CPU Scheduling
(Topic 3)

홍 성 수

서울대학교 공과대학 전기 공학부
Real-Time Operating Systems Laboratory

CPU Scheduling (1)

- Resources fall into two classes:
  - Preemptible: Can take resource away, use it for something else, then give it back later.
    Examples: Processor or disk.
  - Non-preemptible: Once given, it can't be reused until process gives it back.
    Examples: File space, terminal
- Distinction is a little arbitrary, like (non-)breakable.

CPU Scheduling (2)

- Until now you have heard about processes:
  - Process implementation
  - Process synchronization/deadlock
  - Process communication
- From now on you'll hear about resources.
  - Resources are things operated upon by processes.
  - Example: CPU time, disk space, disk channel time.

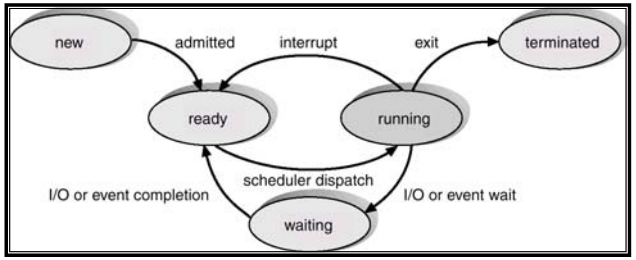
CPU Scheduling (3)

- OS makes two related kinds of decisions about resources:
  - Who gets what?
    Given a set of requests for resources, which processes should be given which resources in order to make most efficient use of the resources?
    Implication is that resources aren't easily preemptible.
  - How long can they keep it?
    When more resources are requested than can be granted immediately, in which order should they be serviced ?
    Examples:
      Processor scheduling: One processor, many processes.
      Memory scheduling in VM systems.
    Implication is that resource is preemptible.

CPU Scheduling (4)

- Resource #1 to examine: The processor
- Processes may be in any of three general scheduling states.
  - Running: Has the CPU.
  - Ready: Wants the CPU.
  - Waiting (Blocked): Waiting for some event:
    Disk I/O, message, semaphore, etc.

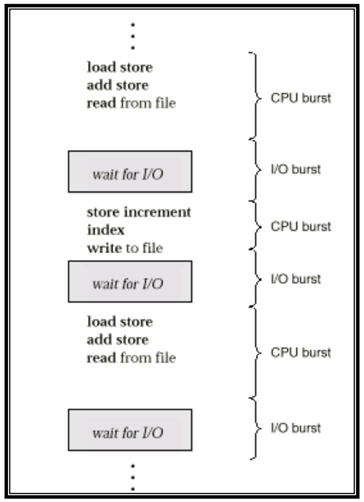
CPU Scheduling (5)



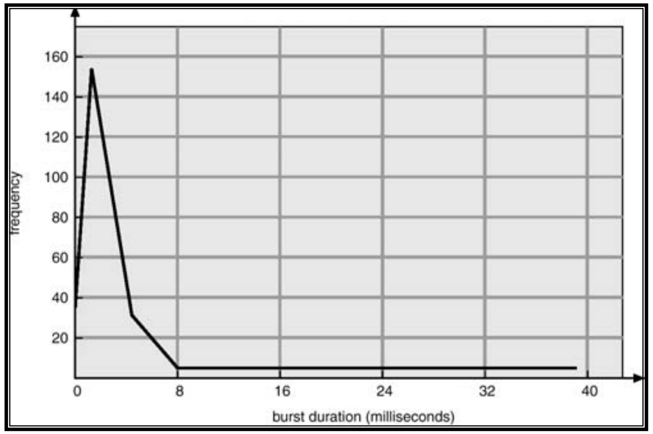
Basic Concepts

- Maximize CPU utilization with multiprogramming
- CPU–I/O burst cycle
  - Process execution consists of a *cycle* of CPU execution and I/O wait.
- CPU burst distribution

Alternating CPU and I/O Bursts



Histogram of CPU Burst Times



CPU Scheduler

- Selects one among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state.
  2. Switches from running to ready state.
  3. Switches from waiting to ready.
  4. Terminates.
- Scheduling under 1 and 4 is *nonpreemptive*.
- All other scheduling is *preemptive.*

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running.

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- 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, **not** output (for time-sharing environment)

Optimization Criteria

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

CPU Scheduling Policies

- Goals of scheduling disciplines:
  - Efficiency of resource utilization.
    Example: Keep CPU and disks busy.
  - Minimize overhead.
    Example: Reduce context switches.
  - Minimize response time.
  - Distribute cycles equitably.
- Scheduling disciplines:
  - FIFO, FCFS, LIFO, RR, STCF, etc.

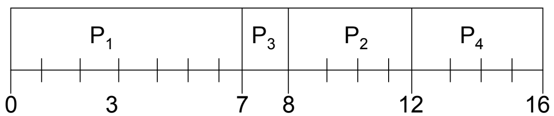
Shortest Job First (SJF) (1)

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

SJF (2): Nonpreemptive

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

- SJF (nonpreemptive)

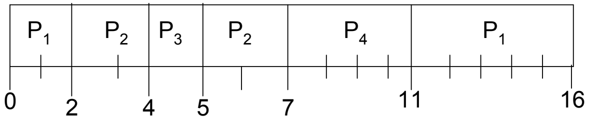


- Average waiting time = (0 + 6 + 3 + 7)/4 - 4

SJF (3): Preemptive

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

- SJF (preemptive)



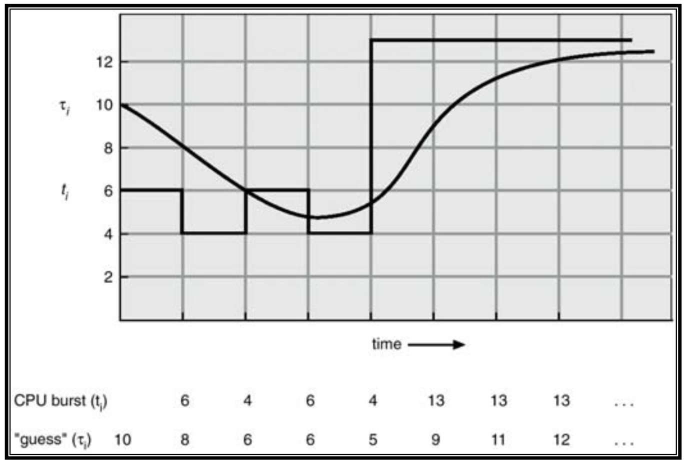
- Average waiting time = (9 + 1 + 0 +2)/4 - 3

Predicting Next CPU Burst Size

- Can only estimate the length.
- Can be done by using the length of previous CPU bursts, using exponential averaging.
  1. $t_n =$ actual lenght 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 :

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

Predicting Next CPU Burst Size



Exponential Averaging

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

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

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

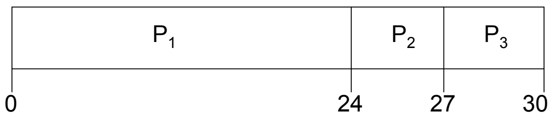
First In First Out (1)

- Run until finish.
  - Also called First Come Fist Served (FCFS).
  - In the simplest case this means uniprogramming.
  - Usually, “finished” means “blocked.”
    - One process can use CPU while another waits on a semaphore.
    - Go to the back of run queue when ready.
  - Problem: One process can monopolize CPU.
  - Solution: Limit maximum amount of time that a process can run without a context switch.
    - →This time is called a time slice.

First In First Out (2)

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:



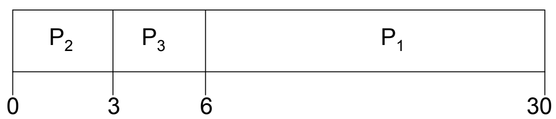
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

First In First Out (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

Round Robin (1)

- Round Robin: Run process for one time slice, then move to the back of queue.
  - Each process gets equal share of the CPU.
  - Most systems use some variant of this.
- What happens if the time slice isn’t chosen carefully:
  - Too long: A process can monopolize the CPU.
  - Too short: Too much context switch overhead.

Round Robin (2)

- Originally, Unix had 1 second time slices. Too long.
  Current systems have time slices of around 100 ms.
- Implementation of priorities:
  - Run highest-priority processes first.
  - Round-robin among processes of equal priority.
  - Re-insert process in run queue behind all processes of greater or equal priority.

Round Robin (3)

- Round-robin can produce bad results occasionally:
  <u>Consider 10 processes each requiring 100 time slices:</u>
  - Under round-robin they all take 1000 time slices to finish.
  - FIFO would average only 500 time slices.

  Round-robin is fair, but uniformly inefficient.
- How to optimized the average response time ?
  <u>STCF: Shortest time to completion first.</u>
  - Results in minimum average response time.

Round Robin (4)

- Example: 2 processes:
  - Proc 1: Run for 1 ms then wait for I/O for 10 ms.
  - Proc 2: No waiting, run continuously.

(1) Round-robin with a 100 ms time slice:

- I/O process runs at 1/10th speed.
- I/O devices are only 10% utilized.

(2) Round-robin with a 1 ms time slice:

- CPU bound process gets interrupted 9 times unnecessarily for each valid interrupt.

Exponential Queues (1)

- STCF works quite nicely.
- Unfortunately, STCF requires knowledge of the future.
  - Must use past behavior to predict future behavior.
  - Example: Long-running process will probably take a long time more.
- Use the dispatcher's priority mechanisms to disfavor long running processes.

Exponential Queues (2)

- Exponential Queues (or Multi-Level Feedback Queues):
  – Give newly runnable processes a high priority and a very short time slice.
    If process uses up the time slice without blocking:
    - Decrease its priority by 1.
    - Double time slice for next time.

    Example:
    - PROC 1 runs for 1 ms and blocks.
    - PROC 2 runs for 1 ms and doesn't block.
      PROC 2 gets priority-1, time slice 2.

Exponential Queues (3)

- PROC 2 runs for 2 ms and doesn't block.
  PROC 2 gets priority-2, time slice 4.
- PROC 2 runs for 4 ms and doesn't block.
  PROC 2 gets priority-3, time slice 8.
- PROC 2 runs for 3 ms and gets preempted.
- PROC 1 runs for 1 ms and blocks.
- PROC 2 runs for 5
- ……
- PROC 1 runs for 1 ms and blocks.
- PROC 2 runs until PROC 1 is ready and preempts it.

– Techniques like this one are called adaptive.
  Common in interactive systems.
– The CTSS systems (MIT around 1962) was the first to use exponential queues.

Fair Share Scheduling

- Fair share scheduling (similar to what's implemented in Unix):
  – Keep history of recent CPU usage for each process.
  – Give highest priority to process that has used the least CPU time recently.
    Highly interactive jobs, like editors, will use little CPU and get high priority.
    CPU-bound jobs, like compilers, will get lower priority.
  – Can adjust priorities by changing "billing factors" for processes.
    E.g., to make high-priority process, only use half its recent CPU usage in computing history.

CPU Scheduling

- Summary
  – In principle, scheduling algorithms can be arbitary, Since the system should produce the same results in any event.
  – However, the algorithms have strong effects on the system's overhead, efficiency, and response time.
  – The best schemes are adaptive. To do absolutely best, se'd have to be able to predict the future.
- Best scheduling algorithms tend to give highest priority to the processes that need the least !