CIIC 4050 / ICOM 5007
Operating Systems

Lecture 8:
CPU Scheduling

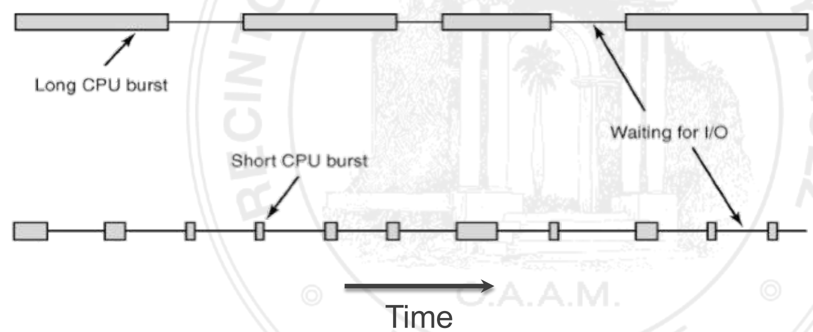
CPU Scheduling Basic Concepts (1)

- **Maximum CPU utilization** is obtained with multiprogramming since another process can be assigned to use the CPU while other processes are waiting for something.
  - without multiprogramming the CPU sits idle while a process waits
- **CPU Burst** - time that a process is expected to be executing without requesting a blocking operation (I/O, wait, etc)
- **I/O Burst** - time that a process is expected to spend in an I/O operation

CPU Scheduling Basic Concepts (2)

- **CPU–I/O Burst Cycle** – Process execution consists of a *cycle* of CPU execution and I/O wait
- **CPU burst** distribution is generally characterized as exponential.
  - there is usually a large number of short CPU bursts and a small number of large CPU bursts.

Process Behavior (1)



Scheduling Algorithms

All systems
Fairness — giving each process a fair share of the CPU
Policy enforcement — seeing that stated policy is carried out
Balance — keeping all parts of the system busy

Batch systems
Throughput — maximize jobs per hour
Turnaround time — minimize time between submission and termination
CPU utilization — keep the CPU busy all the time

Interactive systems
Response time — respond to requests quickly
Proportionality — meet users' expectations

Real—time systems
Meeting deadlines — avoid losing data
Predictability — avoid quality degradation in multimedia systems

- Figure 2-23. Some goals of the scheduling algorithm under different circumstances.

When to Schedule (1)

- When scheduling is absolutely required:
  1. When a process exits.
  2. When a process blocks.
- When scheduling usually done (though not absolutely required)
  3. When a new process is created.
  4. When an I/O interrupt occurs.
  5. When a clock interrupt occurs.

When to Schedule (2)

- 1, 2, 3 are **nonpreemtive**
  - a process can not be removed from running state unless it finishes or voluntarily request being moved to ready state
- 4, 5 are **preemptive**
  - a process can be forced out of running state at any moment

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)

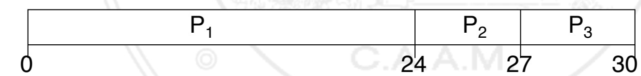
First-Come, First-Served (FCFS) Scheduling

FCFS Scheduling - processes are served in the order of arrival. Consider the following example:

- Consider three processes with given burst times.

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:



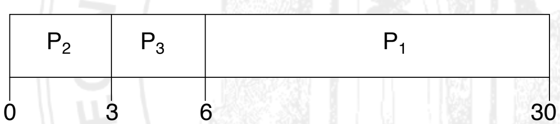
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

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 better than previous case
- *Convoy effect* - short process behind long process

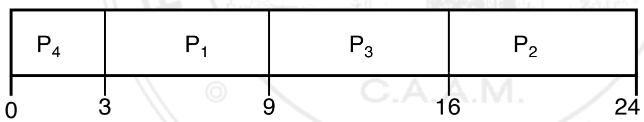
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU burst

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

Shortest Remaining Job First (SRJF)

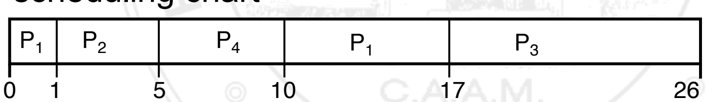
- An extension to SRJF is to preempt the running process whenever a new process arrives with burst time that is smaller than the remaining time of the one in Running state.

Example of SRJF

- Consider 4 process with the arrival/burst times shown

Process	Arrival Time	Burst Time
P_1	0.0	8
P_2	1.0	4
P_3	2.0	9
P_4	3.0	5

- SRJF scheduling chart



- Average waiting time = ((10-1)+(1-1)+(17-2)+(5-3) / 4
- = 26/4 = 6.5

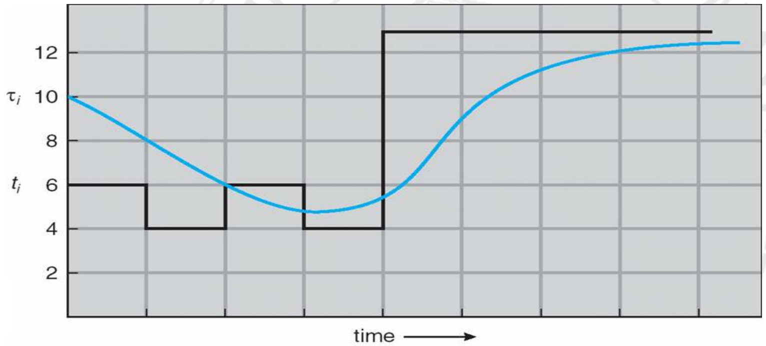
Note that process 1 is preempted at time 1, then it waits up to time 10, etc…

Determine Length of Next CPU Burst

- The major problem of SJF-based scheduling is how to know the next CPU burst of processes.
- Can only estimate it by using statistical strategies.
- Can be done by using the length of previous CPU bursts, using exponential averaging - *averages the past bursts to estimate the next.*

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

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	…

- Exponential average with α = 1/2 and τ = 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:

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

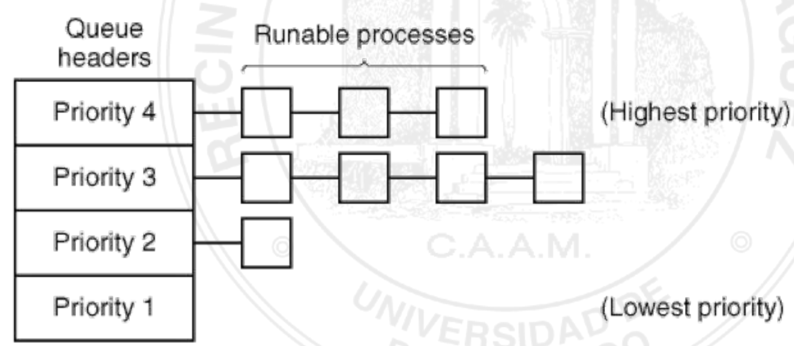
- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem : **Starvation** – low priority processes may never execute
- Solution : **Aging** – as time progresses increase the priority of the process

Priority Scheduling

- Figure 2-27. A scheduling algorithm with four priority classes.



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)

P_2	P_5	P_1	P_3	P_4
1	6	16	18	19

- Average waiting time = (6 + 0 + 16 + 18 + 1)/5 = 8.2

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds.
- After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If ready queue has n processes and q is the time quantum, 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 $(n\text{-}1)q$ time units.
- Performance
  – q large $\Rightarrow$ FIFO
  – q small $\Rightarrow$ increases context switch
  – q must be large with respect to context switch, otherwise overhead is too high

Round-Robin Scheduling

- Figure 2-26. Round-robin scheduling.
- (a) The list of runnable processes.
- (b) The list of runnable processes after B uses up its quantum.

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

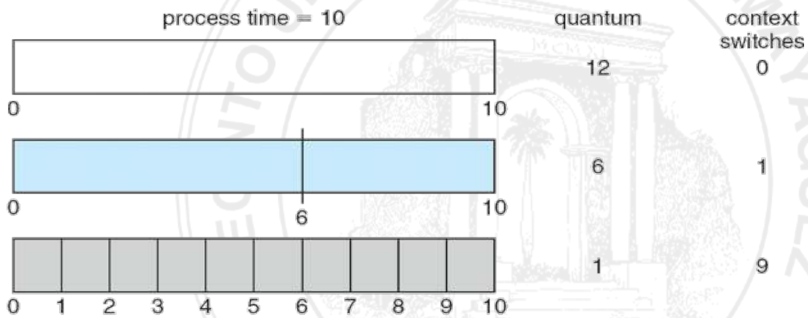
- The Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

- Typically, higher average turnaround than SJF, but better *response*

Time Quantum and Context Switch Time



- Most modern systems have **time quanta** in range **10-100 msec.**
- Context switch is typically less than 10 microseconds.

Turnaround Time Varies With Quantum Length

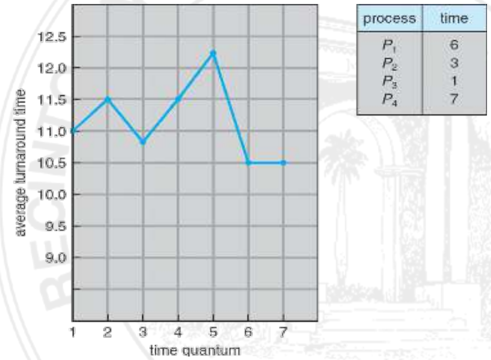


Figure suggests that average turnaround time (ATT) does not necessarily improves as time-quantum increases…

- For the processes given and q=1 we have:
  12341241241414144 => ATT = (3+9+14+17)/4 = 11
- For q=2, we have:
  11223441134411444 => ATT = (5+10+14+17)/4 = 11.5

Average Turnaround Time Varies With Quantum Length (1)

- ATT can be improved if most processes finish their next CPU burst in a single time quantum.
- Example: Consider 3 process with CPU burst of 10 each.
  - q=1 => ATT = 29
  - q=10 => ATT = 20
- If context switch time is added, att increases more as q becomes smaller.

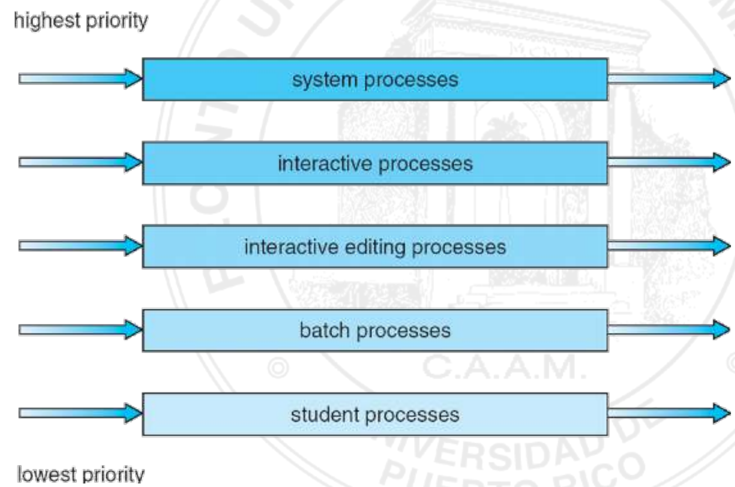
Turnaround Time Varies With Quantum Length (2)

- Time q should be large compared to cs time, but not too large.
  - If too large, RR degenerates to FCFS.
  - Rule of thumb is that *80 percent of the CPU bursts should be shorter than the time q*.

Multilevel Queue

- Ready queue is partitioned into separate queues: foreground (interactive), background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - **Fixed priority 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, 20% to background in FCFS

Multilevel Queue Scheduling



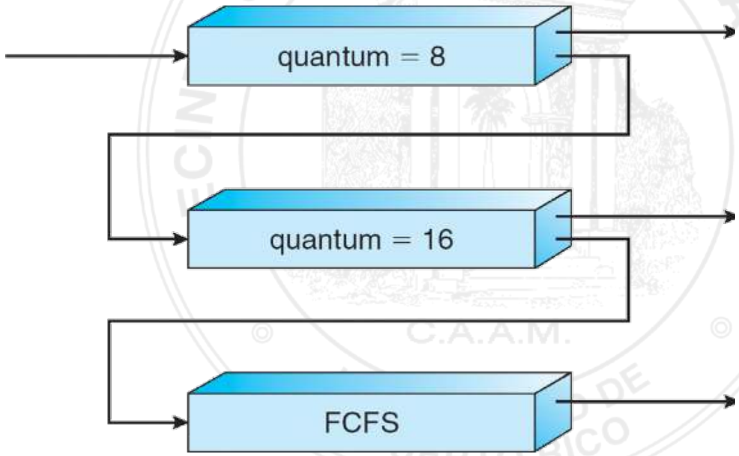
Multilevel Feedback Queue

- A process can move between the queues; Ex: aging can be implemented this way
- Scheduler defined by the following:
  - 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 it needs service

Example of Multilevel Feedback Queue

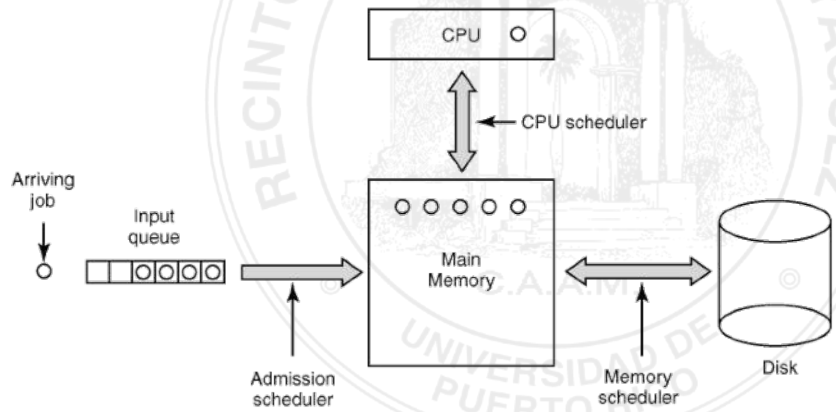
- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters Q_0 which is served FCFS. When it gains CPU, job receives 8 ms. If it does not finish, job is moved to Q_1.
  - At Q_1 job is again served FCFS and receives 16 ms. If it still does not complete, it is moved to queue Q_2.

Multilevel Feedback Queues



Three Level Scheduling (1)

- Figure 2-25. Three-level scheduling.



Three Level Scheduling (2)

- Criteria for deciding which process to choose:
- How long has it been since the process was swapped in or out?
- How much CPU time has the process had recently?
- How big is the process? (Small ones do not get in the way.)
- How important is the process?

Processor Affinity

- **Processor affinity** – process has affinity for processor on which it is currently running. For example: there may be benefits in cached data items - some may still be there from previous CPU bursts of the process…
  - **soft affinity** - whenever the system tries to achieve processor affinity but cannot guarantee it
  - **hard affinity** - some systems (e.g. Linux) provide system calls that allow a process to establish that it is not ti migrate to other processors…

Multi-Processor Scheduling (1)

- Scheduling problem is more complex if multiple CPUs are available
- We are using a model of **homogeneous processors** within a multiprocessor
  - all processors have equal functionality in terms of hardware design

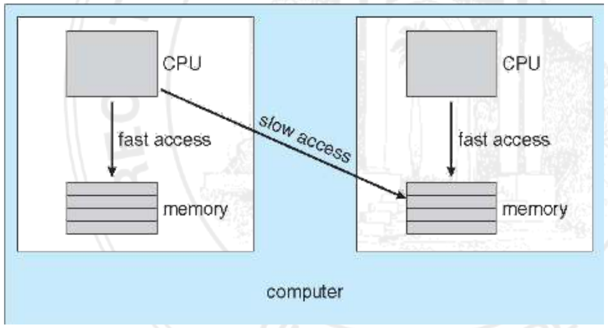
Multi-Processor Scheduling (2)

- **Asymmetric multiprocessing**
  - one CPU (master) handles all kernel operations
  - other processors (slaves) execute processes assigned by master
  - kernel operations are in general as in a single CPU system
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in shared ready queue, or each has its own private queue of ready processes

Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

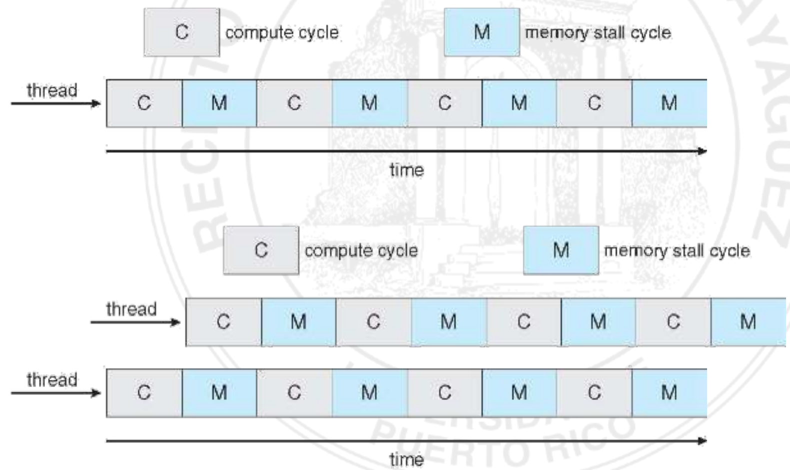
NUMA and CPU Scheduling



Multicore Processors

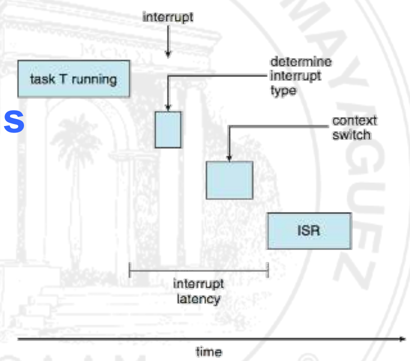
- Recent trend to place multiple processor cores on same physical chip
- Faster and consume less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Multithreaded Multicore System



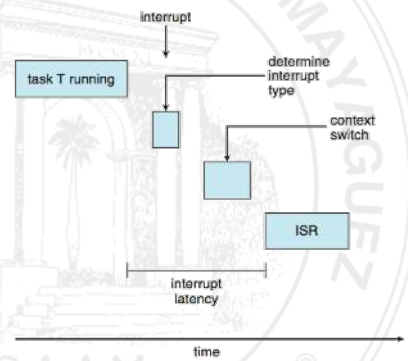
Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline



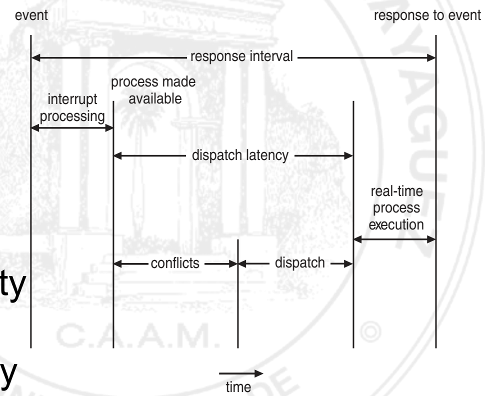
Real-Time CPU Scheduling

- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another



Real-Time CPU Scheduling (Cont.)

- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes

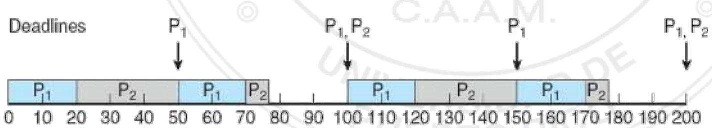


Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
  – Not knowing it does not own the CPUs
  – Can result in poor response time
  – Can effect time-of-day clocks in guests
- Can undo good scheduling algorithm efforts of guests

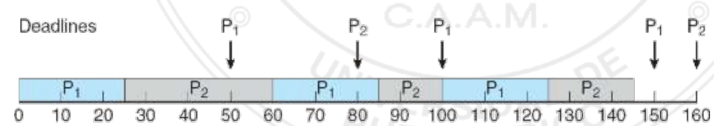
Rate Montonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2.



Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  the earlier the deadline, the higher the priority;
  the later the deadline, the lower the priority



Proportional Share Scheduling

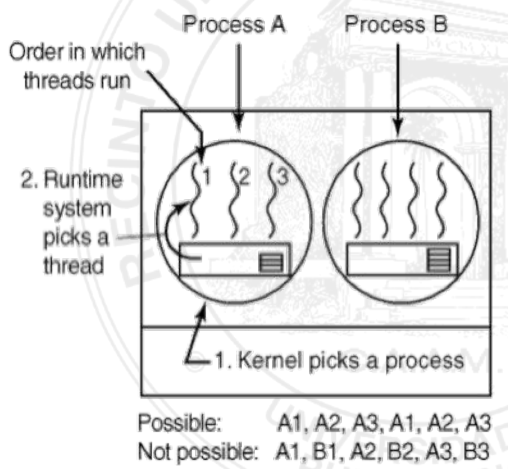
- *T* shares are allocated among all processes in the system
- An application receives *N* shares where *N* < *T*
- This ensures each application will receive ***N*** / *T* of the total processor time

Thread Scheduling (1)

- Distinction between user-level and kernel-level threads
  – On many-to-one and many-to-many models, user level threads are scheduled by the thread libraries, whereas kernel threads are scheduled by the kernel.
  – The thread library schedules user-level threads to run on available LWPs.
  – The scheme is known as **process-contention scope (PCS)** since scheduling competition is within the process

> PCS is based on priorities of threads. Thread in CPU is preempted if a higher priority thread arrives.
> - No guarantee of time slice among threads of equal priority.

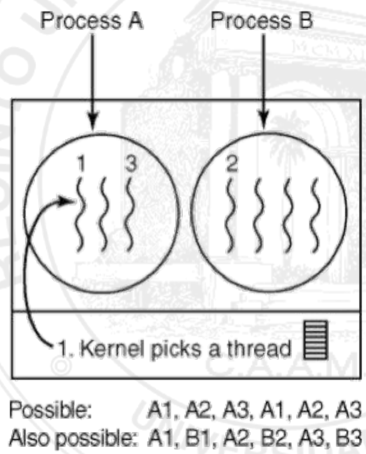
Thread Scheduling (PCS)



Thread Scheduling (2)

– Kernel threads are scheduled onto available CPU based on **system-contention scope (SCS)** – competition among all threads in system
  – this includes those systems that implement the one-to-tone model between user and kernel threads…

Thread Scheduling (SCS)



PTHREAD Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD SCOPE PROCESS schedules threads using PCS scheduling
  - PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.

PTHREAD Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
/* Each thread will begin control in this function */
void *runner(void *param)     {
    printf("I am a thread\n");
    pthread exit(0);
}

int main(int argc, char *argv[])   {
    int i;
    pthread_t tid[NUM THREADS];
    pthread_attr_t attr;

    /* get the default attributes */
    pthread_attr_init(&attr);

    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER);

    /* create the threads */
    for (i = 0; i < NUM THREADS; i++)
     pthread_create(&tid[i],&attr,runner,NULL);

    /* now join on each thread */
    for (i = 0; i < NUM THREADS; i++)
     pthread join(tid[i], NULL);
}
```

Operating System Examples

- Solaris
- Windows XP
- Linux
- MINIX

Solaris Scheduling (1)

- Priority-based thread scheduling:
  - a thread with a given priority has a chance only if no thread of higher priority is not waiting for CPU
- Each thread belongs to one of *six clases*:
  - *time sharing, interactive, real time, systems, fair share, and fixed priority*
- Priorities are numbers: 0-169 (global priority)
- The higher the number the higher the priority

Solaris Scheduling (2)

- Each class has a range of possible priorities
- Priorities of threads in time-sharing and interactive classes may change as they execute. The table on the next slide shows how these transitions are determined for some of the priorities on those two classes…
  – The table also shows the quantum assigned to each priority.
  – Lower priorities are assigned larger quantum

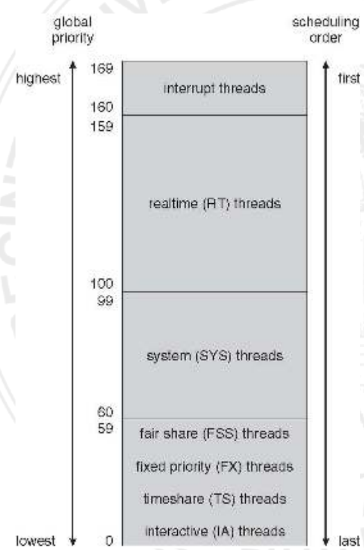
Solaris Dispatch Table

New priority to have when the heading action occurs.

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Priorities of threads in *time-sharing* and *interactive* classes may change as they execute.

Solaris Scheduling Order



Windows XP Priorities

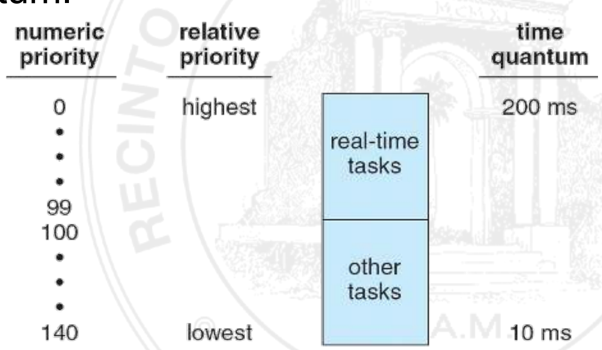
	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Linux Scheduling up to V2.5

- Linux uses a preemptive, priority-based, algorithm.
- Constant order *O*(1) scheduling time
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  – the lower the value, the higher the priority.

Linux Scheduling up to V2.5

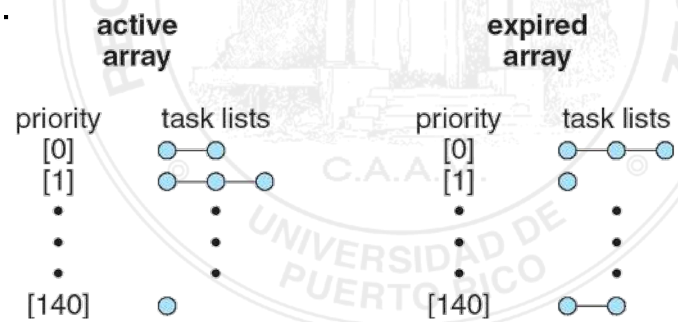
- Figure shows relationship between priorities and quantum.



- A runnable task is considered eligible for execution on the CPU as long as it has time left in its time-slice.

Two array Task Indexing

- When its time-slice has been exhausted, if task is still runnable, it passes to expired array.
- When active array has no task, references to both arrays are swapped - expired becomes active…



Linux Scheduling in V2.6.23 +

- ***Completely Fair Scheduler*** (CFS)
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time

Linux Scheduling in V2.6.23 +

  - 2 scheduling classes included, others can be added
    1. default
    2. real-time
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority

Linux Scheduling in V2.6.23 +

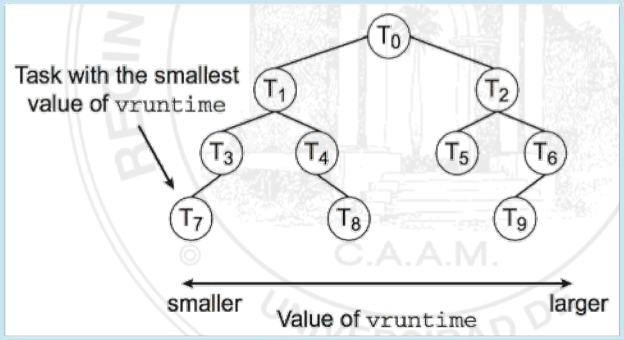
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**

`vruntime`

  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

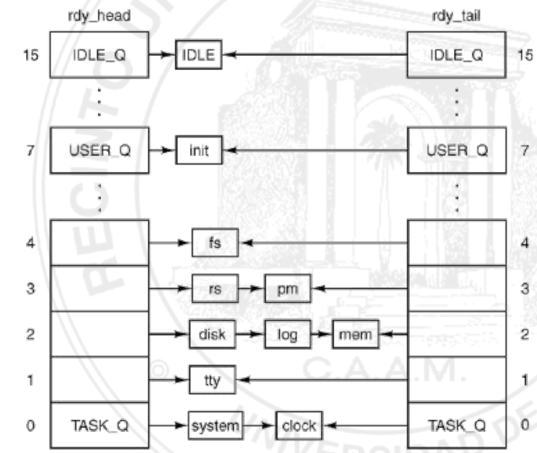
CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



- From O(log N) to O(1) (cached)

Scheduling in MINIX



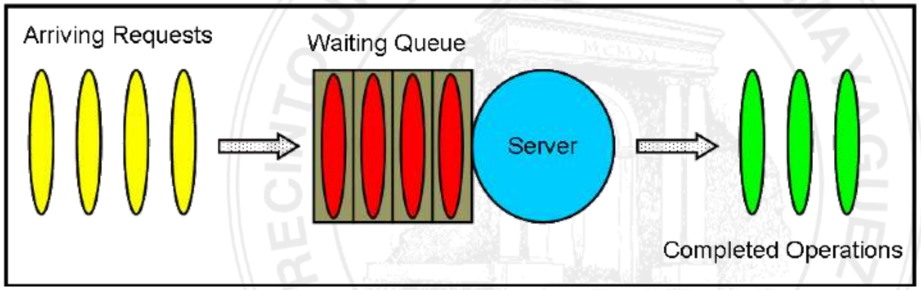
Algorithm Evaluation (1)

- How do we select a CPU scheduling algorithm for a particular system?
- First, define the criteria (e.g. minimize average waiting time)
- Deterministic modeling
  - Given a predetermined workload is known
  - Guestimate the performance of each algorithm (e.g. SJF, FCFS, RR)
  - Useful if behavior repeats or the same workload repeats

Algorithm Evaluation (2)

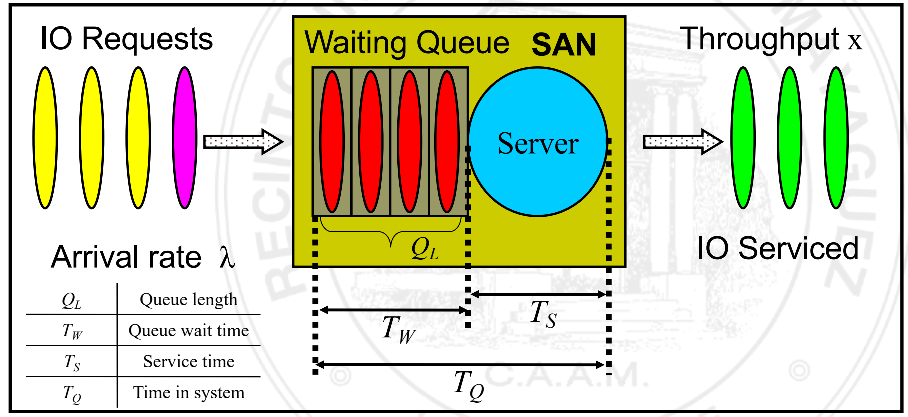
- Queueing models
  - Deterministic model is not realistic
- Simulation
  - Use random number generator to generate processes
  - Use traces
- Implementation

Queueing Theory



- Arriving requests enter the queue
- Wait until service provided

Queueing Theory Example: Storage Area Network (SAN)



Q_L	Queue length
T_W	Queue wait time
T_S	Service time
T_Q	Time in system

- Little's Theorem: For a system in steady state:

$$Q_L = \lambda T_Q$$

Queue Descriptors (1)

- Generic descriptor: A/S/m/k
  - Kendall's notation
- A denotes the arrival process
  - For Poisson arrivals we use M (for Markovian)
- S denotes the service-time distribution
  - M: exponential distribution
  - D: deterministic service times
  - G: general distribution

Queue Descriptors (2)

- Generic descriptor: A/S/m/k
  - Kendall's notation
- m is the number of servers
- k is the max number of customers allowed in the system – either in the buffer or in service
  - k is omitted when the buffer size is infinite

Queue Descriptors: Examples

- M/M/1: Poisson arrivals, exponentially distributed service times, one server, ∞ buffer
- M/M/m: same as previous with m servers
- M/M/m/m: Poisson arrivals, exponentially distributed service times, m server, no buffering
- M/G/1: Poisson arrivals, identically distributed service times (general distribution), one server, ∞ buffer
- */D/∞ : A constant delay system

Evaluation of CPU schedulers by Simulation