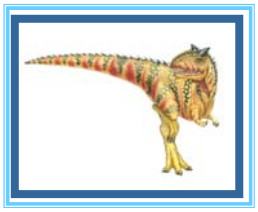
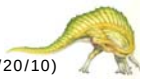
Chapter 5: CPU Scheduling



(slides selected/reordered/modified by R. Doemer, 04/20/10)

Chapter 5: CPU Scheduling

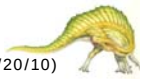
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation



(slide modified by R. Doemer, 04/20/10)

Objectives

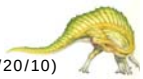
- To introduce CPU scheduling,
  which is the basis for multi-programmed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria
  for selecting a CPU-scheduling algorithm for a particular system



(slide modified by R. Doemer, 04/20/10)

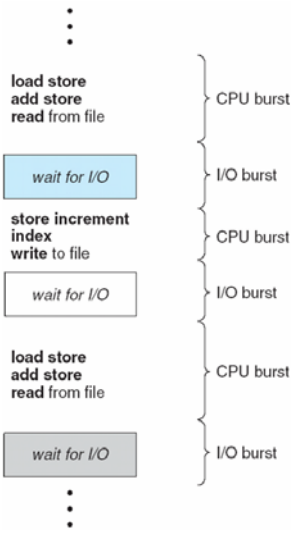
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- **CPU–I/O Burst Cycle** –
  Process execution consists of a *cycle* of
  - CPU execution and
  - I/O wait
- **CPU burst** distribution

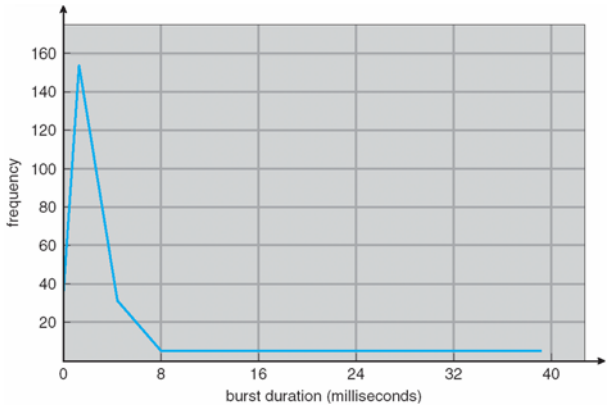


(slide modified by R. Doemer, 04/20/10)

Alternating Sequence of CPU And I/O Bursts

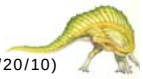


Histogram of CPU-burst Times



CPU Scheduler

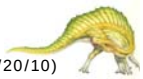
- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates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



(slide modified by R. Doemer, 04/20/10)

Dispatcher

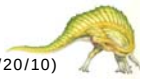
- **Dispatcher** module gives control of the CPU to the process selected by the *short-term scheduler*
- Dispatching involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency**
  - time it takes for the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



(slide modified by R. Doemer, 04/20/10)

Scheduling Algorithm Criteria

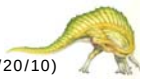
- **CPU utilization**
  - keep the CPU as busy as possible
- **Throughput**
  - number of processes that complete their execution per time unit
- **Turnaround time**
  - amount of time to execute a particular process
- **Waiting time**
  - amount of time a process has been waiting in the ready queue
- **Response time**
  - amount of time it takes from when a request was submitted until the first response is produced (not the time to output result!)
  - for time-sharing environment



(slide modified by R. Doemer, 04/20/10)

Scheduling Algorithm *Optimization* Criteria

- *Maximize* **CPU utilization**
  - keep the CPU as busy as possible
- *Maximize* **Throughput**
  - number of processes that complete their execution per time unit
- *Minimize* **Turnaround time**
  - amount of time to execute a particular process
- *Minimize* **Waiting time**
  - amount of time a process has been waiting in the ready queue
- *Minimize* **Response time**
  - amount of time it takes from when a request was submitted until the first response is produced (not the time to output result!)
  - for time-sharing environment

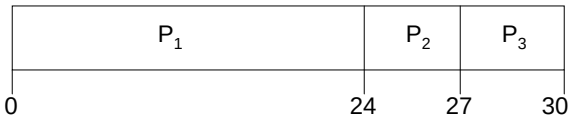


(slide modified by R. Doemer, 04/20/10)

First-Come, First-Served (FCFS) Scheduling

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:



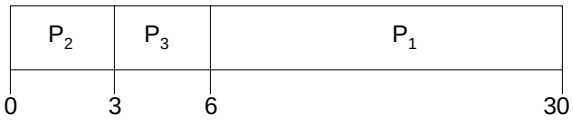
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont)

Suppose that the processes arrive in the order

P_2 , P_3 , P_1

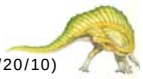
- The Gantt chart for the schedule is:



- 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
  - SJF gives minimum average waiting time for a given set of processes
- However, there's a problem:
  - The difficulty is knowing the length of the next CPU request…

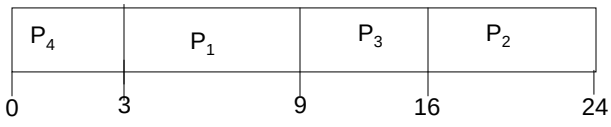


(slide modified by R. Doemer, 04/20/10)

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

(slide fixed by R. Doemer, 01/07/09)

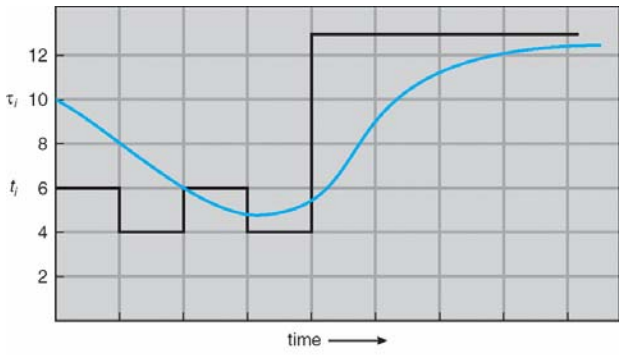
Estimating Length of Next CPU Burst

- Can only *estimate* the length!
  - Note: Text book calls this estimation prediction.
- 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. $\alpha, 0 \le \alpha \le 1$
4. Define: $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$

(slide modified by R. Doemer, 04/21/10)

Estimating 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 ...

(in this example, alpha is 1/2)

(slide modified by R. Doemer, 04/20/10)

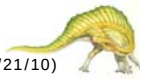
Examples of Exponential Averaging

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

- 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:

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

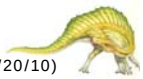
- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor



(slide modified by R. Doemer, 04/21/10)

Priority Scheduling

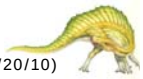
- A **priority number** (an integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Non-preemptive
- SJF is an example of priority scheduling where priority is the predicted next CPU burst time
-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



(slide modified by R. Doemer, 04/20/10)

Round Robin (RR) Scheduling

- 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 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.
- Performance
  - *q* large ⇒ RR degenerates to FCFS
  - *q* small ⇒ *q* should be large with respect to context switch, otherwise overhead is too high

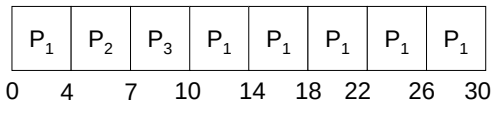


(slide modified by R. Doemer, 04/20/10)

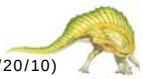
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

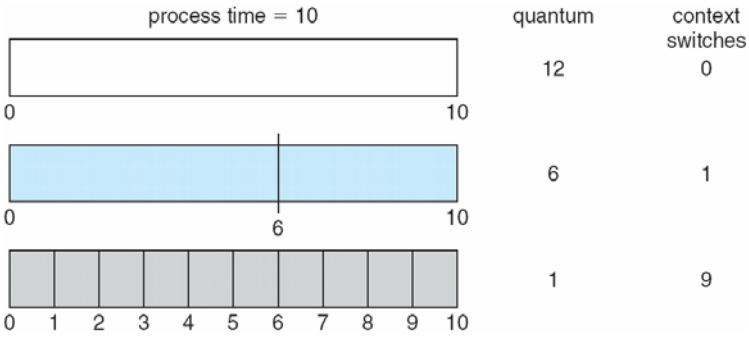


- Typically, higher average *turnaround* than SJF, but better *response*

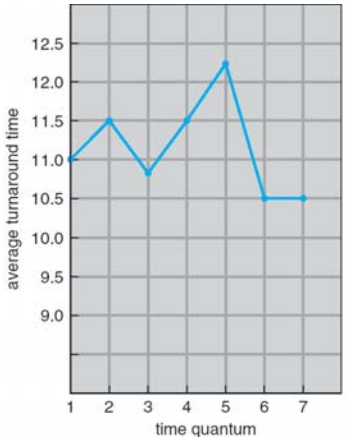


(slide modified by R. Doemer, 04/20/10)

Time Quantum and Context Switch Time



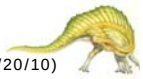
Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

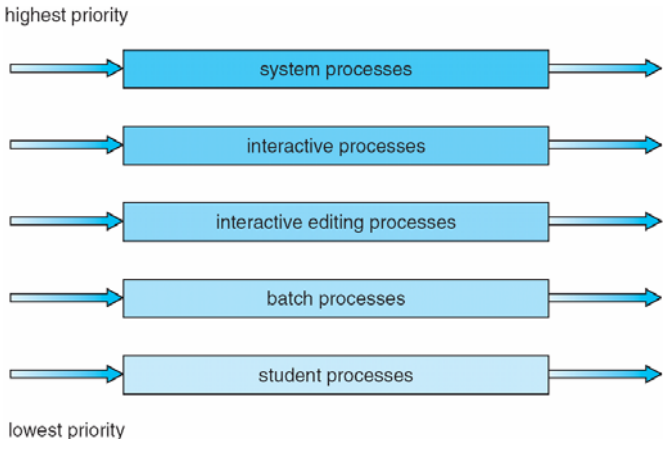
Multilevel Queue Scheduling

- 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



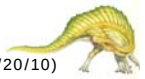
(slide modified by R. Doemer, 04/20/10)

Multilevel Queue Scheduling



Multilevel Feedback Queue

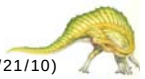
- A process can move between the various queues
  - aging can be implemented this way
- Multilevel-feedback-queue scheduler is defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service



(slide modified by R. Doemer, 04/20/10)

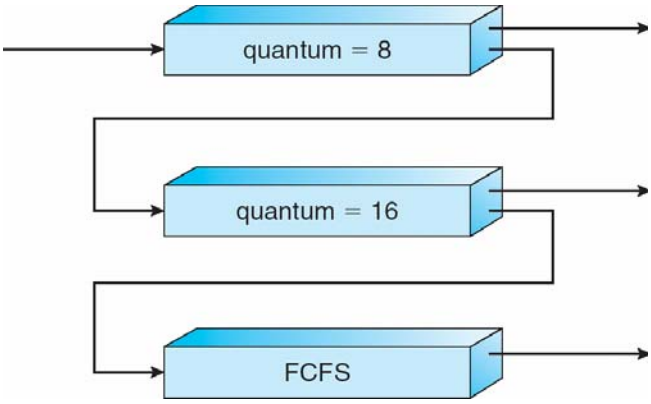
Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – time quantum 8 milliseconds
  - Q_1 – time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is scheduled FCFS.
  - 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 scheduled FCFS and receives 16 additional milliseconds.
  - If it still does not complete, it is preempted and moved to queue Q_2.



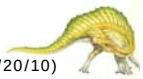
(slide modified by R. Doemer, 04/21/10)

Multilevel Feedback Queues



Thread Scheduling

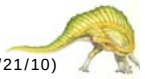
- Distinction between *user-level* and *kernel-level* threads
- With many-to-one and many-to-many models, thread library schedules user-level threads to run on *light-weight processes* (LWP)
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** –
  - competition among all threads in system



(slide modified by R. Doemer, 04/20/10)

Pthread Scheduling API

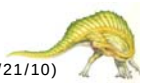
- POSIX API allows specifying either PCS or SCS as thread attribute
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Contention scope can be obtained using
  - **`pthread_attr_getscope(&attr, &scope)`**
- Contention scope can be set using
  - **`pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS)`**
  - **`pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM)`**



(slide modified by R. Doemer, 04/21/10)

Pthread Scheduling API

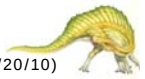
```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
   int i;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;

   /* 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, RR, 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);

   ...
```



(slide modified by R. Doemer, 04/21/10)

Pthread Scheduling API (continued)

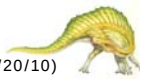
```
...
   /* now join on each thread */
   for (i = 0; i < NUM THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   printf("I am a thread\n");
   pthread_exit(0);
}
```



(slide modified by R. Doemer, 04/20/10)

Multiple-Processor Scheduling

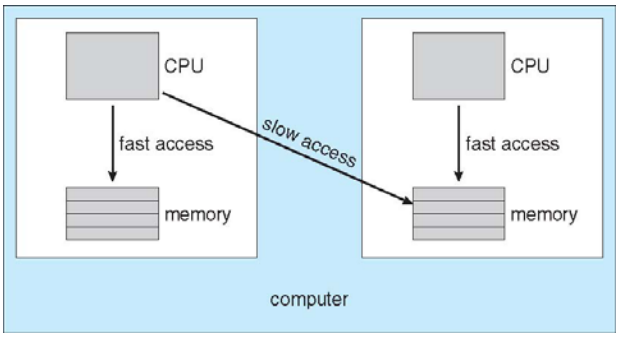
- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** –
  - only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** –
  - each processor is self-scheduling
  - all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity for processor(s) on which it is currently running
  - **soft affinity:** process *should* run on specified processor(s)
  - **hard affinity:** process *must* only run on specified processor(s)

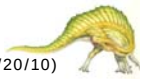


(slide modified by R. Doemer, 04/20/10)

NUMA and Processor Affinity

- Non-Uniform Memory Access (NUMA)
  - A CPU has faster access to some memory than to others

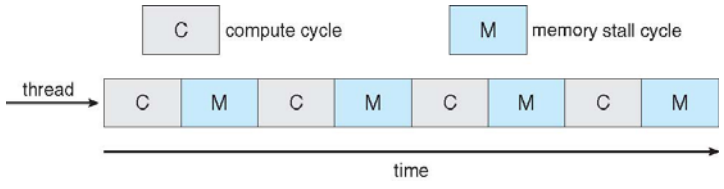


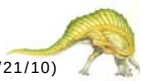


(slide modified by R. Doemer, 04/20/10)

Multi-Core Processors

- Recent trend to place multiple processor cores on same physical chip
  - Typically faster, consumes less power than single-core architecture
- Also, multiple threads per core growing ("hyper-threading")
  - Takes advantage of *memory stall* to make progress on another thread while memory retrieve happens
  - Example:

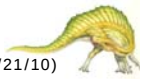




(slide modified by R. Doemer, 04/21/10)

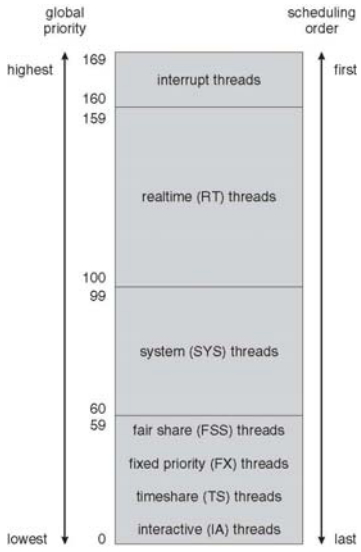
Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling



(slide modified by R. Doemer, 04/21/10)

Solaris Scheduling

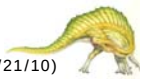


Solaris Dispatch Table

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

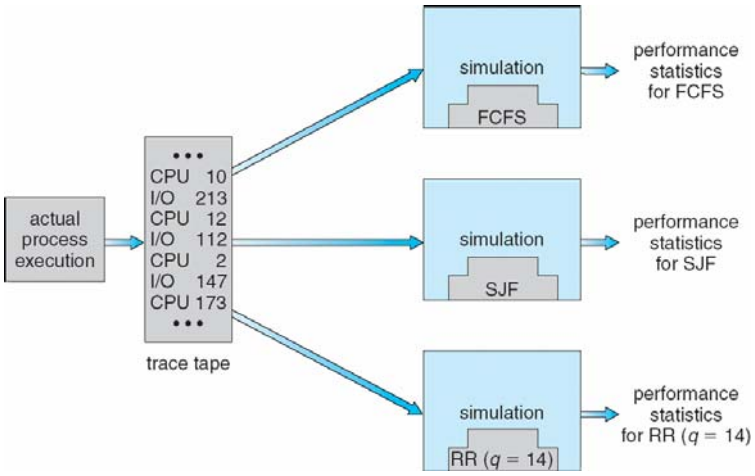
Algorithm Evaluation

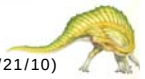
- Deterministic modeling
  - takes a particular predetermined workload and defines the performance of each algorithm for that workload
  - this has been done in this chapter for the algorithms in the beginning
  - Advantages: simple, fast, exact
  - Disadvantages: specific case only
- Queueing models
  - [skipped]
- Simulation
  - see next slide
- Implementation
  - Real evaluation, but expensive…



(slide modified by R. Doemer, 04/21/10)

Evaluation of CPU Schedulers by Simulation





(slide modified by R. Doemer, 04/21/10)

End of Chapter 5