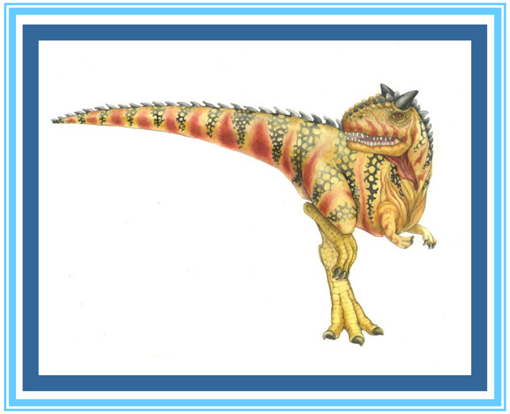
Chapter 5: CPU Scheduling



Chapter 6: CPU Scheduling

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

Process States

- As a **process executes**, it changes **state**.
- The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:
    1. **New**. The process is being created.
    2. **Running**. Instructions are being executed.
    3. **Waiting**. The process is waiting for some event to occur (such as an I/O
    4. completion or reception of a signal).
    5. **Ready**. The process is waiting to be assigned to a processor
- It is important to realize that only **one process can be *running* on any single processor core** at any instant.
- Many processes may be *ready* and *waiting,* however. The state diagram corresponding to these states is presented in **Figure 3.2**.

Process States

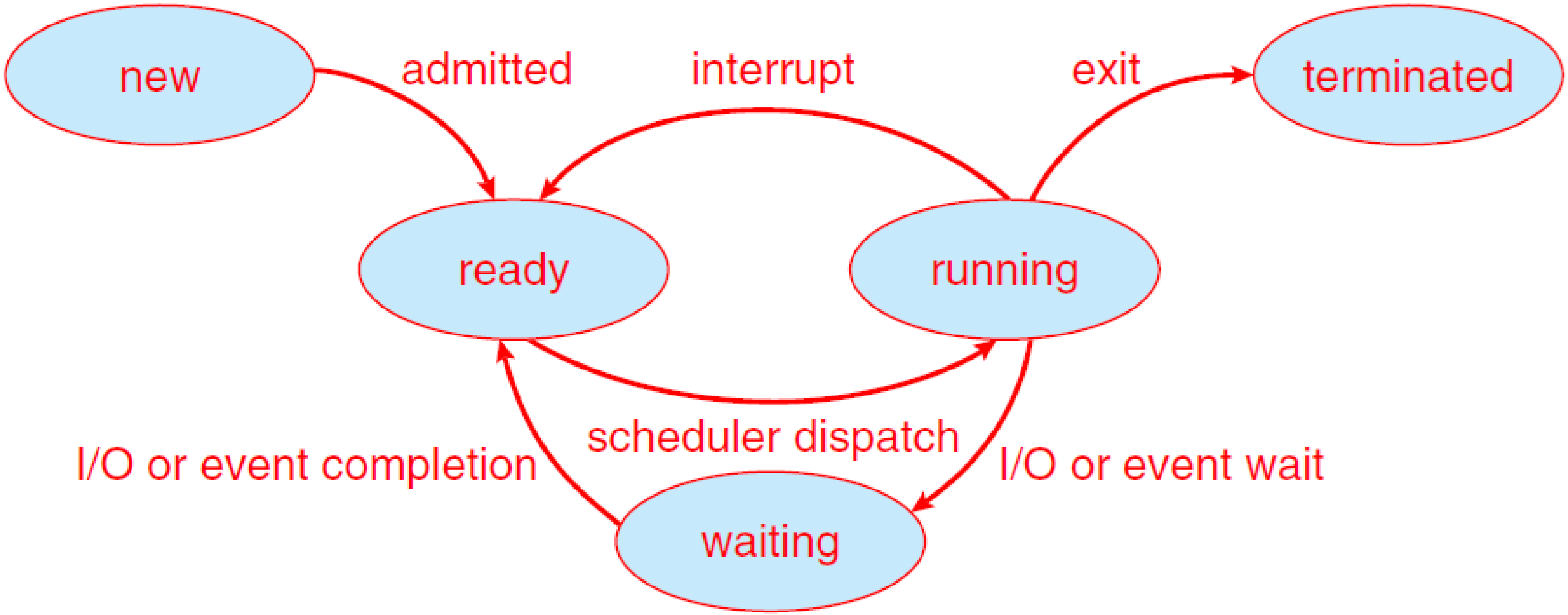


Figure 3.2 Diagram of process state.

Objectives

- **CPU scheduling is the basis of multi-programmed operating systems**
  - By switching the **CPU** among **processes**, the OS can make the computer more productive.
- This topic describes various CPU-scheduling algorithms used by various multiprogrammed operating systems.
  - To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
  - To examine the scheduling algorithms of several operating systems

Basic Concepts

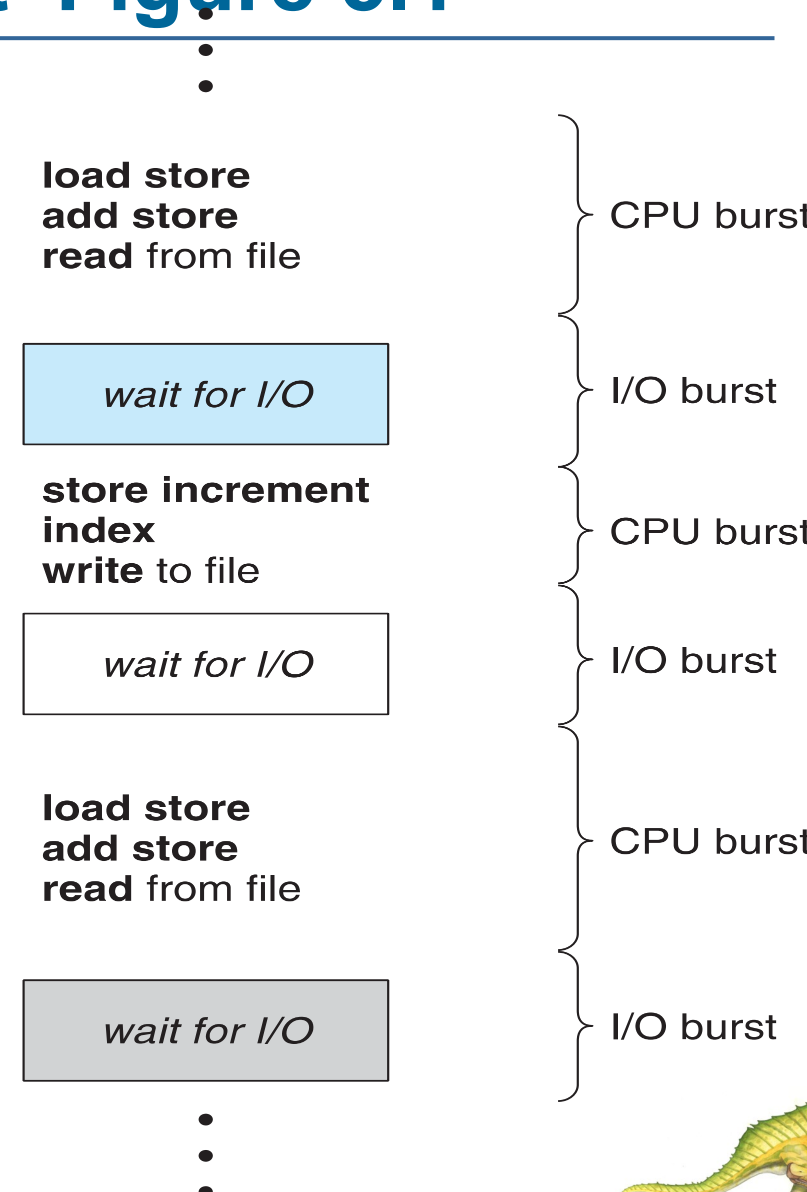
- In a **single-processor** system, only one process can run at a time.
  - Others must wait until the CPU is free.
- The **objective of multiprogramming** is to have some process always running, to **maximize CPU utilization**.
  - The idea is relatively simple: **A process is executed until it must wait for the completion of some I/O request.**
    - Several processes are kept in **main memory** at one time.
    - In this scheme, when one process has to **wait (I/O wait)**, the OS takes the CPU away from that process and gives the CPU to another process.
    - Every time one process has to wait, another process can take over the use of the CPU.
    - This pattern continues until all processes have completed their execution.

Basic Concepts

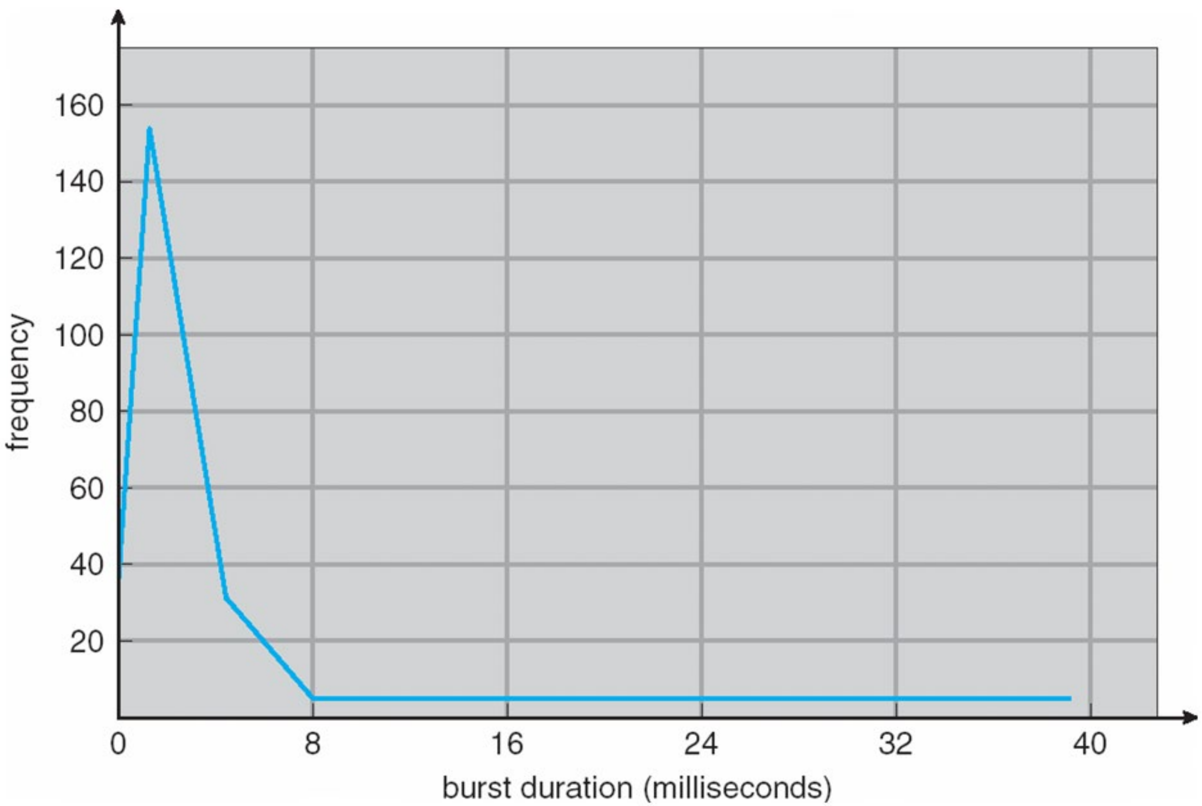
- Scheduling of this kind is a fundamental **operating-system** function.
  - Almost all **computer resources** are scheduled before use.
  - The **CPU** is one of the **primary computer resources**.
- Thus, **CPU scheduling** is central to operating-system design.
- The success of **CPU scheduling** depends on an observed **property of processes**:
  - Process execution consists of a **CPU cycle** and **I/O wait**.
  - Processes alternate between these two cycle states.
  - A **process execution time depends on CPU burst (CPU cycle time) and I/O burst (I/O cycle time)**.
  - Eventually, the **final CPU burst** ends with a **system request** to terminate execution (**Figure 5.1**).

CPU–I/O Burst Figure 5.1

- **Multiprogramming maximizes CPU utilization**
- **CPU burst** followed by **I/O burst**
- An I/O-burst has many short CPU bursts.
- A CPU-bound program may experience several prolonged CPU bursts.
- The distribution of **CPU and I/O bursts (Figure 5.2)** in a process is important when implementing a **CPU-scheduling algorithm.**



Histogram of CPU-burst Times Figure 5.2



Process Scheduler

- To **schedule the CPU**, the **Process Scheduler** takes advantage of a common trait among most computer programs: they alternate between **CPU cycles** and **I/O cycles**:

```
{
printf("\nEnter the first integer: ");
scanf("%d", &a);                              I/O cycle
printf("\nEnter the second integer: ");
scanf("%d", &b);

c = a+b
d = (a*b)-c                                   CPU cycle
e = a-b
f = d/e

printf("\n a+b= %d", c);
printf("\n (a*b)-c = %d", d);                 I/O cycle
printf("\n a-b = %d", e);
printf("\n d/e = %d", f);
}
```

Preemptive and Non-Preemptive

- **CPU-scheduling** decisions by an OS may take place under the following **four circumstances**:
  1. When a process switches from the **running state** to the **waiting state** (for example, as the result of an I/O request).
  2. When a process switches from the **running state** to the **ready state** (for example, when an ***interrupt*** occurs).
  3. When a process switches from the **waiting state** to the **ready state** (for example, ***completion of an I/O***).
  4. When a **process terminates**.
- When **scheduling** takes place only under circumstances **1** and **4**, we say that the scheduling scheme is **non-preemptive** or **cooperative**. Otherwise, it is **<u>preemptive</u>**.

Preemptive and Non-Preemptive

- Under **non-preemptive scheduling**, *once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state*.
  - Virtually all modern operating systems, including **Windows**, **macOS**, **Linux**, and **UNIX,** use **preemptive scheduling algorithms**.
  - Unfortunately, **preemptive scheduling** can result in ***race conditions*** when data are shared among several processes.
    - Consider the case of two processes that **share data**.
    - While one process is **updating the data**, it is **preempted** so that the **second process** can run.
    - The **second process** then tries to **read the data**, which is in an **inconsistent state**.

Preemptive and Non-Preemptive

- **Preemption** affects the design of the operating-system kernel.
- Operating-system **kernels** can be designed as either **non-preemptive** or **preemptive**.
- A **non-preemptive kernel:**
  - A **non-preemptive kernel** will wait for a **system call** to complete a process or to block while **waiting for I/O** to complete to take place before doing a multiprogramming **context switch**.
  - Unfortunately, this kernel-execution model is a **poor one for supporting real-time computing**, where tasks must complete execution within a given time frame **without preemption**.

Preemptive and Non-Preemptive

- A **preemptive kernel:**
  - A **preemptive kernel** requires mechanisms such as **mutex locks** (or **mutual exclusion locks**) to prevent **race conditions** when accessing shared **kernel data structures**.
  - Most modern operating systems are now **fully preemptive when running in kernel mode**.

Dispatcher

- The CPU-scheduling function is a **dispatcher** function.
- The **dispatcher** is the module that gives control of the CPU to the **process** selected by the **CPU scheduler**. This **dispatcher** function involves the following:
  - **Switching context** from one process to another
  - Switching to **user mode**
  - Jumping to the proper location in the user program to resume that program after a **kernel mode**
- The **time** it takes for the **dispatcher** to stop one process and start another running is known as the **dispatch latency** and is illustrated in **Figure 5.3**.

Dispatcher

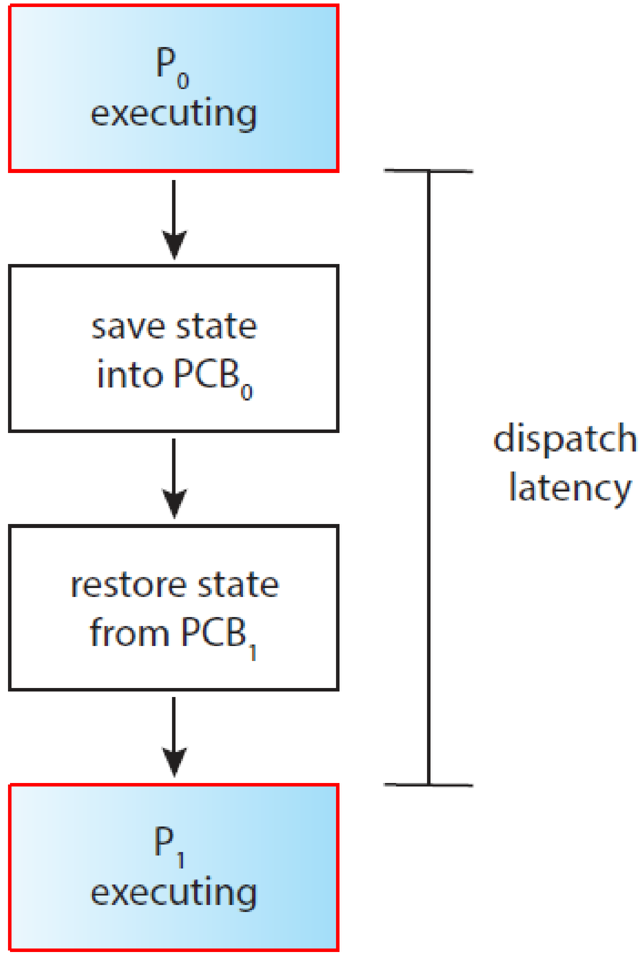


Figure 5.3 The role of the dispatcher.

Scheduling Criteria

- Many criteria have been suggested for comparing **CPU-scheduling** algorithms. The criteria include the following:
- **CPU utilization** – keep the CPU as busy as possible (can range from 0 to 100 percent).
- **Throughput** – # of processes that complete their execution per time unit.
- **Turnaround time** – amount of time to execute a particular process (is the sum of the clock periods spent waiting to get into *memory*, waiting in the *ready queue*, *executing on the CPU*, and *doing I/O*).
- **Waiting time** – 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–*is the time it takes to start responding.*

Scheduling Algorithm Optimization Criteria

The following are the **optimization criteria** of a CPU scheduling algorithm:

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

Scheduling Algorithms

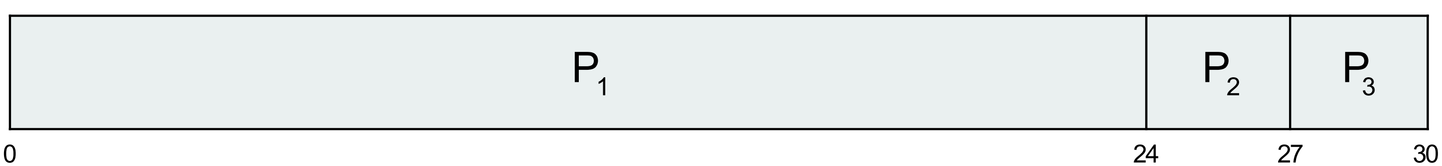
- **CPU scheduling** deals with the problem of deciding which of the processes in the **ready queue** is to be allocated the CPU.
- There are many different **CPU-scheduling algorithms**. This section describes the following:
  - **First-Come, First-Served Scheduling (FCFS)**
  - **Shortest-Job-First Scheduling (SJFS)**
  - **Priority Scheduling (PS)**
  - **Round-Robin Scheduling (RR)**
  - **Multilevel Queue Scheduling (MLQ)**
  - **Multilevel Feedback Queue Scheduling (MLFQ)**
  - **Thread Scheduling (TS)**

First- Come, First-Served (FCFS) Scheduling

Consider the following set of processes that arrive at time 0,with the length of the CPU burst given in milliseconds (ms):

Process	Burst Time(ms)
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**: $(0 + 24 + 27)/3 = 17$ ms

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**: $(6 + 0 + 3)/3 = 3$ ms
- Much better than previous case
- **Convoy effect - short process** behind **long process** (in previous case)
  - Consider one CPU-bound and many I/O-bound processes
  - This effect results in lower CPU and device utilization

FCFS Scheduling (Cont.)

- The FCFS scheduling algorithm is a **non-preemptive** one.
- 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.

Shortest-Job-First (SJF) Scheduling

- In the **SJF algorithm** each process is associated with the length of its next CPU burst
  - Use these lengths to schedule the process with the **shortest time**
- **SJF is optimal** – it 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

Consider the following set of processes (in the ready queue) with the length of the CPU burst given in milliseconds:

Process	Burst Time (ms)
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart

P_4	P_1	P_3	P_2

0 3 9 16 24

- **Waiting time** for P_4 = 0; P_1 = 3; P_3 = 9; P_2 = 16
- **Average waiting time** = (3 + 16 + 9 + 0) / 4 = 28/4 = 7 ms

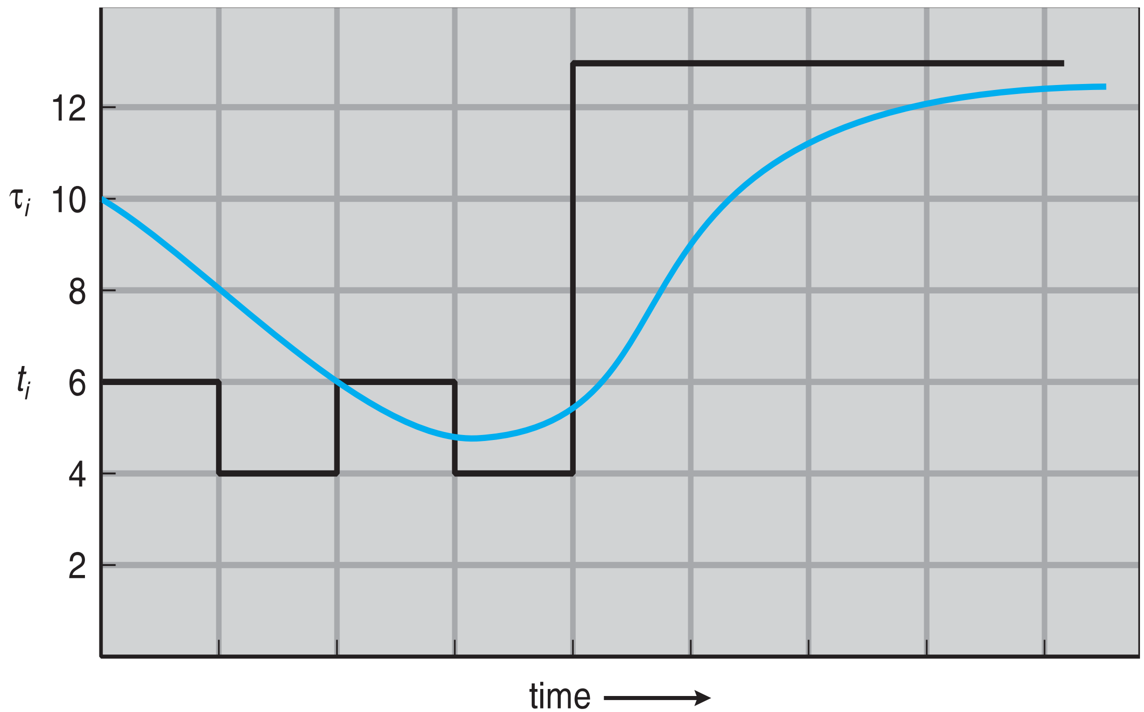
Determining Length of Next CPU Burst

- The real difficulty with the **SJF algorithm** is knowing **the length** of the next CPU request.
  - It can only estimate the length – should be similar to the previous one
  - Then pick process with **shortest predicted next CPU burst**
- Can be done by using the length of previous CPU bursts, using exponential averaging

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

- Commonly, α set to ½
- <u>A **Preemptive SJF** algorithm is called **shortest-remaining-time-first (SRTF)**</u>
- A **non-preemptive SJF** algorithm will allow the currently running process to finish its CPU burst.

Prediction of the Length of the Next CPU Burst



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

Shortest-remaining-time-first

- A **Preemptive** version **SJF** is called **shortest-remaining-time-first** (**SRTF**) algorithm
- Now we add the concepts of varying ***arrival times*** and ***preemption*** to the analysis

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

- ***Preemptive* SJF Gantt Chart**

P_1	P_2	P_4	P_1	P_3
0–1	1–5	5–10	10–17	17–26

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

Shortest-remaining-time-first

- Process *P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 ms) is* larger than the time required by process *P2 (4 ms), so process P1 is* preempted, and process *P2 is scheduled. The average waiting time for this* example is

  [(10 − 1) + (1 − 1) + (17 − 2) + (5 − 3)]/4 = 26/4 = 6.5 ms.

  P1 = final-start – initial start = 10 -1 = 9 ms

  P2 = start – arrival = 1-1 = 0 ms

  P3 = start – arrival = 17-2 = 15 ms

  P4 = start – arrival = 5-3 = 2 ms

  Total time /4 = (9 + 0 + 15 + 2)/4 = 26/4 = 6.5 ms

- **Non-preemptive SJF** scheduling would result in an average waiting time of 7.75 ms

Priority Scheduling

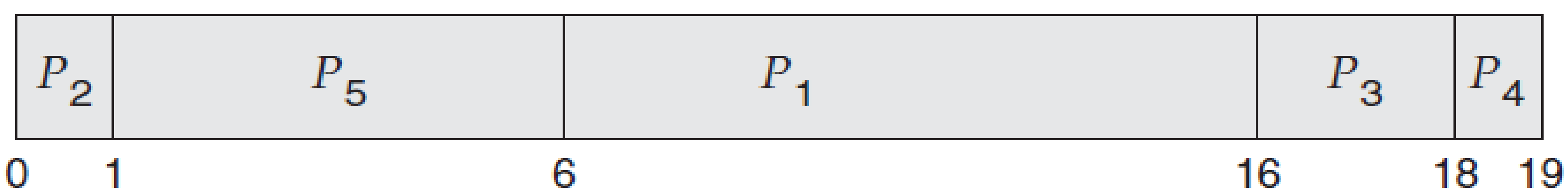
- A **priority number** (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (**smallest integer represents highest priority**)
  - **Preemptive**
  - **Non-preemptive**
- **Equal-priority processes** are scheduled in FCFS order
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- <u>**Problem**</u> $\equiv$ **Starvation** – low priority processes may never execute
- <u>**Solution**</u> is **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Consider the following set of processes, assumed to have arrived at time 0 in the order *P1, P2, · · ·, P5, with the length of the CPU burst* given in ms:

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

- Priority scheduling Gantt Chart



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

Round Robin (RR) Scheduling

- The **round-robin (RR) scheduling algorithm** is designed especially for **timesharing** systems. It is similar to **FCFS** scheduling, but **preemption** is added to enable the system to switch between processes.
- *Each process gets a small unit of CPU time* (**time quantum *q***), usually 10-100 milliseconds. After this time has elapsed, the process is **preempted** and added to the end of the **ready queue**.
  - The **ready queue** is treated as a circular 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** $\Rightarrow$ **FIFO**
  - ***q* small** $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high

Round Robin (RR) Scheduling

- The **ready queue** is treated as a **circular queue**. The *CPU scheduler* goes around the **ready queue**, allocating the CPU to each process for a time interval of up to **1 time quantum**.
- New processes are added to the tail of the **ready queue**. The *CPU scheduler* picks the first process from the **ready queue**, sets a timer to interrupt after **1 time quantum**, and **dispatches** the process.
- If the process have a CPU burst of **less than 1 time quantum**, then the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the **ready queue**.
- If the CPU burst of the currently running process is **longer than 1 time quantum**, the timer will go off and will cause an interrupt to the operating system.
  - A context switch will be executed, and the process will be put at the tail of the **ready queue**. The CPU scheduler will then select the next process in the ready queue.

Example of RR with Time Quantum = 4

Consider the following **set of processes that arrive at time 0**, with the length of the CPU burst given in ms (and the **time quantum** is 4 ms):

Process	Burst Time (ms)
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

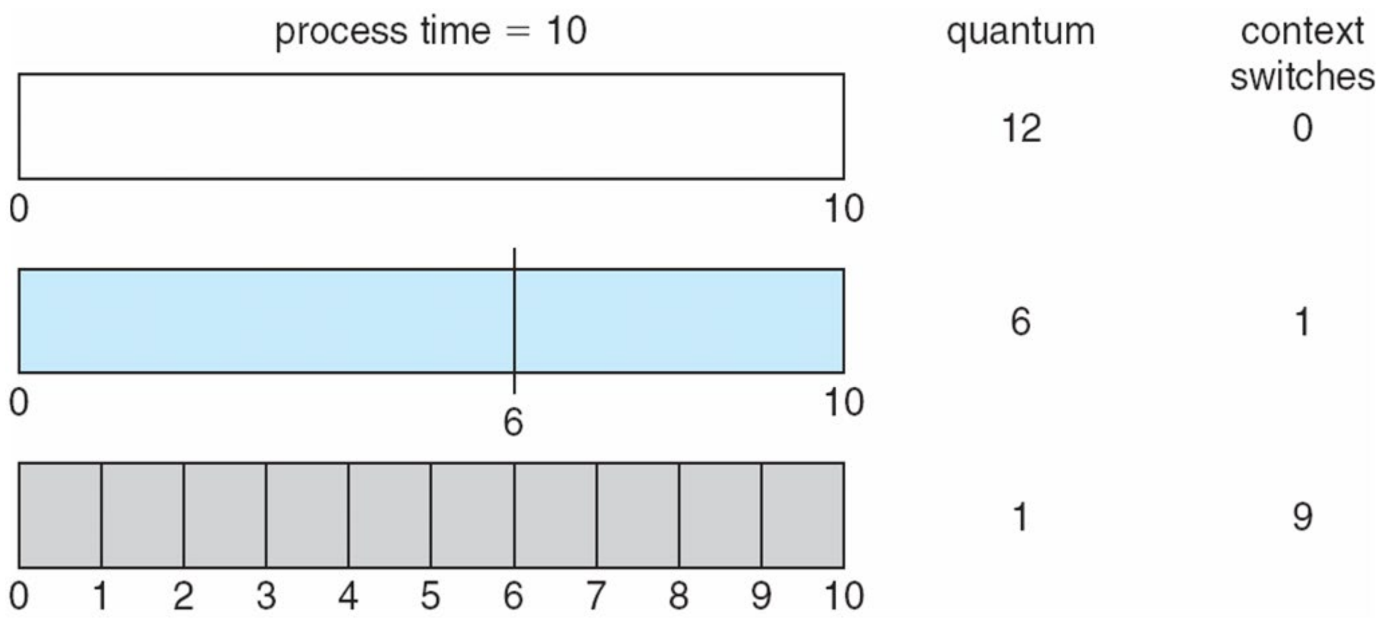
Let's calculate the **average waiting time** for this schedule. P1 waits for 6 ms (10 - 4), P2 waits for 4 ms, and *P*3 *waits for* 7ms

- Thus, the **average waiting time** is (6+4+7)/3 = 17/3 = 5.66ms

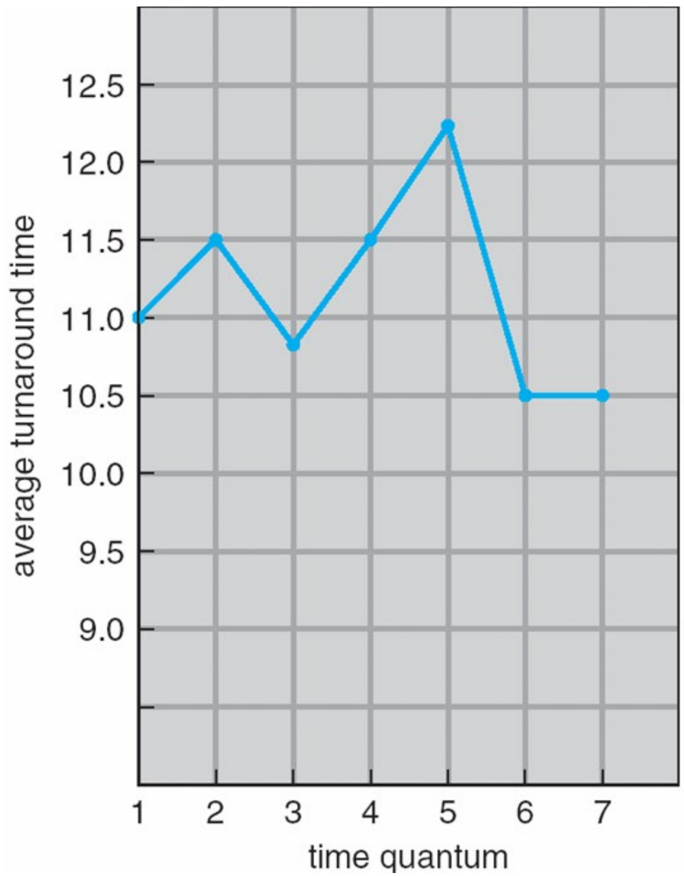
Example of RR with Time Quantum = 4

- If we use a time **quantum of 4 milliseconds**, then process *P1 gets the first 4* ms.
- Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process *P2. Process P2 does not need 4 milliseconds, so it quits before its time* quantum expires.
- The CPU is then given to the next process, process *P3. Once* each process has received 1 time quantum, the CPU is returned to process *P1* for an additional time quantum.
- Typically, higher average turnaround than SJF, but better ***response***
- The **time quantum, q should be large compared to context switch time**
  - **q** usually 10ms to 100ms, context switch < 10 μsec

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

80% of CPU bursts
should be shorter than q

Multilevel Queue

- The **Ready queue** is partitioned into separate queues, eg:
  - **foreground** (interactive) processes
  - **background** (batch) processes
- **foreground processes** may have higher priority (externally defined) over **background processes**.
- 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 Scheduling

- A **multilevel queue scheduling algorithm** partitions the **ready queue** into several separate queues (**Figure 6.6**).
- The processes are permanently assigned to one queue, generally based on some property of the process, such as *memory size*, *process priority*, or *process type*.
- Each queue has its own **scheduling algorithm**. For example, separate queues might be used for **foreground** and **background** processes.
- The **foreground queue** might be scheduled by an **RR algorithm**, while the **background queue** is scheduled by an **FCFS algorithm**.
- The **foreground queue** may have absolute priority over the **background queue.**

Multilevel Queue Scheduling

- Let's look at an example of a **multilevel queue scheduling algorithm** with **five queues, listed below in their order of priority**:
  1. **System processes**
  2. **Interactive processes**
  3. **Interactive editing processes**
  4. **Batch processes**
  5. **Student processes**
- 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.

Multilevel Queue Scheduling

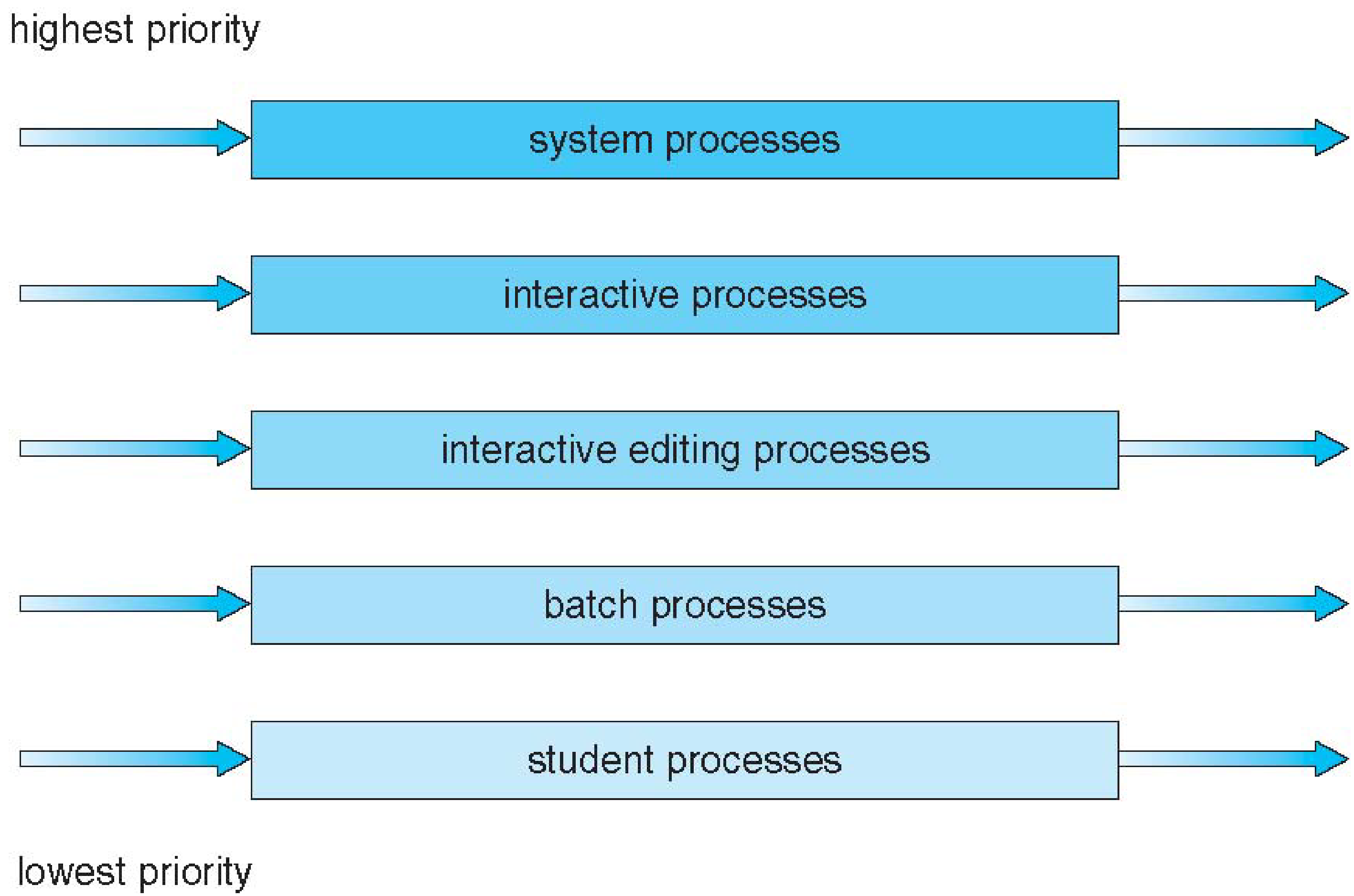


Figure 5.6 Multilevel queue scheduling.

User-Level Thread Scheduling

- We have seen the distinction between **user-level** and **kernel-level** threads
- As per operating systems **the 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)**.
- **Thread library schedules user-level threads to run on LWP**
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
  - Typically done via priority set by programmer
- **Kernel thread** scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

User-Level Thread Scheduling

- When we say the ***thread library*** schedules **user threads** onto available **LWP**s, we do not mean that the threads are actually running on a CPU.
- That would require the OS to schedule the **kernel thread** onto a physical CPU. To decide which **kernel-level thread** to schedule onto a CPU, the kernel uses **system-contention scope (SCS).**
- Typically, **process-contention scope (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**.

Multiple-Processor Scheduling

- **CPU scheduling is** more complex when **multiple CPUs** are available
- **<u>Multi-core system:</u> Homogeneous processors** within a **multiprocessor**
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes are in a **common ready queue**, or each has its own private **queue of ready processes.**
- The core idea behind the **multi-processor scheduling** is organizing the **threads** eligible to be scheduled:
  - All **threads** may be in a common **ready queue**.
  - Each processor may have its own **private queue of threads**.
- These two strategies are contrasted in **Figure 5.11**.

NUMA and CPU Scheduling

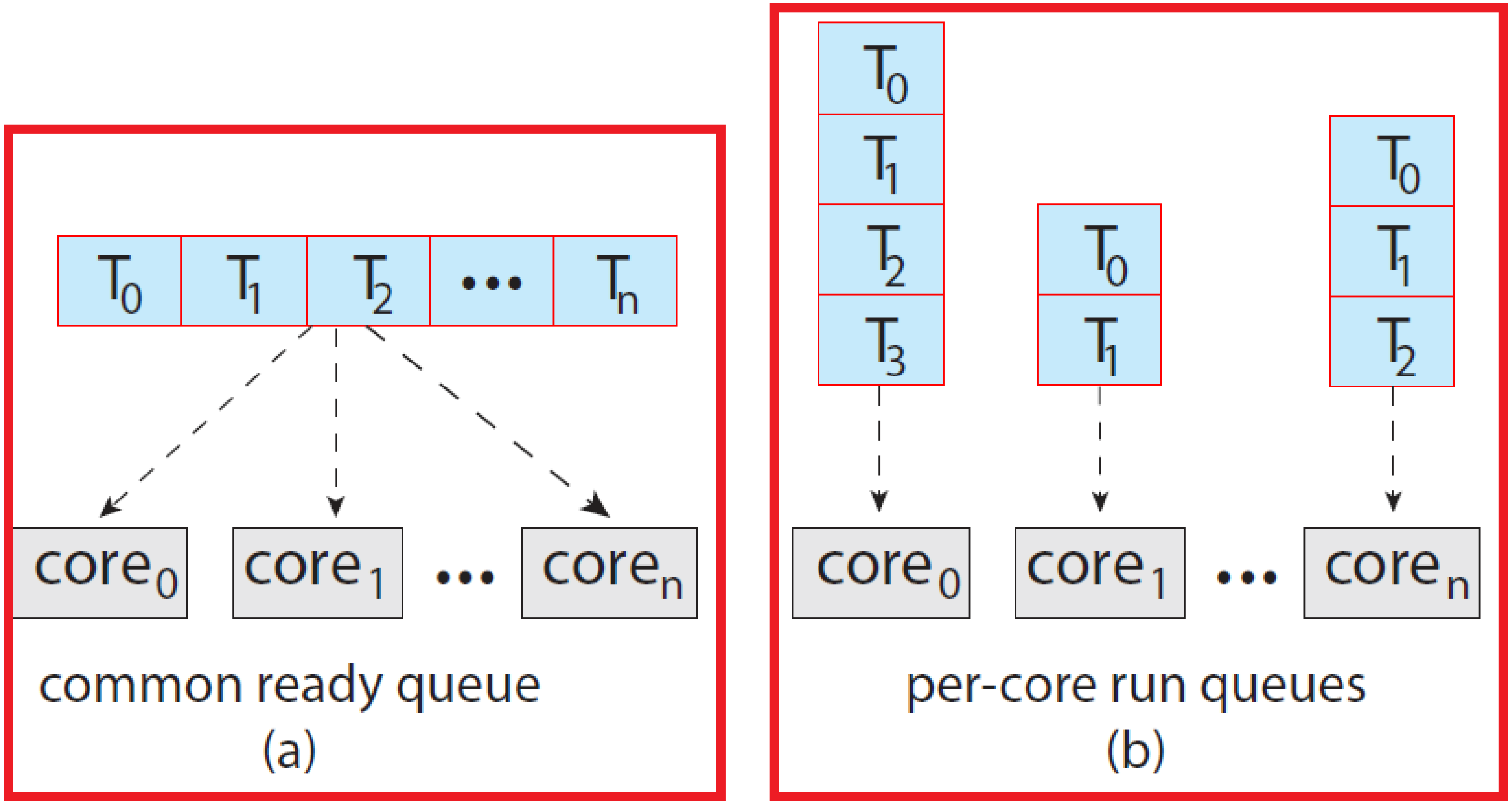
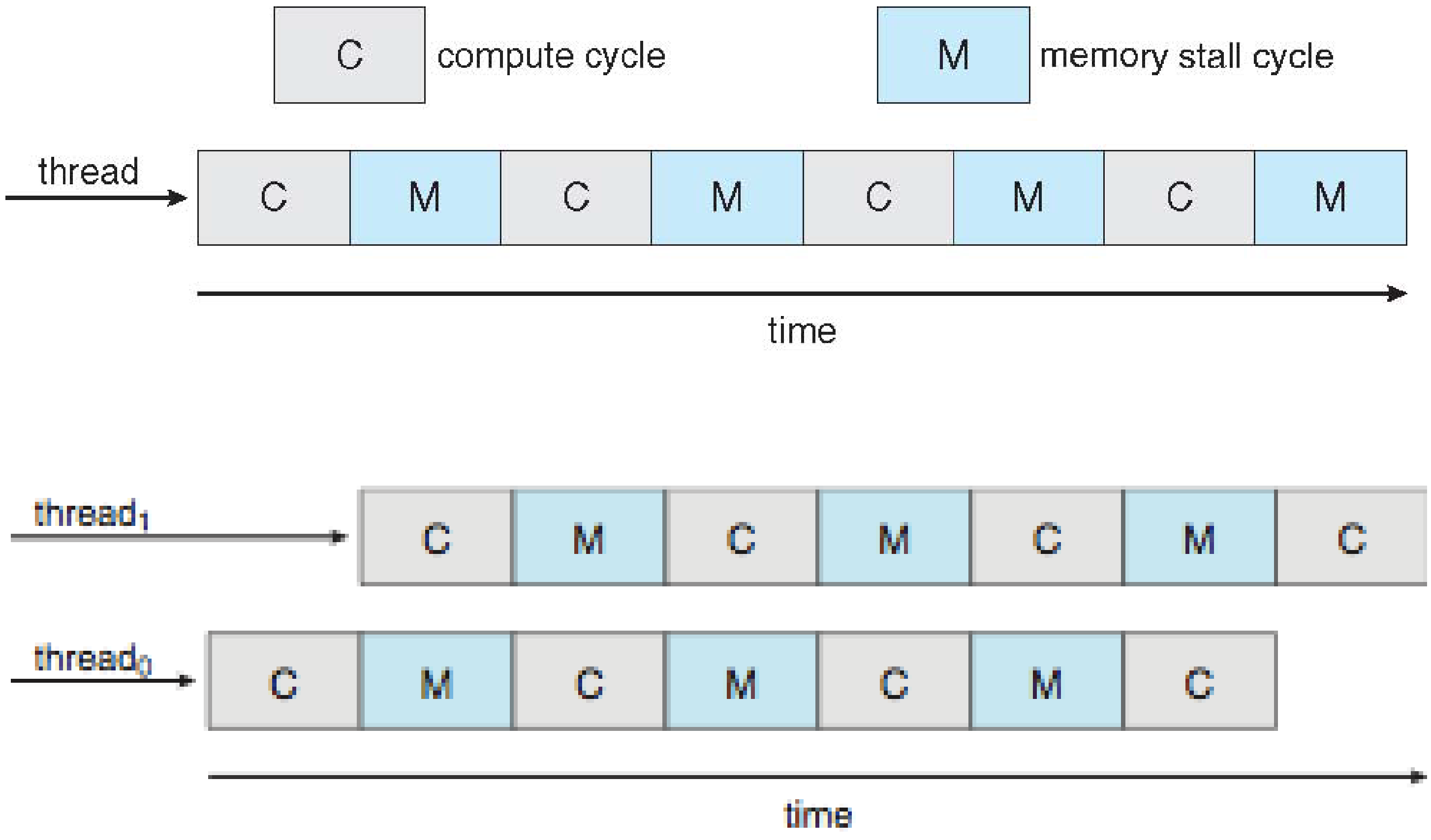


Figure 5.11 Organization of ready queues.

Multiple-Processor Scheduling – Load Balancing

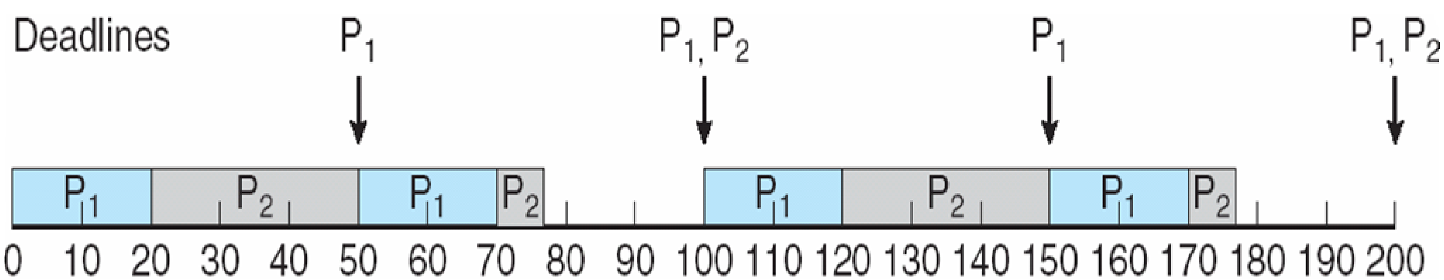
- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task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

Multithreaded Multicore System

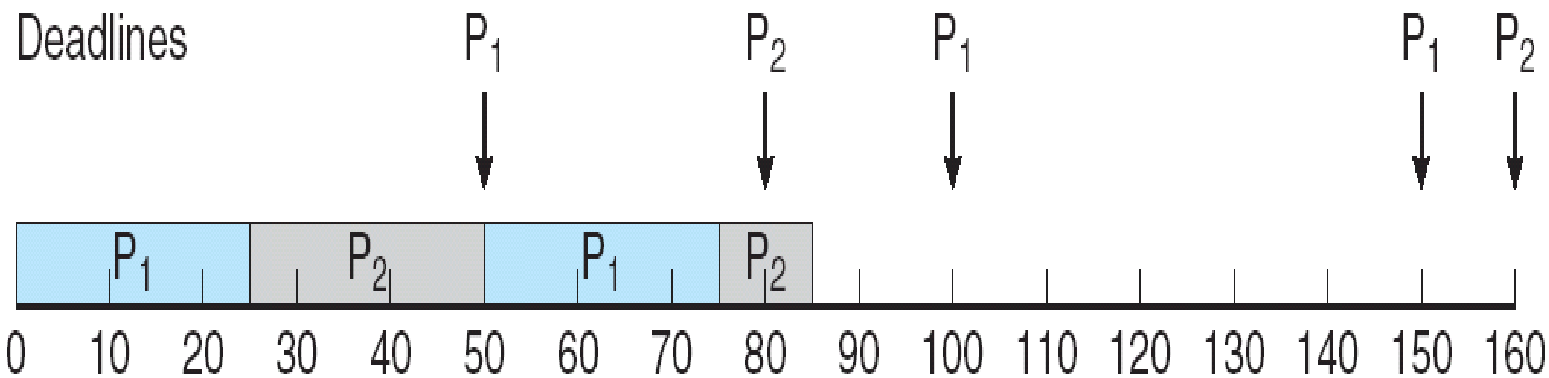


Rate Montonic Scheduling

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



Missed Deadlines with Rate Monotonic Scheduling

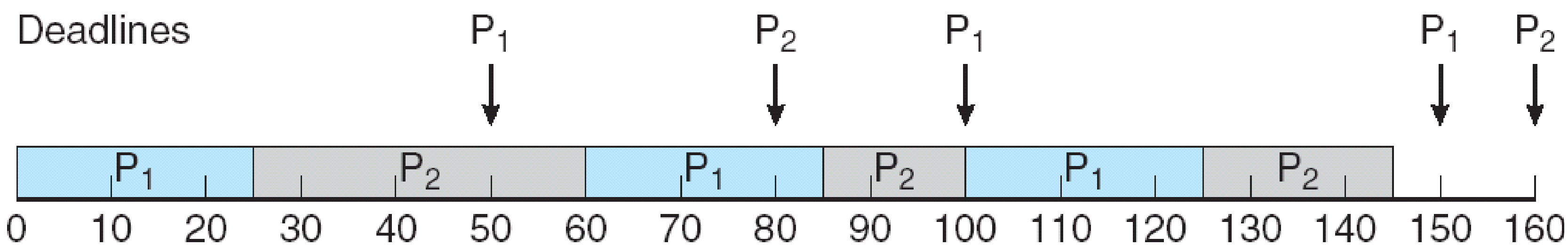


Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

  the earlier the deadline, the higher the priority;

  the later the deadline, the lower the priority



Scheduling in Operating Systems

- Linux scheduling
- Windows scheduling
- Solaris scheduling

Linux Scheduling Through Version 2.5

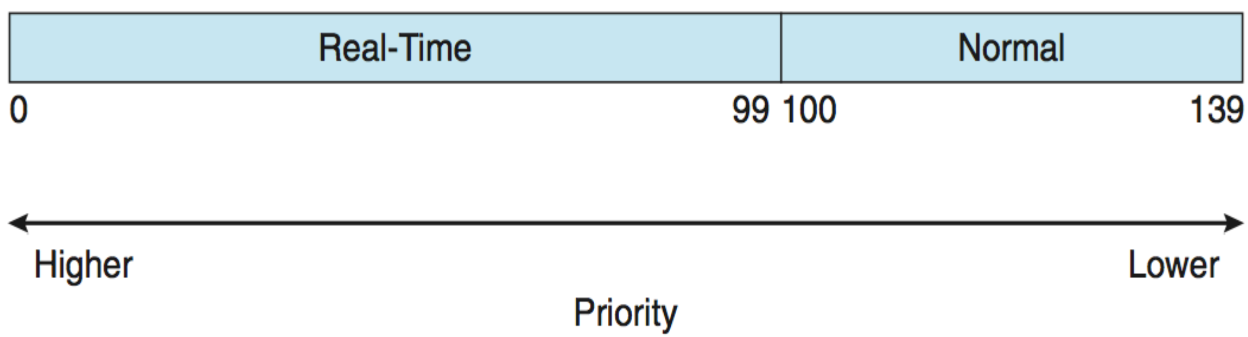
- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order *O*(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes

Linux Scheduling in Version 2.6.23 +

- ***Completely Fair Scheduler*** (CFS)
- **Scheduling classes**
  - Each 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: **default** and **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 scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate and 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
- **Dispatcher** is scheduler
- 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 Priority Classes

- **Win32 API** identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base

Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling** (**UMS**)
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework

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

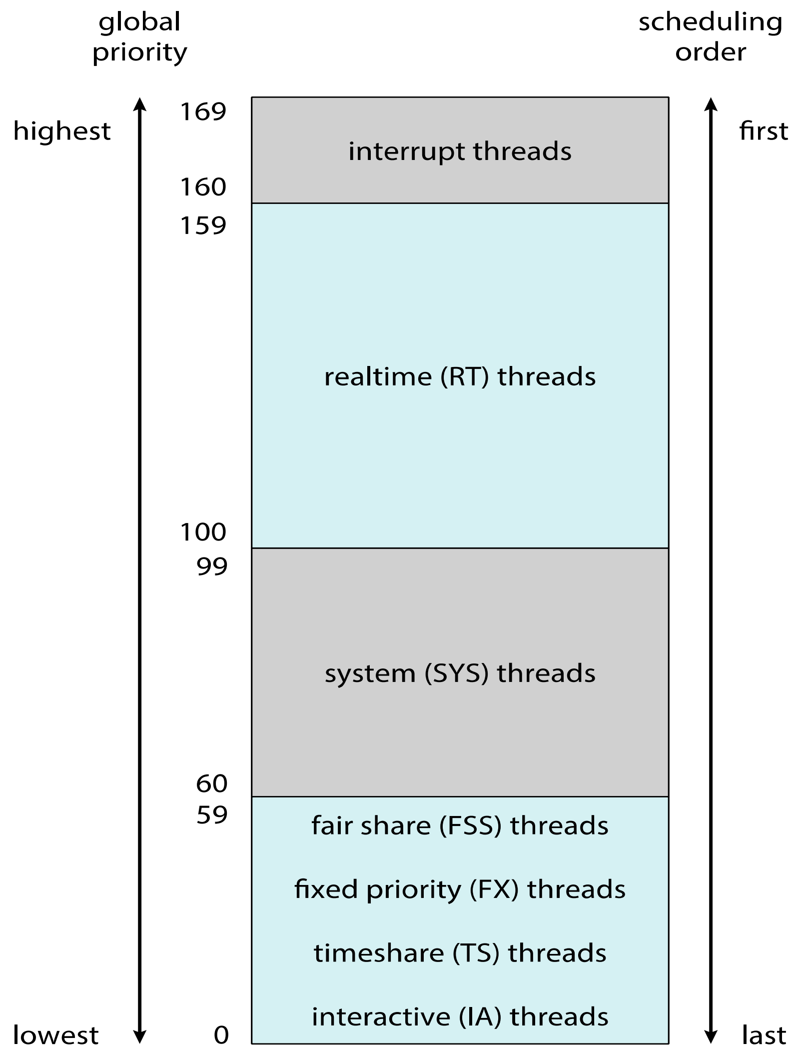
Solaris

- Priority-based scheduling
- Six classes available
  - **Time sharing (default) (TS)**
  - **Interactive (IA)**
  - **Real time (RT)**
  - **System (SYS)**
  - **Fair Share (FSS)**
  - **Fixed priority (FP)**
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sys.admin

Solaris Dispatch Table

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

Solaris Scheduling



Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

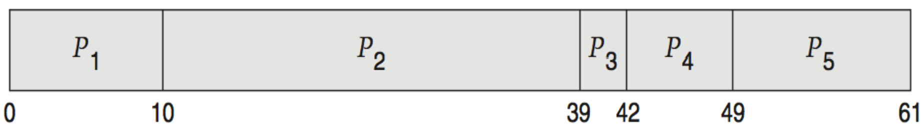
Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **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:

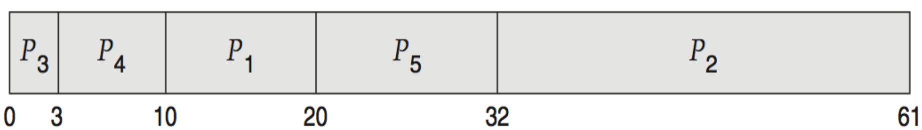
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

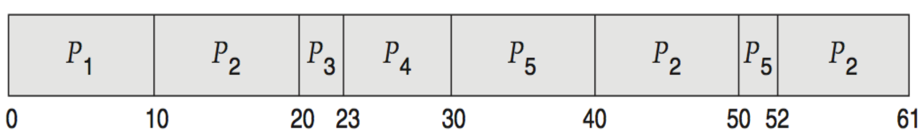
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:



  - Non-preemptive SFJ is 13ms:



  - RR is 23ms:



Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system 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

Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

  $n = \lambda \times W$

  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

Evaluation of CPU Schedulers by Simulation