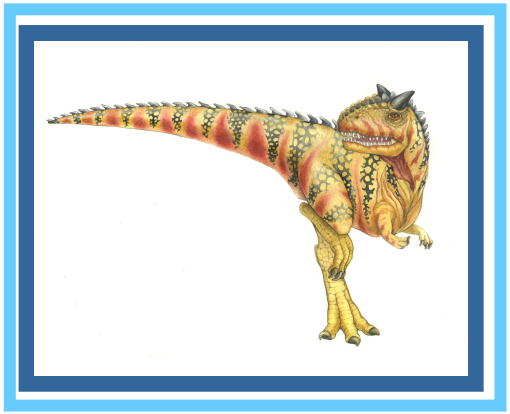
Chapter 5: CPU Scheduling



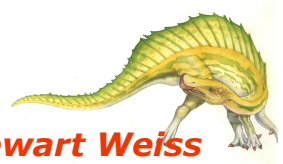
Chapter 5: CPU Scheduling

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

modified by Stewart Weiss

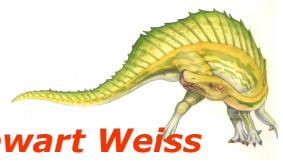
Objectives

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

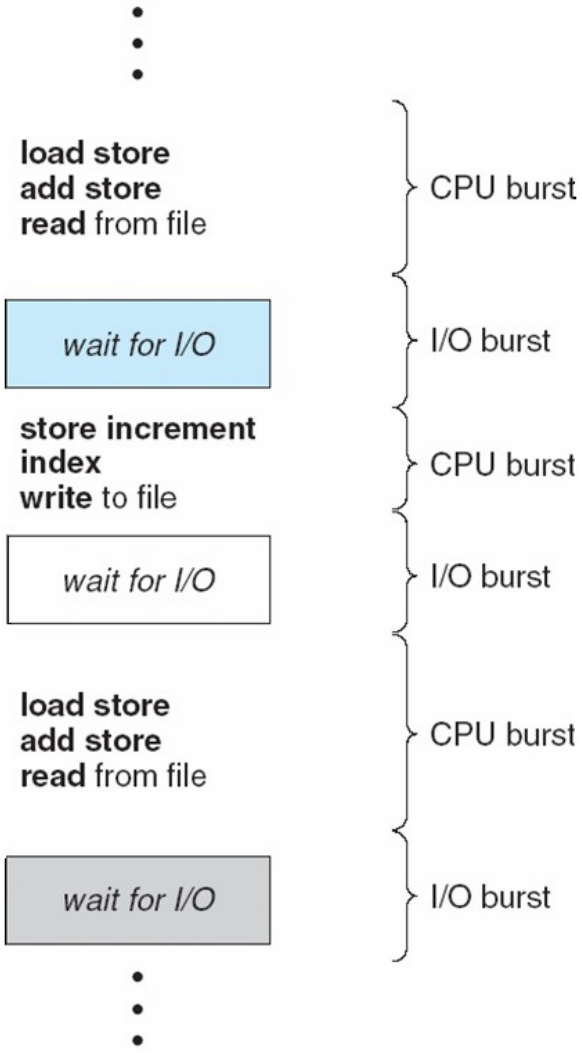


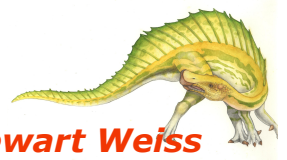
Basic Concepts

- Maximum CPU utilization is obtained by using multiprogramming – keep CPU busy by removing waiting processes and scheduling new one
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of alternating CPU execution and I/O wait, not of equal lengths
- **CPU burst** distribution – historical measurements of CPU burst lengths measured over long periods show exponential or hyperexponential distribution.

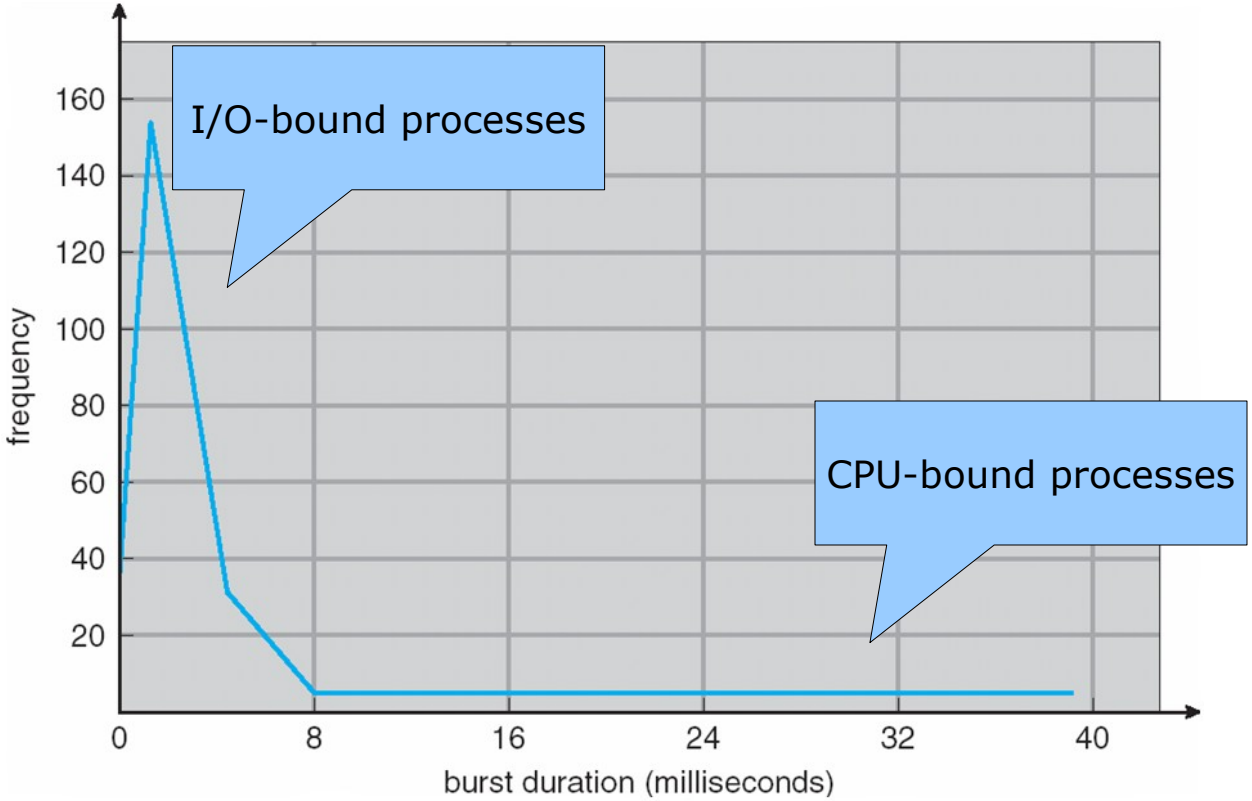


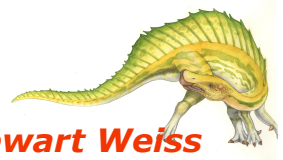
Alternating Sequence of CPU And I/O Bursts





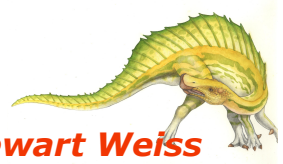
Histogram of CPU-burst Times





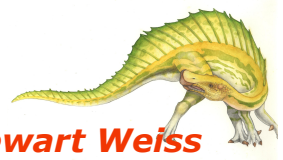
CPU Scheduler

- 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. issues a call that puts it in waiting state
  2. is removed because of an interrupt, and moved to ready queue
  3. is moved to ready queue from wait queue (I/O completion)
  4. terminates
  5. is created and put on ready queue
- If scheduling only occurs for events 1 and 4 , scheduler is called ***nonpreemptive***
- Otherwise it is ***preemptive***



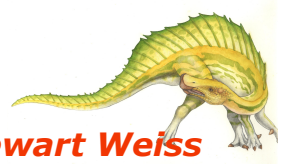
Preemptive Scheduling

- In non-preemptive scheduling, process holds CPU until it issue a wait request or terminates. Easier to implement but processes can hold CPU.
- Preemptive scheduling
  - how is shared data protected when processes can be preempted?
  - can the kernel itself be preempted?
    - different operating systems handle this in different ways
    - certain kernel routines must complete so cannot be preempted
    - what about real time processes?



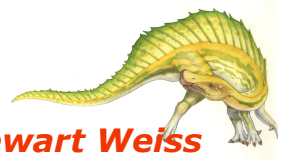
Dispatcher

- The dispatcher is the module that actually 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
- ***Dispatcher runs so frequently that it must be extremely fast***
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running.



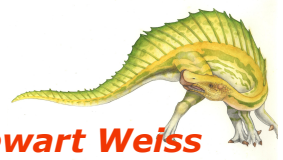
Scheduling Criteria

- The criteria for evaluating how "good" a scheduling algorithm is include:
  - **CPU utilization** – fraction of time CPU is busy (%age)
  - **Throughput** – # of processes that complete their execution per time unit
  - **Turnaround time** – amount of time to execute a particular process = time waiting to enter memory + wait in ready queue + time in CPU + I/O wait time
  - **Waiting time** – total amount of time a process spends 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 example, when the enter key is typed and a shell starts to process a command, until the first output is ready to be sent to terminal.



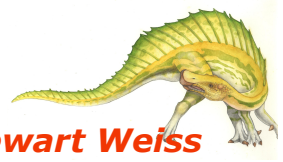
Scheduling Algorithm Optimization Criteria

- Deciding which criteria to satisfy depends on what type of system.
- Ideally we want to
  - Maximize CPU utilization
  - Maximize throughput
  - Minimize turnaround time
  - Minimize waiting time
  - Minimize response time
  - but cannot do all because it is impossible.
- It is an ***optimization problem*** – optimize max, min or average
- E.g., minimize the maximum waiting time, or minimize average waiting time
- Usually in timesharing systems, minimize variance in response time



Scheduling Algorithm Evaluation

- Different measures of performance will result in different rankings of algorithms.
- To analyze algorithms here, we model processes by the lengths of their CPU bursts and use the average waiting time as a measure of performance.
- Other ways to measure are found in the book in Section 5.7

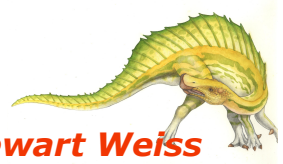


Scheduling Algorithms

- A CPU scheduling algorithm is an algorithm that decides which process in the ready queue to schedule next.
- Input to the algorithm is a sequence of process ids together with the lengths of their CPU bursts.
- E.g.:

```
P1::24,  P2:6, P3:14,  P4:36
```

- The output can be represented as a Gantt Chart,which is a bar chart that shows the start and finish times of each process.



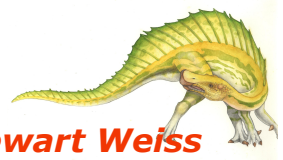
First-Come, First-Served (FCFS) Scheduling

- **First-Come-First-Served** (**FCFS**) schedules the process at the front of the ready queue, and each new process is put on the back of the ready queue.

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

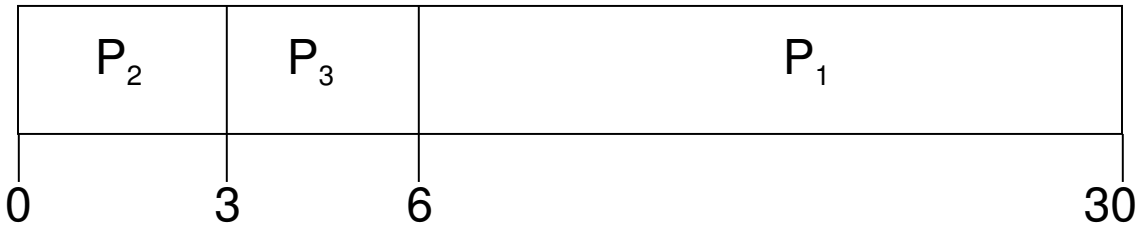


FCFS Scheduling (Cont)

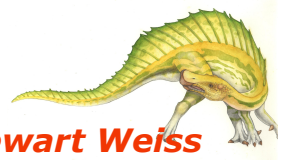
Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

- 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* is when many short processes are behind long process (see next slide)

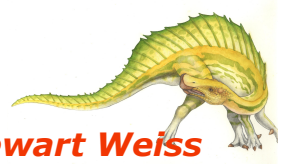


FCFS Scheduling : Performance

- FCFS is nonpreemptive
- If job mix is bad, we can get poor CPU & I/O utilization. Consider the **convoy effect**:
- One CPU-bound & many I/O bound

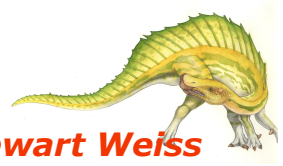
1. The long CPU bound proces hogs CPU while all I/O bound processes finish their I/O and move into ready queue; I/O devices are idle
2. CPU-bound process issues I/O request
3. All I/O-bound processes run through CPU quickly, move into I/O queues
4. CPU-bound process is still waiting, so CPU is idle
5. CPU-bound process finishes I/O moves onto CPU, and Step 1 repeats

- FCFS is bad because of possibility of long response time; mostly used for low-level process queues and batch jobs.



Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. The next process to get the CPU is the one with the shortest time. If two or more processes have the same burst time, ties are broken with FCFS.
- SJF is **provably optimal** – gives minimum average waiting time for a given set of processes, but ...
  - The difficulty is that the length of the next CPU request is usually not known!
- In batch systems, users usually submit estimates of running time in the job control "cards", and those estimates are used for a method of long term scheduling like SJF. If a user estimates too low, the job is penalized by removing it from the CPU and moving it to the end of the queue, to prevent users from purposely underestimating to get their jobs to run first.
- Not used exactly in short-term scheduler.

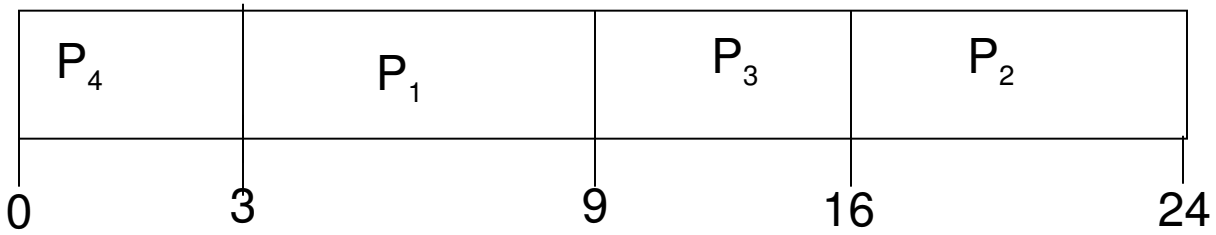


Example of SJF

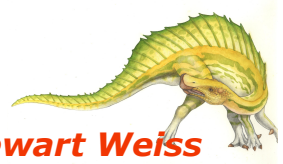
- Assume all processes are in ready queue initially.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



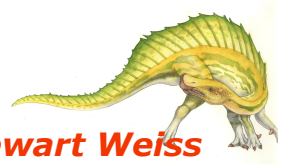
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7



Determining Length of Next CPU Burst

- Can predict the length of next CPU burst from ***history of process***
- Can be done by using the length of previous CPU bursts, using exponential averaging
- Let t_n = length of n^{th} CPU burst.
- Let τ_{n+1} = predicted length of $(n+1)^{st}$ CPU burst
- Let c be any number such that 0 <= c <= 1. Define

  $\tau_{n+1} = c^*t_n + (1-c)^*\tau_n$
- This is called an ***exponential average with parameter c***. τ_n embodies past history, and t_n is the most recent event. As c approaches 0, the past is weighted more in the prediction; as c approaches 1, the past is ignored more.



Exponential Averaging

- Expanding the formula repeatedly,

$$\tau_{n+1} = c*t_n + (1-c)*\tau_n$$

$$\tau_{n+1} = c*t_n + (1-c)*\tau_n$$

$$= c*t_n + (1-c)c*t_{n-1} + (1-c)^2*\tau_{n-1}$$

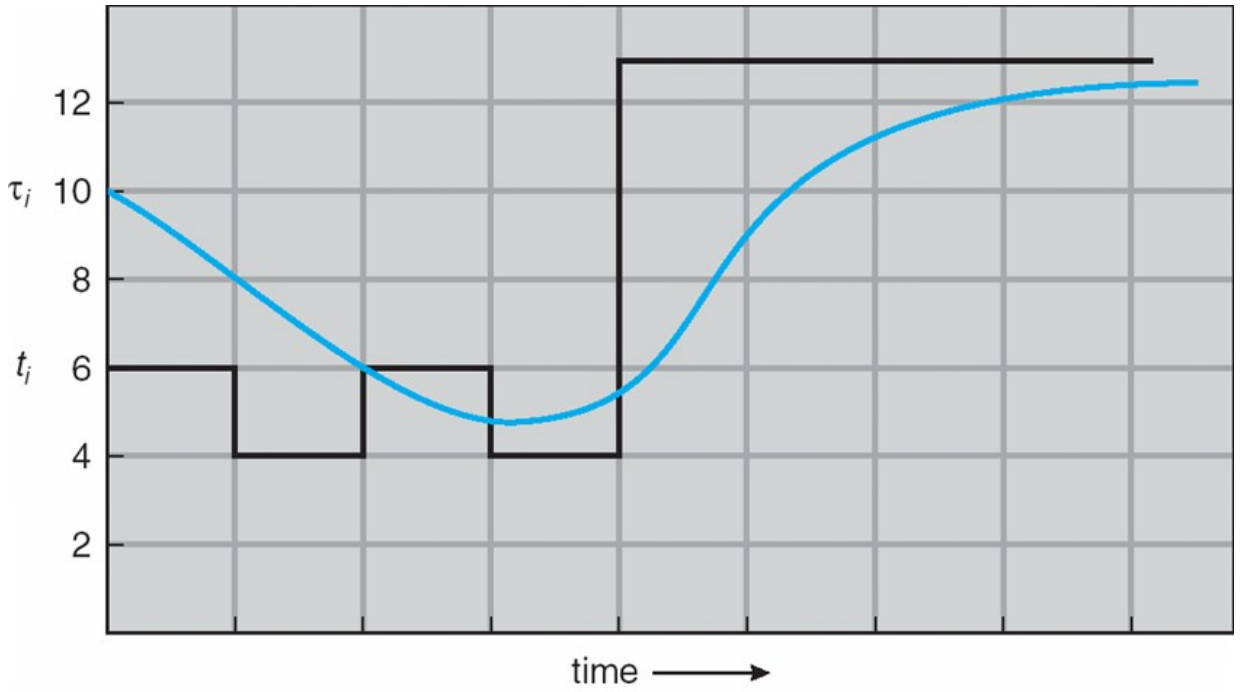
$$= c*t_n + (1-c)c*t_{n-1} + (1-c)^2c*t_{n-2} + (1-c)^3*\tau_{n-2}$$

$$\ldots$$

$$= c*t_n + (1-c)c*t_{n-1} + \ldots + (1-c)^kc*t_{n-k} + \ldots + (1-c)^{n+1}*\tau_0$$

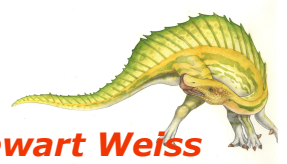
- Since c and (1-c) are both less than 1, successive terms are smaller and smaller.

Prediction of the Length of the Next CPU Burst



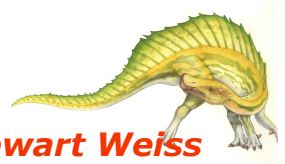
CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

c=0.5, τ0 = 10



Shortest-Remaining-Time-First

- SJF can also be preemptive.
- When a process is put in the ready queue, if its next CPU burst is smaller than the ***remaining time of the currently running process***, the running process is preempted and put back in the ready queue, and the shorter one is run.
-

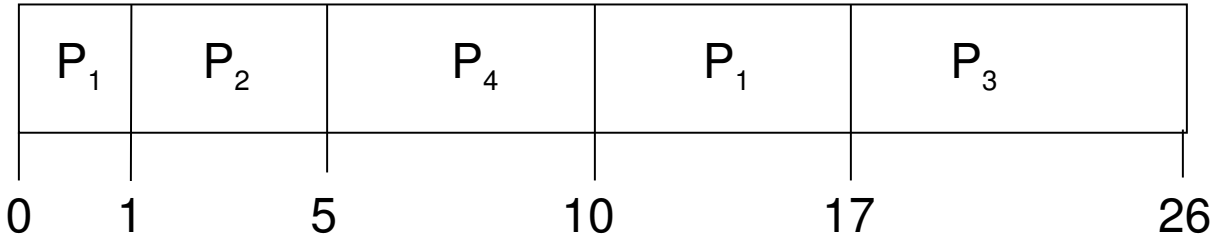


Example of SRTF

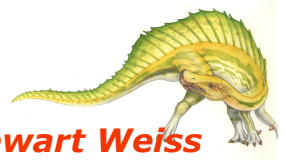
- Assume all processes are in ready queue initially.

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- SRTF scheduling chart



- Average waiting time is the sum over all processes of (finishing time – arrival time – burst time)/(number of processes)
- [(17-0-8) + (5-1-4) + (26-2-9) + (10-3-5)]/4 = 26/4 = 6.5

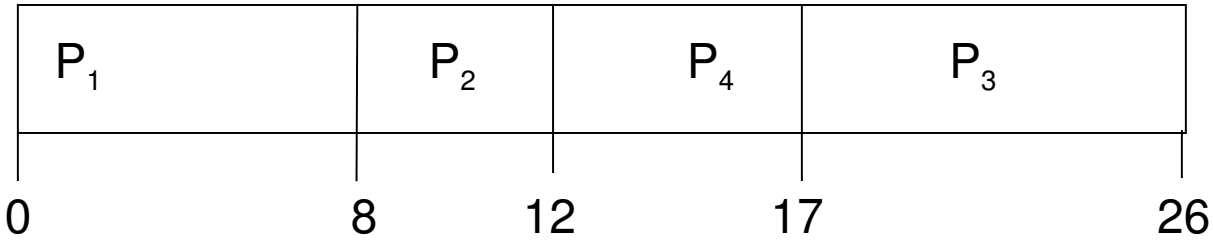


Example of SJF

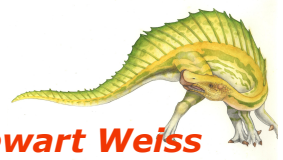
- Under SJF, this same set of processes, with the same arrival times:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- has the Gantt chart

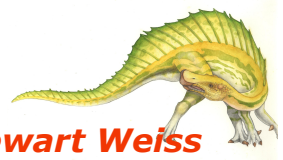


- Average waiting time is the sum over all processes of (finishing time – arrival time – burst time)/(number of processes)
- [(8-0-8) + (12-1-4) + (26-2-9) + (17-3-5)]/4 = 29/4 = 7.75



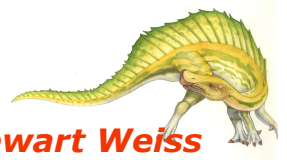
Priority Scheduling

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



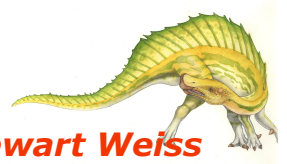
Priority Scheduling Examples

- UNIX Priorities
  - System processes all have higher priorities than user processes
  - System process priority determined by event that signals its wake-up
    - process that just got I/O buffer >
    - process whose child just terminated >
    - process just resumed from suspension
  - Priority of user process = actual CPU time so far/elapsed time
- I/O bound processes favored because they get little CPU time



Round Robin (RR)

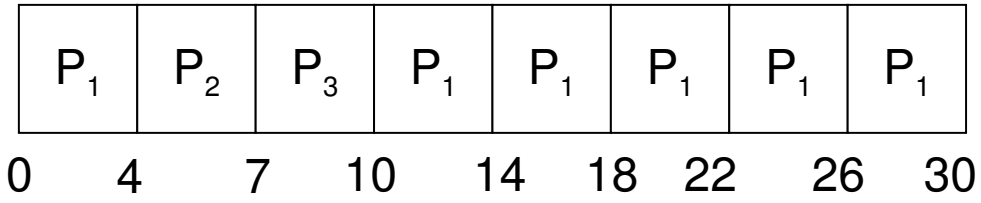
- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue on average and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- If a process blocks itself, before quantum expires, it loses CPU.
- Performance
  - *If q* large it is like FIFO (FCFS), response time long
  - *q* small, response time is faster but *q* must be large with respect to context switch, otherwise overhead is too high



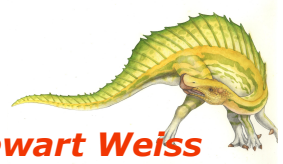
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

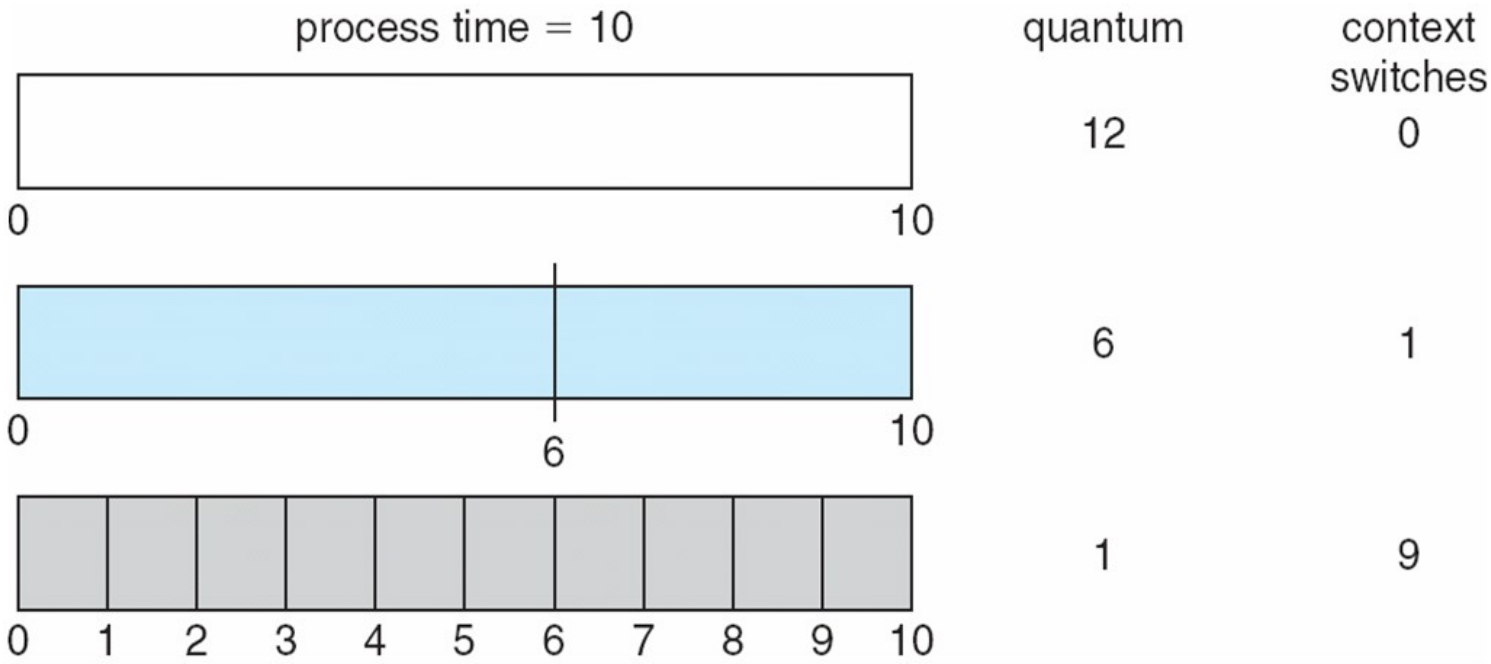
- The Gantt chart is (ignoring context-switch times):



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



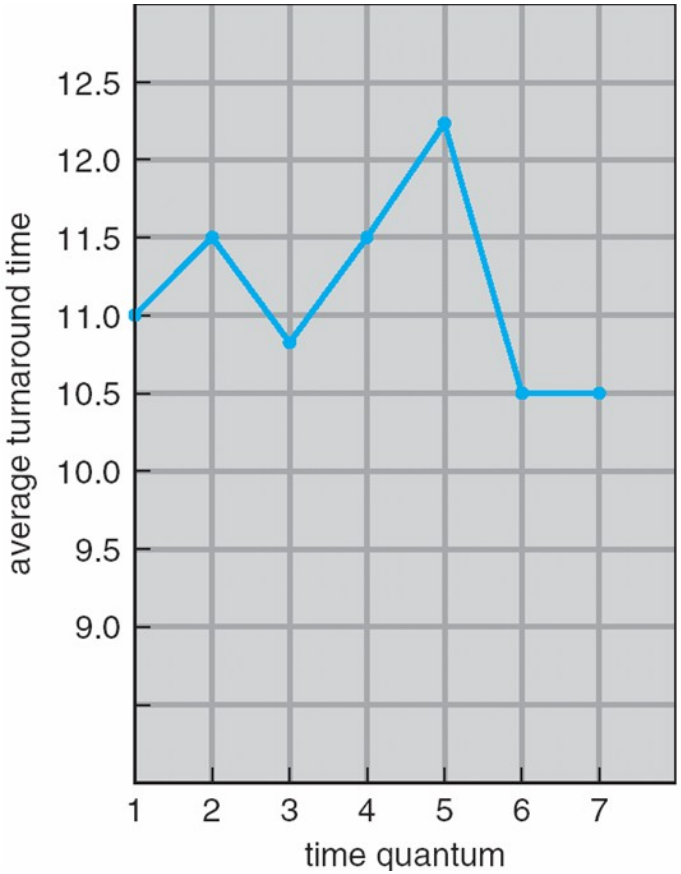
Time Quantum and Context Switch Time



If q is quantum size and c is context-switch time,
0 <= overhead = c/(q+c) <= 1
For q near 0, overhead ~ 1. For very large q, overhead ~ 0

General rule: best quantum is smallest size such that 80% of processes finish burst in a single quantum.

Turnaround Time Varies With The Time Quantum

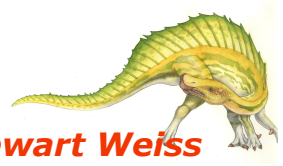


process	time
P_1	6
P_2	3
P_3	1
P_4	7

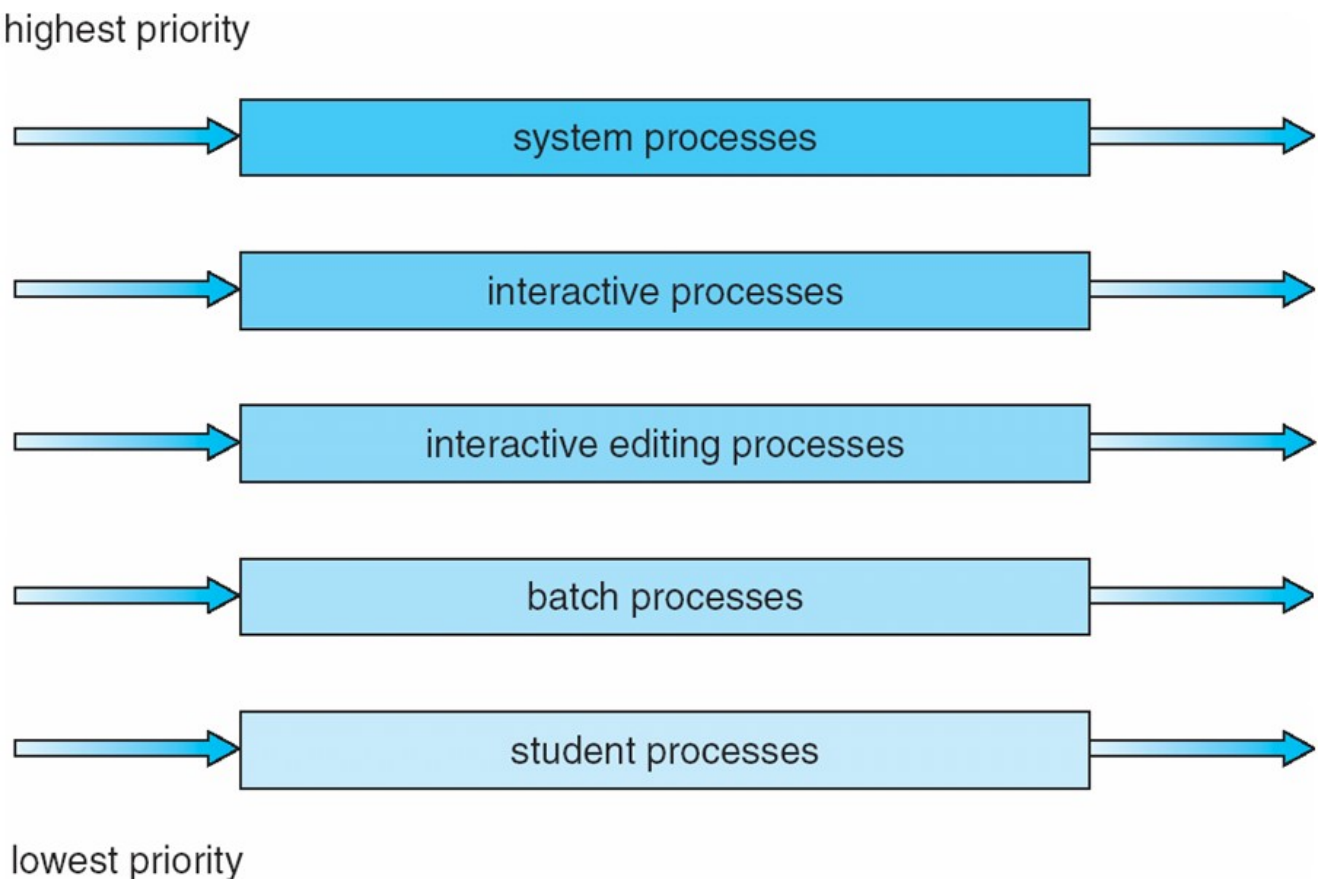
Turnaround time varies as the quantum varies, but not linearly – it depends on the burst sizes of the mix of processes in the ready queue.

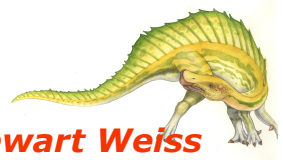
Multilevel Queue

- Two-level queuing scheme:
  foreground queue (interactive) – uses RR
  background queue (batch) -- uses FCFS
- Multi-level queuing scheme: multiple queues, each with its own scheduler, each queue having lower priority than preceding one.
- Processes are assigned to queues
- Scheduling must be done between the queues, i.e., which queue goes next
  - 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



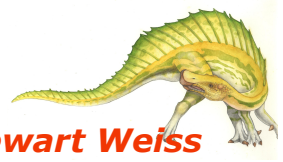
Multilevel Queue Scheduling





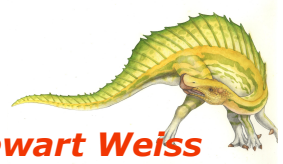
Multilevel Feedback Queue

- When a process can move between the various queues, it is called multi-level feedback queuing; aging can be implemented this way
- Multilevel-feedback-queue scheduler 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

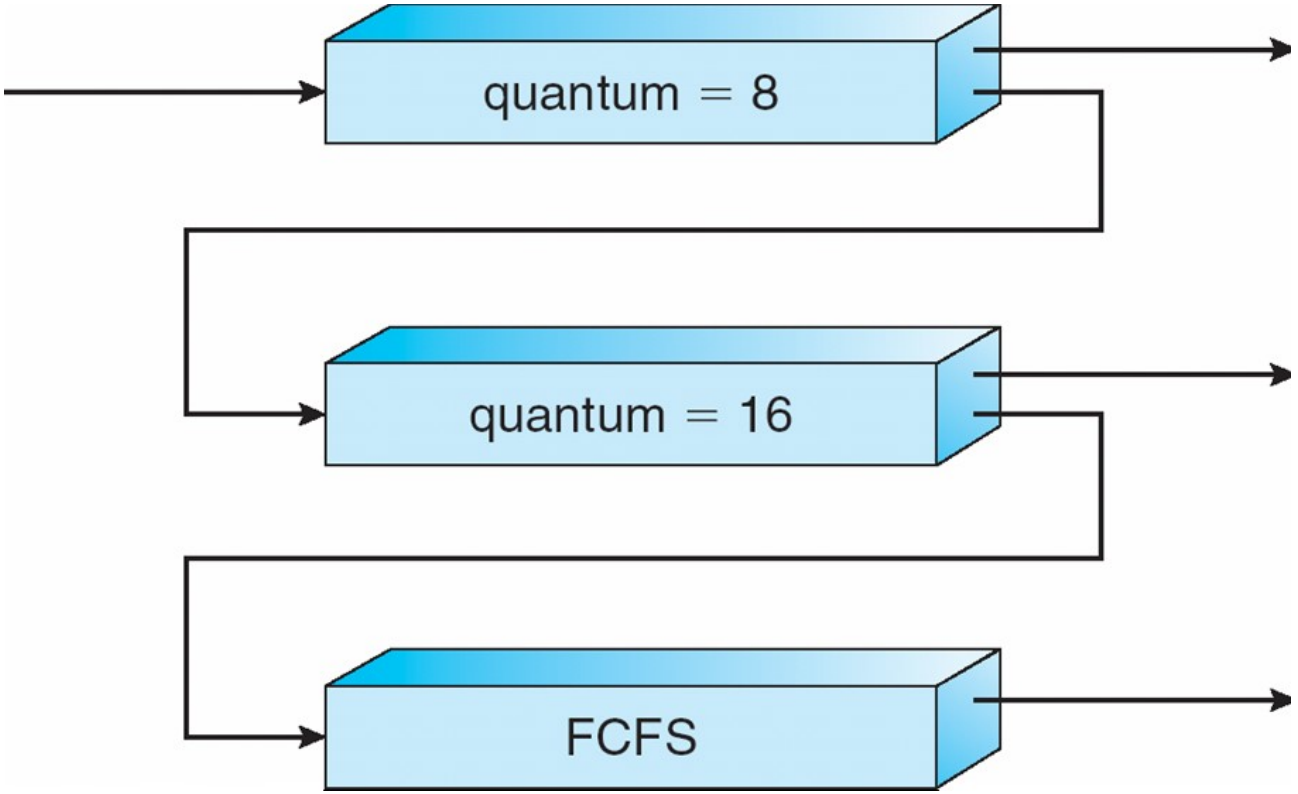


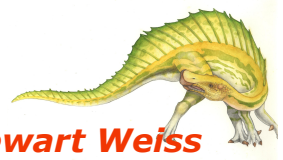
Example of Multilevel Feedback Queue

- 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 queue Q_0 which is served 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 served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.



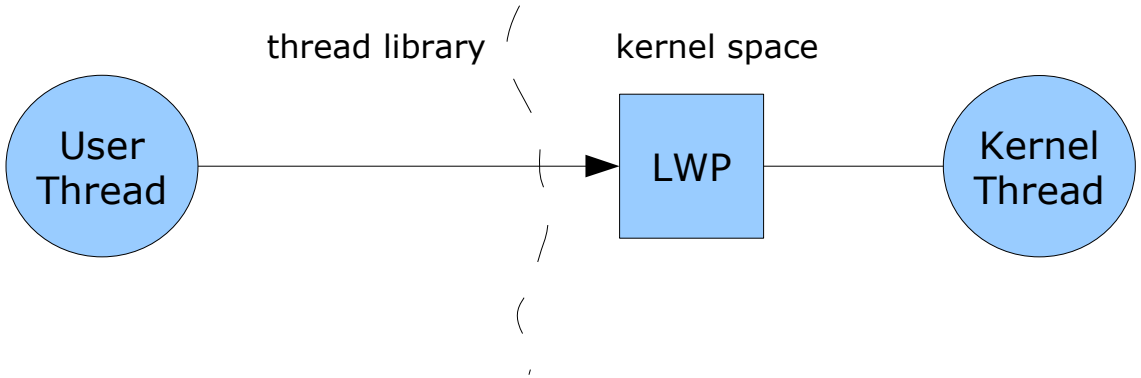
Multilevel Feedback Queues

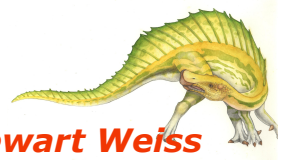




Threads and LWPs

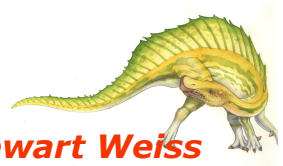
- Some operating systems (e.g., Linux, Solaris) that support user-level threads use an intermediary between a user thread and a kernel thread that is (unfortunately) known as a **Lightweight Process**, or **LWP**.
- A LWP is essentially attached to a kernel thread; each kernel thread has a single LWP, and the pair are scheduled together. The LWP is a kernel object, not a user object. It is a data structure that presents the abstraction of a processor available to that thread.
- The LWP is attached to the user thread within the user thread library.





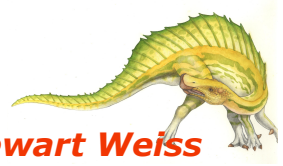
Threads and LWPs

- Operating systems that use LWPs maintain a pool of them. If a user thread is to be scheduled, it needs a free LWP. If one is available, it is scheduled onto it. If not it must wait. If a user thread issues a blocking call, the LWP is blocked. The task can be given a new LWP if one is available, for other threads that need execution.
- Operating systems like Solaris maintain lists of blocked and free LWPs; they also communicate with the thread library, notifying it when threads become blocked, via an ***upcall***. The library has an ***upcall handler***, which is code that runs asynchronously when an upcall signal occurs.



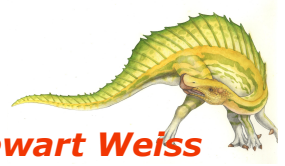
Thread Scheduling

- Many-to-one and many-to-many models use this LWP method of scheduling.
  - Known as **process-contention scope (PCS)** since threads within a process compete for virtual CPUs.
- Kernel threads are scheduled onto physical CPUs by the kernel. This is called **system-contention scope (SCS)** because all threads in the system compete for the real CPU resources.



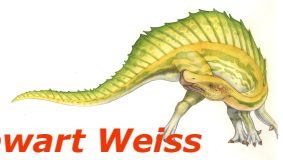
Pthread Scheduling

- 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.
  - Some systems (Linux, Mac OS X) will not allow the process to choose PTHREAD_SCOPE_SYSTEM.

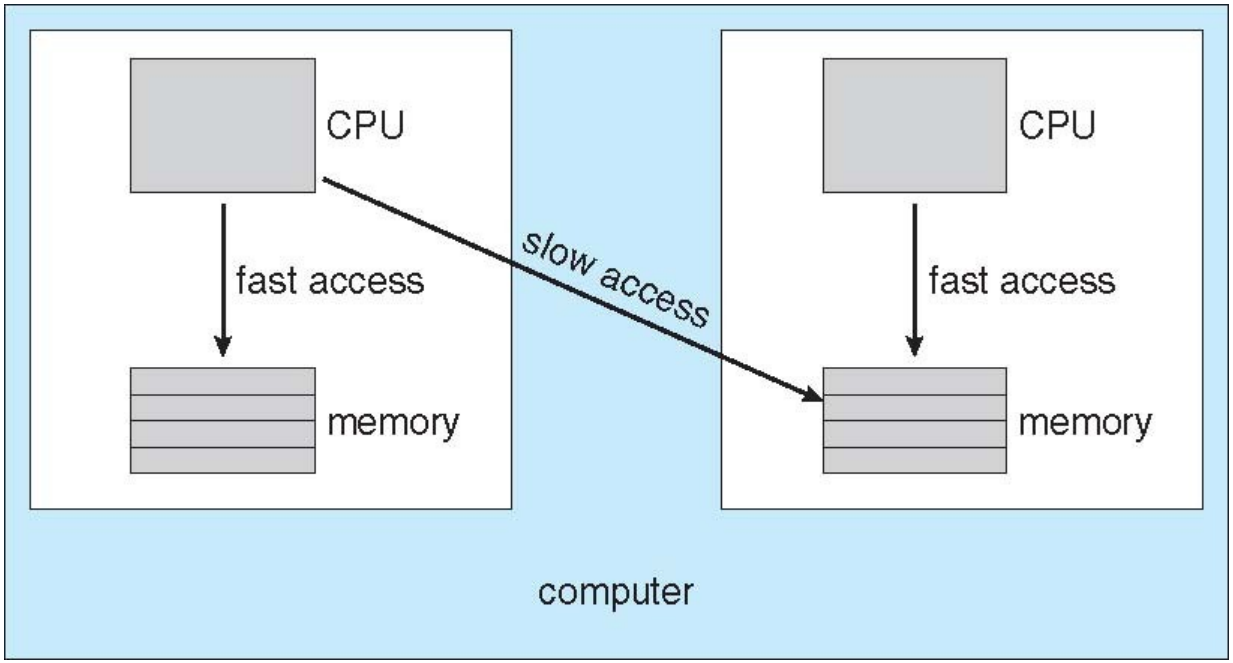


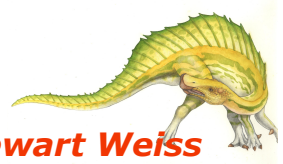
Multiple-Processor Scheduling

- 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 on which it is currently running
  - **soft affinity**
  - **hard affinity**



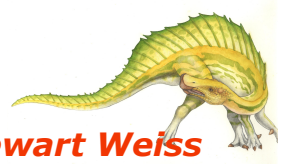
NUMA and CPU Scheduling



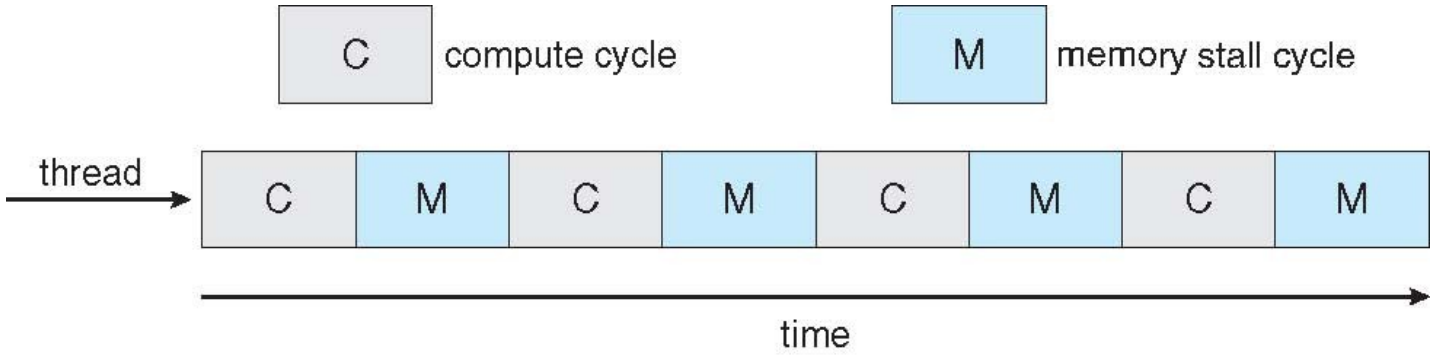


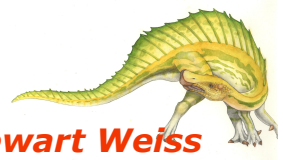
Multicore Processors

- 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



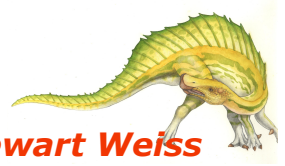
Multithreaded Multicore System





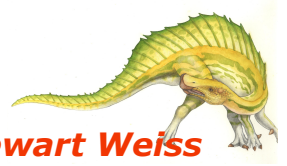
Operating System Examples

- Solaris scheduling
- Linux scheduling



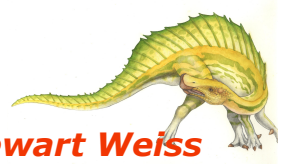
Solaris Scheduling

- Each thread is assigned one of six possible priority classes:
  - Time sharing (default class)
  - Interactive
  - Real time (get highest priority)
  - System (only kernel processes)
  - Fair share
  - Fixed priority
- Within classes there is further prioritizing.
- Time sharing and interactive classes use multilevel feedback queue:
  - higher priority get smaller quanta
  - CPU-bound get lower priority; interactive, higher



Solaris Scheduling

- System class excludes the kernel phase of user processes – only kernel processes are in this class and have fixed priority.
- There are class-specific priorities, but all get converted to global priorities which are then compared.

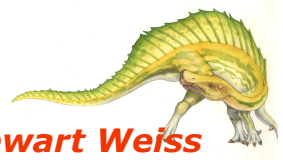


Solaris Dispatch Table

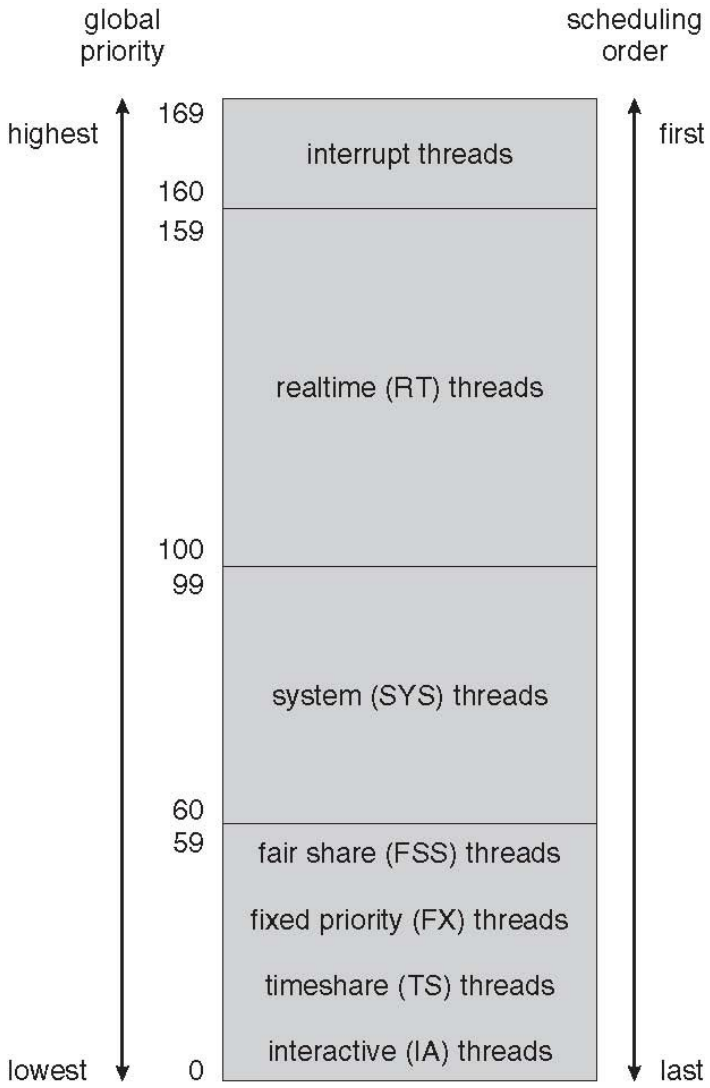
- The time-quantum-expired column is the priority given to a thread that already used a full quantum. Note that it is penalized with lower priority.

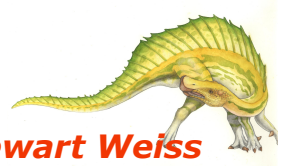
- The return-from-sleep column in the priority given to a process back in the ready queue because I/O was satisfied; note the reward for issuing I/O request.

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



Solaris Global Priorities





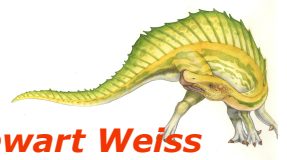
modified by Stewart Weiss

Linux Scheduling

- Constant order *O*(1) scheduling time
- Preemptive, priority-based algorithm
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
- Higher priority get larger quanta

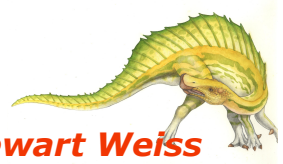
Priorities and Time-slice length

numeric priority	relative priority		time quantum
0	highest	real-time tasks	200 ms
•			
•			
•			
99			
100		other tasks	
•			
•			
•			
140	lowest		10 ms



Linux Scheduling Details

- Each processor has its own ready queue called a runqueue.
- Every task is either active (has not used up its quantum) or has expired (used up its last quantum).
- Expired tasks must wait for all active processes to run.
- When all tasks in the active queue have finished their quanta, the two queues are swapped.
- Each runqueue has its own scheduler and the highest priority task in that queue is run next.
- A task that sleeps long has its priority increased by 5; a task that sleeps little has its priority decreased by 5.
- All tasks in the expired array have their priorities adjusted when the arrays are swapped.



List of Tasks Indexed According to Priorities