Chapter 5: CPU Scheduling

Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Thread Scheduling
- Operating Systems Examples
- Java Thread Scheduling
- Algorithm Evaluation
- **Project Proposal and Teams Formation: Deadline 16th November 2009**

Before we start

- 27th November, Chair of Microsoft Europe, **Mr. Jan Muehlfeit,** is coming at UNYT and will hold a discussion session (**to be confirmed!!!)**
  - As part of the Participation section of OS you are required to attend. Absences will be taken and recorded.
  - The mid-term exam will be shifted with one week.
- 26th November, afternoon (**time to be confirmed**)
  - CSD will organize mini-workshops on Microsoft solutions
  - As part of the Participation section of OS you are required to attend. Absences will be taken and recorded.

Before we start

- **Scheduling** is a key concept in computer multitasking and multiprocessing operating system design, and in real-time operating system design. In modern operating systems, there are typically many more processes running than there are CPUs available to run them.
- **Automated planning and scheduling** is a branch of artificial intelligence that concerns the realisation of strategies or action sequences, typically for execution by intelligent agents, autonomous robots and unmanned vehicles. Unlike classical control and classification problems, the solutions are complex, unknown and have to be discovered and optimised in multidimensional space.
- **Scheduling** as well as **game theory** are two classical areas in applied mathematics and artificial intelligence of great academic interest and broad practical relevance.

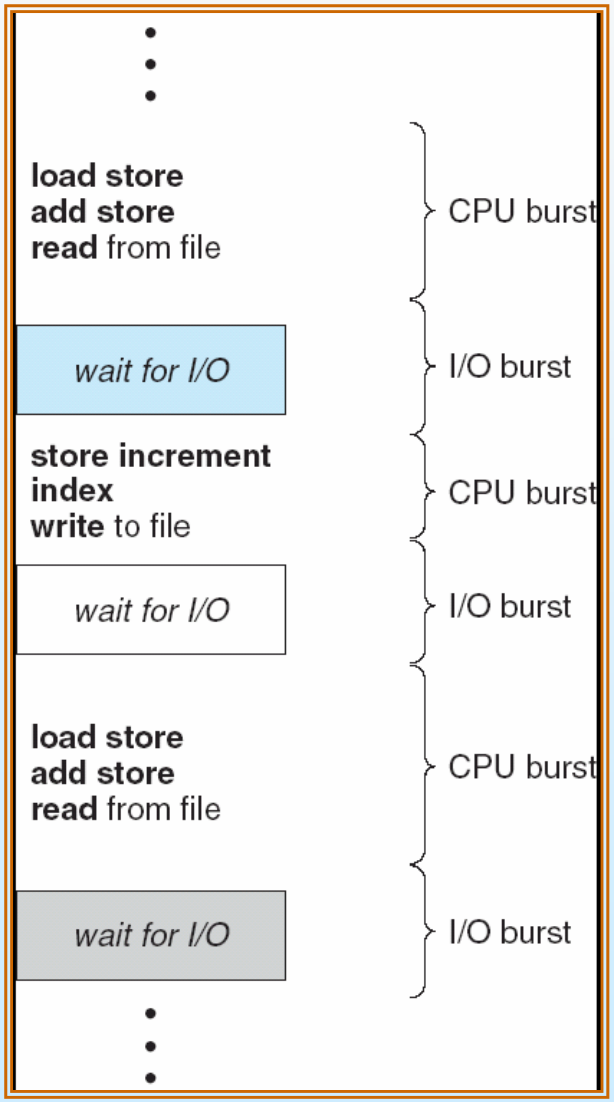
Introduction to Scheduling

- In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled.
  - In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. A process is executed until it must wait, typically for the completion of some I/O request.
- With multiprogramming, we try to use this time productively.
  - Several processes are kept in memory at one time.
  - When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

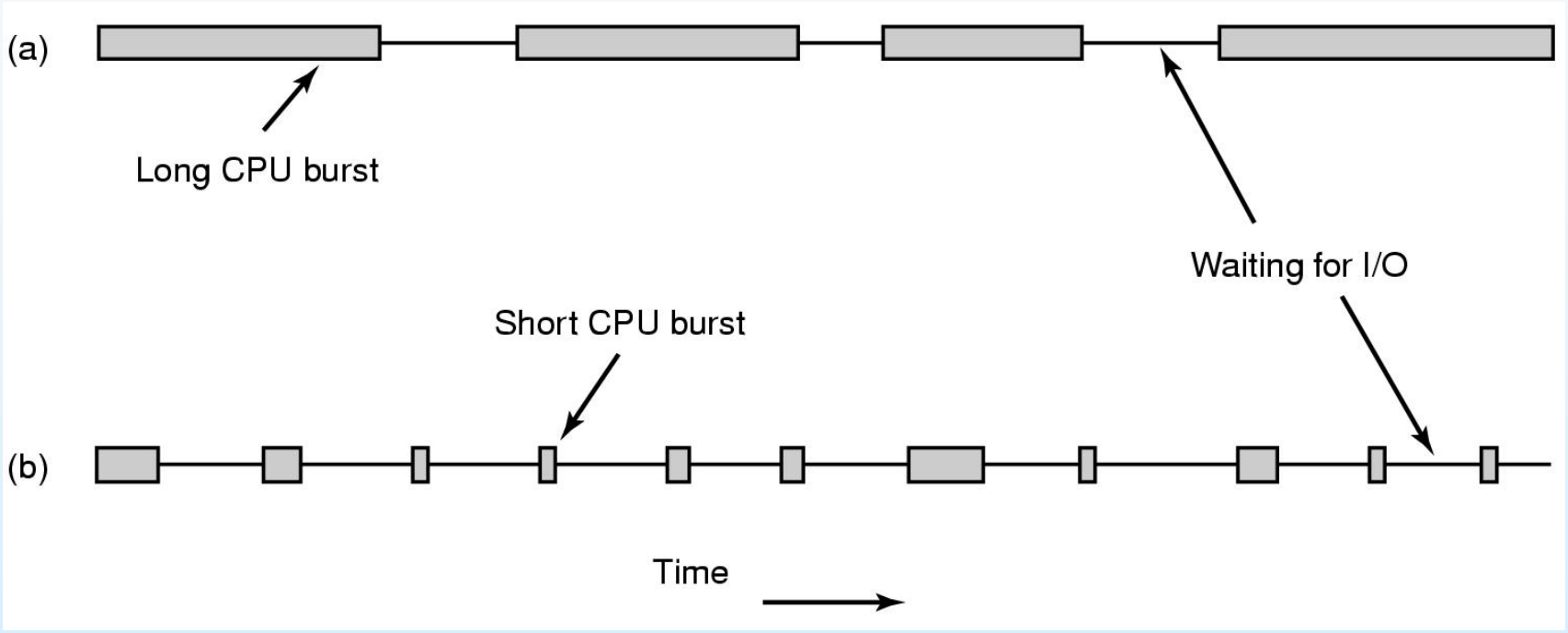
CPU-I/O Burst Cycle

- The success of CPU scheduling depends on an observed property of processes:
  - Process execution consists of a cycle of CPU execution and I/O wait.
  - Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
  - Eventually, the final CPU burst ends with a system request to terminate execution

Alternating Sequence of CPU And I/O Bursts



CPU and I/O Burst

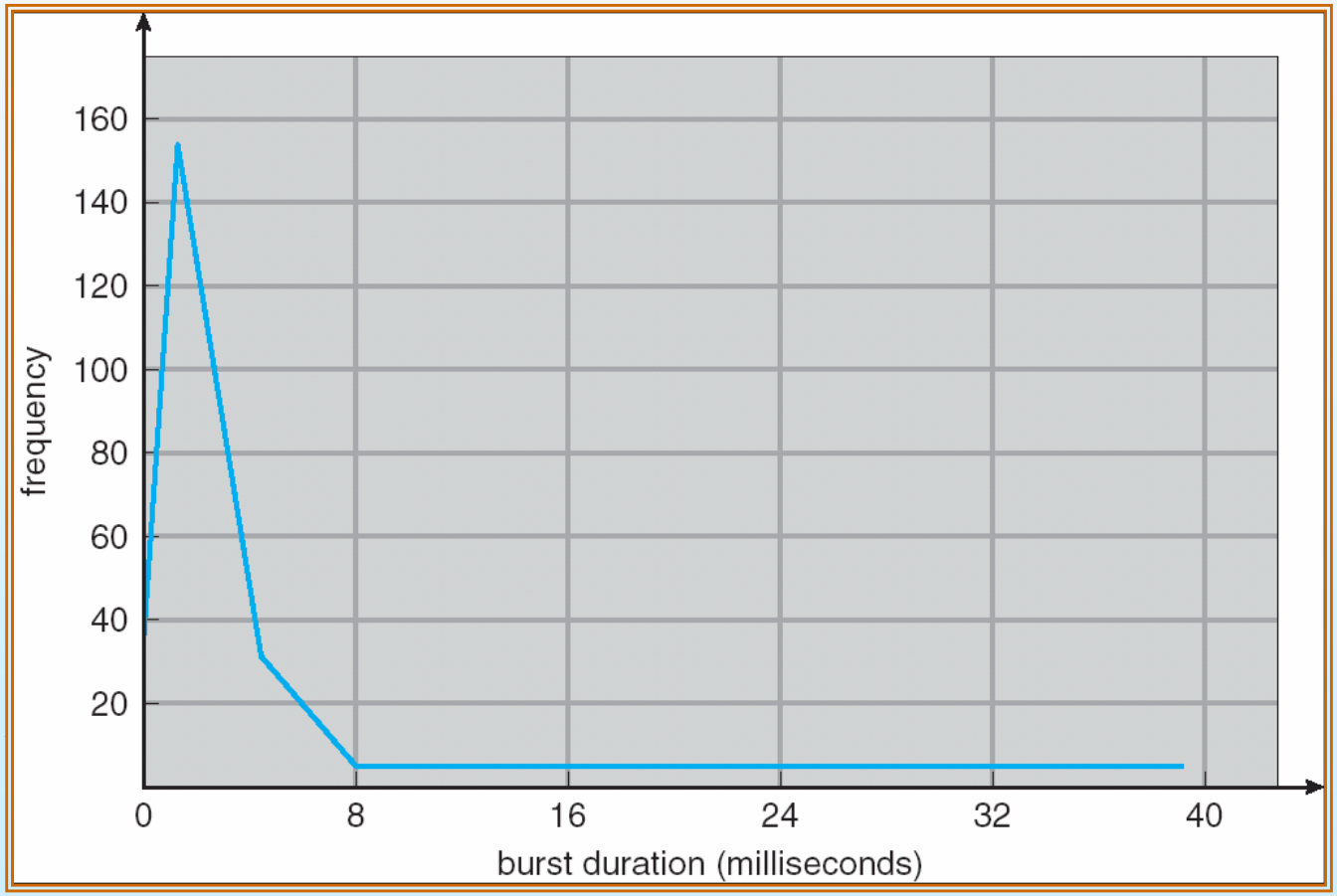


- Bursts of CPU usage alternate with periods of I/O wait
  - a CPU-bound process
  - an I/O bound process

CPU-I/O Burst Cycle

- The durations of CPU bursts have been measured extensively.
- Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in the figure in the next slide.
- 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.
- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.
- This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

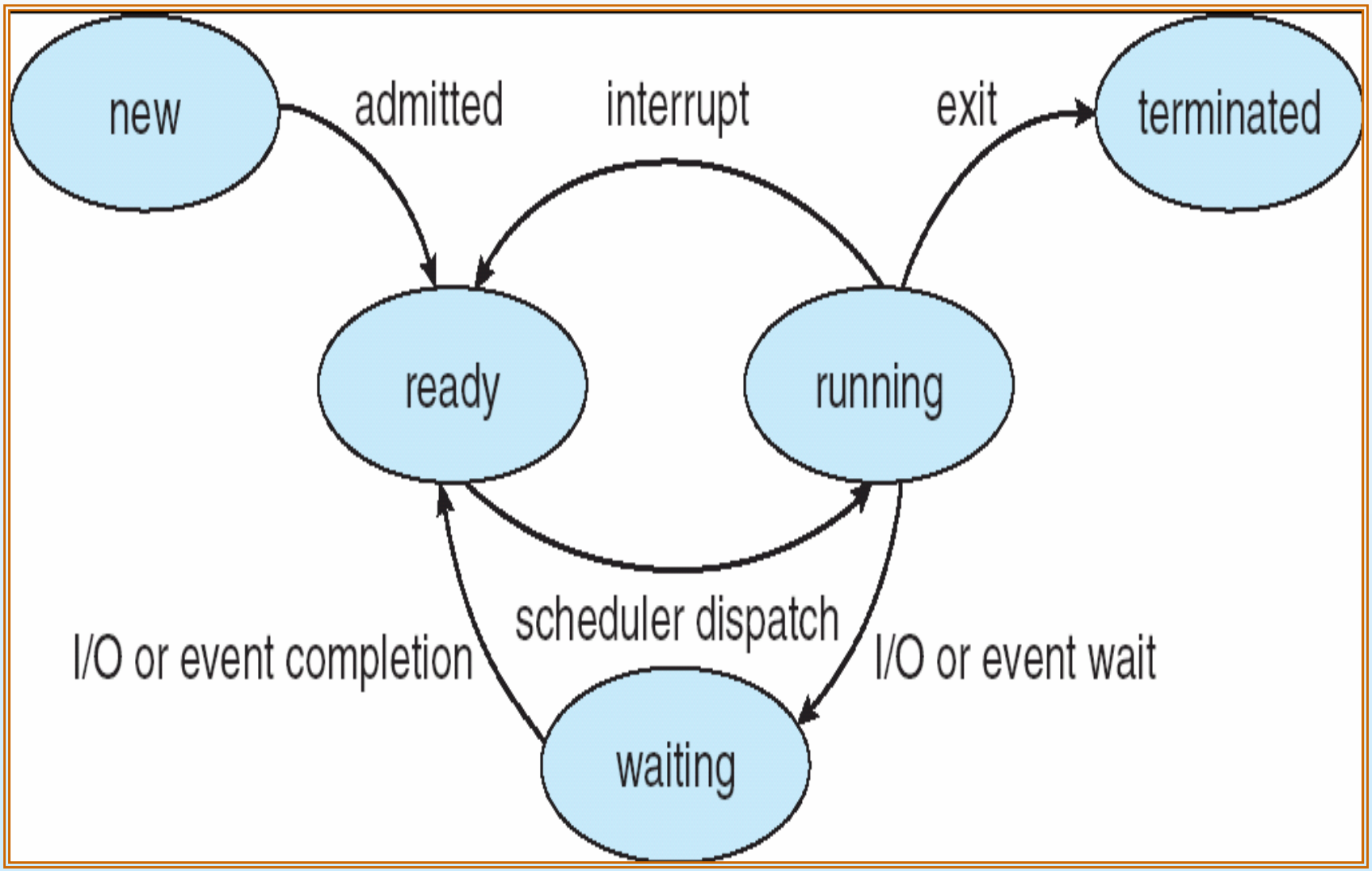
Histogram of CPU-burst Times



CPU scheduler

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.
- Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue.
- As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.
- Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

Recollect from Chap.3



Preemptive scheduling

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
     - For example, as the result of an *IIO* request or an invocation of wait for the termination of one of the child processes)
  2. Switches from running to ready state
     - For example, when an interrupt occurs
  3. Switches from waiting to ready
     - For example, at completion of I/O
  4. Terminates
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.
- Scheduling under 1 and 4 is *nonpreemptive*
- All other scheduling is *preemptive*

Preempt

- **1 :** to acquire (as land) by preemption
- **2 :** to seize upon to the exclusion of others **:** take for oneself <the movement was then preempt*ed* by a lunatic fringe>
- **3 :** to replace with something considered to be of greater value or priority **:** take precedence over <the program did not appear, having been preempt*ed* by a baseball game — Robert MacNeil>
- **4 :** to gain a commanding or preeminent place in
- **5 :** to prevent from happening or taking place **:** forestall, preclude
- **From Merriam-Webster Online Dictionary**

Non-preemptive scheduling

- Under non-preemptive 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.
- This scheduling method was used by Microsoft Windows 3.x;
  - Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling.
  - The Mac OS X operating system for the Macintosh uses preemptive scheduling; previous versions of the Macintosh operating system relied on cooperative scheduling.
- Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- The dispatcher should be as fast as possible, since it is invoked during every process switch.
  - The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Scheduling Criteria 1

- Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.
- In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
  - **(Analysis for your project)**
- Many criteria have been suggested for comparing CPU scheduling algorithms.
- Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best.

Scheduling Criteria 2

- **CPU utilization**
  - We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput** – # of processes that complete their execution per time unit
  - If the CPU is busy executing processes, then work is being done. One measure of work is the **throughput**. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.
- **Turnaround time** – amount of time to execute a particular process
  - From the point of view of a particular process, the important criterion is how long it takes to execute that process.
  - The interval from the time of submission of a process to the time of completion is the **turnaround *time***.
  - **Turnaround time** is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Scheduling Criteria 3

- **Waiting time** – amount of time a process has been waiting in the ready queue
  - The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue.
  - *Waiting time* is the sum of the periods spent 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)
  - In an interactive system, **turnaround time** may not be the best criterion.
  - Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user.
  - Thus, another measure is the time from the submission of a request until the first response is produced. The **response time**, is the time it takes to start responding, not the time it takes to output the response.
  - The turnaround time is generally limited by the speed of the output device.

Optimization Criteria 1

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

Optimization Criteria 2

- It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.
  - In most cases, we optimize the average measure.
- However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average.
  - For example, to guarantee that all users get good service, we may want to minimize the maximum response time.
- Investigators have suggested that, for interactive systems (such as timesharing systems), it is more important to minimize the *variance* in the response time than to minimize the average response time.
  - A system with reasonable and *predictable* response time may be considered more desirable than a system that is faster on the average but is highly variable.
  - However, little work has been done on CPU-scheduling algorithms that minimize variance.

Scheduling Algorithm Goals

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms.

All systems

- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

Batch systems

- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

Interactive systems

- Response time - respond to requests quickly
- Proportionality - meet users' expectations

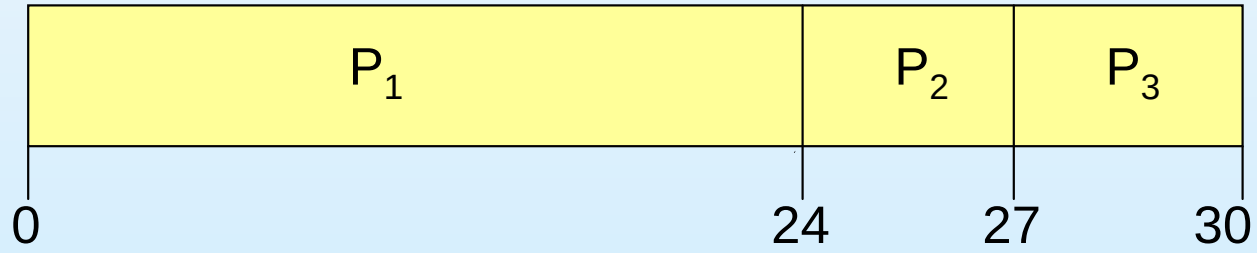
Real-time systems

- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

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 order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:



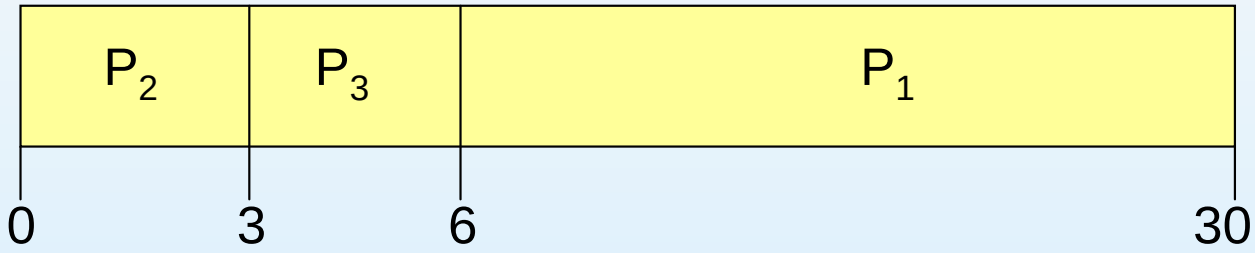
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

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
- **Much better than previous case**
  - **That's why we need scheduling algorithms!!!**
- *Convoy effect* short process behind long process

FCFS is nonpreemptive

- The FCFS scheduling algorithm is nonpreemptive.
  - Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
  - The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.
  - It would be disastrous to allow one process to keep the CPU for an extended period.

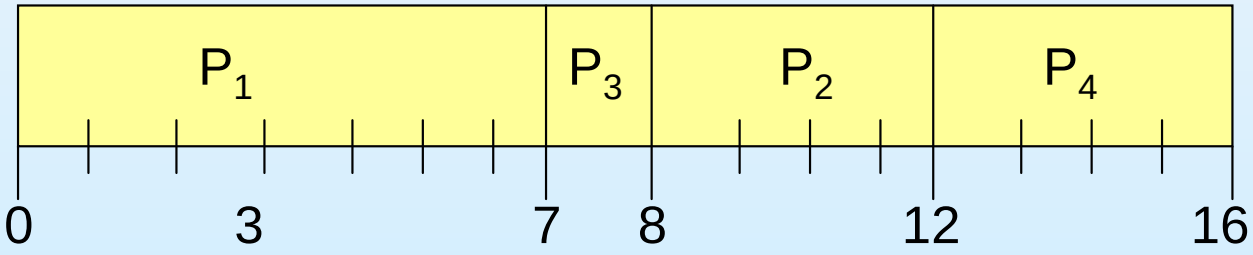
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the **Shortest-Remaining-Time-First (SRTF)**
- SJF is optimal – gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

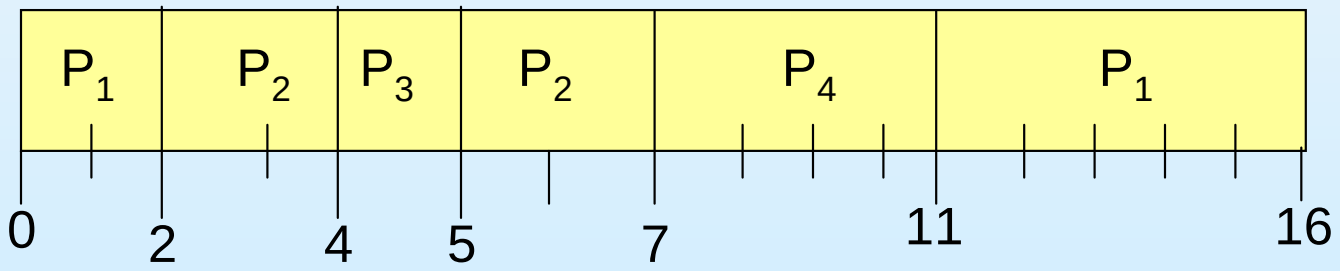


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



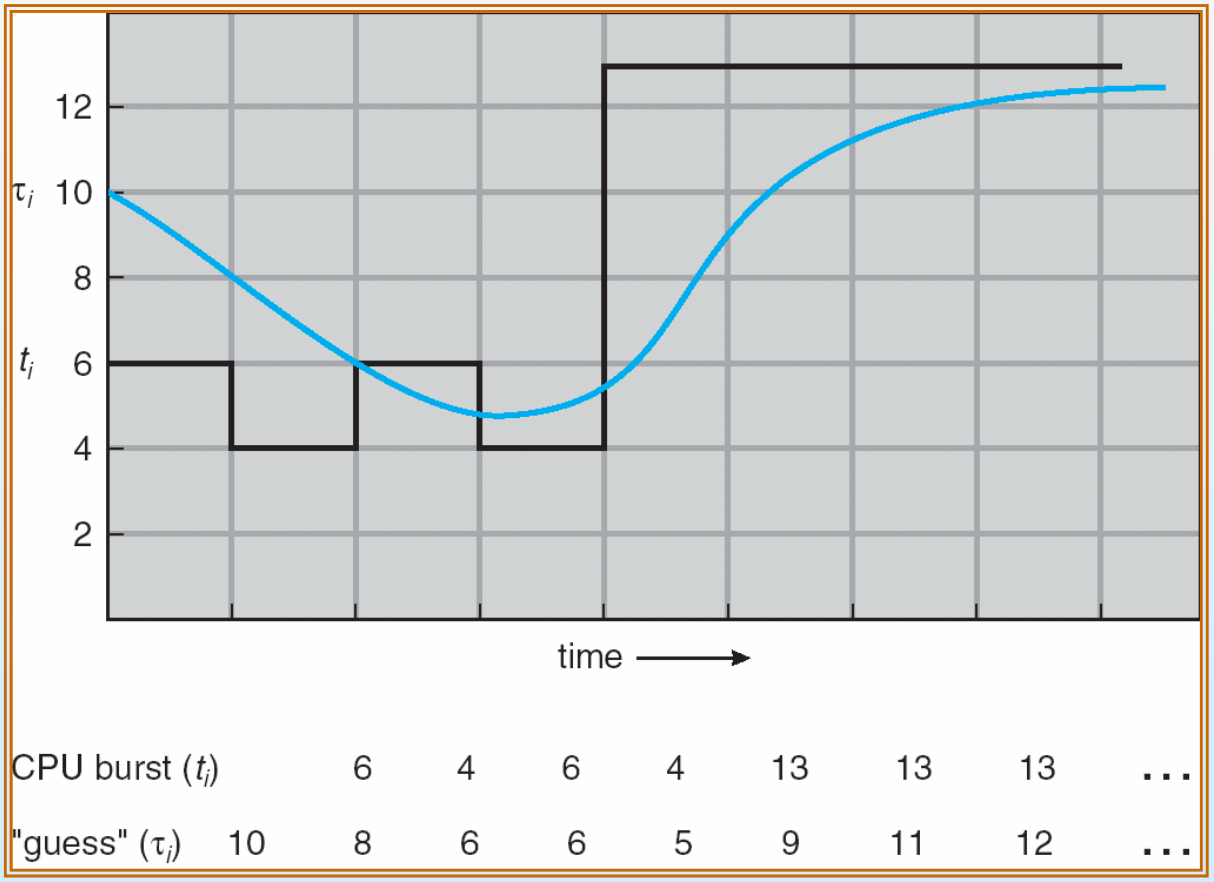
- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define : $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n$.

Prediction of the Length of the Next CPU Burst



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 a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- (it is told that when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run. ☺)
- Solution ≡ **Aging** – as time progresses increase the priority of the process

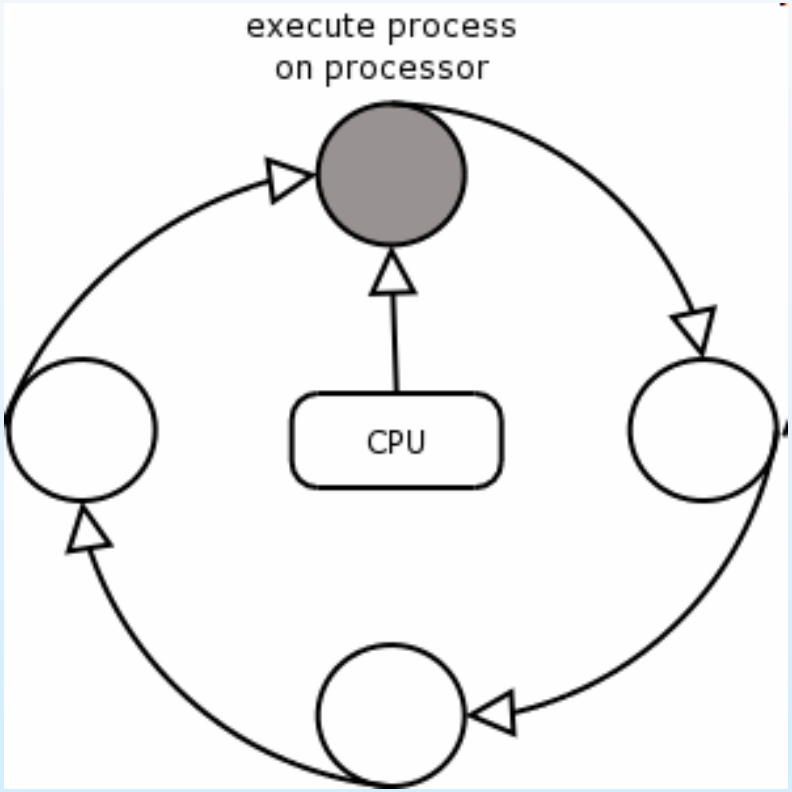
Aging

- Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.
  - For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.
  - Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.

Round Robin (RR)

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

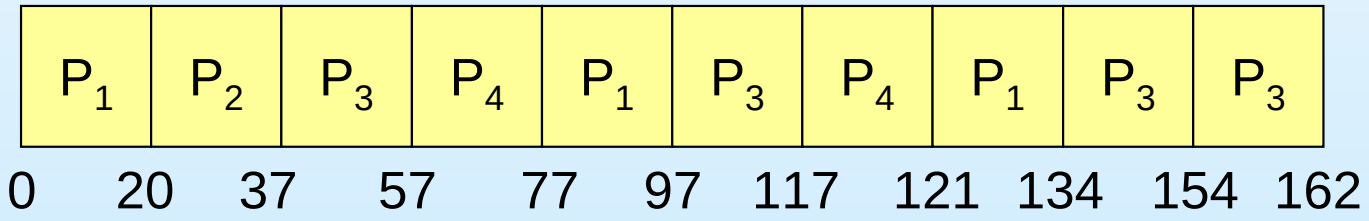
Round-Robin



Example of RR with Time Quantum = 20

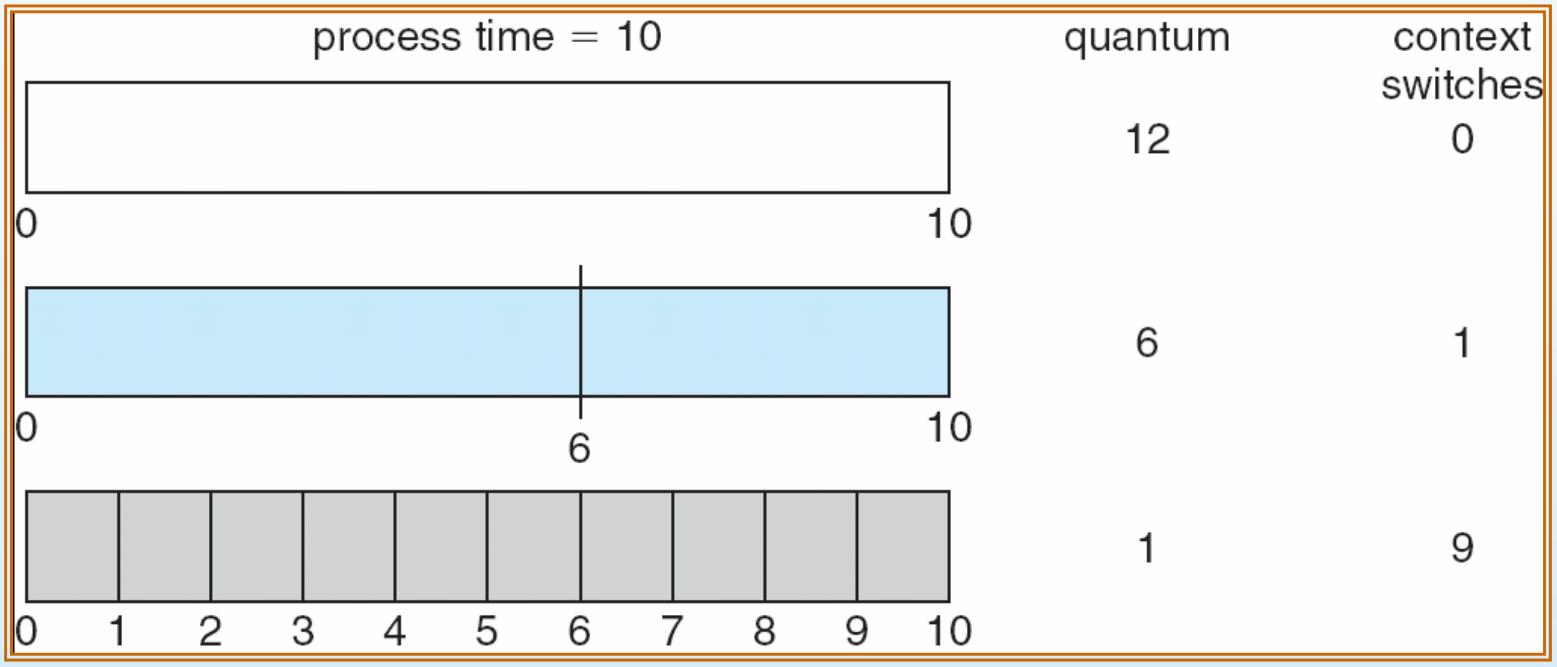
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

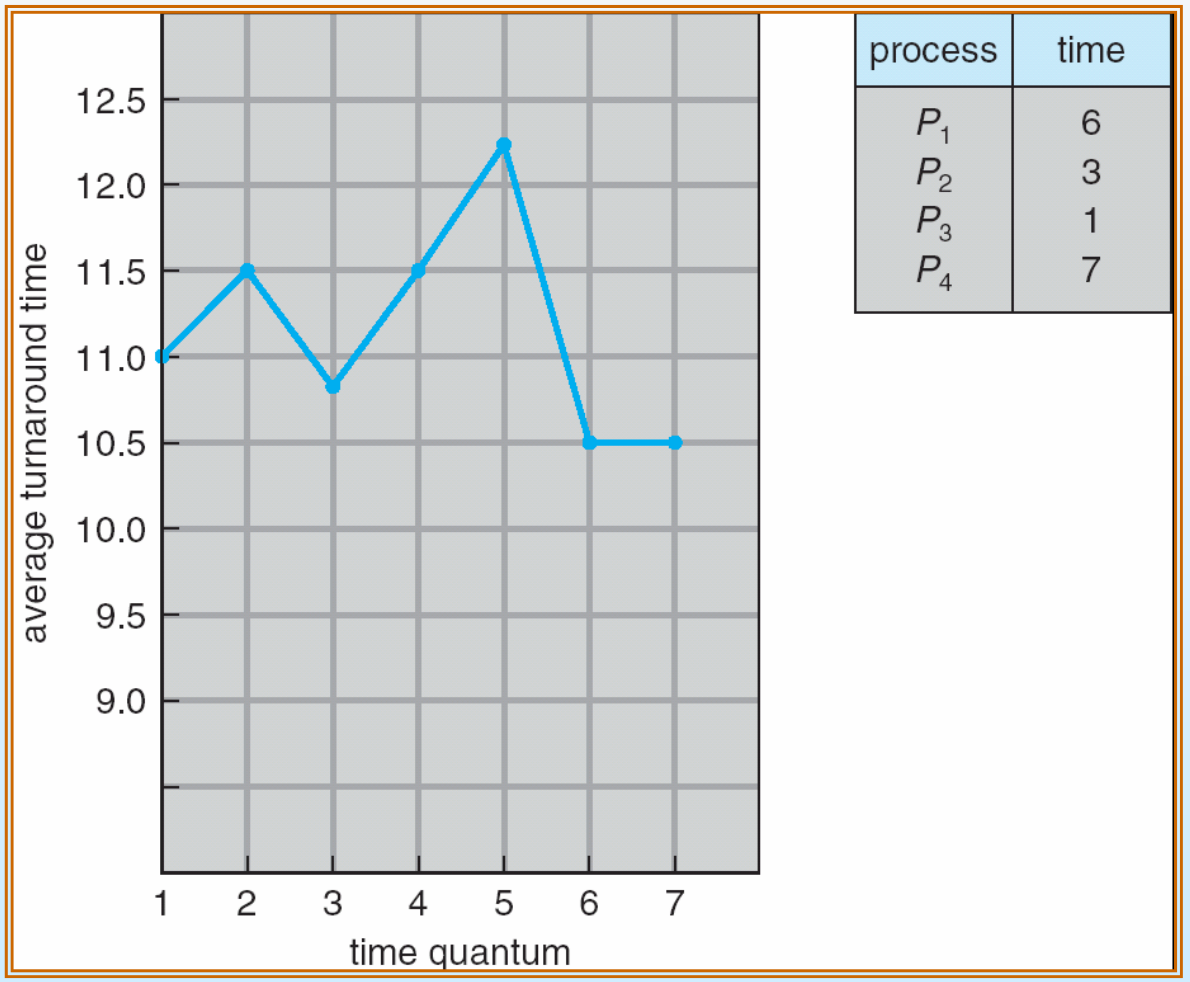


- Typically, higher average turnaround than SJF, but better *response*

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum

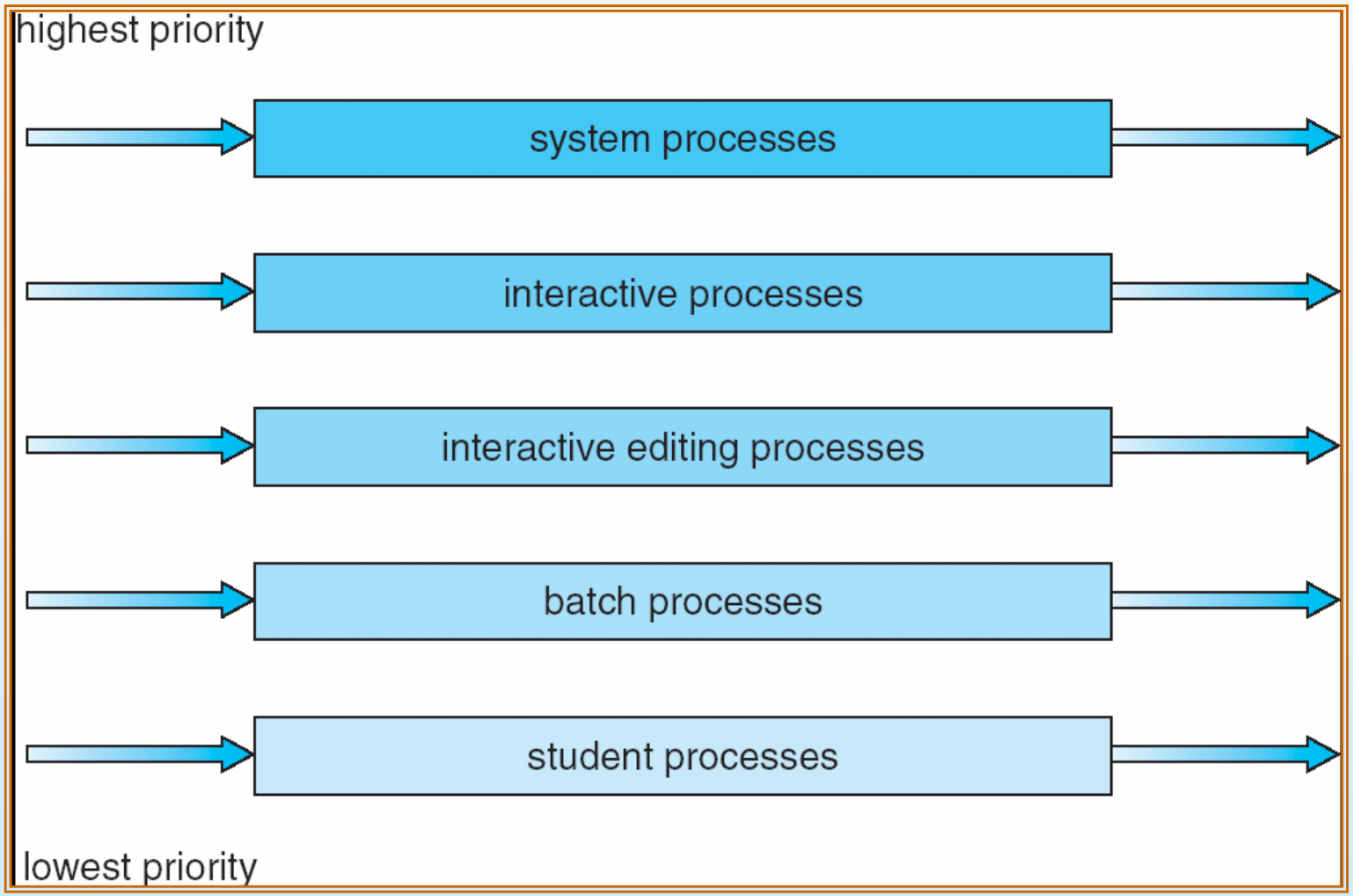


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

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

Multilevel Queue Scheduling



Multilevel Queue Scheduling

- Each queue has absolute priority over lower-priority queues.
  - No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty.
  - If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.
- Another possibility is to time-slice among the queues.
  - Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.
  - For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

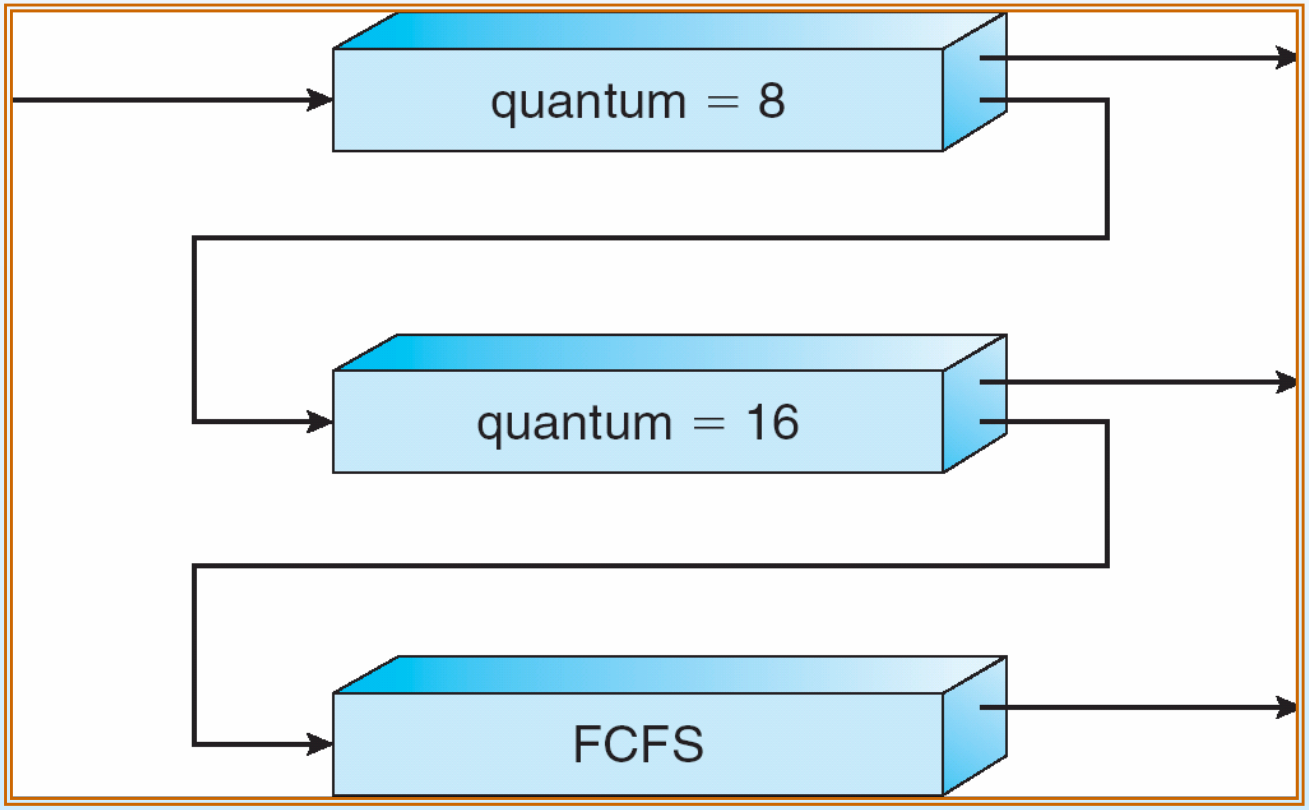
Multilevel Feedback Queue

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

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR 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.
- This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less.
  - Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst.
  - Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes.
  - Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

Multilevel Feedback Queues



Multiple-Processor Scheduling

- If multiple CPUs are available, load sharing becomes possible;
  - however the scheduling problem becomes correspondingly more complex.
  - just as with single processor CPU scheduling, there is no one best solution.
- We concentrate on systems in which the processors are identical – **homogeneous -** in terms of their functionality
  - we can use any available processor to run any process in the queue.

All schedulers comparison

Scheduling algorithm	CPU Utilization	Throughput	Turnaround time	Response time	Deadline handling	Starvation free
First In First Out	Low	Low	High	High	No	Yes
Shortest remaining time	Medium	High	Medium	Medium	No	No
Fixed priority pre-emptive scheduling	Medium	Low	High	High	Yes	No
Round-robin scheduling	High	Medium	Medium	Low	No	Yes

There is no universal “best” scheduling algorithm, and many operating systems use extended or combinations of the scheduling algorithms above.

OS schedulers

Operating System	Preemption	Algorithm
Windows 3.1x	None	Cooperative Scheduler
Windows 95,98,ME	Half	Only for 32 bit operations
Windows NT,XP,Vista	Yes	Multilevel Feedback Queue
Mac OS pre 9	None	Cooperative Scheduler
Mac OS X	Yes	Mach (kernel)

Linux pre 2.5	Yes	Multilevel Feedback Queue
Linux 2.5-2.6.23	Yes	O(1) scheduler
Linux post 2.6.23	Yes	Completely Fair Scheduler
Solaris	Yes	Multilevel feedback queue
NetBSD	Yes	Multilevel feedback queue
FreeBSD	Yes	Multilevel feedback queue

Asymmetric multiprocessing

- One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor - **the master server**.
- The other processors execute only user code
  - This asymmetric multiprocessing is simple because only **one processor** accesses the system data structures, reducing the need for data sharing.

Symmetric multiprocessing

- A second approach uses symmetric multiprocessing (SMP), where each processor is self-scheduling.
  - All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.
  - Regardless, scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.
  - If we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully: **We must ensure that two processors do not choose the same process and that processes are not lost from the queue.**
  - Virtually all modern operating systems support SMP, including Windows XP, Windows 2000, Solaris, Linux, and Mac as X.

Real-Time Scheduling

- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time
- *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones

Processor Affinity

- Consider what happens to cache memory when a process has been running on a specific processor:
  - The data most recently accessed by the process populates the cache for the processor; and as a result, successive memory accesses by the process are often satisfied in cache memory.
- Now consider what happens if the process migrates to another processor:
  - The contents of cache memory must be invalidated for the processor being migrated from, and the cache for the processor being migrated to must be re-populated.
  - Because of the high cost of invalidating and re-populating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor.
  - This is known as **processor affinity**, meaning that a process has an affinity for the processor on which it is currently running.

Load Balancing

- On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.
  - Otherwise, one or more processors may sit idle while other processors have high workloads along with lists of processes awaiting the CPU.
  - **Load balancing** attempts to keep the workload evenly distributed across all processors in an SMP system.
- It is important to note that load balancing is typically only necessary on systems where each processor has its own private queue of eligible processes to execute.
  - On systems with a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue.
  - It is also important to note, however, that in most contemporary operating systems supporting SMP, each processor does have a private queue of eligible processes.

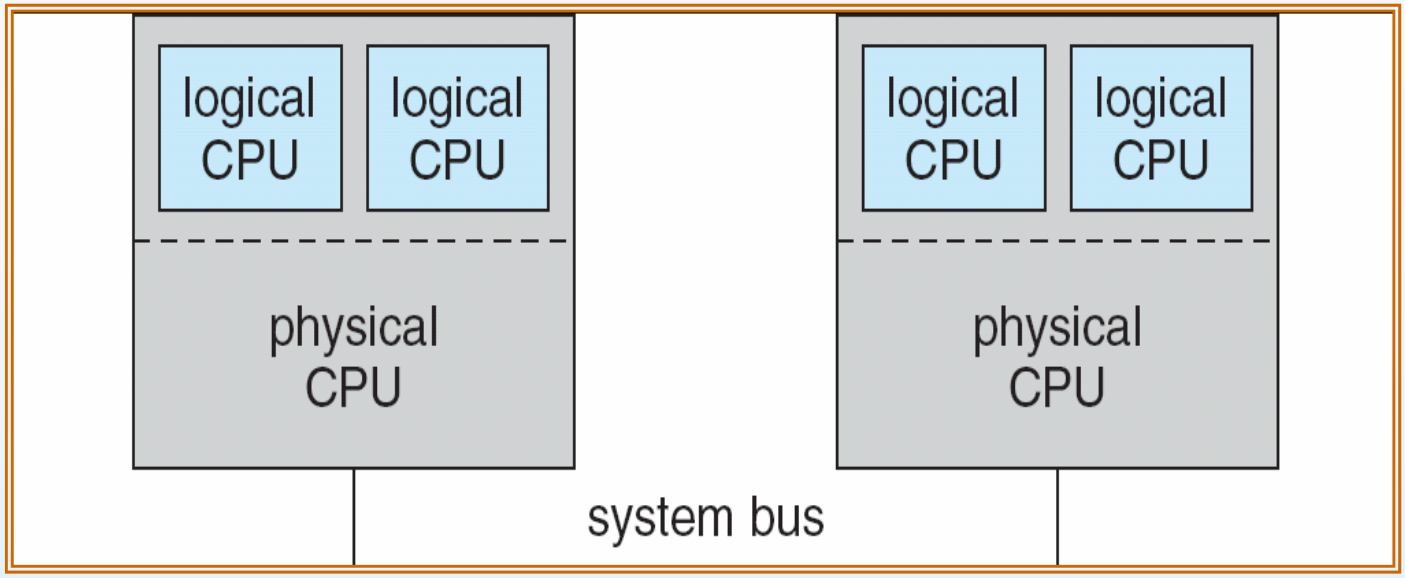
Push/Pull Migration

- There are two general approaches to load balancing:
  - With **push migration**, a specific task periodically checks the load on each processor and - if it finds an imbalance - evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.
  - **Pull migration** occurs when an idle processor pulls a waiting task from a busy processor.
  - Push and pull migration need not be **mutually exclusive** and are in fact often implemented in parallel on load-balancing systems.
- For example, the Linux scheduler and the ULE scheduler available for FreeBSD systems implement both techniques.
  - Linux runs its load-balancing algorithm every 200 milliseconds (push migration) or whenever the run queue for a processor is empty (pull migration).

Symmetric Multithreading

- SMP systems allow several threads to run concurrently by providing multiple physical processors. An alternative strategy is to provide multiple *logical* rather than *physical-processors.*
- Such a strategy is known as symmetric multithreading (or SMT); it has also been termed **hyperthreading technology** on **Intel** processors.
- The idea behind SMT is to create multiple logical processors on the same physical processor:
  - presenting a view of several logical processors to the operating system, even on a system with only a single physical processor.
  - each logical processor has its own architecture state, which includes general-purpose and machine-state registers.
  - each logical processor is responsible for its own interrupt handling, meaning that interrupts are delivered to- and handled by - logical processors rather than physical ones.
  - each logical processor shares the resources of its physical processor, such as cache memory and buses.

Symmetric Multithreading

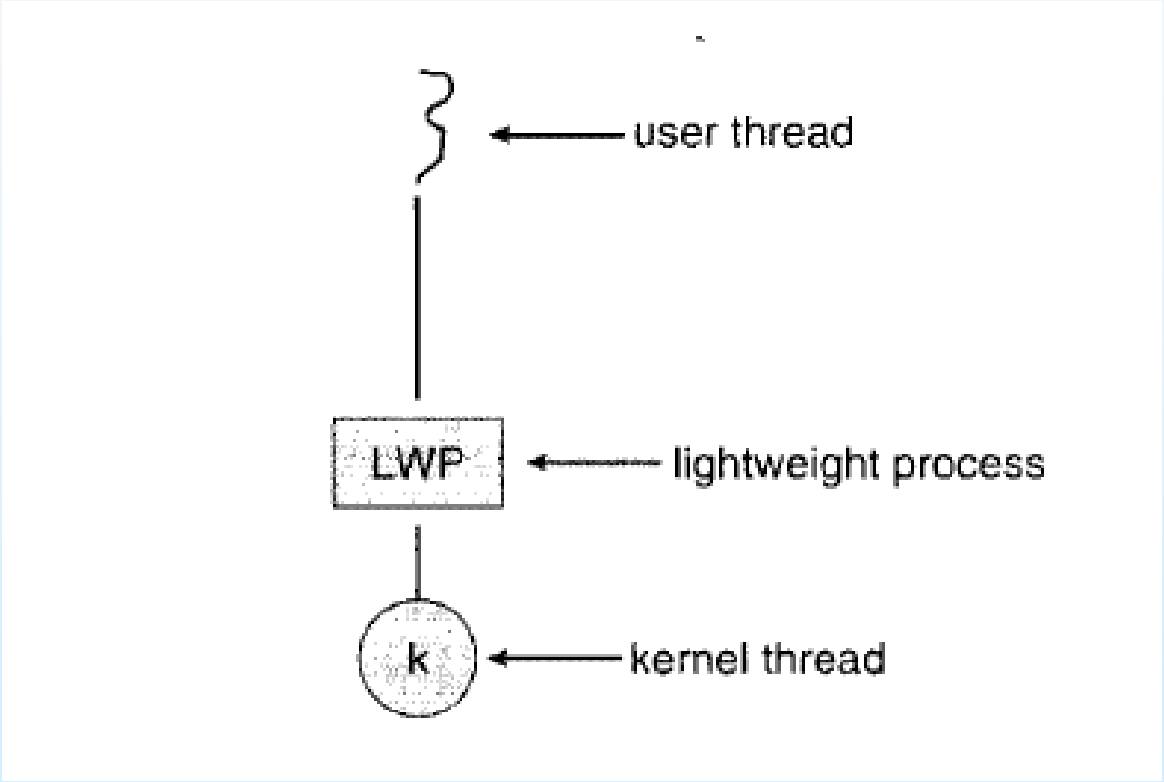


- It is important to recognize that SMT is a feature provided in hardware, not software.
  - That is, hardware must provide the representation of the architecture state for each logical processor, as well as interrupt handling.
- Operating systems need not necessarily be designed differently if they are to run on an SMT system;
  - however, certain performance gains are possible if the operating system is aware that it is running on such a system.

Thread Scheduling

- On operating systems that support them, it is kernel-level threads-not processes-that are being scheduled by the operating system.
- User-level threads are managed by a thread library, and the kernel is unaware of them.
- To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).

Recollect from Chap. 4



Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP
- Global Scheduling – How the kernel decides which kernel thread to run next

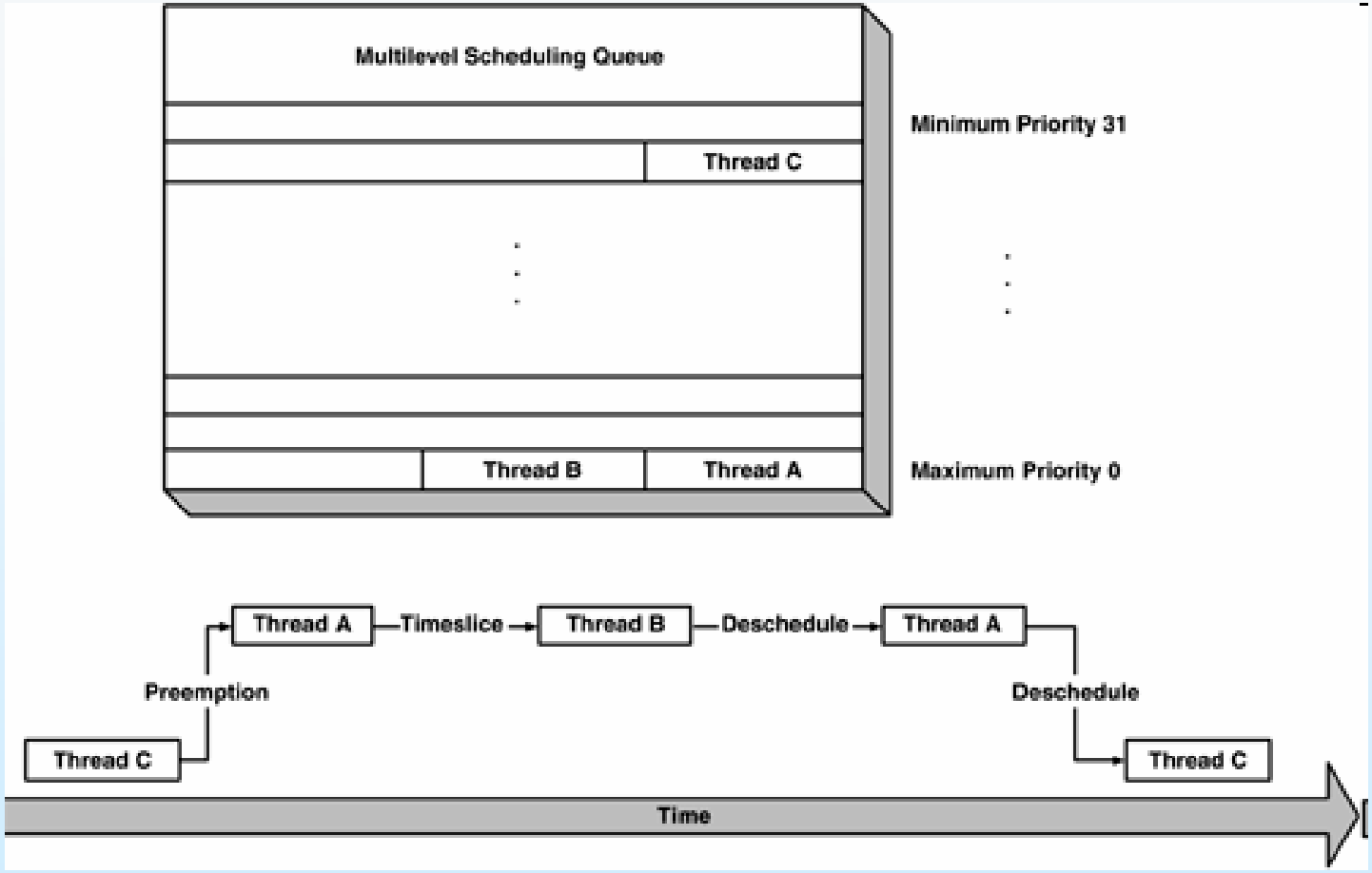
Thread Scheduling

- One distinction between user-level and kernel-level threads lies in how they are scheduled.
- On systems implementing the many-to-one and many-to-many models:
  - the thread library schedules user-level threads to run on an available LWP, a scheme known as **process-contention scope** (**PCS**), since competition for the CPU takes place among threads belonging to the same process.
- When we say the thread library *schedules* user threads onto available LWPs, we do not mean that the thread is actually running on a CPU
  - this would require the operating system to schedule the kernel thread onto a physical CPU.
  - to decide which kernel thread to schedule onto a CPU, the kernel uses **system-contention scope** (**SCS**).
  - competition for the CPU with **SCS** scheduling takes place among all threads in the system.
- Systems using the one-to-one model (such as Windows XP, Solaris 9, and Linux) schedule threads using only SCS.

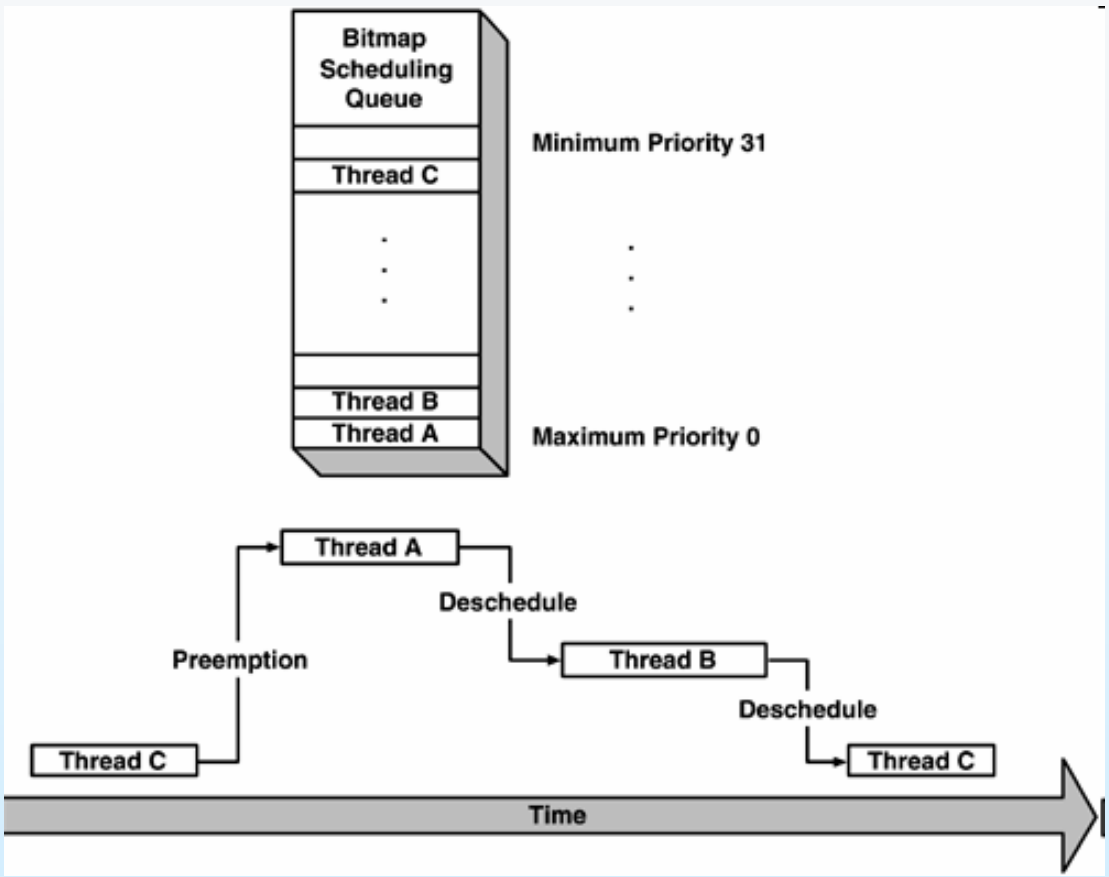
Thread Scheduling

- Typically, PCS is done according to priority - the scheduler selects the runnable thread with the highest priority to run.
- User-level thread priorities are set by the **programmer** and are not adjusted by the thread library,
  - Although some thread libraries may allow the programmer to change the priority of a thread.
- It is important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread;
  - however, there is no guarantee of time slicing among threads of equal priority.

Multilevel Thread Scheduling



Bitmap scheduler



The bitmap scheduler allows the execution of threads at multiple priority levels; however, only a single thread can exist at each priority level.

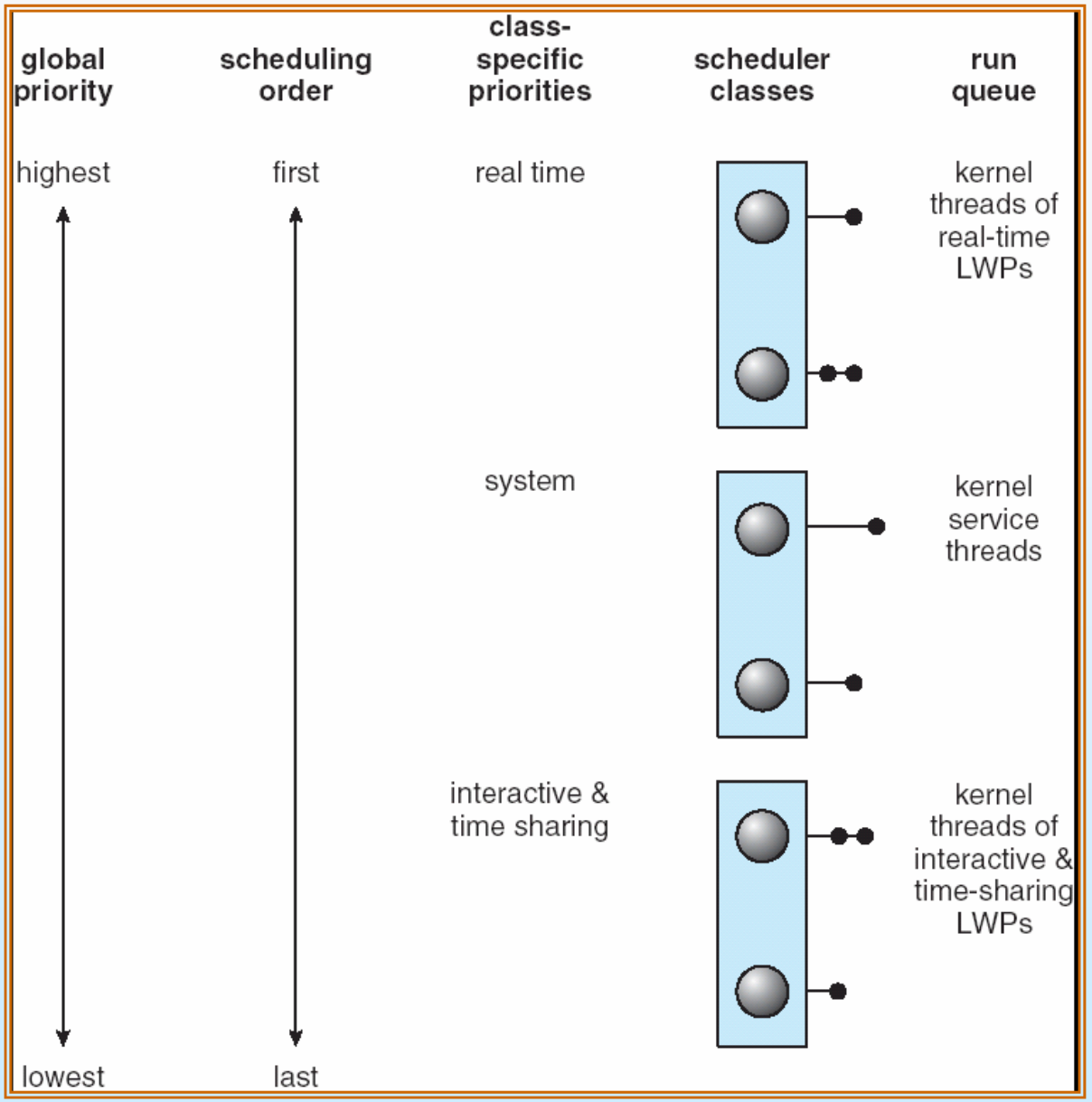
Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Solaris 2 Scheduling

- Solaris uses priority-based thread scheduling. It has defined four classes of scheduling, which are, in order of priority:
  1. Real time
  2. System
  3. Time sharing
  4. Interactive
- Within each class there are different priorities and different scheduling algorithms.

Solaris 2 Scheduling



Solaris 2 Scheduling

- The default scheduling class for a process is time sharing.
- The scheduling policy for time sharing dynamically alters priorities and assigns time slices of different lengths using a **multilevel feedback queue**.
- By default there is an inverse relationship between priorities and time slices:
  - The higher the priority, the smaller the time slice; and the lower the priority, the larger the time slice.
- Interactive processes typically have a higher priority.
- CPU-bound processes, a lower priority.
- This scheduling policy gives good response time for interactive processes and good throughput for CPU-bound processes.
- The interactive class uses the same scheduling policy as the time-sharing class, but it gives windowing applications a higher priority for better performance.

Solaris Dispatch Table

- Dispatch table for interactive and time-sharing threads. (60 priority levels)
- **Priority**. The class-dependent priority for the time-sharing and interactive classes. A higher number indicates a higher priority.
- **Time quantum.** The time quantum for the associated priority. This illustrates the inverse relationship between priorities and time quanta:
  - The lowest priority (priority 0) has the highest time quantum (200 milliseconds), and the highest priority (priority 59) has the lowest time quantum (20 milliseconds) .

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

- **Time quantum expired.** The new priority of a thread that has used its entire time quantum without blocking. Such threads are considered CPU-intensive. As shown in the table, these threads have their priorities lowered.
- **Return from sleep.** The priority of a thread that is returning from sleeping (such as waiting for I/O). As the table illustrates, when I/O is available for a waiting thread, its priority is boosted to between 50 and 59, thus supporting the scheduling policy of providing good response time for interactive processes.

Windows XP scheduling

- Windows XP schedules threads using a priority-based, preemptive scheduling algorithm.
- The Windows XP scheduler ensures that the highest-priority thread will always run.
- The portion of the Windows XP kernel that handles scheduling is called the *dispatcher.*
- A thread selected to run by the dispatcher will run until it is preempted by a higher-priority thread, until it terminates, until its time quantum ends, or until it calls a blocking system call, such as for I/O.
- If a higher-priority real-time thread becomes ready while a lower-priority thread is running, the lower-priority thread will be preempted. This preemption gives a real-time thread preferential access to the CPU when the thread needs such access.

Windows XP Scheduling

- The dispatcher uses a 32-level priority scheme to determine the order of thread execution.
- Priorities are divided into two classes:
  - **variable class** contains threads having priorities from 1 to 15,
  - **real-time class** contains threads with priorities ranging from 16 to 31.
- There is also a thread running at priority 0 that is used for memory management.
- The dispatcher uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run.
- If no ready thread is found, the dispatcher will execute a special thread called the idle thread.

Windows XP 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

Relatioship between Win32 and priorities

- There is a relationship between the numeric priorities of the Windows XP kernel and the Win32 API.
- The Win32 API identifies several priority classes to which a process can belong. These include:
  - REALTIME_PRIORITY _CLASS
  - HIGH_PRIORITY_CLASS
  - ABOVE_NORMALPRIORITY _CLASS
  - NORMALPRIORITY _CLASS
  - BELOW _NORMALPRIORITY _CLASS
  - IDLE-PRIORITY _CLASS

Linux Scheduling

- Two algorithms (Linux does not distinguish between processes and threads, we talk about **tasks**):
  - **time-sharing**
  - **real-time**
- **Time-sharing**
  - Prioritized credit-based – process with most credits is scheduled next
  - Credit subtracted when timer interrupt occurs
  - When credit = 0, another process chosen
  - When all processes have credit = 0, recrediting occurs
    - Based on factors including priority and history
- **Real-time**
  - Soft real-time
  - Posix.1b compliant – two classes
    - FCFS and RR
    - Highest priority process always runs first

Linux Scheduling: UNIX algorithm

- Prior to version 2.5, the Linux kernel ran a variation of the traditional UNIX scheduling algorithm.
- Two problems with the traditional UNIX scheduler are that it does not provide adequate support for SMP systems and that it does not scale well as the number of tasks on the system grows.
- With version 2.5, the scheduler was overhauled, and the kernel now provides a scheduling algorithm that runs in constant time - known as O(1) - regardless of the number of tasks on the system.
- The new scheduler also provides increased support for SMP, including processor affinity and load balancing, as well as providing fairness and support for interactive tasks.

Linux Scheduling 1

- The Linux scheduler is a preemptive, priority-based algorithm with two separate priority ranges:
  - a **real-time** range from a to 99
  - **nice value** ranging from 100 to 140.
- These two ranges map into a global priority scheme whereby numerically lower values indicate higher priorities.
- **Unlike schedulers for many other systems, including Solaris (5.6.1) and Windows XP (5.6.2), Linux assigns higher-priority tasks longer time quanta and lower-priority tasks shorter time quanta.**

The Relationship Between Priorities and Time-slice length

numeric priority	relative priority		time quantum
0	highest	real-time tasks	200 ms
•			
•			
•			
99			
100		other tasks	
•			
•			
•			
140	lowest		10 ms

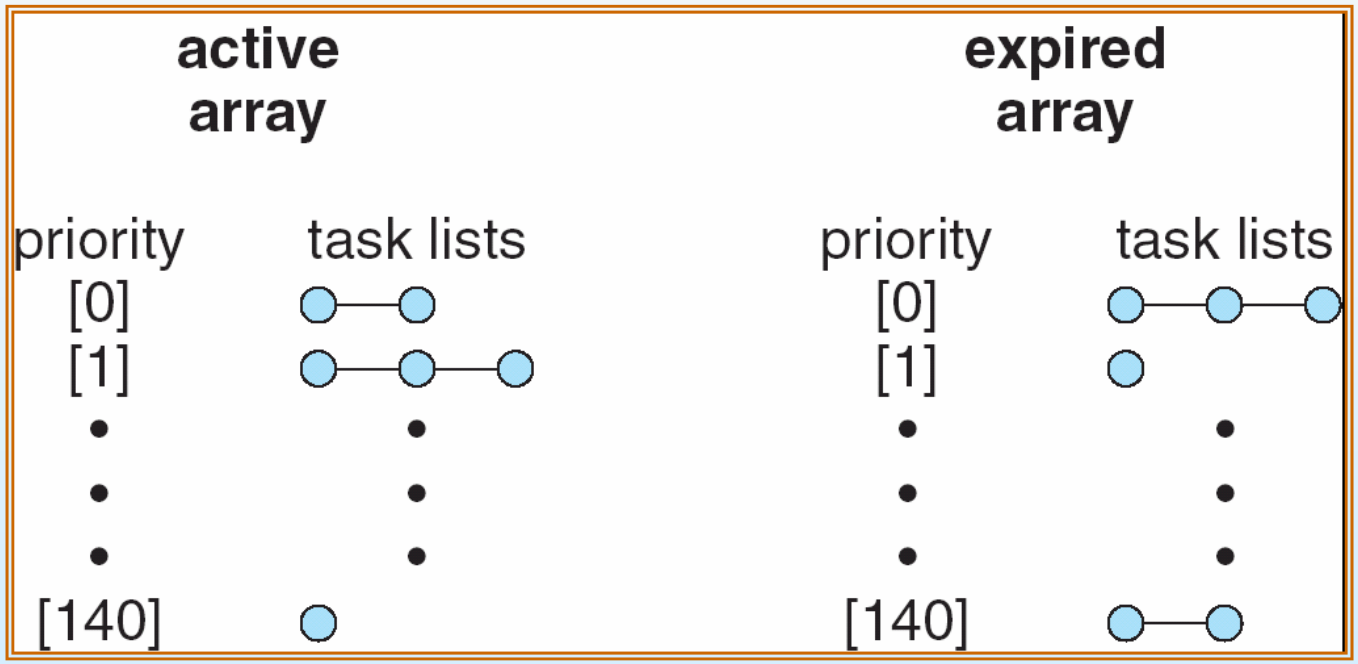
Linux Scheduling 2

- A runnable task is considered eligible for execution on the CPU as long as it has time remaining in its time slice.
- When a task has exhausted its time slice, it is considered expired and is not eligible for execution again until all other tasks have also exhausted their time quanta.
- The kernel maintains a list of all runnable tasks in a **runqueue** data structure. Because of its support for SMP, each processor maintains its own **runqueue** and schedules itself independently.
- Each runqueue contains two priority arrays - **active** and **expired**.
- The active array contains all tasks with time remaining in their time slices, and the expired array contains all expired tasks. Each of these priority arrays contains a list of tasks indexed according to priority.

Linux Scheduling 3

- The scheduler chooses the task with the highest priority from the active array for execution on the CPU.
- On multiprocessor machines, this means that each processor is scheduling the highest-priority task from its own **runqueue** structure.
- When all tasks have exhausted their time slices (that is, the active array is empty), the two priority arrays are exchanged; the expired array becomes the active array, and vice versa.

List of Tasks Indexed According to Prorities



Selecting Scheduling Algorithms

- How do we select a CPU scheduling algorithm for a particular system?
- Since there are many scheduling algorithms, each with its own parameters, selecting an algorithm can be difficult.
- The first problem is **defining the criteria** to be used in selecting an algorithm.
  - criteria are often defined in terms of CPU utilization, response time, or throughput.
  - **For your project: to select an algorithm, we must first define the relative importance of these measures.**
  - Our criteria may include several measures, such as:
    - Maximizing CPU utilization under the constraint that the maximum response time is 1 second
    - Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time
- Once the selection criteria have been defined, we want to evaluate the algorithms under consideration.

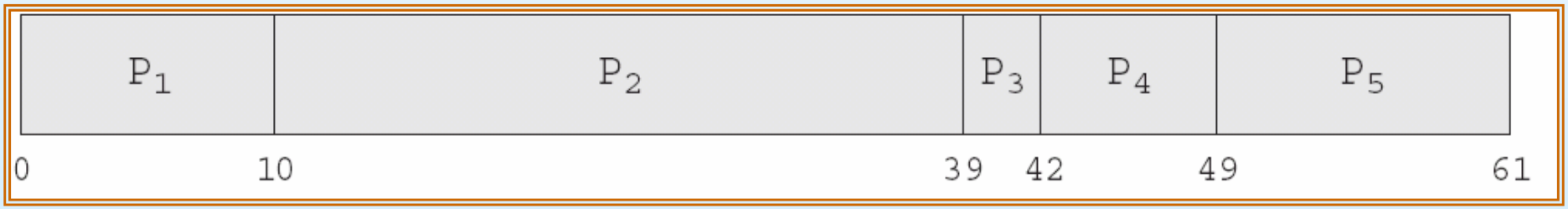
Deterministic Modeling

- One major class of evaluation methods is **analytic evaluation**.
  - It uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload.
- One type of analytic evaluation is **deterministic modeling**. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload. For example, assume that we have the workload shown below:

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

FCFS

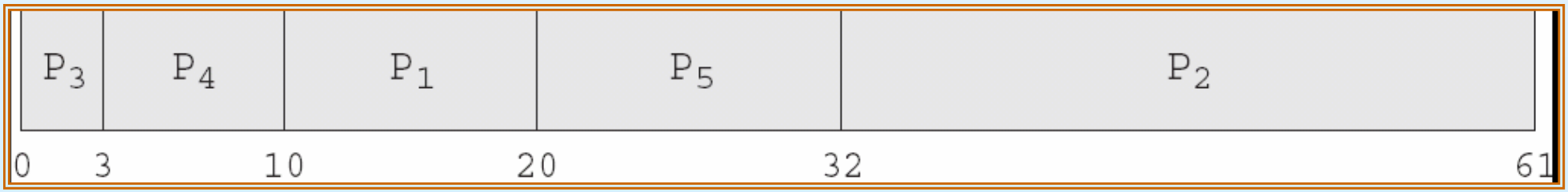
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12



The waiting time is 0 milliseconds for process PI, 10 milliseconds for process *P2,* 39 milliseconds for process *P3,* 42 milliseconds for process *P4,* and 49 milliseconds for process Ps. Thus, the average waiting time is (0 + 10 + 39 + 42 + 49)/5 = 28 milliseconds.

Nonpreemptive SJF scheduling

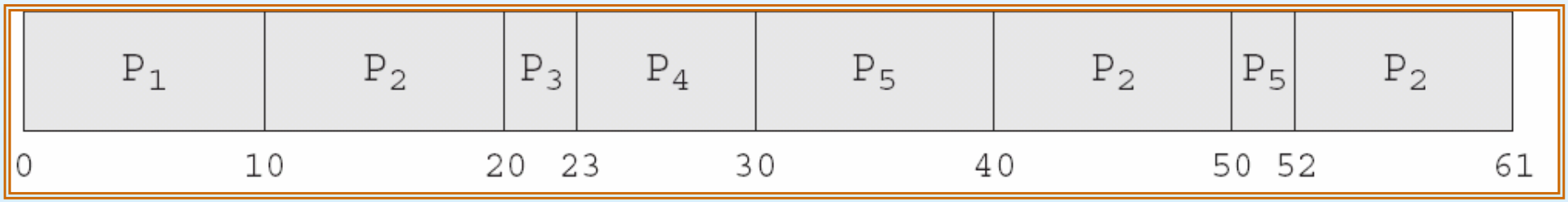
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12



The waiting time is 10 milliseconds for process *PI,* 32 milliseconds for process *P2,* 0 milliseconds for process *P3,* 3 milliseconds for process *P4* , and 20 milliseconds for process Ps. Thus, the average waiting time is (10 + 32 + 0 + 3 + 20)/5 = 13 milliseconds.

Round-Robin

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



The waiting time is 0 milliseconds for process *PI,* 32 milliseconds for process *P2,* 20 milliseconds for process *P3* , 23 milliseconds for process *P4* , and 40 milliseconds for process Ps. Thus, the average waiting time is (0 + 32 + 20 + 23 + 40)/5 = 23 milliseconds.

We see that, *ill this case,* the average waiting time obtained with the SJF policy is less than half that obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.

Deterministic modeling

- Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms.
- However, it requires exact numbers for input, and its answers apply only to those cases.
- The main uses of deterministic modeling are in describing scheduling algorithms and providing examples.
- In cases where we are running the same program over and over again and can measure the program's processing requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm.
- Furthermore, over a set of examples, deterministic modeling may indicate trends that can then be analyzed and proved separately.
- For example, it can be shown that, for the environment described (all processes and their times available at time 0), the SJF policy will always result in the minimum waiting time.

Queueing Models

- On many systems, the processes that are run vary from day to day, so there is no static set of processes (or times) to use for deterministic modeling.
- What can be determined, however, is the distribution of CPU and I/O bursts.
  - These distributions can be measured and then approximated or simply estimated.
  - The result is a mathematical formula describing the probability of a particular CPU burst. Commonly, this distribution is exponential and is described by its mean.
- Similarly, we can describe the distribution of times when processes arrive in the system (the arrival-time distribution).
- From these two distributions, it is possible to compute the average throughput, utilization, waiting time, and so on for most algorithms.

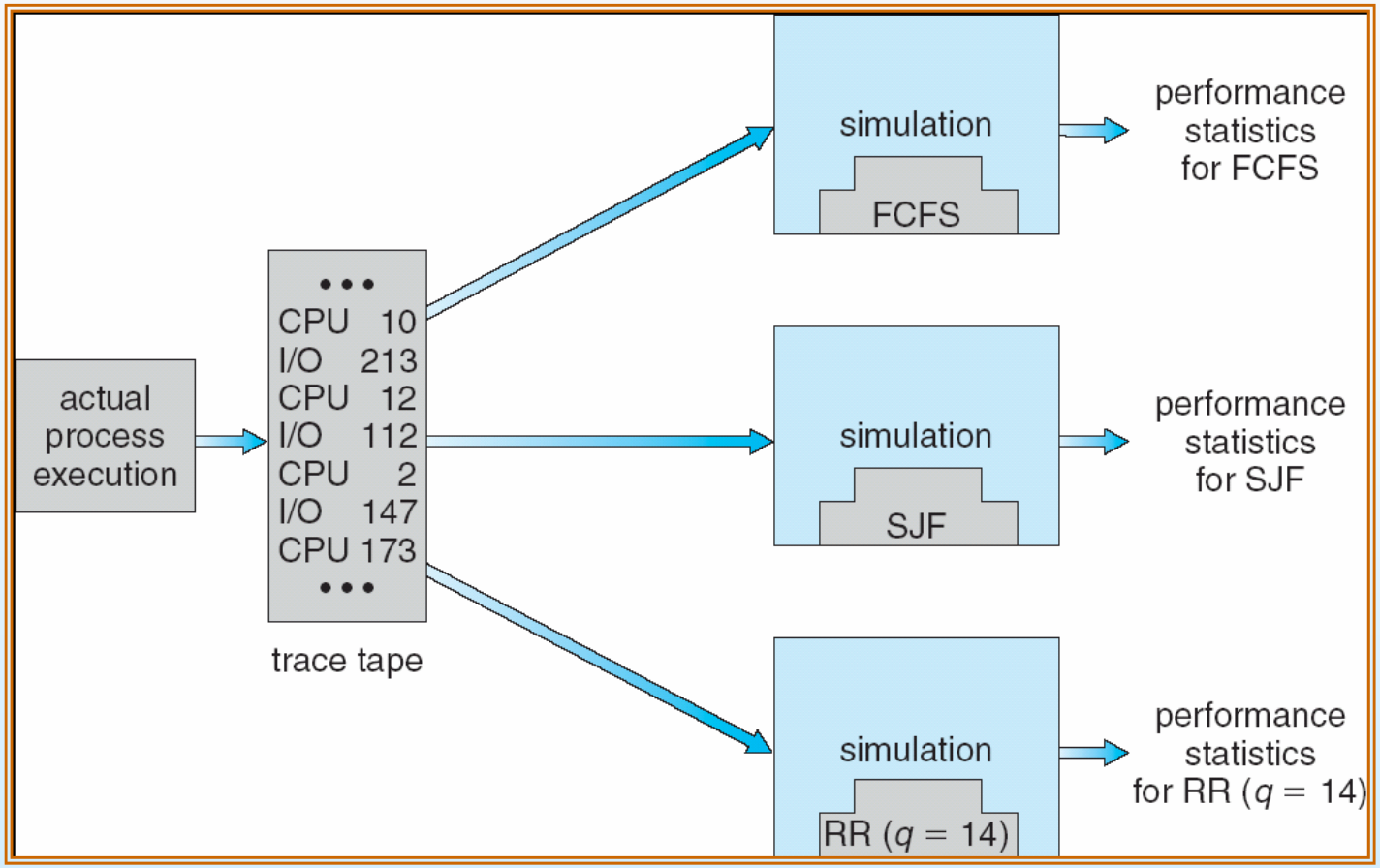
Simulation

- To get a more accurate evaluation of scheduling algorithms, we can use simulations.
  - Running simulations involves programming a model of the computer system.
  - Software data structures represent the major components of the 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.
  - As the simulation executes, statistics that indicate algorithm performance are gathered and printed.

Simulation

- The data to drive the simulation can be generated in several ways.
- The most common method uses a random-number generator, which is programmed to generate processes, CPU burst times, arrivals, departures, and so on, according to probability distributions.
- The frequency distribution indicates only how many instances of each event occur; it does not indicate anything about the order of their occurrence. To correct this problem, we can use trace tapes.
- We create a trace tape by monitoring the real system and recording the sequence of actual events. We then use this sequence to drive the simulation.
- **Trace tapes** provide an excellent way to compare two algorithms on exactly the same set of real inputs. This method can produce accurate results for its inputs.

Simulation


actual process execution
CPU 10
I/O 213
CPU 12
I/O 112
CPU 2
I/O 147
CPU 173
trace tape
simulation
FCFS
performance statistics for FCFS
simulation
SJF
performance statistics for SJF
simulation
RR (q = 14)
performance statistics for RR (q = 14)

End of Chapter 5

From PDP-x to UNIX: the inventors

Dennis Richie and Ken Thompson working on a PDP-11. (1971-1973)

Ken Thompson (left) and Dennis Ritchie receive the
National Medal of Technology
from President Clinton. (1999)

Project

- Project is on scheduling
- You will be given a problem/scenario and you are required to propose the best solution and implement it.
- Note: you are going to implement a **simulation** of the scheduling algorithm
- The project (100 points) consists of:
  - Analysis 40%
  - Design 40%
  - Implementation 20%
- The project starts on November 12th 2009
  - Analysis due December 4th
  - Design due December 21st
  - Implementation due January 26th 2010
- The project will be performed in groups
- Final project will be discussed with instructor on 27, 28, 29 of January'10

Project proposal

- Project proposal is due within November 16th together with the group composition.
- For every day of delay in submitting each part of the project there is a penalty of **one point**.
- Any tentative of copying the others' work will automatically lead to project rejection!
- Recommendations
  - Start working now!
  - Even though you might be ahead with the analysis and/or design, go ahead! Implementation takes time!
  - Do not try to rely only on the other members of the group and do nothing for yourself!
  - Do not try to copy from other groups!

GOOD LUCK and HAVE FUN!

Project Analysis

- What will you be given?
  - A description of a real-world scenario requiring scheduling algorithms to be implemented in a system.
- Tasks for analysis
  - Read carefully the scenario
  - Identify the problem;
    - describe the problem and related issues
  - Identify a solution;
    - describe the solution by motivating the choices through analyses of the alternatives.
    - describe the solution strategy and how you are going to implement it.
- Not more than **5 pages**!

What-if

- What if analysis is not appropriate?
  - You will get less points but…
  - You will also get advice on how to proceed with a better solution
  - This will lead you to a better design and hopefully to a better implementation process.

Project Design

- Tasks for design
  - Map the solution identified during analysis in a design schema
    - Design the general architecture of the solution by identifying the major components
    - Identify and motivate a design methodology (example Object-Oriented)
  - Identify the proper data structures to be used
    - Design each of the components
    - Design the algorithms
  - Identify the proper technological framework
    - Identify a programming language to implement the solution.
    - Identify a representation formalism to describe the project design.
- Not more than **20 pages**!

What-if

- What if design is not appropriate?
  - You will get less points but…
  - You will also get advice on how to proceed with a better solution
  - This will lead you to a better implementation process.

Implementation

- Project Implementation
  - Implement the code
  - Simulate the scheduling algorithms by running them with different parameters
  - Record the tests with your system
  - Document your system
    - Not more than **10 pages**!

What-if

- What if implementation is not appropriate?
  - The program will not execute ☺
  - You may get less points if your implementation is far from the **executable** state. ☹
  - If most of your code is right but you have not been able to complete the code, your work will be awarded! ☺

Scenario 1

- A financial company runs large servers for transactions. It wants to optimize the waiting time for each transaction. On the other side, since there are many transactions coming at the same time from the Internet, it has to complete a transaction as soon as possible once it starts in order for it not to get lost over the networks due to large waiting times.
- Among the transactions there are some of them with amounts more than 500.000 Euro. These have to be treated with priority and must be completely executed before going on with other transactions.
- The company servers are also responsible for cashiers operations which need fast responses. Each operation is handled through a process and the cashier needs a response as soon as possible.

Design and implement scheduling algorithms that will guarantee to the base to optimally perform all the above tasks.

Scenario 2

- A military base controls some remote devices whose feedback and reaction time is critical. The devices are managed through some software programs. Each program receives a signal from the device, elaborates the signal and must reply within a certain time.
- There are also some local devices which are handled by some other programs. These local devices have a response time larger than the remote ones. Once activated a signal from a local device it must be dealt with completely until the end of the procedure
- The military base also runs frequent periodic scans of the security system through a program which checks if there have been intrusions in the system.
- The military base has also a software program that in case of immediate emergency activates the general alarm and closes all the gateways.

Design and implement scheduling algorithms that will guarantee to the base to optimally perform all the above tasks.

Scenario 3

A nuclear plant has many sensors that continuously (run-time) collect data which are then stored in a database. Sensors are managed through programs which are always running and are considered to be critical.

Data are analyzed frequently through complex mathematical programs which perform hard computations. If a certain result is obtained, the programs become of critical importance by launching the security procedure. Since data analysis is critical it should be minimized the average computation time for all the programs.

Once the security program is launched it cannot be suspended. It must complete the whole procedure.

If more than one security programs are launched, the CPU should schedule them to complete in the minimum time possible.

- **Design and implement scheduling algorithms that will guarantee to the base to optimally perform all the above tasks.**

Scenario 4

- An astronomic laboratory observes remote objects and each observation is handled trough a software program. During the observation it is possible that data may get lost due to the movement of the object or any other external factors. Thus it is important to save data as soon the object is observed.
- The problem that arises is that for any object observed there is a new process running and it must be selected among these who is going to get the CPU. The objects are divided into fast and slow in terms of their movement speed. For slow objects it is important to complete their observation data storage until the end once the object starts being observed.
- For fast objects it is essential to save as much data as possible about each object.

Design and implement scheduling algorithms that will guarantee to the base to optimally perform all the above tasks.