Service Oriented Computing (6)

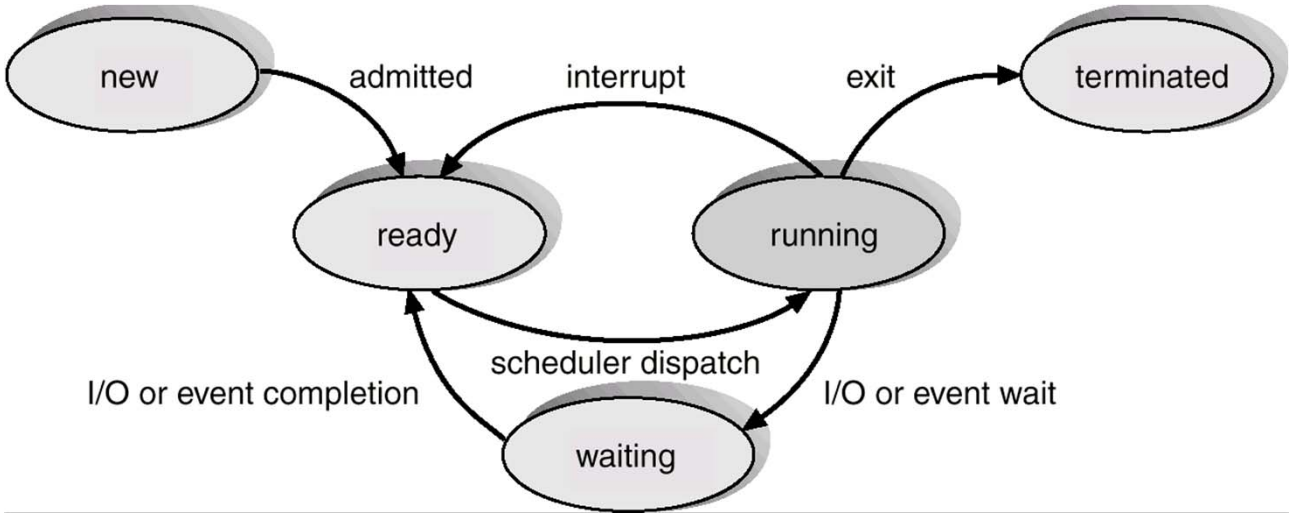
Chan-Hyun YOUN

Dept of Electrical Engineering, KAIST

Process/Thread/VM Scheduling

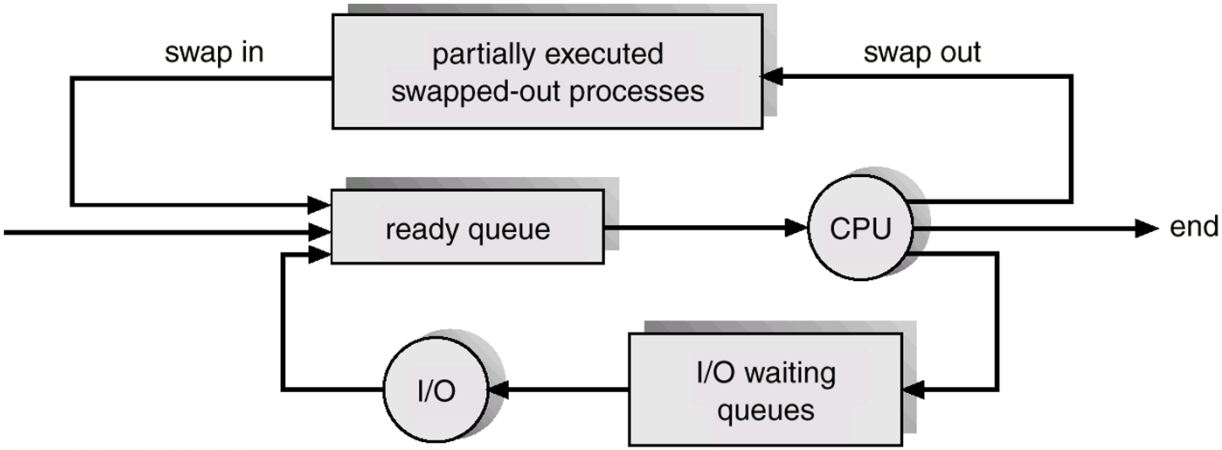
Recap: Process State

- As a process executes, it changes *state*
  - new: The process is being created.
  - running: Instructions are being executed.
  - waiting: The process is waiting for some event to occur.
  - ready: The process is waiting to be assigned to a process.
  - terminated: The process has finished execution.



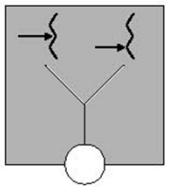
Recap: Process Schedulers

- Long-term scheduler (or job scheduler) – selects which process should be brought into the ready queue.
  - Invoke infrequently (seconds, minutes)
  - Control the degree of concurrency
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU.
  - Invoke frequently (millisecond), must be fast
  - Optimize throughput, average response time, slowdown, revenue, etc
  - Slowdown is a normalized queuing delay w.r.t. exec time



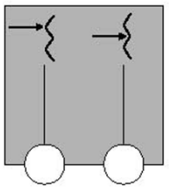
Recap: Thread Scheduling Options

- M-1 model (user-level): portable, easy to programming, but no concurrency
- 1:1 model (kernel-level): each user level thread is known to the kernel and all threads can access the kernel at the same time but, hard to program. Win32 put limits on the number of threads
- M:M model (2-level model): minimizes programming effort while reducing the cost and weight of each thread.



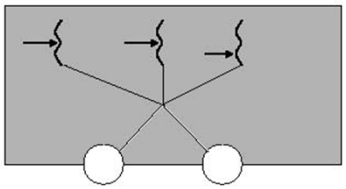
M:1

HP-UX 10.20 (via DCE) Green Threads



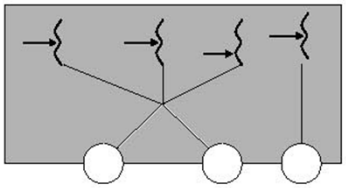
1:1

Win32, OS/2, AIX 4.0



M:M (strict)

Linux, Solaris 5.9 and later, NetBSD5, FreeBSD 8



2-Level (aka: M:M)

Solaris, DEC, IRIX, HP-UX 10.30, AIX 4.1

Solaris 5.2 to 5.9, NetBSD2 to 4, FreeBSD 5 and 6

Scheduling Disciplines

- A set of rules based on which processes/threads are scheduled
- Performance metrics:
  - Throughput: the amount of work to be finished in a time unit
  - Utilization: the fraction of a system is busy with useful work
  - Turnaround time: the time from start to completion, including
    - waiting time to be loaded into memory,
    - waiting time in ready queue,
    - execution time
    - Blocked time waiting for an event, for example waiting for I/O
  - Response time: the time from arrival time of a request to the time its response is produced
  - Waiting time: the delay time in Ready queue, directly impacted by scheduling discipline
  - Fairness: each process/thread gets fair share of resources (cpu time, etc)
  - Deadline in real-time scheduling
  - etc
- Maximize/minimize metrics,
  - Bounded vs average result; Deterministic vs stochastic

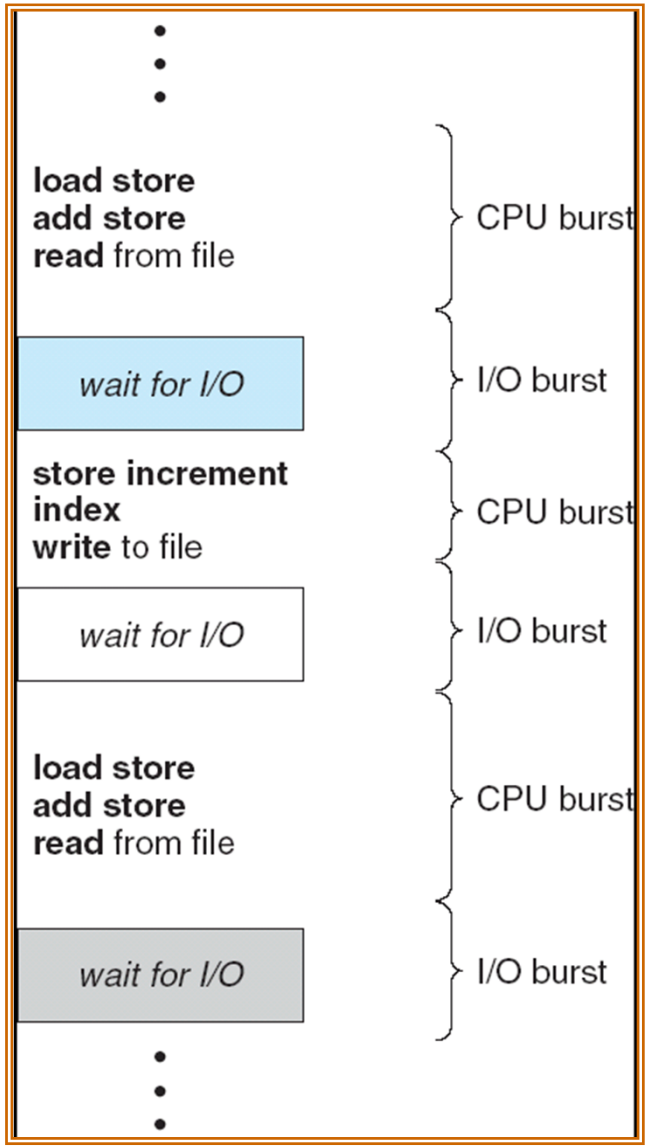
CPU-I/O bursts

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

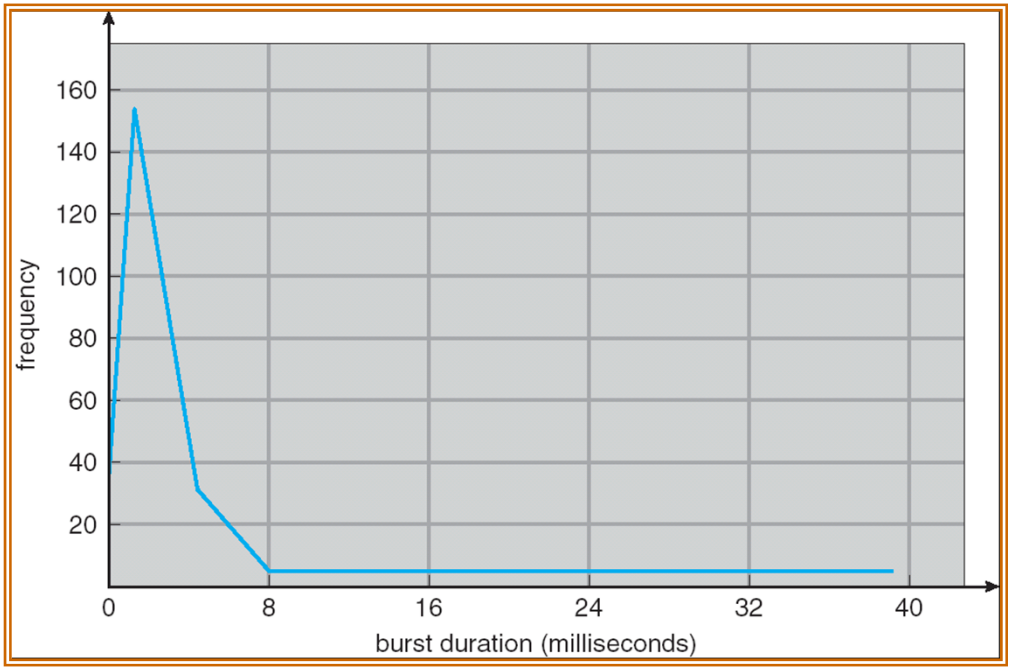
  I/O-bound process: performs lots of I/O followed by short bursts of computation
- ideally, the system admits a mix of CPU-bound and I/O-bound processes to maximize CPU and I/O device usage



Burst distribution

CPU bursts tends to have an exponential or hyperexpo distribution

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

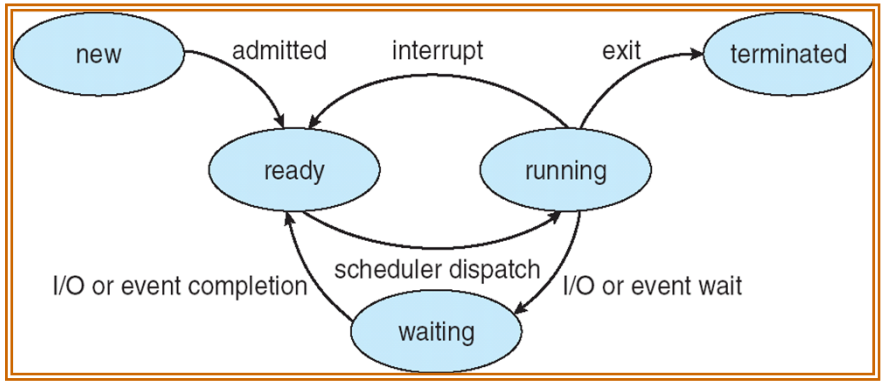


What does this distribution pattern imply about the importance of CPU scheduling?

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

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

Priority Scheduling

Each process is assigned a numeric priority

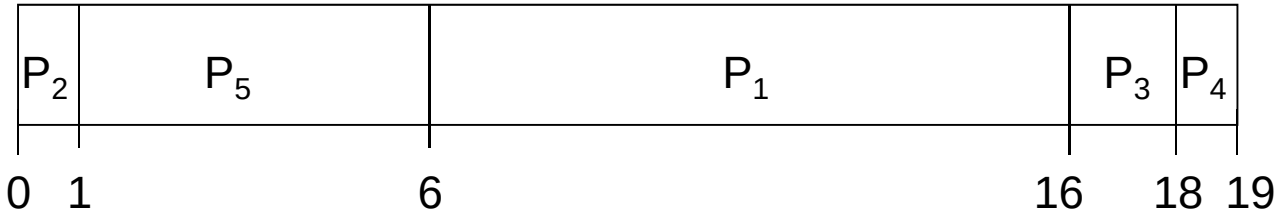
- CPU is allocated to the process with the highest priority

- Fixed vs dynamic priority
  - Priorities can be external (set by user/admin) or internal (based on resources/history)
  - can be made fair using aging – as time progresses, increase the priority (dynamic priority)
- May be preemptive or nonpreemptive
  - *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, preemp
- Fixed priority preemptive scheduling has no particular advantage in terms of throughput over FIFO scheduling
- Not fair: starvation is possible, low pri processes may never execute

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

Shortest-Job-First (SJF) scheduling

More accurately, 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 is priority scheduling where priority is the predicted next CPU burst time

SJF can be preemptive or nonpreemptive

- Preemptive SJF
  - known as Shortest-Remaining-Time-First (SRTF)

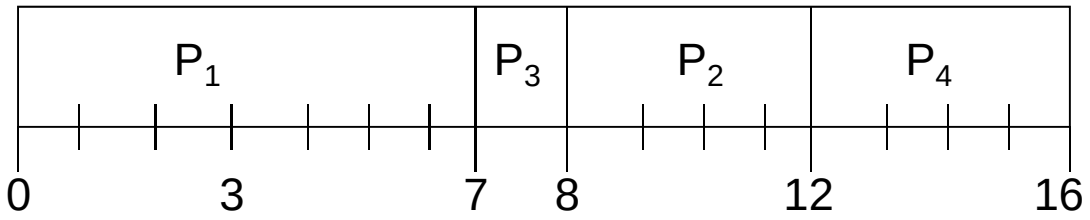
If you can accurately predict CPU burst length, SJF is optimal

- it minimizes average waiting time for a given set of processes
- Waiting time and response time increase as the process' computational requirements increase

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:



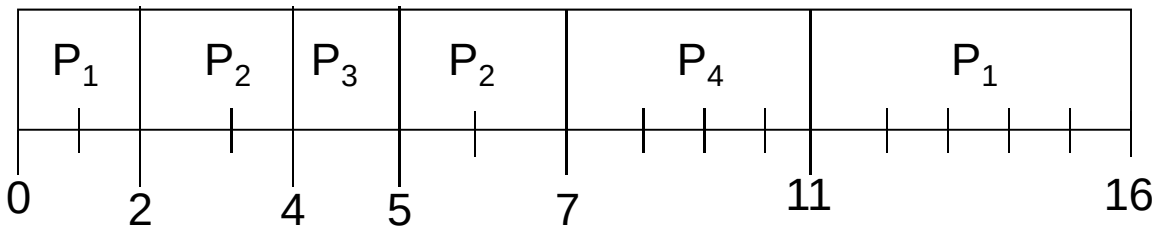
average waiting time: (0 + 6 + 3 + 7)/4 = 4

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

But 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

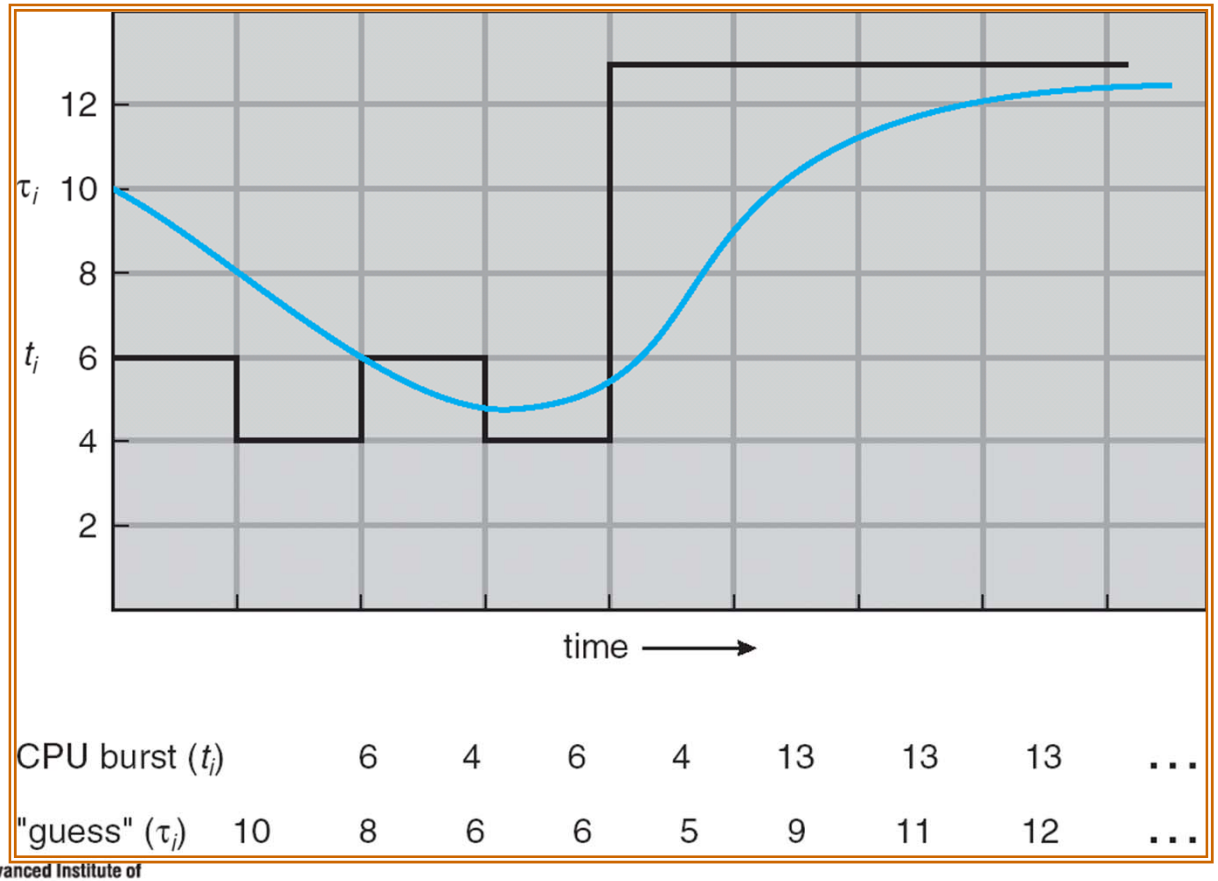
t_n = actual length for nth CPU burst

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

Exponential averaging

consider the following example, with $\alpha = 0.5$ and $\tau_0 = 10$

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



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

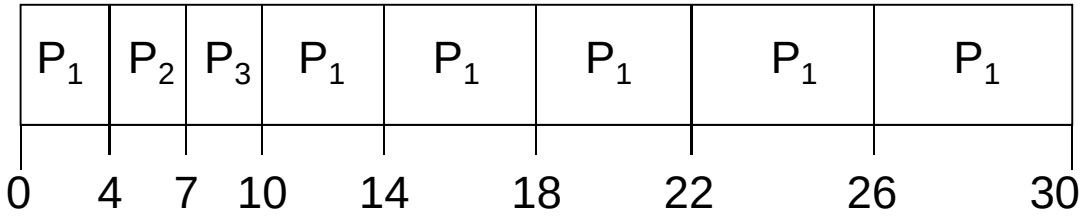
if there are *n* processes in the ready queue and the time quantum is *q*

- 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:



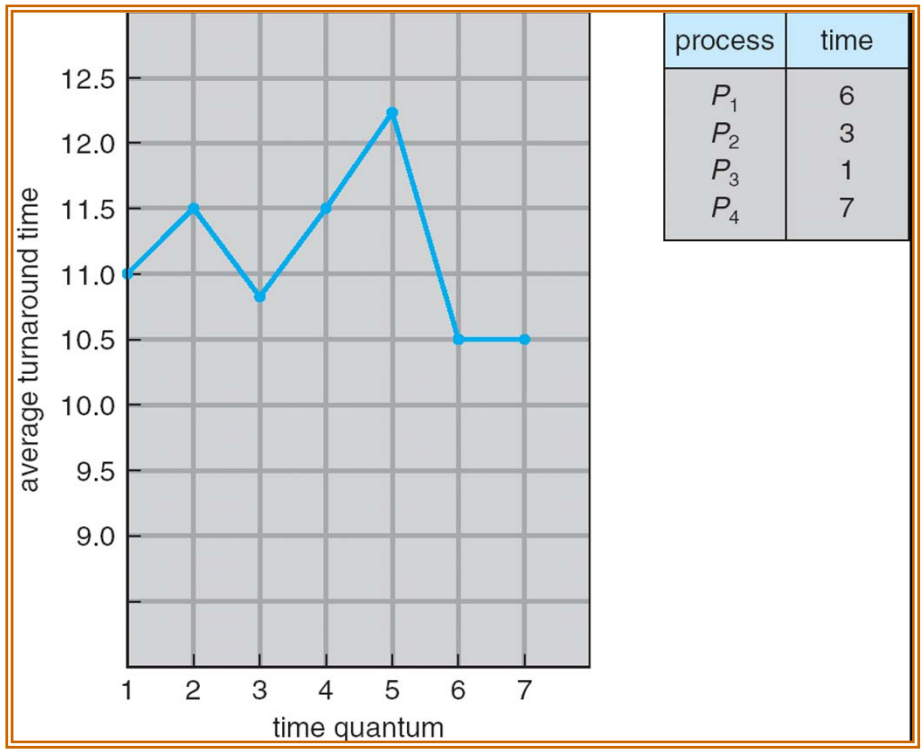
average waiting time: (6 + 2 + 3)/3 = 3.67

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



process	time
P_1	6
P_2	3
P_3	1
P_4	7

How to provide guarantee of fairness via RR??

Summary of Basic Scheduling Algorithms

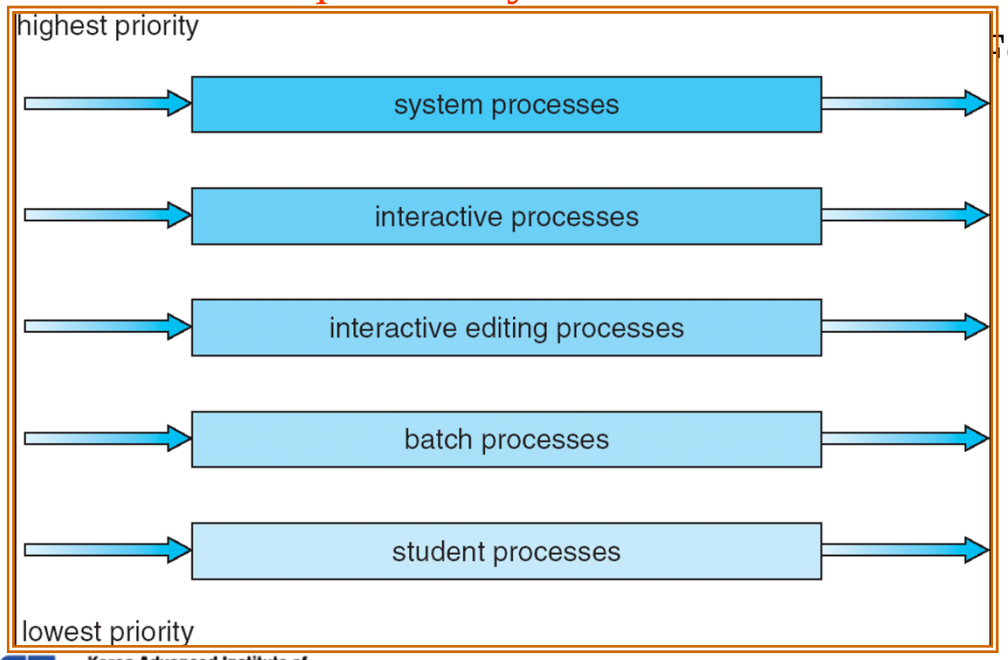
Scheduling algorithm	CPU Utilization	Throughput	Turnaround time	Response time	Deadline handling	Starvation free
First In First Out	Low	Low	High	Low	No	Yes
Shortest remaining time	Medium	High	Medium	Medium	No	No
Fixed priority pre-emptive scheduling	Medium	Low	High	High	Yes	No
Round-robin scheduling	High	Medium	Medium	Low	No	Yes

Multilevel Queue Scheduling

For situations in which processes can easily be classified

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



CFS for batch processes)

must be scheduling among queues

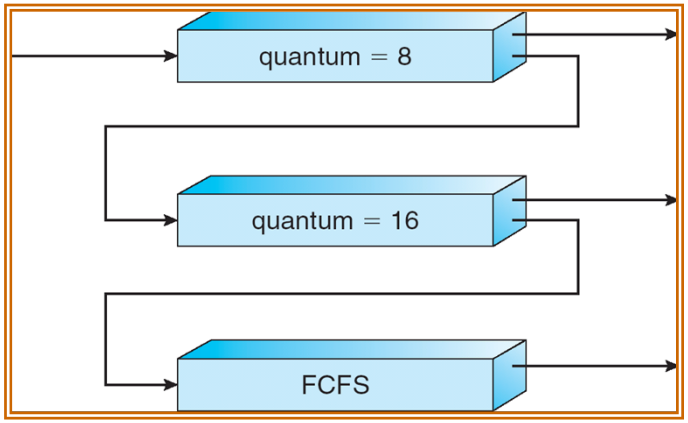
- absolute priorities
- (uneven) time slicing

Multilevel Feedback Queue Scheduling

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:

  separate queues for each processor, processes are entered into the shortest ready queue

  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

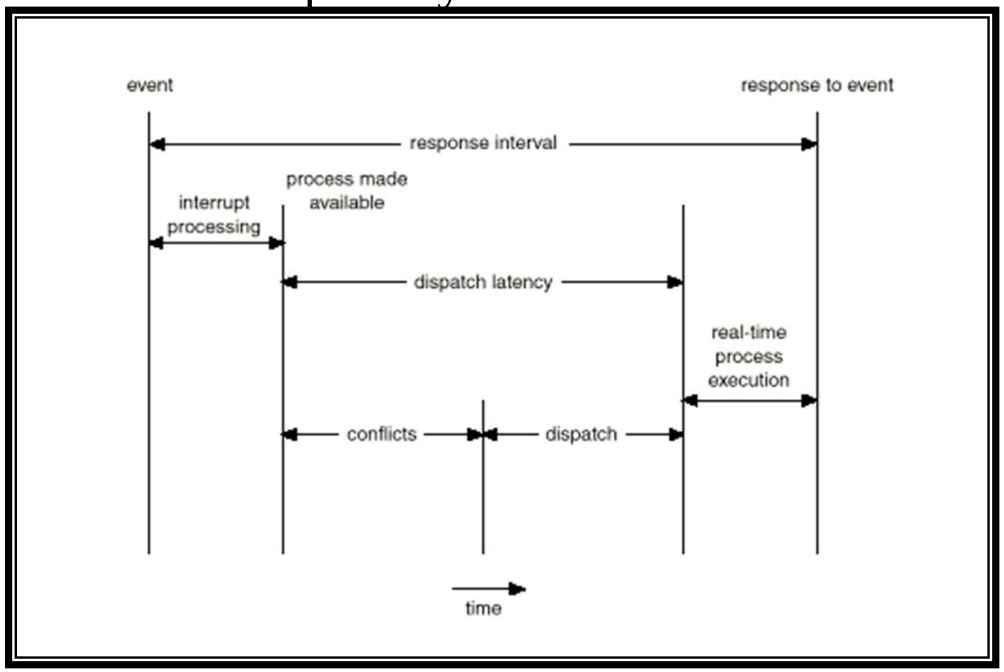
- requires that critical processes receive priority over less fortunate ones

Delays happens:

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



Conflicts: Preemption of process running in kernel; Release resource hold by low-priority processes, but needed by high-priority process

Scheduling algorithm evaluation

Various techniques exist for evaluating scheduling algorithms

- Deterministic model, Simulation, Queueing Model, Implementation

Deterministic model

use predetermined workload, evaluate each algorithm using it

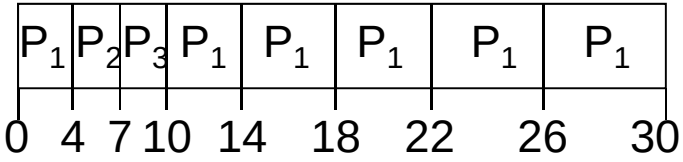
this is what we have done with the Gantt charts

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

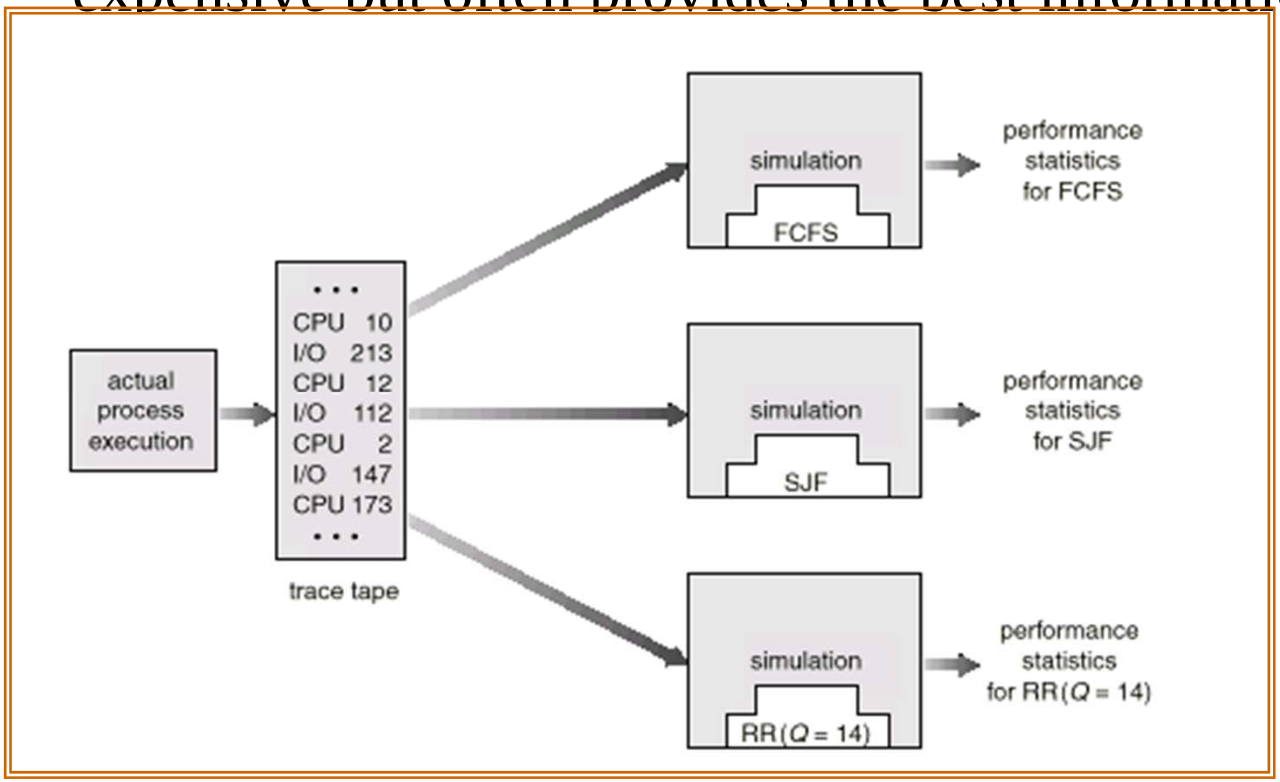
average turnaround time: (30 + 5 + 6)/3 = 13.67

Scheduling algorithm evaluation (cont.)

Simulations

use statistical data or trace data to drive the simulation

expensive but often provides the best information



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)
```

Implementation, just build it!

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

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 (the higher priority, the smaller the time slices) → good throughput for CPU-bound processes; good response time for I/O bound processes

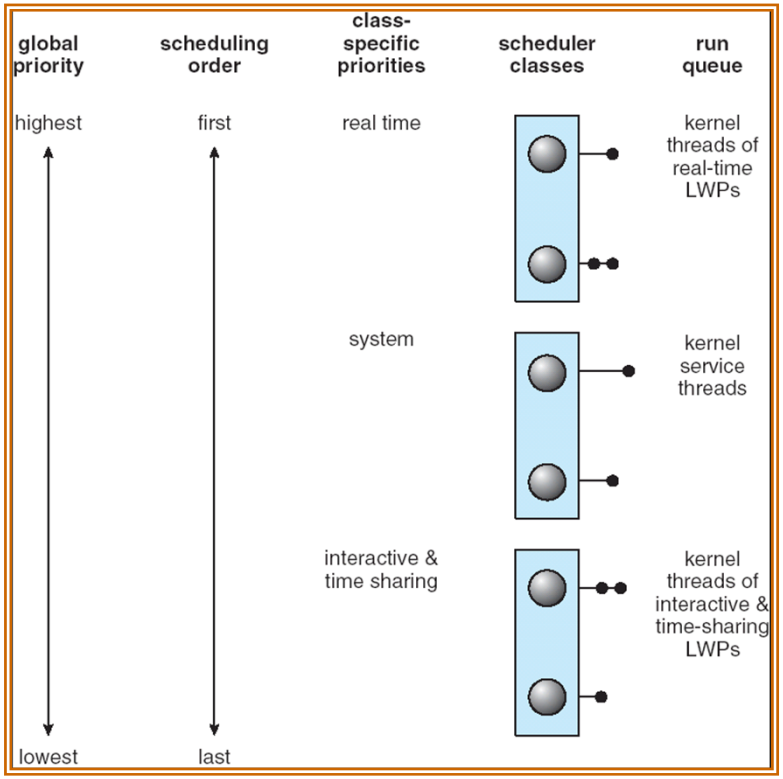
interactive class same as time-sharing

- 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

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

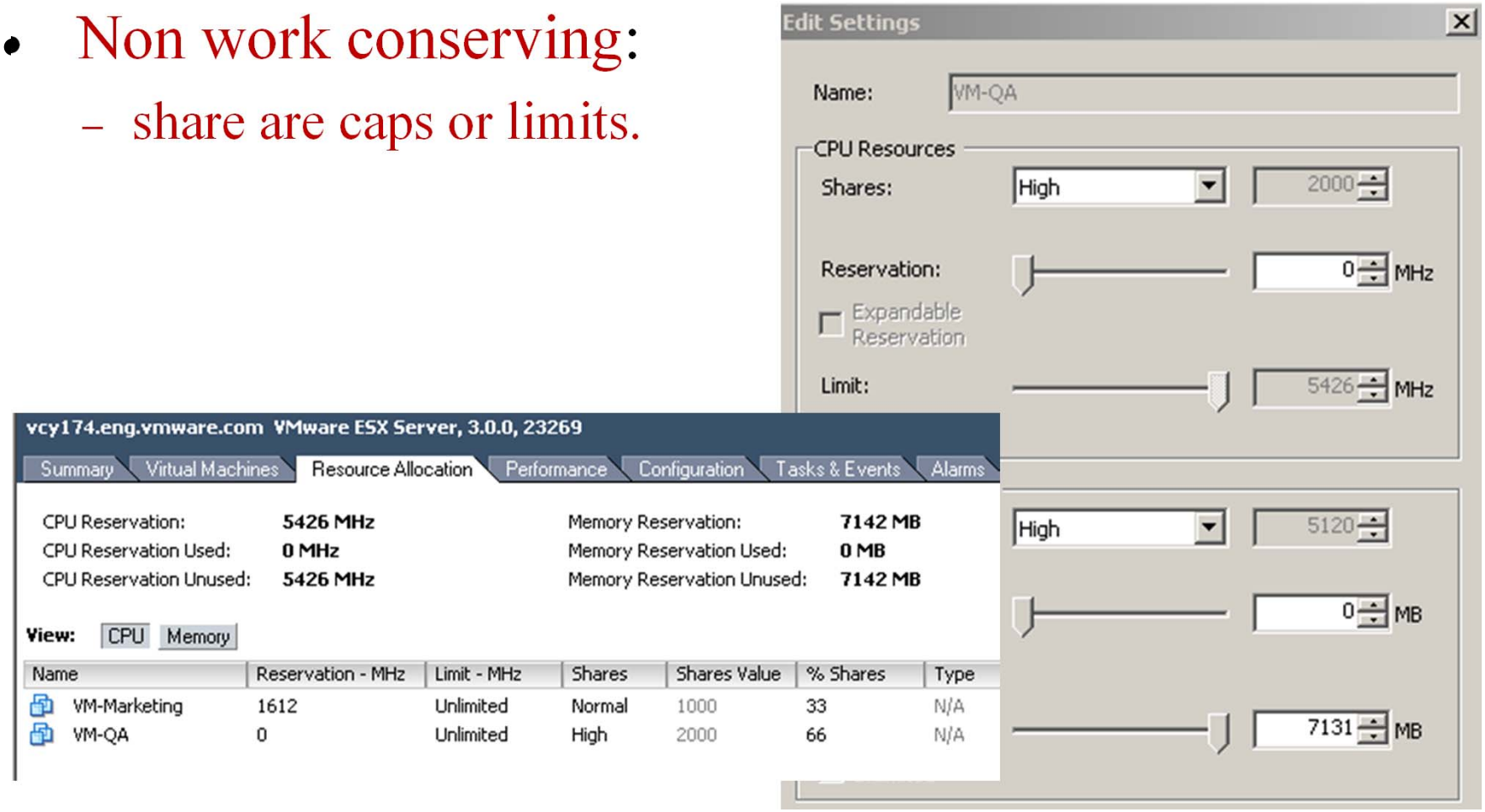
  tasks with short wait times (i.e., CPU bound) have priority decreased

Resource Alloc in VM

- Resource Management Guide of Vmwae ESX
  - www.vmware.com/pdf/vi3_35/esx_3/r35/vi3_35_25_resource_mgmt.pdf
- Resource configuration for a VM
  - Number of virtual CPUs
    - In Vmware, equal share of CPU per vCPU by default. E.g a VM with one vcpu is assigned half of the resources of a VM of 2 vcpus
  - Reservation in absolute values: Amount of cpu in MHz and memory in MB
    - E.g. on a 2GHz cpu, reserve a VM with 512MHz
    - Equal share per MB of VM: a VM with 8GB is entitled to eight times as much mem as a 1GB VM
  - Shares: entitlement in proportion to specified shared
    - In Vmware ESX, high (2000 shares per vcpu, 20 shares per MB), normal (1000 shares per vcpu, 10 shares per MB), low (500 shares per vcpu, and 5 shares per MB)
    - A normal config of VM with 2 vcpus and 1GB should have 2x1000 shares of cpus and 10x1024=20140 shares of mem

VM Resource Alloc (cont')

- Work conserving: Idle only iff there is no runnable vm
- Non work conserving:
  - share are caps or limits.



Proportional Share Scheduling

- In Vmware ESX, *("The CPU scheduler in Vmware ESX")*
  - dynamic priority scheduling
  - Priority is set to the ratio of consumed cpu resource to entitled resource

Figure 3. CPU time between vm0 and vm1 that have different shares with the ratio of 1:7, 2:6, 3:5, 4:4, 5:3, 6:2, and 7:1.

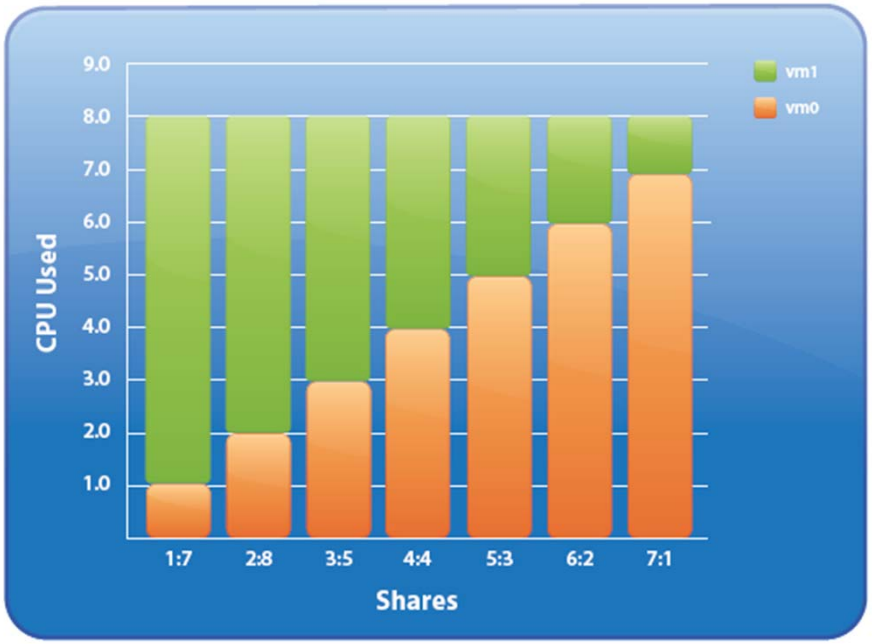
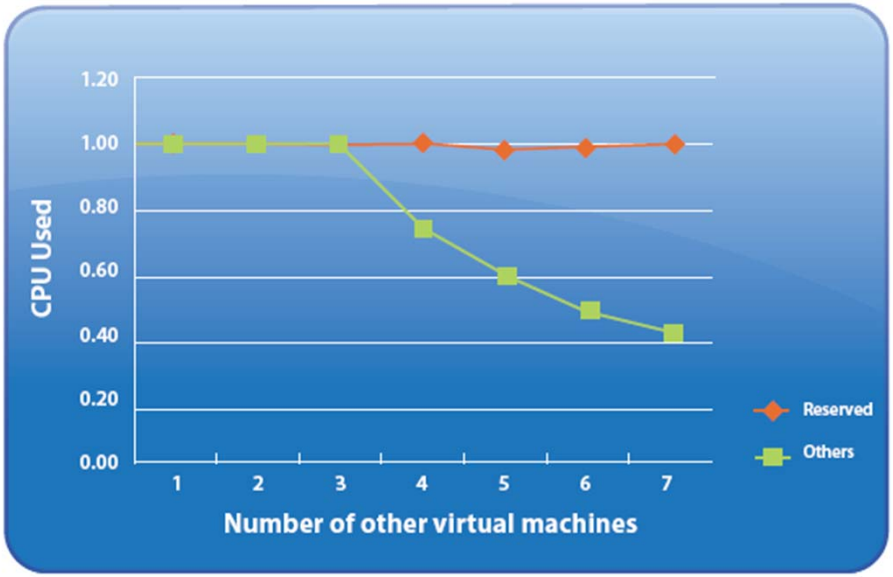


Figure 4. CPU time of a fully-reserved virtual machine and the average CPU time of others without reservation.



Credit Scheduling in Xen

- http://wiki.xensource.com/xenwiki/CreditScheduler.
  - Comparison: www.hpl.hp.com/techreports/2007/HPL-2007-25.pdf
- Concepts:
  - Weight: a VM with a weight of 512 will get twice as much cpu as a VM of weight 256
  - Cap: the maximum amount of cpu, in percentage of one physical cpu
    - 100 equiv to 1 pcpu, 50 is half a pcpu, 200 means 2 cpus
- Scheduling
  - The scheduler transforms the weight into a credit allocation for each vcpu, using a system-wide accounting thread. As vcpu runs, it consumes credit: in a period of 10ms, the current running vcpu is debited 100 credits
  - Negative credits means a priority of "over" its share. Otherwise, a priority of "under"
  - Periodically, when the sum of credits goes negative, he accounting thread gives everyone more credits.
  - Each CPU maintains a sorted queue of runnable vcpus in terms of their priority
  - At each scheduling epoch (30ms), the accounting thread recomputes the credits for each active VM.