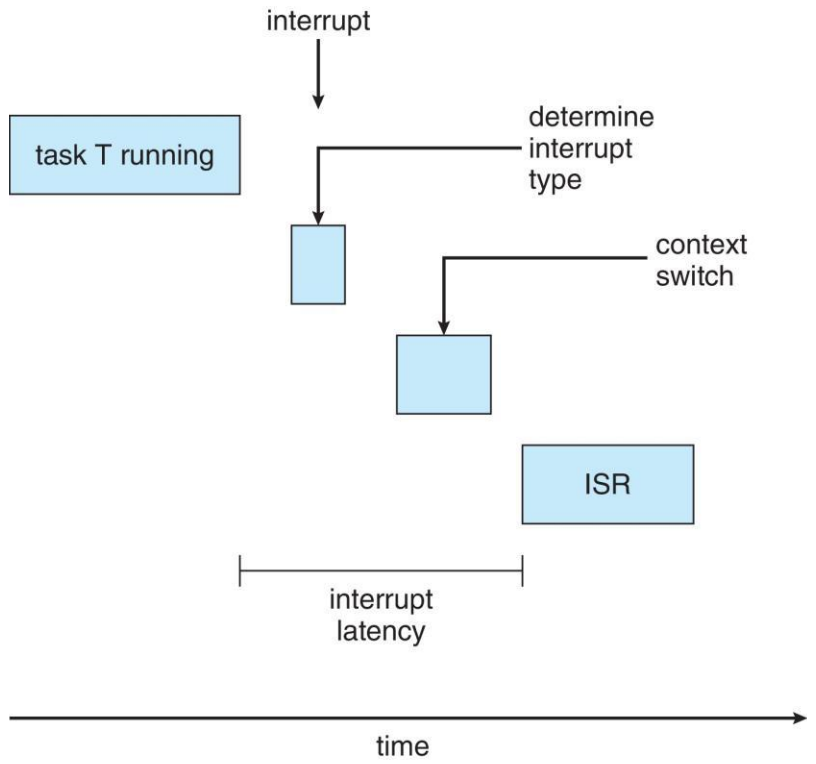
- Can present obvious challenges
- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled (best try only)
- **Hard real-time systems** – a task must be serviced by its deadline (guarantee)

Real-Time CPU Scheduling

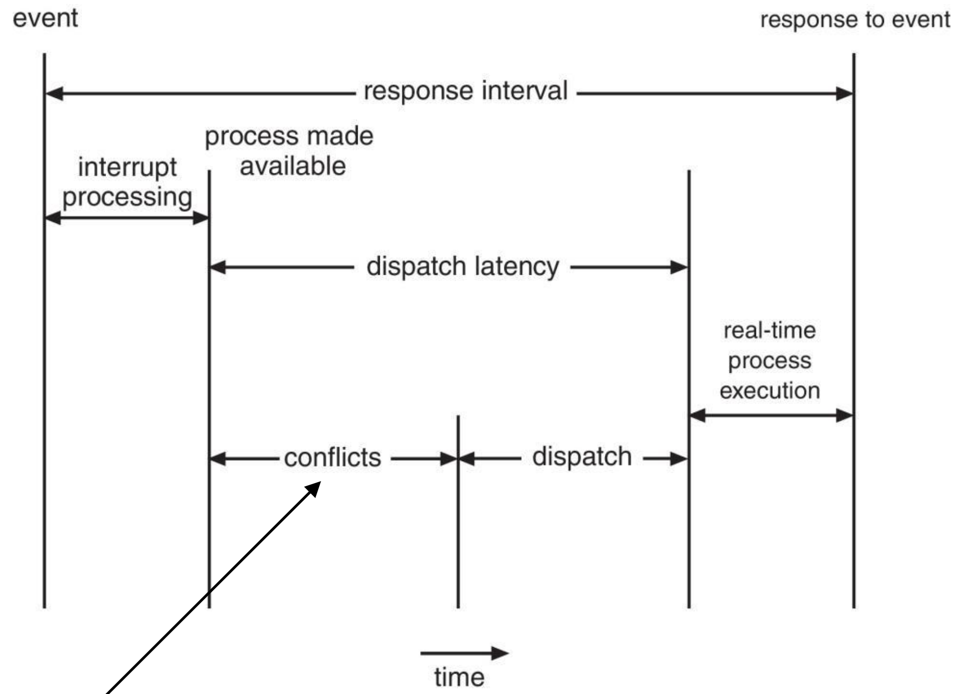
- **Event latency** – the amount of time that elapses from **when an event occurs to when it is serviced**.
- Two types of latencies affect performance
  1. **Interrupt latency** – time from arrival of interrupt to start of kernel interrupt service routine (ISR) that services interrupt
  2. **Dispatch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



Interrupt Latency



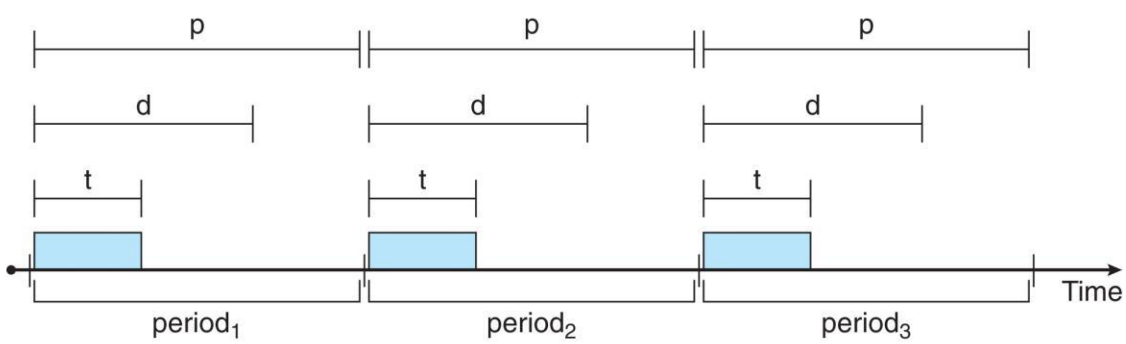
Dispatch Latency



- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes

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*
  - $0 \le t \le d \le p$
  - **Rate** of periodic task is $1/p$



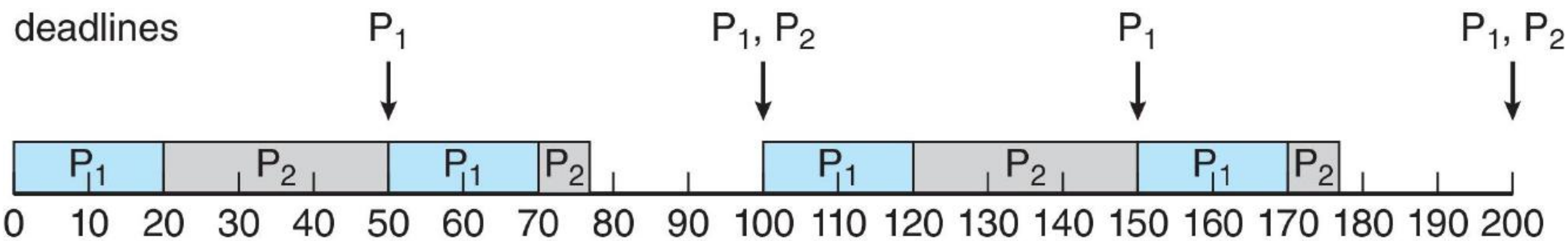
Rate Monotonic Scheduling

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

  Shorter periods = higher priority

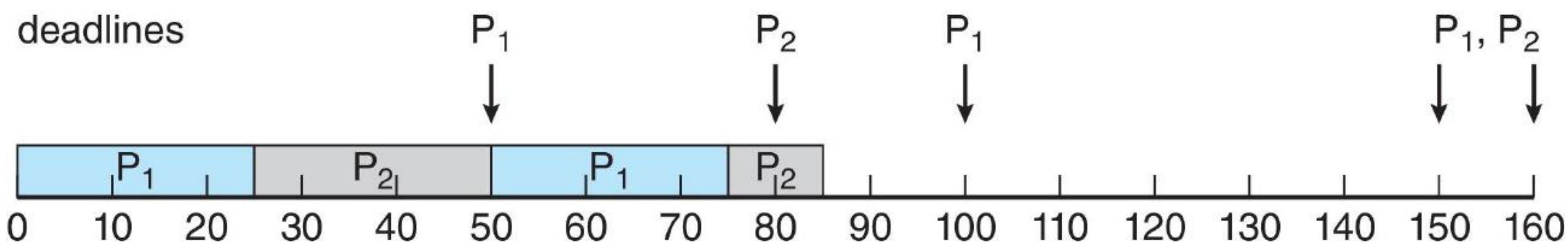
  Longer periods = lower priority

- 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.
  - P_2 needs to run for 35 ms every **100** ms.



Missed Deadlines with Rate Monotonic Scheduling

- Example:
  - P_1 needs to run for 25 ms every 50 ms.
  - P_2 needs to run for 35 ms every 80 ms.



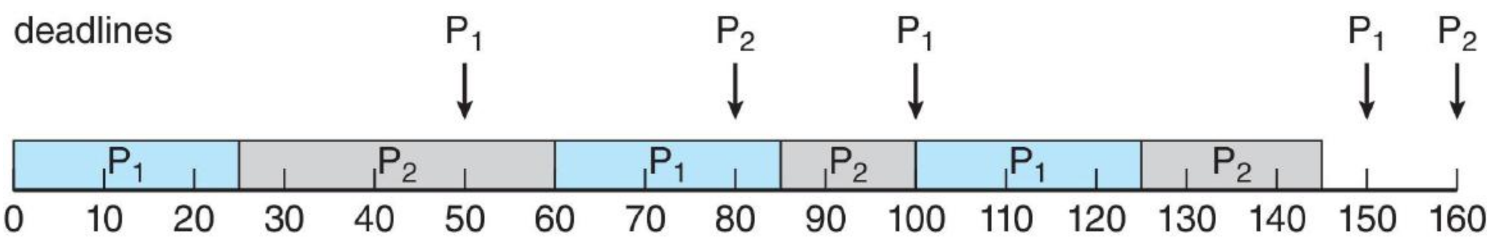
- Process P_2 **misses** its deadline **at time 80 ms**.
- Notes: **if P_2 is allowed to run from 25 to 60 and P_1 then runs from 60 to 85 then both processes can meet their deadline.**
- So the problem is not a lack of CPU time, the problem is that **rate monotonic scheduling is not a very good algorithm.**

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 $N < T$
  - Example: an application receives *N* = 5 shares
- This ensures each application will receive ***N*** / *T* of the total processor time
  - Example: 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.

Operating System Examples

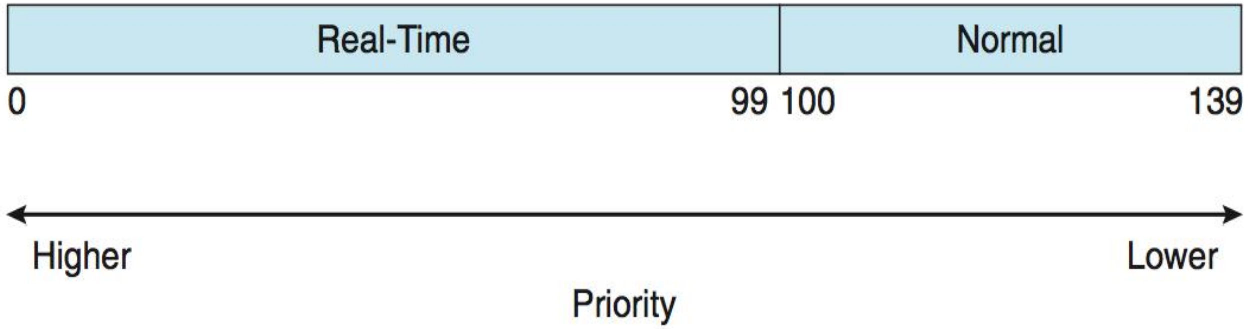
- Linux scheduling
- Windows scheduling

Linux Scheduling

- ***Completely Fair Scheduler*** (CFS)
- **Scheduling classes**
  - Each process/task has **specific priority**
  - Scheduler picks **highest priority task** in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on **proportion of CPU time**
  - 2 scheduling classes included, others can be added
    1. default
    2. real-time
- **Quantum** calculated based on **nice value** from **-20 to +19**
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- 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 yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with **lowest virtual run time**

Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



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
- Queue for each priority
- If no run-able thread, runs **idle thread**

Windows Priorities

	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

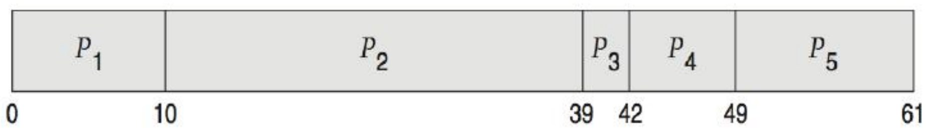
Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- **Determine criteria, then evaluate algorithms**
  - What is the computer used for?
  - Then find which algorithm is the best one for that kind of usage.
- **Deterministic modeling**
  - 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:

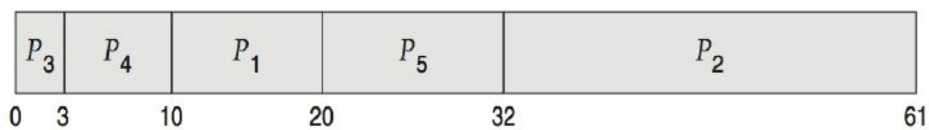
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

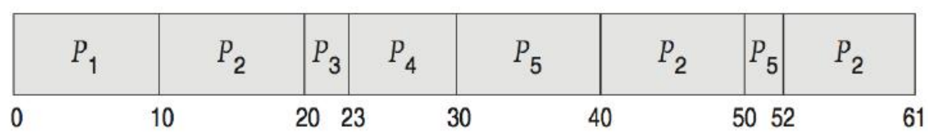
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCFS is 28ms:



  - Non-preemptive SJF 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. The CPU is allocated to the selected process by the **dispatcher**.
- Scheduling algorithms:
  - FCFS(First-Come,First-Served)
  - SJF(Shortest-Job-First or **shortest-remaining-time-first**)
  - RR(Round-Robin)
  - Priority-Based
  - Multilevel Queue
  - Multilevel Feedback Queue

The FCFS algorithm is nonpreemptive; the RR algorithm is preemptive.

The SJF and priority algorithms may be either preemptive or nonpreemptive.

Starvation problem => aging (solution)

Multilevel queue algorithms allow different algorithms to be used for different classes of processes. The most common model includes a foreground interactive queue that uses RR scheduling and a background batch queue that uses FCFS scheduling. Multilevel feedback queues allow processes to move from one queue to another.

Summary

- Thread Scheduling
  - Process-contention scope
  - System-contention scope
- Multiprocessor scheduling

  Typically, each processor maintains its own private queue of processes (or threads), all of which are available to run. Additional issues related to multiprocessor scheduling include processor affinity, load balancing, and multicore processing.
- Real-time CPU Scheduling
  - Priority-Based
  - **Rate-Monotonic**
  - ✓ **Earliest-Deadline-First**
  - Proportional Share
- The POSIX Pthread API provides various features for scheduling real-time threads.

End of Lecture 6