COMP3511 Operating Systems

Topic 5: CPU Scheduling

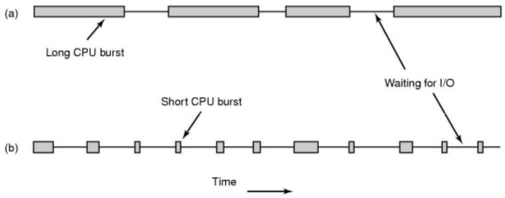
Dr. Desmond Tsoi

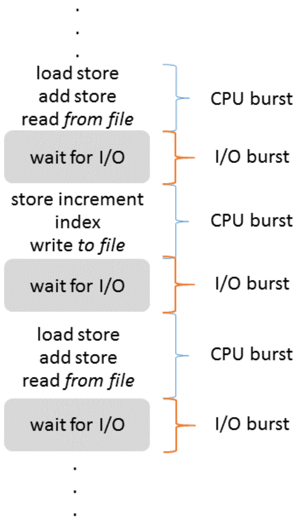
Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Acknowledgment: The lecture notes are based on various sources on the Internet

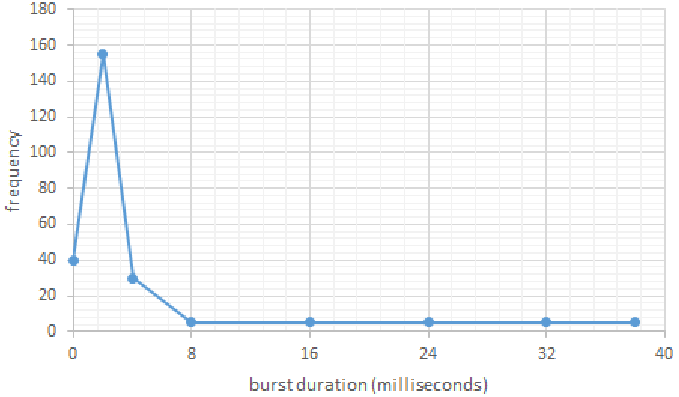
Motivation

- Many programs have a characteristics CPU-I/O burst cycle, i.e. alternating phases of CPU activity and I/O inactivity
  (a) CPU-bound programs have fewer, longer CPU bursts
  (b) I/O-bound programs have more, shorter CPU bursts





Histogram of CPU-burst Times



- The extensive measurement of CPU bursts shows that the CPU burst duration is consisted of a large number of short CPU bursts and a small number of long CPU bursts
- The curve is generally characterized as exponential or hyper-exponential

CPU Scheduling

- Scheduling problem:
  - Have K jobs ready to run
  - Have N $\geq$ 1 CPUs
  - Which jobs to assign to which CPU(s)?

CPU scheduling

CPU scheduling (a.k.a. short-term scheduling) is the act of selecting the next process for the CPU to "service" once the current process leaves the CPU idle

- Many algorithms for making the selection of the next process
- General rule
  - Keep the CPU busy: an idle CPU is a wasted CPU
    Major source of CPU idleness: I/O (or waiting for it)

CPU Scheduling

- Implementation-wise: CPU scheduling manipulates the operating system's various PCB queues
- CPU scheduling manipulates the queues in the following 4 circumstances
  1. With process switches from running to waiting state (I/O request or wait())
  2. With process switches from running to ready state (interrupts)
  3. With process switches from waiting to ready (the completion of I/O)
  4. With process terminates
- Scheduling under 1 and 4 is non-preemptive (i.e. non-stop or non-pause). The process releases the CPU voluntarily
- Scheduling under 2 and 3 is preemptive (i.e. stop or pause), which may cause race condition (more on this discussed in the next topic)

Dispatcher

Dispatcher

Dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler

- Dispatch latency: time it takes for the dispatcher to stop one process and start another running
- The dispatcher should be as fast as possible, since it is invoked during each process switch
- Dispatching involves the following
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program

Scheduling Criteria

- CPU utilisation: Keep the CPU as busy as possible (maximisation)
- Throughput: Number of processes that complete their execution per time unit (Maximisation)
- Turnaround time: Amount of time to execute a particular process. The interval from the time of submission of a process to the time of completion (Minimisation)
- Waiting time: The sum of the periods spent waiting in the ready queue (Minimisation)
  - Notice that CPU scheduling algorithm does not affect the total amount of time during which a process executes (on CPU) and does I/O. It affects only the amount of time that a process spends waiting in ready queue
- Response time: The amount of time it takes from when a request was submitted until the first response is produced, not output, in a time-sharing or interactive environment (Minimisation)

Process Scheduling Algorithms

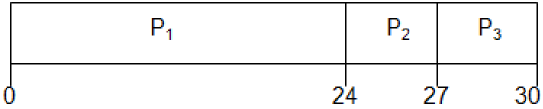
- First-Come, First-Served (FCFS)
- Shortest-Job-First (SJF)
- Shortest-Remaining-Time-First
- Priority Scheduling
- Round Robin (RR)
- Multilevel Queue

Scheduling Algorithm - First-Come, First-Served (FCFS)

- Processes are dispatched according to the arrival time on the ready queue

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

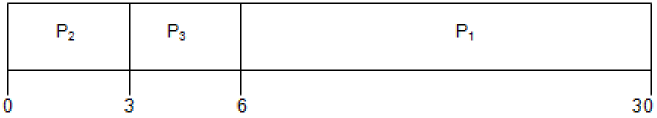


- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27) / 3 = 17$
- The FCFS scheduling algorithm is non-preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O

Scheduling Algorithm - First-Come, First-Served (FCFS) (Cont'd)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- 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
- Convoy effect: Short process behind long process (Consider one CPU-bound and many I/O-bound processes)

Scheduling Algorithm - Shortest-Job-First (SJF)

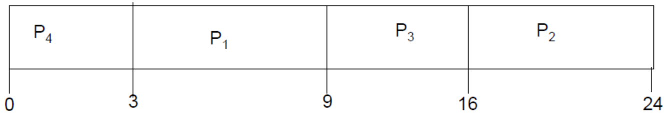
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
  - If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie
- SJF is optimal, i.e. gives minimum average waiting time for a given set of processes
  - Moving a short process before a long process decreases the waiting time of the short process more than it increases the waiting time for the long process. Consequently, the average (or the total) waiting time decreases

Scheduling Algorithm - Shortest-Job-First (SJF)

Suppose the length of next burst time of 4 processes are as follows:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- The Gantt chart for the schedule is



- Waiting time for $P_1 = 3$; $P_2 = 16$; $P_3 = 9$, $P_4 = 0$
- Average waiting time: $(3 + 16 + 9 + 0) / 4 = 7$

Determining Length of Next CPU Burst

- The difficulty of SJF is knowing the length of the next CPU request
- We can only estimate the length. It 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. α, $0 \leq \alpha \leq 1$, commonly, α is set to $\frac{1}{2}$
    4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
- t_n contains the most recent information, τ_n stores the past history, α controls the relative weight of recent and the past history
    - $\alpha = 0$, $\tau_{n+1} = \tau_n$, recent history does not count
    - $\alpha = 1$, $\tau_{n+1} = t_n$, only the actual last CPU burst counts

To understand the behaviour of the exponential average, we expand the formula as follows:

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

Since both α and (1-α) are less than or equal to 1, each successive term has less weight than its predecessor, i.e., the impact on the estimate is reduced exponentially

Prediction of the Length of the Next CPU Burst

Example:

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

- Suppose $t_1 = 6$, $\alpha = \frac{1}{2}$, $\tau_1 = 10$

$$\begin{aligned} \tau_2 &= \alpha t_1 + (1-\alpha)\tau_1 \\ \tau_2 &= \frac{1}{2}(6) + (1-\frac{1}{2})10 = 8 \end{aligned}$$

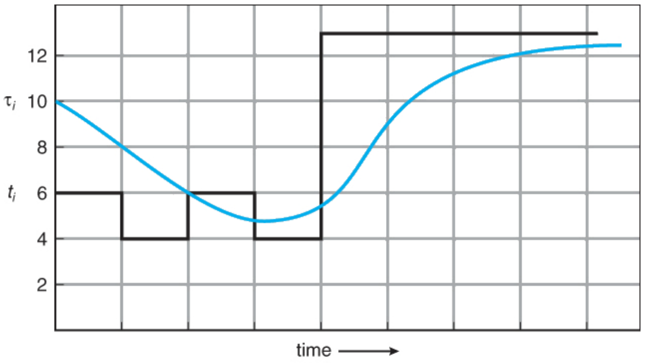
- Suppose $t_2 = 4$, $\alpha = \frac{1}{2}$, $\tau_2 = 8$

$$\begin{aligned} \tau_3 &= \alpha t_2 + (1-\alpha)\tau_2 \\ \tau_3 &= \frac{1}{2}(4) + (1-\frac{1}{2})8 = 6 \end{aligned}$$

- Suppose $t_3 = 6$, $\alpha = \frac{1}{2}$, $\tau_3 = 6$

$$\begin{aligned} \tau_4 &= \alpha t_3 + (1-\alpha)\tau_3 \\ \tau_4 &= \frac{1}{2}(6) + (1-\frac{1}{2})6 = 6 \end{aligned}$$

Prediction of the Length of the Next CPU Burst (Cont'd)



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

Note: Black line refers to the actual CPU bursts, while blue line refers to the predicted values, i.e., guessed values

Preemptive and Non-preemptive SJF

- SJF can be either preemptive and non-preemptive
- The choice arises when a process arrives at the ready queue while another process is still executing on the CPU
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. If this is the case, the next CPU burst length of this newly arrived process will be shorter than the CPU burst lengths of all processes currently in the ready queue (the SJF determines)
- A preemptive SJF will preempt the currently running process (returning to the ready queue with the remaining CPU time, whereas a non-preemptive SJF will allow the currently running process to finish its CPU burst)
- Preemptive SJF scheduling algorithm is sometime called shortest-remaining-time first scheduling

Scheduling Algorithm: Shortest-Remaining-Time-First

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

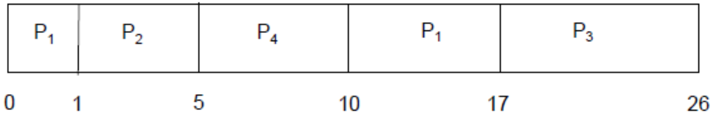
Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Burst time for each process at different time

Process	t = 0	t=1	t=2	t=3	t=5	t=10	t=17	t=26
P_1 AT = 0	8	7	7	7	7	7	0	0
P_2 AT = 1	4	4	3	2	0	0	0	0
P_3 AT = 2	9	9	9	9	9	9	9	0
P_4 AT = 3	5	5	5	5	5	0	0	0

Scheduling Algorithm: Shortest-Remaining-Time-First (Cont'd)

- Preemptive SJF Gantt Chart



- Waiting time for
  - $P_1 = (0 - 0) + (10 - 1) = 9$
  - $P_2 = (1 - 1) = 0$
  - $P_3 = (17 - 2) = 15$
  - $P_4 = (5 - 3) = 2$
- Average waiting time $= (9 + 0 + 15 + 2)/4 = 6.5$ msec

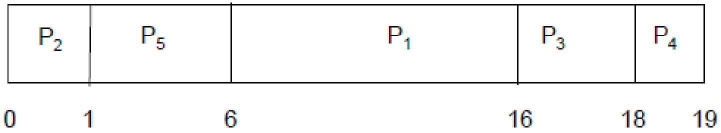
Scheduling Algorithm: Priority Scheduling

- A priority number (an integer) is associated with each process
- The CPU is allocated to the process with the highest priority (small integer = highest priority)
- SJF is priority scheduling where priority is the inverse of the predicted next CPU burst time
- Priority scheduling can be either preemptive or non-preemptive
- Major problem is starvation, i.e., lower priority processes may never execute
- Solution is aging, i.e. as time progresses, increase the priority of the process

Scheduling Algorithm: 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 = $(6 + 0 + 16 + 18 + 1)/5 = 8.2$ msec

Can we have different arrival time for each process? If so, how to compute the average waiting time?

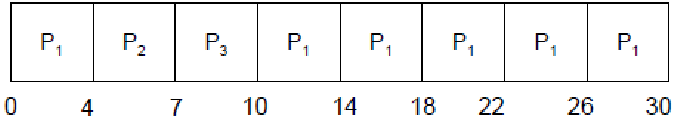
Scheduling Algorithm: Round Robin (RR)

- A round-robin scheduling is designed especially for time-sharing systems. It is similar to FCFS, but preemption is added to restrict the maximum amount of time that a process can occupy the CPU, thus enabling 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 (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

Scheduling Algorithm: RR with Time Quantum = 4

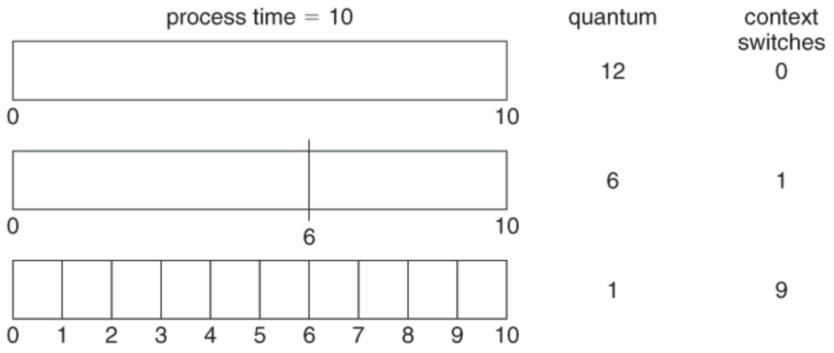
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is



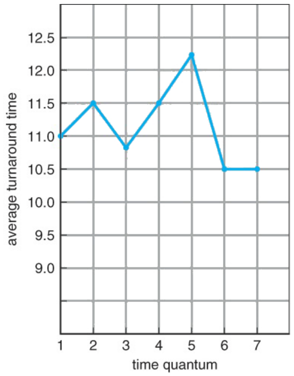
- Typically, higher average turnaround time (i.e., amount of time taken to get the job done) than SJF, but better response time
- q should be large compared to context switch time
- q is usually 10ms to 100ms, context switch $<$ 10 microseconds

Time Quantum vs. Context Switch Time



A smaller time quantum increases the number of context switches

Turnaround Time Varies With the Time Quantum



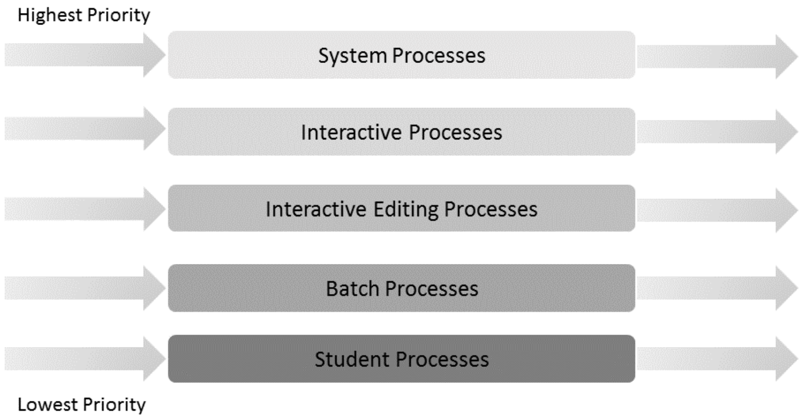
Process	Time
P_1	6
P_2	3
P_3	1
P_4	7

- 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 next CPU burst in a single quantum
- The time quantum cannot be too big, in which RR degenerates to an FCFS policy
- A rule of thumb: 80% of CPU bursts should be shorter than the time quantum q

Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.:
  - Foreground (interactive)
  - Background (batch)
- Processes are permanently assigned to one queue when they enter the system, based on some property of the process, such as memory size, priority, or process type
- Each queue has its own scheduling algorithm:
  - Foreground: RR
  - Background: FCFS
- Scheduling must be done among the queues
  - Fixed-priority preemptive 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 and 20% to background in FCFS

Multilevel Queue Scheduling

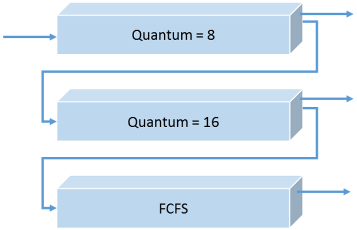


Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - 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

Multilevel Feedback Queue (Cont'd)

- Three queues
  - Q_0 - RR with time quantum 8 ms
  - Q_1 - RR with time quantum 16 ms
  - Q_2 - FCFS
- Scheduling - preemptive
  - A new job enters queue Q_0
    - ★ When it gains CPU, job receives 8 milliseconds
    - ★ If it does not finish in 8 ms, job is moved to queue Q_1
  - At Q_1 job receives 16 additional milliseconds
    - ★ If it still does not complete, it is preempted and moved to queue Q_2
  - A process in queue 1 or 2 will be preempted by a process arriving for queue 0

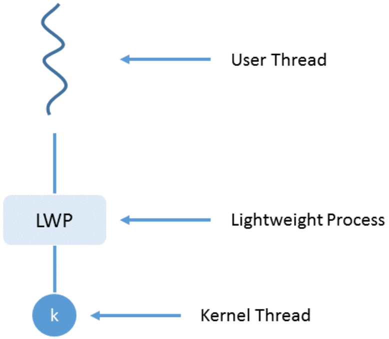


Others

- Thread Scheduling
- Multiple-Processor Scheduling
- NUMA and CPU Scheduling
- Load Balancing
- Multicore Processors
- Multithreaded Multicore System
- CPU-scheduling Algorithm Evaluation

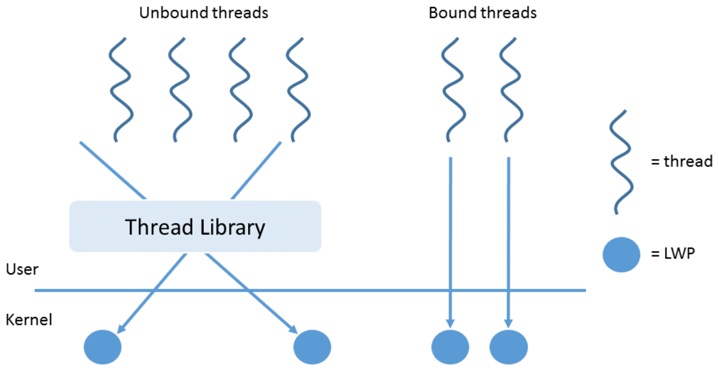
Thread Scheduling

- When the OS supports threads, the kernel-level threads are being scheduled, not processes
  - User-level threads are managed by a thread library
- To run on CPU, user-level threads must ultimately be mapped to an associated kernel-level threads, although this mapping may be indirect and may use a lightweight process (LWP)
  - LWPs bridge the user level and the kernel level
  - LWP appears to be a virtual processor on which process can schedule user threads to run on it
  - Each LWP attached to kernel thread



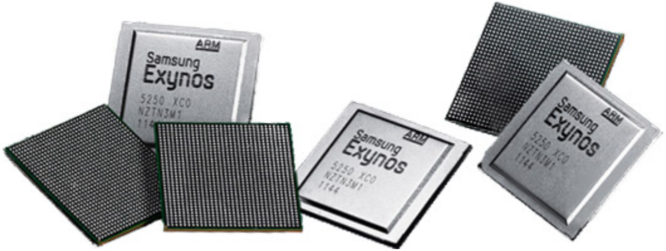
Thread Scheduling (Cont'd)

- Unbound Threads / Process-Contention Scope (PCS)
  - Used in many-to-one and many-to-many models
  - The association of these threads with LWPs is managed by the thread library (i.e., threads are scheduled by the thread library)
- Bound Threads / System-Contention Scope (SCS)
  - Used in one-to-one model
  - A bound thread is permanently attached to an LWP



Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available, because now there is more than one CPU which must be kept busy and in effective use at all times
- Multi-processor systems may be heterogeneous (different kinds of CPUs), or homogeneous (all the same kind of CPU)
  - Here, we only focus on homogeneous processors within a multiprocessor, i.e., cores of the same kind

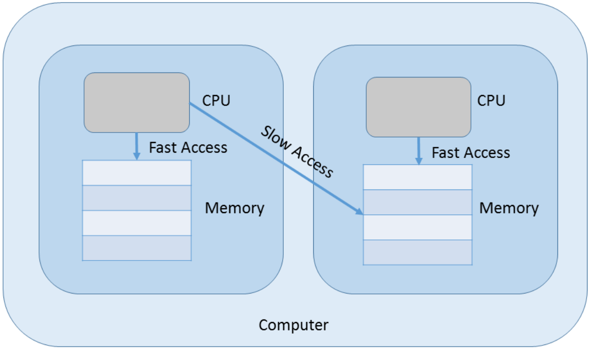


Multiple-Processor Scheduling (Cont'd)

- Approaches to multiple-processor scheduling
  - Asymmetric multiprocessing - one processor is the master, controlling all activities and running all kernel code, while the other runs only user code
  - Symmetric multiprocessing (SMP) - each processor schedules its own jobs, either from common ready queue or from separate ready queues for each processor
    - ★ Currently, this is more common (e.g., Windows 2000, XP, Solaris, Linux, and Mac OS X)
- Processor affinity - process has affinity for processor or which it is currently running, especially the cache content
  - Soft affinity - the OS attempt to keep a process running on the same processor, not guaranteeing it
  - Hard affinity - allow a process to specify a subset of processors on which it may run

Non-Uniform Memory Access (NUMA) & CPU Scheduling

- Main memory architecture of a system can affect processor affinity, if particular CPUs have faster access to memory on the same chip or board than to other memory loaded elsewhere (Non-Uniform Memory Access, NUMA)
- As shown below, if a process has an affinity for a particular CPU, then it should preferentially be assigned memory storage in "local" fast access



Load Balancing

- Obviously an important goal in a multiprocessor system is to balance the load between processors, so that one processor will not be sitting idle while another is overloaded
- System using a common ready queue are naturally self-balancing, and do not need any special handling. Most systems, however, maintain separate ready queues for each processor
- Balancing can be achieved through either push migration or pull migration
  - Push migration: Involves a separate process that runs periodically, and moves processes from heavily loaded processors onto less loaded ones
  - Pull migration: Involves idle processors taking processes from the ready queues of other processors
- Push and pull migration need not to be mutually exclusive and are in fact often implemented in parallel on load-balancing systems

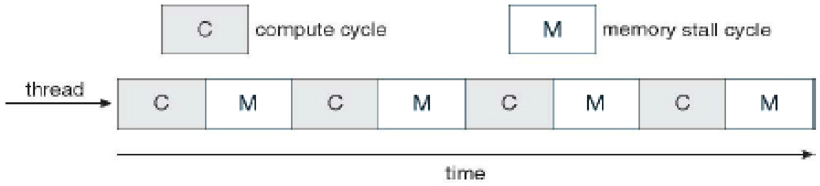
Note that moving processes from processor to processor to achieve load balancing works against the principle of processor affinity

Multicore Processors

- Traditional SMP required multiple CPU chips to run multiple kernel threads concurrently
- Recent trends are to put multiple CPUs (cores) onto a single chip, which appear to the system as multiple processors
  - Faster and consumes less power

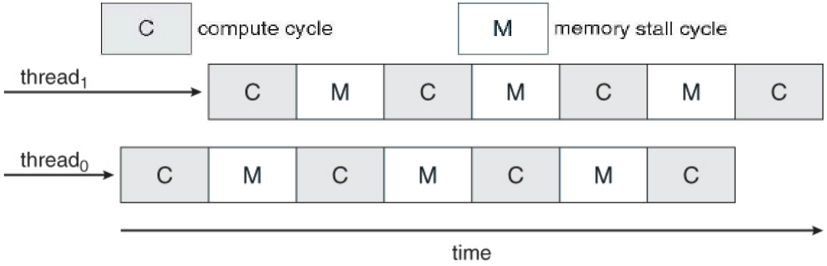
Memory stall

A situation when a processor accesses memory, it spends a significant amount of time waiting for the data to become available, due to various reasons such as cache miss



Multithreaded Multicore System

- The scheduling can takes advantage of memory stall to make progress on another thread while memory retrieval happens
  - If one thread stalls while waiting for memory, the core can switch to another thread. This becomes a dual-thread processor core, or two logical processors
  - A dual-threaded, dual-core system presents four logical processors to the operating system



Algorithm Evaluation

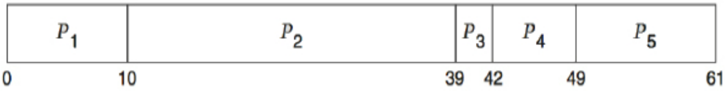
- How to select CPU-scheduling algorithm for a particular system?
  - Determine what criteria are to be used, what goals are to be targeted, and what constraints if any must be applied
  - Then analyze different algorithms and determine a "best choice"
- Deterministic modeling
  - A type of analytic evaluation
  - Takes a particular pre-determined workload and defines the performance of each algorithm for that workload

Example: Consider 5 processes arriving at time 0

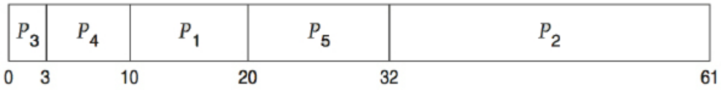
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Example: Deterministic Evaluation

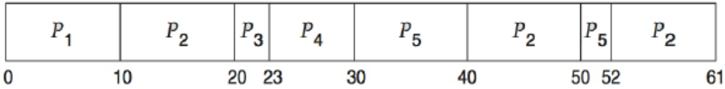
- For each algorithm, calculate minimum average waiting time
  - FCS is 28ms



  - Non-preemptive SFJ is 13ms



  - RR is 23ms



- Simple and fast, but requires exact numbers for input, applies only to those inputs

Queuing Models

- Specific process data is often not available, particularly for future times
- However a study of historical performance can often produce statistical descriptions of certain important parameters, such as
  - the rate at which new processes arrive
  - the ratio of CPU bursts to I/O times
  - the distribution of CPU burst times and I/O burst times, etc
- Armed with those probability distributions and some mathematical formulas, it is possible to calculate certain performance characteristics of individual waiting queues

Little's Formula

Little's Formula

It says that for an average queue length of n, with an average waiting time in the queue of W, and an average arrival of new jobs in the queue of λ, then these three terms can be related by

$$n = \lambda \times W$$

- Example: Suppose on average 7 processes arrive per second, and normally 14 processes in queue (i.e., $\lambda = 7$, n = 14), then average wait time per process is computed by

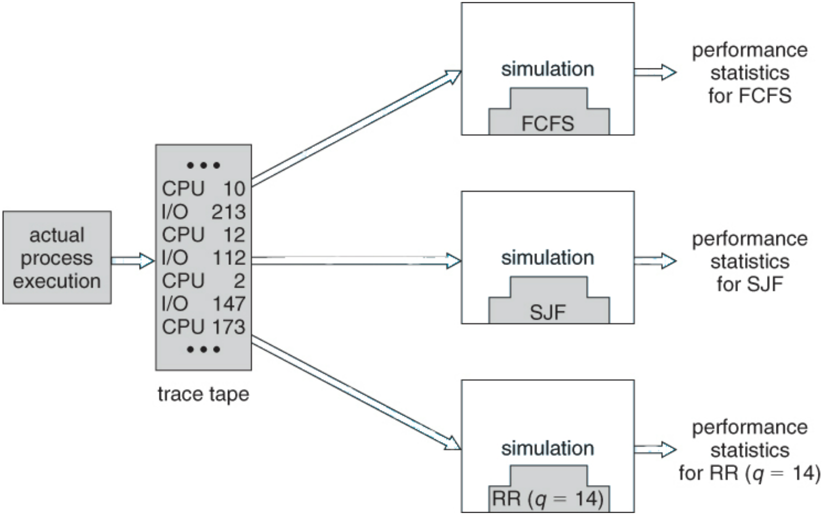
$$\begin{aligned} 4 &= 7 \times W \\ W &= 2 \texttt{ seconds} \end{aligned}$$

- Queuing models treat the computer as a network of interconnected queues, each of which is described by its probability distribution statistics and formulas such as Little's formula

Simulations

- Queuing models limited for a very few known distributions in order to compute the performance mathematically
- Simulations are more accurate and general
    - 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



Implementation

- The only real way to determine how a proposed scheduling algorithm is going to operate is to implement it on a real system
  - High cost, high risk
- The measured results may not be definitive, since system environments vary
- Most modern systems provide some capability for the system administrator to adjust scheduling parameters, either on the fly or as the result of a reboot or a kernel rebuild

That's all!

Any questions?