CSC 539: Operating Systems Structure and Design

Spring 2005

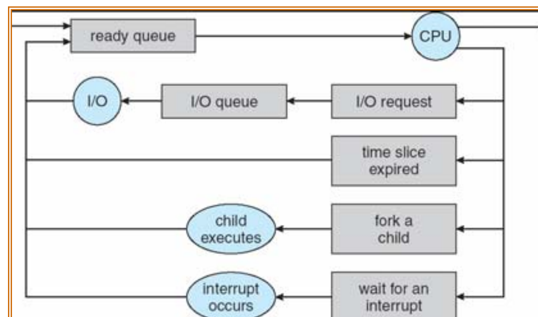
CPU scheduling

- historical perspective
- CPU-I/O bursts
- preemptive vs. nonpreemptive scheduling
- scheduling criteria
- scheduling algorithms: FCFS, SJF, Priority, RR, multilevel
- multiple-processor & real-time scheduling
- real-world systems: BSD UNIX, Solaris, Linux, Windows NT/2K, …

CPU scheduling

Recall : short-term scheduler (CPU scheduler) selects from among the ready processes in memory and allocates the CPU to one of them

- only one process can be running in a uniprocessor system
- a running process may be forced to wait (e.g., for I/O or some other event)
- multiprocessing revolves around the system's ability to fill the waiting times of one process with the working times of another process
- *scheduling is a fundamental operating system function*



Historical perspective

50's: process scheduling was not an issue

- either single user or batch processing

60's – early 80's: multiprogramming and timesharing evolved

- process scheduling needed to handle multiple users, swap jobs to avoid idleness

mid 80's – early 90's: personal computers brought simplicity (& limitations)

- DOS and early versions of Windows/Mac OS had NO sophisticated CPU scheduling algorithms
- one process ran until the user directed the OS to run another process

mid 90's – present: advanced OS's reintroduced sophistication

- graphical interfaces & increasing user demands required multiprocessing

CPU-I/O bursts

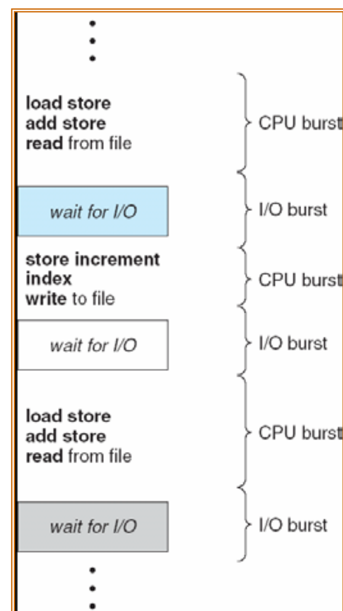
process execution consists of a *cycle* of CPU execution and I/O wait

- different processes may have different distributions of bursts

  CPU-bound process: performs lots of computations in long bursts, very little I/O

  I/O-bound process: performs lots of I/O followed by short bursts of computation

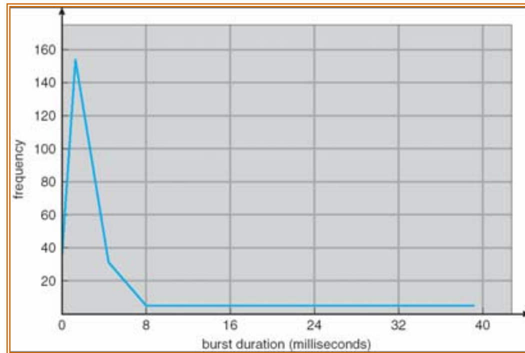
- ideally, the system admits a mix of CPU-bound and I/O-bound processes to maximize CPU and I/O device usage



Burst distribution

CPU bursts tends to have an exponential or hyperexponential distribution

- there are lots of little bursts, very few long bursts
- a typical distribution might be shaped as here:



What does this distribution pattern imply about the importance of CPU scheduling?

Mechanism vs. policy

a key principle of OS design:

- separate mechanism (how to do something) vs. policy (what to do, when to do it)

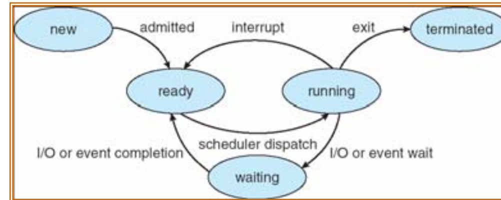
an OS should provide a context-switching *mechanism* to allow for processes/threads to be swapped in and out

a process/thread scheduling *policy* decides when swapping will occur in order to meet the performance goals of the system

Preemptive vs. nonpreemptive scheduling

CPU scheduling decisions may take place when a process:

1. switches from running to waiting state
   e.g., I/O request
2. switches from running to ready state
   e.g., when interrupt or timeout occurs
3. switches from waiting to ready
   e.g., completion of I/O
4. terminates



scheduling under 1 and 4 is *nonpreemptive*

- once a process starts, it runs until it terminates or willingly gives up control
  simple and efficient to implement – few context switches
  examples: Windows 3.1, early Mac OS

all other scheduling is *preemptive*

- process can be "forced" to give up the CPU (e.g., timeout, higher priority process)
  more sophisticated and powerful
  examples: Windows 95/98/NT/XP, Mac OS-X, UNIX

CPU scheduling criteria

CPU utilization: (time CPU is doing useful work)/(total elapsed time)

- want to keep the CPU as busy as possible
- in a real system, should range from 40% (light load) to 90% (heavy load)

throughput: # of processes that complete their execution per time unit

- want to complete as many processes/jobs as possible
- actual number depends upon the lengths of processes (shorter → higher throughput)

turnaround time: average time to execute a process

- want to minimize time it takes from origination to completion
- again: average depends on process lengths

waiting time: average time a process has spent waiting in the ready queue

- want to minimize time process is in the system but not running
- less dependent on process length

response time: average time between submission of request and first response

- in a time-sharing environment, want to minimize interaction time for user
- *rule-of-thumb: response time of 0.1 sec req'd to make interaction seem instantaneous*
  response time of 1.0 sec req'd for user's flow of thought to stay uninterrupted
  response time of 10 sec req'd to keep user's attention focused

CPU scheduling criteria (cont.)

in a batch system, throughput and turnaround time are key

in an interactive system, response time is usually most important

CPU scheduling may also be characterized w.r.t. fairness

- want to share the CPU among users/processes in some equitable way
- what is fair? communism? socialism? capitalism?

minimal definition of fairness: freedom from starvation

- starvation = indefinite blocking
- want to ensure that every ready job will eventually run
  (assuming arrival rate of new jobs ≤ max throughput of the system)

- note: fairness is often at odds with other scheduling criteria
  e.g., can often improve throughput or response time by making system less fair
  analogy?

Scheduling algorithms

First-Come, First-Served (FCFS)

- CPU executes job that arrived earliest

Shortest-Job-First (SJF)

- CPU executes job with shortest time remaining to completion*

Priority Scheduling

- CPU executes process with highest priority

Round Robin (RR)

- like FCFS, but with limited time slices

Multilevel queue

- like RR, but with multiple queues for waiting processes (i.e., priorities)

Multilevel feedback queue

- like multilevel queue, except that jobs can migrate from one queue to another

First-Come, First-Served (FCFS) scheduling

the ready queue is a simple FIFO queue

- when a process enters the system, its PCB is added to the rear of the queue
- when a process terminates/waits, process at front of queue is selected

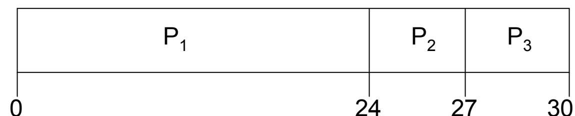
FCFS is nonpreemptive

- once a process starts, it runs until it terminates or enters wait state (e.g., I/O)
- average waiting and turnaround times can be poor
- in general, nonpreemptive schedulers perform poorly in a time sharing system since there is no way to stop a CPU-intensive process (e.g., an infinite loop)

FCFS example

Process	Arrival Time	Burst Time
P_1	0	24
P_2	2	3
P_3	4	3

Gantt Chart for the schedule is:



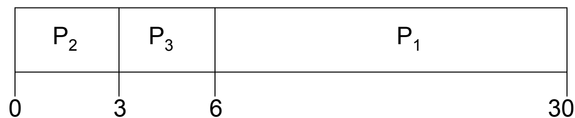
average waiting time: (0 + 22 + 23)/3 = 15

average turnaround time: (24 + 25 + 26)/3 = 25

FCFS example (cont.)

Process	Arrival Time	Burst Time
P_2	0	3
P_3	2	3
P_1	4	24

Gantt Chart for the schedule is:



average waiting time: (0 + 1 + 2)/3 = 1 *MUCH BETTER*

average turnaround time: (3 + 4 + 26)/3 = 11 *MUCH BETTER*

Convoy effect : short process behind long process degrades wait/turnaround times

Shortest-Job-First (SJF) scheduling

more accurate name would be Shortest Next CPU Burst (SNCB)

- associate with each process the length of its next CPU burst (???)
- use these lengths to schedule the process with the shortest time

SJF can be:

- *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
  - – known as Shortest-Remaining-Time-First (SRTF)

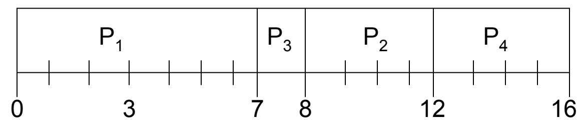
if you can accurately predict CPU burst length, SJF is optimal

- it minimizes average waiting time for a given set of processes

Nonpreemptive SJF example

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Gantt Chart for the schedule is:

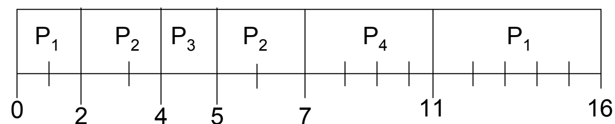


average waiting time: (0 + 6 + 3 + 7)/4 = 4
average turnaround time: (7 + 10 + 4 + 11)/4 = 8

Preemptive SJF example

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Gantt Chart for the schedule is:



average waiting time: (9 + 1 + 0 +2)/4 = 3
average turnaround time: (16 + 5 + 1 + 6)/4 = 7

SJF: predicting the future

in reality, can't know precisely how long the next CPU burst will be

- consider the Halting Problem

can estimate the length of the next burst

- simple: same as last CPU burst
- more effective in practice: exponential average of previous CPU bursts

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

where: τ_n = predicted value for nth CPU burst

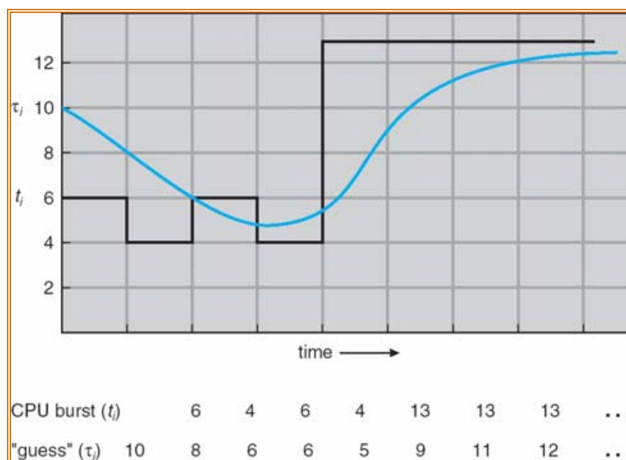
t_n = actual length for nth CPU burst

α = weight parameter ($0 \le \alpha \le 1$, larger α emphasizes last burst)

Exponential averaging

consider the following example, with α = 0.5 and τ_0 = 10

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



Priority scheduling

each process is assigned a numeric priority

- CPU is allocated to the process with the highest priority
- priorities can be external (set by user/admin) or internal (based on resources/history)
- SJF is priority scheduling where priority is the predicted next CPU burst time

priority scheduling may be preemptive or nonpreemptive

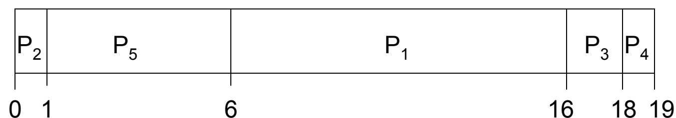
priority scheduling is not fair

- starvation is possible – low priority processes may never execute
- can be made fair using aging – as time progresses, increase the priority

Priority scheduling example

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

assuming processes all arrived at time 0, *Gantt Chart* for the schedule is:



average waiting time: (6 + 0 + 16 + 18 + 1)/5 = 8.2

average turnaround time: (16 + 1 + 18 + 19 + 6)/5 = 12

Round-Robin (RR) scheduling

RR = FCFS with preemption

- time slice or time quantum is used to preempt an executing process
- timed out process is moved to rear of the ready queue
- some form of RR scheduling is used in virtually all operating systems

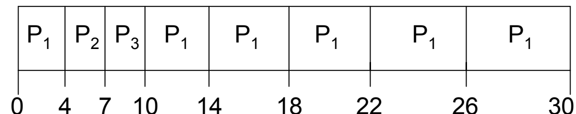
if there are *n* processes in the ready queue and the time quantum is *q*

- 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.

RR example

Process	Arrival Time	Burst Time
P_1	0	24
P_2	2	3
P_3	4	3

assuming q = 4, *Gantt Chart* for the schedule is:



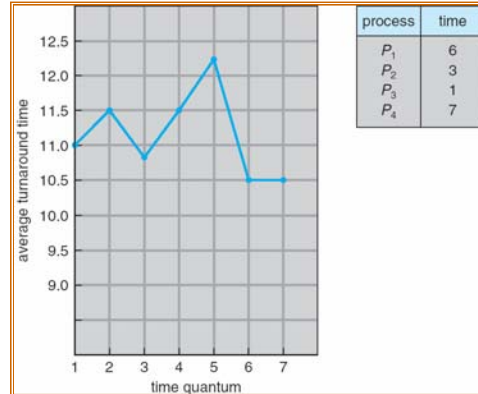
average waiting time: (6 + 2 + 3)/3 = 3.67

average turnaround time: (30 + 5 + 6)/3 = 13.67

RR performance

performance depends heavily upon quantum size

- if q is too large, response time suffers (reduces to FCFS)
- if q is too small, throughput suffers (spend all of CPU's time context switching)



- rule-of-thumb: quantum size should be longer than 80% of CPU bursts

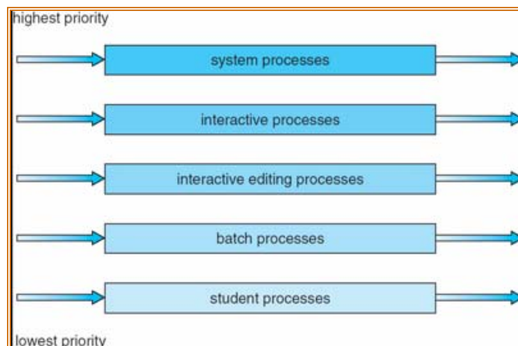
- in practice, quantum of 10-100 msec, context-switch of 0.1-1msec
  →CPU spends 1% of its time on context-switch overhead

Multilevel queue

combination of priority scheduling and other algorithms (often RR)

- ready queue is partitioned into separate queues
- each queue holds processes of a specified priority

- each queue may have its own scheduling algorithm
  (e.g., RR for interactive processes, FCFS for batch processes)



must be scheduling among queues

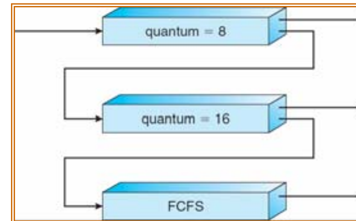
- absolute priorities
- (uneven) time slicing

Multilevel feedback queue

similar to multilevel queue but processes can move between the queues

e.g., a process gets lower priority if it uses a lot of CPU time

process gets a higher priority if it has been ready a long time (aging)



example: three queues

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

scheduling

- new job enters queue Q_0 which is served RR
  when it gains CPU, job receives 8 milliseconds
  if it does not finish in 8 milliseconds, job is moved to queue Q_1.
- at Q_1 job is again served RR and receives 16 additional milliseconds
  if it still does not complete, it is preempted and moved to queue Q_2.

Multiprocessor scheduling

CPU scheduling is more complex when multiple CPUs are available

symmetric multiprocessing:

- when all the processors are the same, can attempt to do real load sharing
- 2 common approaches:
  1. separate queues for each processor,
     processes are entered into the shortest ready queue
  2. one ready queue for all the processes,
     all processors retrieve their next process from the same spot

asymmetric multiprocessing:

- can specialize, e.g., one processor for I/O, another for system data structures, …
- alleviates the need for data sharing

Real-time scheduling

hard real time systems

- requires completion of a critical task within a guaranteed amount of time

soft real-time systems

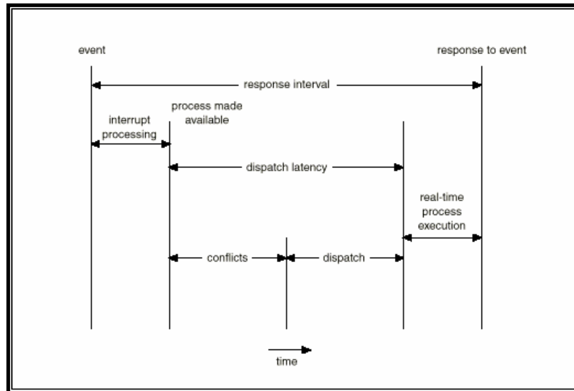
- requires that critical processes receive priority over less fortunate ones

note: delays happen!

when event occurs, OS must:

- handle interrupt
- save current process
- load real-time process
- execute

for hard real-time systems, may have to reject processes as impossible



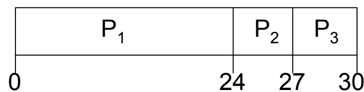
Scheduling algorithm evaluation

various techniques exist for evaluating scheduling algorithms

Deterministic model

- use predetermined workload, evaluate each algorithm using it
- this is what we have done with the Gantt charts

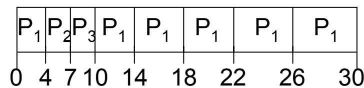
Process	Arrival Time	Burst Time
P_1	0	24
P_2	2	3
P_3	4	3



FCFS:

average waiting time: (0 + 22 + 23)/3 = 15

average turnaround time: (24 + 25 + 26)/3 = 25



RR (q = 4):

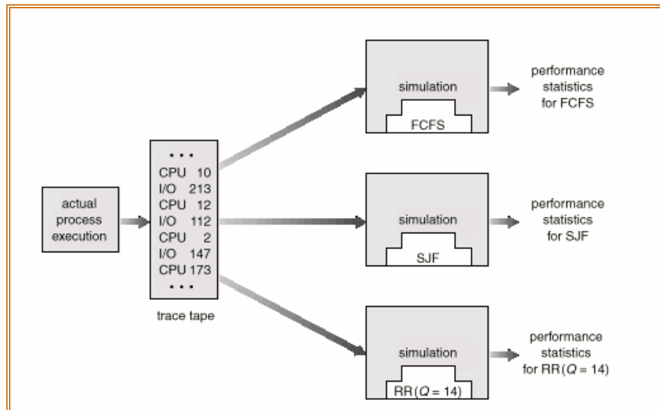
average waiting time: (6 + 2 + 3)/3 = 3.67

average turnaround time: (30 + 5 + 6)/3 = 13.67

Scheduling algorithm evaluation (cont.)

Simulations

- use statistical data or trace data to drive the simulation
- expensive but often provides the best information
- this is what we did with HW1 and HW2



Scheduling algorithm evaluation (cont.)

Queuing models

- statistically based, utilizes mathematical methods
- collect data from a real system on CPU bursts, I/O bursts, and process arrival times

Little's formula: `N = L * W`

where `N` is number of processes in the queue
`L` is the process arrival rate
`W` is the wait time for a process

under simplifying assumptions (randomly arriving jobs, random lengths):

```
response_time = service_time/(1-utilization)
```

- powerful methods, but real systems are often too complex to model neatly

Implementation

- just build it!

Scheduling example: Solaris

utilizes 4 priority classes

each with priorities & scheduling algorithms

time-sharing is default

- utilizes multilevel feedback queue w/ dynamically altered priorities
- inverse relationship between priorities & time slices → good throughput for CPU-bound processes; good response time for I/O bound processes

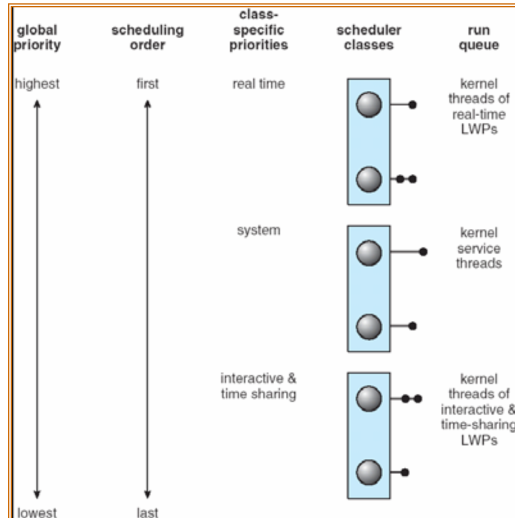
interactive class same as time-sharing

- windowing apps given high priorities

system class runs kernel processes

- static priorities, FCFS

real-time class provides highest priority



Scheduling example: Windows XP

Windows XP utilizes a priority-based, preemptive scheduling algorithm

- multilevel feedback queue with 32 priority levels (1-15 are variable class, 16-31 are real-time class)
- scheduler selects thread from highest numbered queue, utilizes RR
- thread priorities are dynamic
  - priority is reduced when RR quantum expires
  - priority is increased when unblocked & foreground window
- fully preemptive – whenever a thread becomes ready, it is entered into priority queue and can preempt active thread

	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

Scheduling example: Linux

with Linux 2.5 (2002), CPU scheduler was overhauled

- provides better support for Symmetric Muliprocessing (SMP)
- improves performance under high loads (many processes)

Linux scheduler is preemptive, priority-based

- 2 priority ranges: real-time (0-99) & nice (100-140)
- unlike Solaris & XP, direct relationship between priority and quantum size
  highest priority (200 ms) ←→ lowest priority (10 ms)

- real-time tasks are assigned fixed priorities
- nice tasks have dynamic priorities, adjusted when quantum is expired
  tasks with long waits on I/O have priorities increased → favors interactive tasks
  tasks with short wait times (i.e., CPU bound) have priority decreased