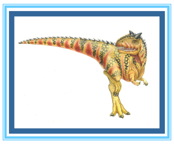
Chapter 5: CPU Scheduling



Chapter 5: CPU Scheduling

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

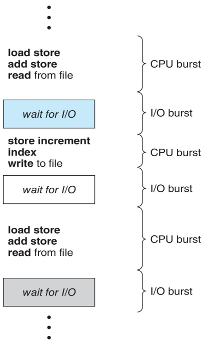
Objectives

- Describe various CPU scheduling algorithms
- Evaluate CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe real-time scheduling algorithms
- Apply modeling and simulations to evaluate CPU scheduling algorithms

---

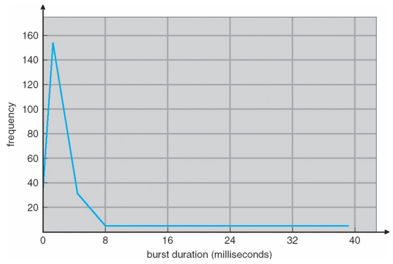
Basic Concepts

- The objective of multiprogramming is to have a process running at all times - maximize CPU utilization
- Process execution consists of a cycle of CPU execution and I/O wait – referred as CPU burst and I/O burst（when not running on CPU)
- Whenever CPU is idle, the OS tries to select one of processes on the ready queue to execute unless the ready queue is empty
- The selection of process is carried out by the CPU scheduler or called process scheduler, short-term scheduler



Histogram of CPU-burst Times

- The durations of CPU bursts have been measured extensively over the years. The frequency curve is similar to that shown below
- There are a large number of short CPU bursts and a small number of long CPU bursts (long-tail distribution). An I/O-bound program typically has many short CPU bursts, while a CPU-bound program might have a few long CPU bursts.
- This distribution is important for designing a CPU-scheduling algorithm

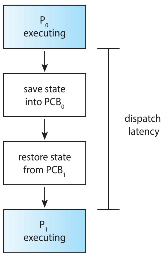


CPU Scheduler

- The CPU scheduler selects one process or thread from the processes on ready queue, and allocates the a CPU core to the selected process
  - Queue may be ordered in various ways, even single or multiple queues
- CPU scheduling may take place in the following four conditions:
  1. Switches from running to waiting state, e.g., I/O request, or wait()
  2. Process terminates
  3. Switches from running to ready state, e.g., interrupt
  4. Switches from waiting to ready, e.g., completion of I/O or from new to ready, a new process arrives on the ready queue with a higher priority
- Scheduling under 1 and 2 is non-preemptive, in which a process gives up CPU voluntarily (by itself)
- Scheduling under 3 and 4 is preemptive
  - Consider access to shared data (discussed in Chapter 6)
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Dispatcher

- Dispatcher module allocates the CPU to the process selected by the short-term scheduler; this involves:
  - switching context from one process to another
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- Dispatch latency – the time it takes for the dispatcher to stop one process and start another running – an overhead
- The number of context switches can be obtained by using `vmstat` on Linux, typically hundreds of context switches per second



Scheduling Criteria

- **CPU utilization** – fraction of the time that CPU is busy
- Throughput – # of processes or jobs completed per time unit (second)
  - Efficient use of the resources (CPU, memory, disk, etc.)
  - May be one process in seconds (long), or tens of processes/sec (short)
- Waiting time – amount of time a process waiting on the ready queue
- Turnaround time – the amount of time to execute a particular process, measured by the CPU burst time, I/O burst time and waiting time
  - Considering single CPU burst, turnaround time = waiting time + CPU burst time
- Response time – the amount of time it takes from when a request was submitted until the first response is produced
  - Time to echo a keystroke in editor, or games
  - This is more relevant to interactive programs (typically using RR scheduling)
  - Considering single CPU burst, this is the time between the completion of first CPU time minus the time that this process joins the ready queue
- Fairness
  - Resources such as CPU are utilized in some "fair" manner

Scheduling Criteria (Cont.)

- It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. But these can be conflicting set of criteria, there are different considerations in practice
- In most cases, we optimize an average measure, e.g., the average waiting time. However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average
  - Considering all users, we may want to minimize the maximum response time
- For interactive systems (such as a desktop or laptop), it might be 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
- Different CPU-scheduling algorithms have different properties. we next describe several scheduling algorithms in the context of only one CPU core – the system is capable of only running one process or thread at a time

---

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
- Average turn-around time: (24+27+30)/3 = 27
- In earlier systems, FCFS means that one program is scheduled to run until completion including all I/O
- In multiprogramming systems, this means a process finishes its current CPU burst time

---

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 shorter!)
- Average turn-around time: (30+3+6)/3 = 13
- **Convoy effect** – A short process stuck behind a long process, bad for short jobs, potentially (depending purely on the arrival order)
  - Consider one CPU-bound and many I/O-bound processes, FCFS results also in low device utilization
  - Waiting in banks: depositing a check, stuck behind new account opening

---

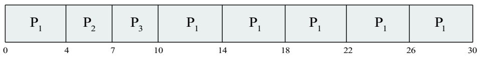
Round Robin (RR)

- The FCFS scheduling algorithm is non-preemptive. Once the CPU core allocated to a process, the process keeps the CPU until it releases CPU
- The FCFS is thus particularly troublesome for interactive systems, where it is important that each process get a share of the CPU at regular intervals.
- The round-robin (RR) scheduling algorithm is similar to FCFS scheduling, but preemption is added to enable 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 (exactly like FCFS).
- Given `n` processes, each process gets `1/n` of the CPU time in chunks of at most `q` time units at once – context switching time is ignored
  - No process waits more than `(n-1)q` time units.
- A timer interrupts every quantum to schedule next process, or the process blocks upon completing its current CPU burst time when its CPU burst time < `q`

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



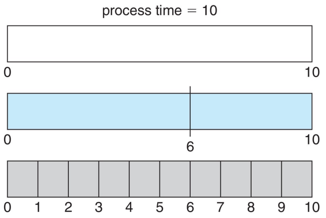
Waiting time for P1=6, P2=4,P3=7

- Average waiting time (6+4+7)/3 = 5.67
- Average turn-around time: (30+7+10)/3 = 15.67
- **Response time** for P1=4, P2=7, P3=10, average = 7
- The average waiting time under RR policy can be long, but is inherently more “fair” (FIFO order), usually perform better for short jobs than FCFS, and offers better average response time – important for interactive jobs

---

Time Quantum and Context Switch Time

- The performance of the RR algorithm depends heavily on the size of the time quantum
  - q large ⇒ FCFS
  - q small ⇒ interleaved, but q must be large with respect to context switch time (usually < 10 usec), otherwise overhead is too high



quantum	context switches
12	0
6	1
1	9

Comparisons between FCFS and RR

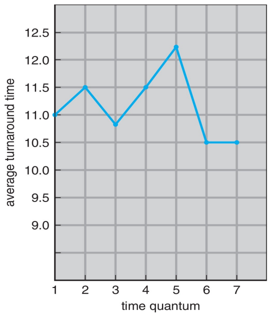
Assuming zero-cost context-switching time, is RR always better than FCFS?

- An example: 10 jobs starting at the same time, each taking 100s of CPU time; RR scheduler quantum of 1s;

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  - The average job turn-around time is much worse under RR!
    - Bad when all jobs have the same length

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

- The average turnaround time does not necessarily improves as the time quantum size increases
- In general, the average turnaround time can be improved if most processes finish their current CPU bursts within a single quantum
- The time quantum can not be too big, in which RR degenerates to an FCFS policy
- **A rule of thumb**: 80% CPU bursts should be shorter than the time quantum q

---

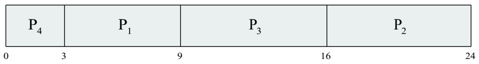
Shortest-Job-First (SJF) Scheduling

- Noticing that in FCFS and RR, we do not need to know the next CPU burst time of each process during scheduling, and scheduling is done based on the arrival order
- What if we knew the future – the next CPU burst time of each process
- Associate with each process the length of its next CPU burst
  - To schedule the process with the shortest next CPU burst
- The Shortest Job First or SJF scheduling algorithm is optimal – produces the minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - The basic idea is to get the short jobs out of the system sooner
  - Big effect on short jobs, relatively small effect on long jobs
  - This can be applied to an entire program or the current CPU burst
  - Perhaps a more precise term should be the shortest-next-CPU-burst algorithm, but shortest job first or SJF is commonly used.

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
- The “best” FCFS perform the same if arrival order happens to be the same
- Sketch Proof: Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases

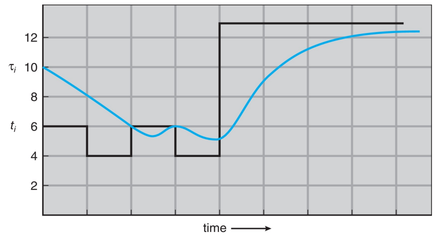
Determining Length of Next CPU Burst

- How to estimate the length based on the past behavior
  - Then pick the process with shortest predicted next CPU burst
- Can be done by using the lengths of previous CPU bursts and exponential averaging algorithm

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

- Commonly, α set to ½ - the relative weight of recent and past history in the prediction
- Preemptive version called shortest-remaining-time-first（SRTF）

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 …

This shows an exponential average with $\alpha = 1/2$ and $\tau_0 = 10$.

Examples of Exponential Averaging

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

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

- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor, thus its effect is diminishing exponentially fast

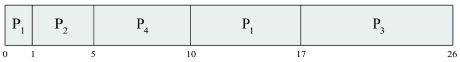
---

Example of Shortest-Remaining-Time-First

- The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while another process is still executing

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
- Now scheduling needs to be considered when there is an arrival to the ready queue (scheduling condition 4)

Comparison of SJF/SRTF and FCFS

- SJF/SRTF are the best we can do towards minimizing the average waiting time. or the average turnaround time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - SRTF is always at least as good as SJF
- SJF/SRTF performs the same as FCFS if all processes have the same CPU burst times
- SJF/SRTF can possibly lead to starvation for long process if there is always shorter process joining the ready queue
  - "fairness" can not be enforced

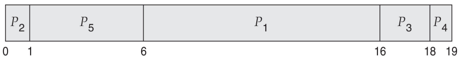
Priority Scheduling

- A priority number (e.g., integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority), it can be
  - Preemptive（upon new arrival of a higher priority process)
  - Nonpreemptive
- Equal-priority processes are scheduled in FCFS order
- SJF is is a special case of the general priority-scheduling algorithm, where priority is the inverse of predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process

Example of Priority Scheduling

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

- Priority scheduling Gantt Chart

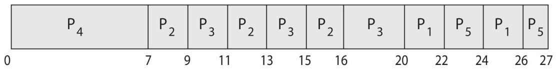


- Average waiting time = 8.2 msec

Priority Scheduling w/ Round-Robin

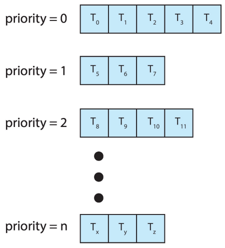
Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Run the process with the highest priority. Processes with the same priority run round-robin
- Gantt Chart with 2 ms time quantum



Multilevel Queue

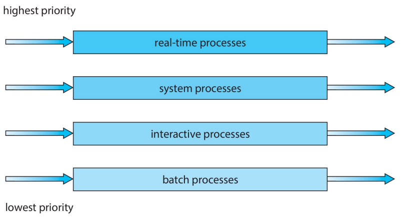
- Multilevel queue scheduling can still be a priority scheduling combined with round-robin
- A priority is assigned statically to each process, and a process remains in the same queue for the duration of its runtime



Multilevel Queue (Cont.)

Partition processes into different queues based on process type

- Each queue can have its own scheduling algorithm based on the needs
- The scheduling among the queues, is commonly implemented as ***fixed-priority preemptive scheduling*** or each queue gets certain amount of CPU time – time-slice (for instance 60%, 20%, 10%, 10%)



Multilevel Feedback Queue (MLFQ)

- A process can move between the various queues; aging can be implemented this way. This provides the flexibility
- Multilevel-feedback-queue or MLFQ scheduler defined by the following parameters:
  - The 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

---

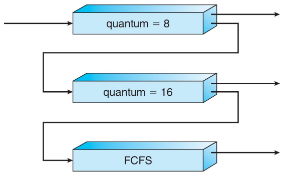
Example of Multilevel Feedback Queue

Three queues:

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

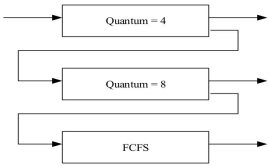
Scheduling

- A new job enters queue Q_0 which is served FCFS, also preempts jobs from Q1 or Q2 if currently running on CPU
  - When it gains CPU, job receives 8 ms
  - If it does not finish in 8 milliseconds, job is moved to the 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
- If a job from Q_1 or Q_2 is preempted by a new job from Q_0, it joins the head of the queue Q_1 or Q_2 , respectively

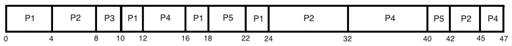


- This approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top

MLFQ Example

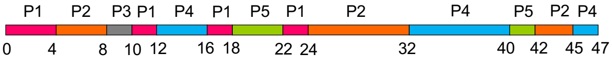


Process	Arrival Time (ms)	Burst Time (ms)
P1	0	10
P2	2	15
P3	5	2
P4	12	14
P5	18	6

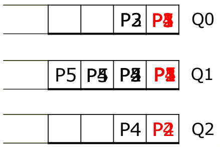


MLFQ Scheduling: Example

	Process	Burst Time	Arrival Time	Remaining Time
√	P1	10	0	10 0
√	P2	15	2	15 0
√	P3	2	5	2 0
√	P4	14	12	10 0
√	P5	6	18	6 0

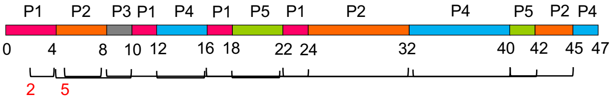


- at time [illegible] and waits in Q1 [illegible] service in Q1



MLFQ Scheduling: Example

Process	Burst Time	Arrival Time	Remaining Time
P1	10	0	0
P2	15	2	0
P3	2	5	0
P4	14	12	0
P5	6	18	0



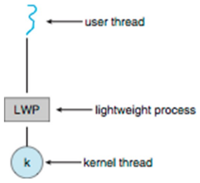
- Waiting time for P1=14, P2=28, P3=3, P4=21, P5=18
- Average waiting time: (14+28+3+21+18)/5=16.8

Multilevel Feedback Queue (MLFQ)

- MLFQ is commonly used in many systems such as BSD Unix, Solaris, Window NT and subsequent Window operating systems
- MLFQ has several distinctive advantages:
  - It does not need prior knowledge on the next CPU burst time
  - It handles interactive jobs well by delivering similar performance as that of SJF or SRTF
  - It is also "fair" by making progress on CPU-bound jobs
- The possible starvation problem can be handled by reshuffling the jobs to different queues periodically
  - E.g., after some period, move all jobs to the top queue

Thread Scheduling

- On modern OS, kernel-level threads are the ones being scheduled, user-level threads are managed by a thread library instead
- The OS can use an intermediate data structure between user threads and kernel threads, a lightweight process (LWP)
  - Appears as a virtual processor on which user threads are scheduled to "run"
  - Each LWP attached to kernel thread (one-to-one)



Thread Scheduling

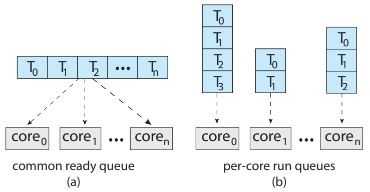
- Under many-to-one and many-to-many models, thread library "schedules" user-level threads to run on LWP. This is known as process-contention scope (PCS).
- Since scheduling competition takes place among the threads belonging to the same process, typically done via ***priority*** set by programmers
  - Thread library usually can not adjust the priority
  - PCS will typically preempt the thread currently running in favour of a higher-priority (user-level) thread
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in the system – CPU scheduling
- Systems using one-to-one mapping model, such as Windows, Linux, and Solaris, schedule threads using only SCS

Multiple-Processor Scheduling

- CPU scheduling far more complex with multiple CPUs – load sharing
- Traditionally, the term multiprocessor referred to systems that provided multiple physical processors, where each physical processor chip contained one single-core CPU
- The definition of multiprocessor has evolved significantly, and in modern computing systems, multiprocessor now applies to multicore CPUs, multithreaded cores, NUMA systems, and heterogeneous multiprocessing
- There are generally two types of multiprocessing systems, asymmetric multiprocessing and symmetric multiprocessing
- Asymmetric multiprocessing – only one processor can access kernel data structures, alleviating the need for data sharing. The other processors execute only user codes
  - All scheduling decisions, I/O processing, and other system activities handled by a single processor — the master server
  - The master server can become a potential bottleneck

Symmetric multiprocessing (SMP)

- Symmetric multiprocessing (SMP) – each processor is self-scheduling, with one common ready queue, or each having its own private ready queue
  - Scheduling proceeds by having the scheduler of each processor examine the ready queue and select a thread to run
  - To ensure two separate processors do not choose to schedule the same thread with a common ready queue - possible race condition (discuss in Chapter 6)
  - All modern OS support SMP, including Window, Linux, Mac OS X, as well as mobile systems including Android and iOS

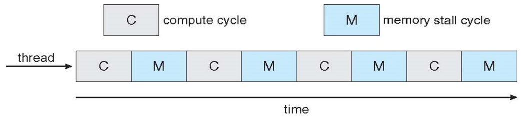


common ready queue
(a)

per-core run queues
(b)

Multicore Processors

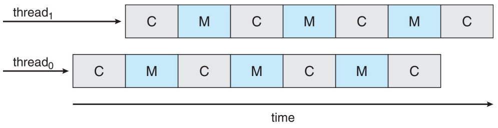
- Recent trend to place multiple processor cores on same physical chip
  - Faster and consumes less power, but complicate the scheduling design
- Memory stall: when a processor accesses memory, it spends a significant amount of time waiting for data to become available, primarily because modern processors operate at much faster speeds than memory, esp. when there is a cache miss
- Multiple hardware threads per core - each hardware thread has its own state, program counter (PC), register set appearing as a logical CPU to run a software thread. This is known as chip multithreading (CMT)



Multithreaded Multicore System

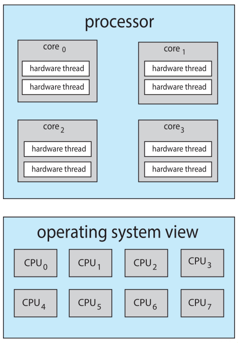
The scheduling can takes advantage of memory stall to make progress on another hardware thread while memory retrieve happens

- If one thread stalls while waiting for memory, the core can switch to another thread. This becomes a dual-thread processor core, or resembles two logical processors
- A dual-threaded, dual-core system presents four logical processors to the operating system
- UltraSPARC T3 CPU has 16 cores per chip and 8 hardware threads per core, from operating system perspective, this appear to be 128 logical processors



Multithreaded Multicore System

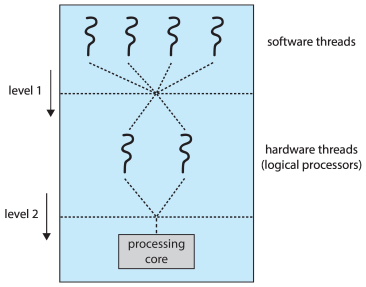
- From an operating system perspective, each hardware thread maintains its architectural state, such as instruction pointer and register set, and thus appears as a logical CPU that is available to run a software thread
- Chip-multithreading (CMT) assigns each core multiple hardware threads. (Intel refers 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 (kernel thread) to run on a logical CPU – CPU scheduling that we cover in this Chapter
2. How each core decides which hardware thread to run on the physical core – could use RR scheduling (UltraSPARC T3)



Multithreaded Multicore Scheduling

- A user-level thread is schedule to a LWP – a kernel-level thread
  - Under many-to-one and many-to-many models, thread library schedules user-level threads known as process-contention scope (PCS), typically based on the **priority** set by programmers
- A kernel thread now referred as a software thread is scheduled onto a logical CPU – a hardware thread
  - CPU or process scheduling – operating system
- A hardware thread is scheduled to run on a CPU core
  - Each CPU core decides the scheduling, typically using RR

Processor Affinity

Processor affinity – process has an affinity for a processor on which it runs

- When a thread has been running on one processor, the cache content 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”)
- There is a high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another
- Essentially, per-processor ready queues provide processor affinity for free!
- Soft affinity – the OS attempt to keep a process running on the same processor (not guaranteeing that), and it is possible for a process to migrate between processors during load balancing
- Hard affinity – allow a process to specify a subset of processors it may run
- Many systems provide both soft and hard affinity
  - For example, Linux implements soft affinity, but it also provides the system call `sched_setaffinity()`, which supports hard affinity by allowing a thread to specify the set of CPUs on which it is eligible to run

Multiple-Processor Scheduling – Load Balancing

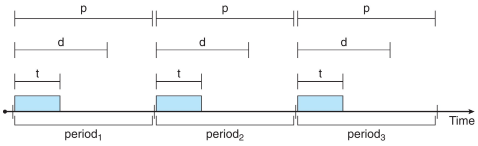
- Load balancing attempts to keep workload evenly distributed
  - On systems where each processor has its own private ready queue of eligible threads to execute
- There are two general approaches to load balancing
  - Push migration – a specific task periodically checks the load on each processor, and if it finds an imbalance, pushes task(s) from overloaded CPU to idle or less-busy CPUs
  - Pull migration – idle processors pulls waiting task(s) from a busy processor
  - Push and pull migration need not to be mutually exclusive and are in fact often both implemented in parallel on load-balancing systems. the Linux CFS implement both techniques
- Load balancing often counteracts the benefits of processor affinity - natural tension between load balancing and minimizing memory access times
  - Thus, scheduling algorithms for modern multicore NUMA systems have become quite complex

Real-Time CPU Scheduling

- Real-time scheduling demands performance guarantee – predictability
- Hard real-time systems – have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all
- Soft real-time systems – provide no guarantee as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes
- The scheduler for a real-time operating system must support a priority based algorithm with preemption

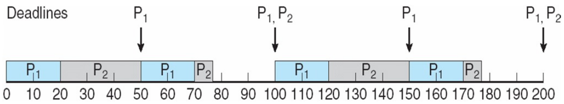
Priority-based Scheduling

- Note that providing a preemptive, priority-based scheduler only guarantees soft real-time functionality. Processes have the characteristics: periodic ones require CPU at constant intervals (periods)
  - Has processing time *t*, deadline *d*, period *p* , in which $0 \le t \le d \le p$
  - The rate of a periodic task is 1/*p*
  - A process may have to announce its deadline requirements to the scheduler. The scheduler decides whether to admit the process or not depending on whether it can guarantee that the process will complete on time (by its deadline)



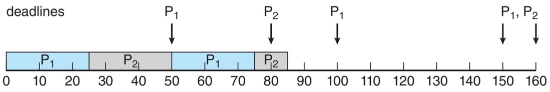
Rate Montonic Scheduling

- A static priority is assigned based on the inverse of its period
  - Shorter (longer) period = higher (lower) priority;
  - The rationale is to assign a higher priority to tasks requiring CPU more often
- Suppose P_1 has a period of 50 (also deadline), and processing time 20. P_2 has a period of 100 (also deadline), and processing time 35.
  - The deadline for each process requires that it complete its CPU burst by the start of its next period. Since 50 < 100, P_1 is assigned a higher priority than P_2
  - The CPU utilization of a process P_i as the ratio of its burst to its period t_i/P_i the CPU utilization of P_1 is 20/50 = 0.40 and that of P_2 is 35/100 = 0.35, so the total CPU utilization of 75 percent



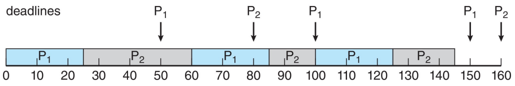
Missed Deadlines with Rate-Monotonic Scheduling

- Suppose P_1 has a period of 50 (also deadline), and processing time 25. P_2 has a period of 80 (also deadline), and processing time 35.
  - Since 50 < 80, P_1 is assigned a higher priority than P_2
  - The CPU utilization of P_1 is 25/50 = 0.50 and that of P_2 is 35/80 = 0.44, for a total CPU utilization of 94 percent
- Process P2 misses the deadline (80) by finishing at time 85



Earliest Deadline First Scheduling (EDF)

- Earliest-deadline-first (EDF) scheduling assigns priorities dynamically according to the deadline
  - the earlier (later) the deadline, the higher (lower) the priority
- Consider the same example, where P1 has a period of 50 (also deadline), and processing time 25. P2 has a period of 80 (also deadline), and processing time 35.



Rate-monotonic vs. EDF Scheduling

- The rate-monotonic scheduling algorithm schedules periodic tasks using a **static priority** policy with preemption
- The rate-monotonic scheduling is considered to be optimal in that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities.
- Unlike the rate-monotonic algorithm, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst. The only requirement is that a process announce its deadline to the scheduler when it becomes runnable
- EDF scheduling is theoretically optimal - it can schedule processes such that each process can meet its deadline requirements and CPU utilization will be 100 percent
  - In practice, however, it is impossible to achieve this level of CPU utilization due to the cost of context switching between processes and interrupt handling

---

Algorithm Evaluation

- Selecting CPU-scheduling algorithm in practice can be difficult – as there are many scheduling algorithms, each with its own set of parameters
- The first problem is defining the criteria to be used in selecting an algorithm - often defined in terms of CPU utilization, response time, or throughput
- Determine criteria – the criteria may include several measures with their relative importance, such as
  - Maximizing CPU utilization under the constraint that the maximum response time is 300 milliseconds
  - Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time

---

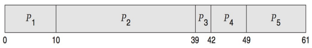
Deterministic Modeling

- Deterministic modeling takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Deterministic modeling is simple and fast. It gives us exact numbers, to compare algorithms. However, it requires exact numbers for input, and its answers apply only to those cases
- How processes run vary from time to time, so there is no static set of processes (or times) to use for deterministic modeling
- Consider 5 processes arriving at time 0:

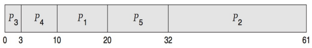
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

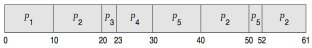
- For each algorithm, calculate the average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:



  - Non-preemptive SFJ is 13ms:



  - RR is 23ms:



Queueing Analysis

- Though the actual numbers (e.g. process arrival time, CPU or I/O bursts) vary from time (system) to time (system), the distributions of CPU and I/O bursts, and process arrival-time can be possibly measured and then approximated or simply estimated
- The computer system can be described as a network of servers, and each server has a queue of waiting processes.
  - The CPU is a server with its ready queue, I/O system with its device queues
  - Commonly use the *exponential* distribution, and described by mean
- Knowing arrival rates and service rates, we can compute the utilization, average queue length, average wait time, and so on.
- This area of study is called queueing-network analysis
- Queueing analysis can be useful in comparing scheduling algorithms, but the classes of algorithms and distributions that can be handled are very limited. Often the assumptions for the mathematical models to be tractable are unrealistic in practice

---

Queueing Models

- Mathematical approach for handling stochastic workloads
- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's Formula – in steady state (there are mathematical assumptions for this to hold, e.g., arrival rate must be smaller than service rate), 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

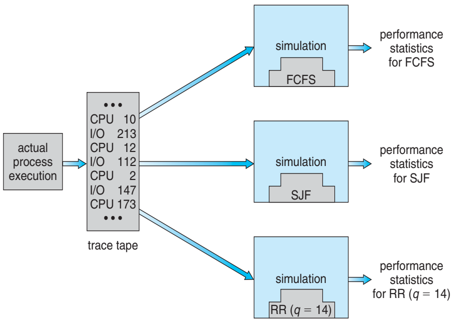
---

Simulations

- Queueing models is restricted to a few known distributions
- Running simulations involves programming a model of the computer system – which is more accurate
- As the clock 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.
- The data to drive the simulation can be generated in several ways
  - Random number generator according to probability distributions - distributions can be defined mathematically (uniform, exponential, Poisson) or empirically
  - Trace files to monitor the real system and record the sequence of actual events

---

Evaluation of CPU Schedulers by Simulation



Implementation

- Even simulations have limited accuracy
- Build a system which allows actual algorithms to run with real data set – more flexible and general.
- Implementing a new scheduler and test in real systems has difficulties:
  - This incurs high cost (coding the new scheduler), and high risk (e.g., potentially introducing new bugs)
  - Environments also changes constantly
- Most flexible scheduling algorithms are those that can be altered by the system managers so that they can be tuned for a specific application
  - A system supporting graphical applications or web (file) service, for instance, may have different scheduling needs
  - Many UNIX systems allow the system manager to fine-tune the scheduling parameters for a particular system configuration
- APIs can be used to modify priority of a process or thread – improving performance of specific application, not overall application performance
  - The Java, POSIX, and Windows APIs provide such functions

End of Chapter 5