Running Programs on a System

Process Scheduling

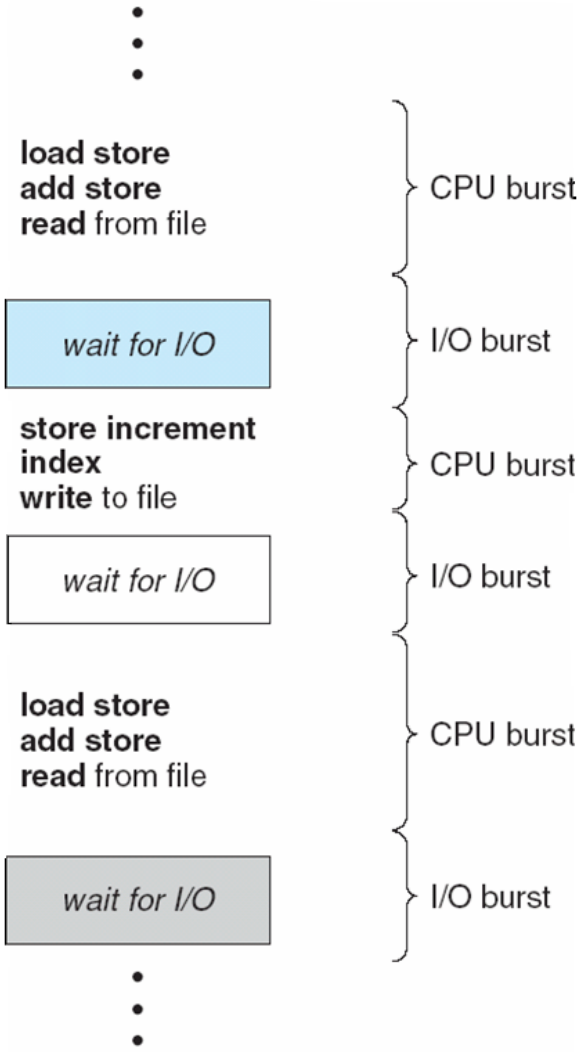
Process Scheduling

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

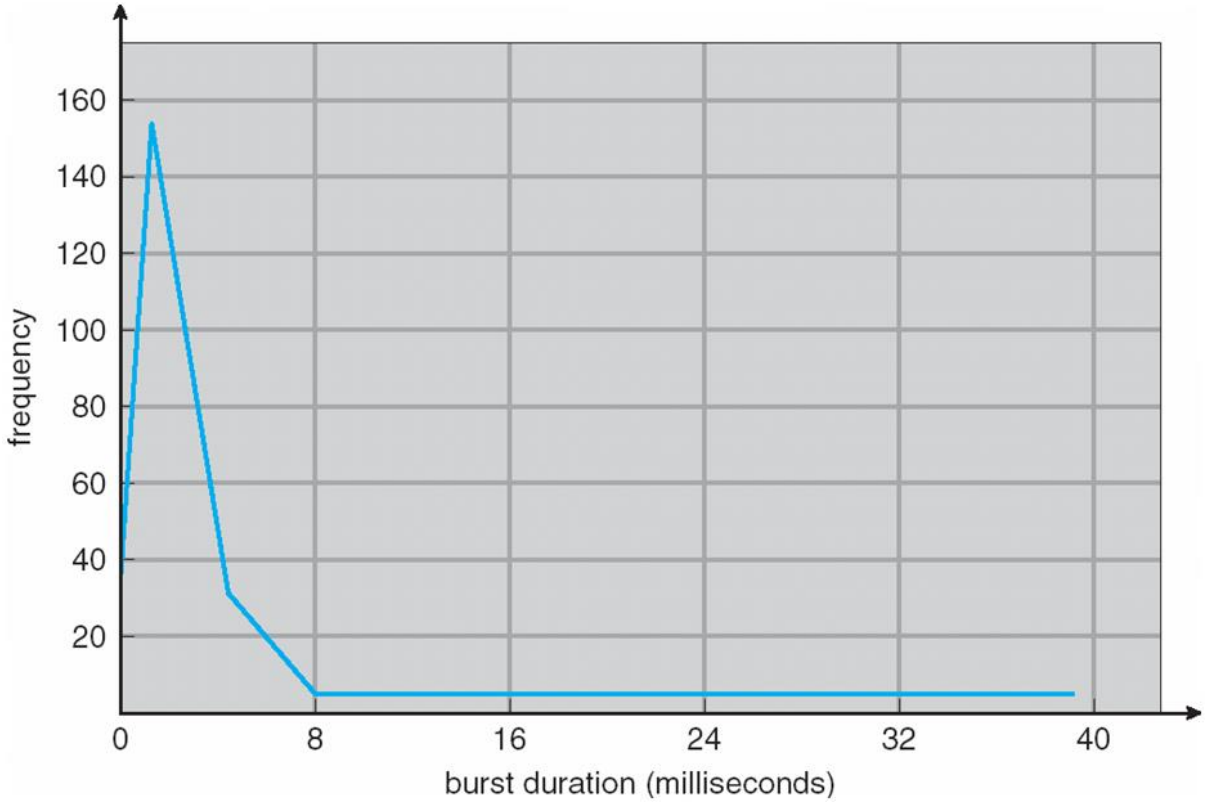
Basic Concepts

- Goals of scheduling:
  - maximize CPU utilization
    have some process running at all times
  - Fair distribution of resources (CPU)
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle
  Process execution consists of a cycle of CPU execution and I/O wait

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
  - not necessarily FIFO (priority queue, tree, unordered list, …)

- 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 **nonpreemptive**
- All other scheduling is **preemptive**

CPU Scheduler

- **Non-preemptive** or **cooperative** scheduling
  - process keeps CPU until it switches to the waiting state or terminates

- **Preemptive** scheduling
  - process interrupted while in running state (due to an interrupt)
  - requires process coordination and locks to protect access to shared data
    - both between processes
    - and kernel data structures
      - UNIX: no preemption while kernel data structures in inconsistent state
        - certain interrupts must be handled (x86: CLI/STI)
        - poor performance for real-time systems and multiprocessing

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
  - includes the time to pick a new process (CPU scheduler)

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)

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time
  - for interactive systems: min variance in response time

Process Scheduling: Algorithms

- First-Come, First-Served
- Shortest-Job-First
- Priority Scheduling
- Round Robin
- Multilevel Queue
- Multilevel Feedback Queue

First-Come, First-Served (FCFS) Scheduling

- Non-preemptive scheduler

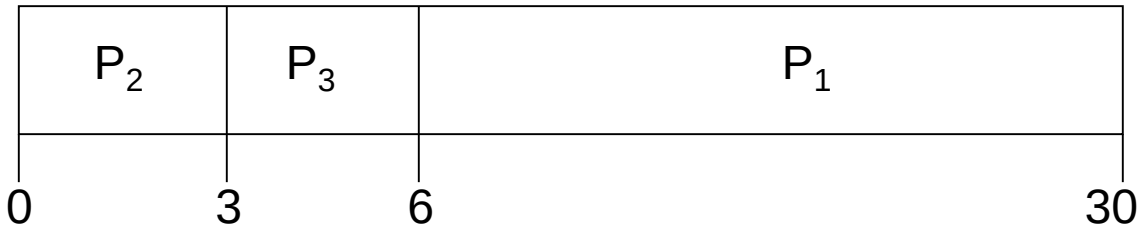
Process	Burst Time
P_1	24
P_2	3
P_3	3

- 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
  - e.g., one CPU-bound, several I/O bound processes

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
(FCFS for equal lengths)

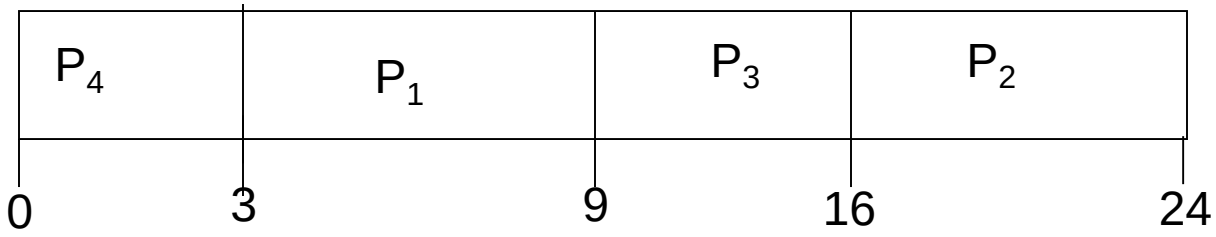
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request

- SJF frequently used in long-term scheduling
  - for short-term scheduling, we need to estimate the length of the next CPU burst

Example of SJF

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

- SJF scheduling chart



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

Determining Length of Next CPU Burst

- Can only estimate the length
- 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 : $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$

Examples of Exponential Averaging

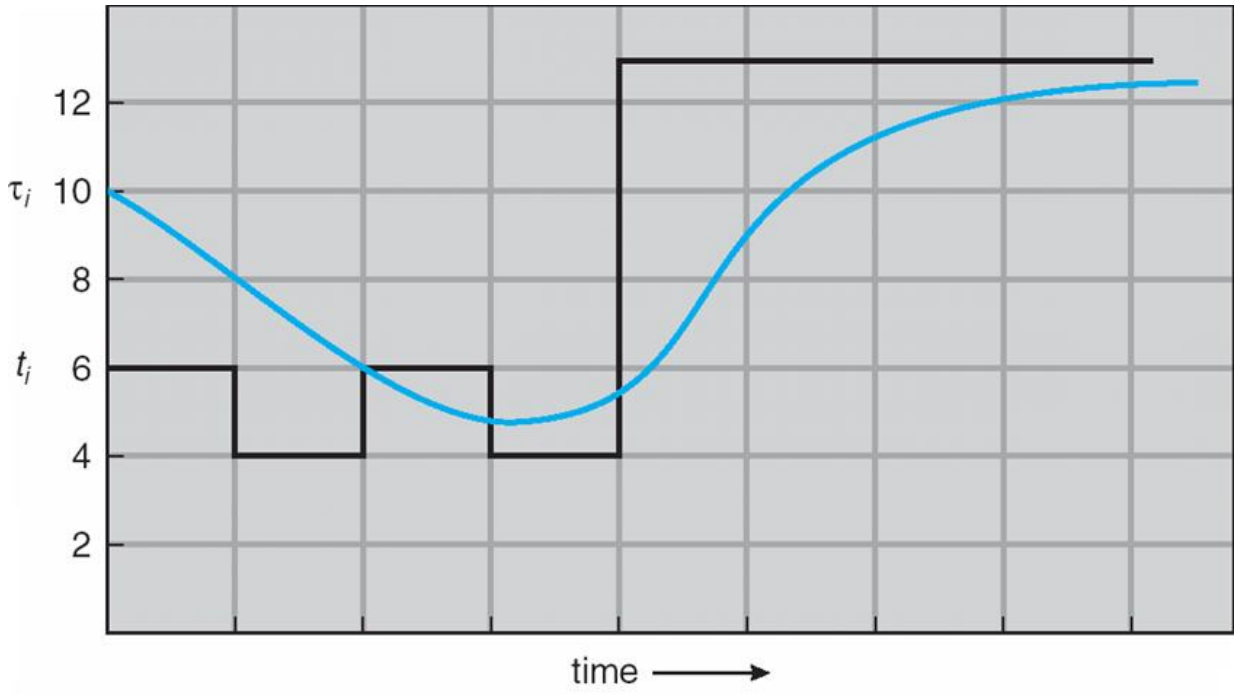
- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts

- 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 $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Prediction of the Length of the Next CPU Burst

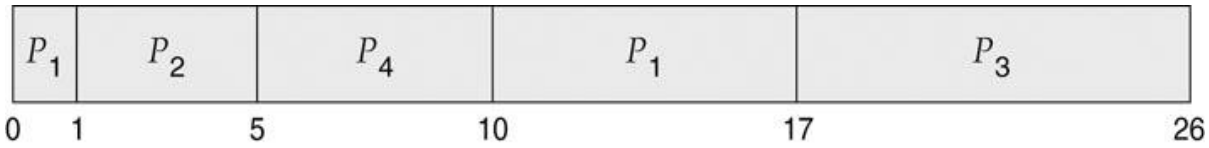


CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

SJF Scheduling

- can be preemptive or cooperative
  - choice arises when a new job arrives
  - **shortest-remaining-time-first** scheduling

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



Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
  (WLOG smallest integer ≡ highest priority)
  - preemptive
  - nonpreemptive
- Priority
  - internal: measurable quantity (time limit, open files, CPU time, …)
  - external: set by criteria outside the OS (importance, funds, …)
- SJF is a priority scheduling where priority is the predicted next CPU burst time

Priority Scheduling

- **Starvation**
  - aka indefinite blocking
  - low priority processes may never execute if there is a constant stream of high priority processes

- A solution to starvation is **aging**
  - gradually increase the priority of the process as the waiting time increases
  - reduce when run

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

all processes arrive at time 0
average waiting time: 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.

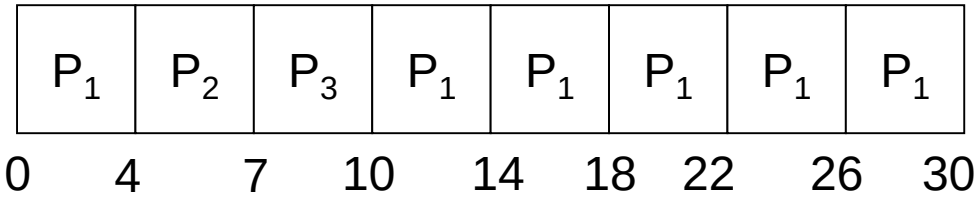
- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once. No process waits more than (n-1)q time units.

- Performance
  - q large ⇒ FIFO
  - q small ⇒ each process running at ~1/n-th the speed of the CPU (→ q must be large with respect to context switch, otherwise overhead is too high)
  - rule of thumb: 80% of CPU bursts should be shorter than q

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

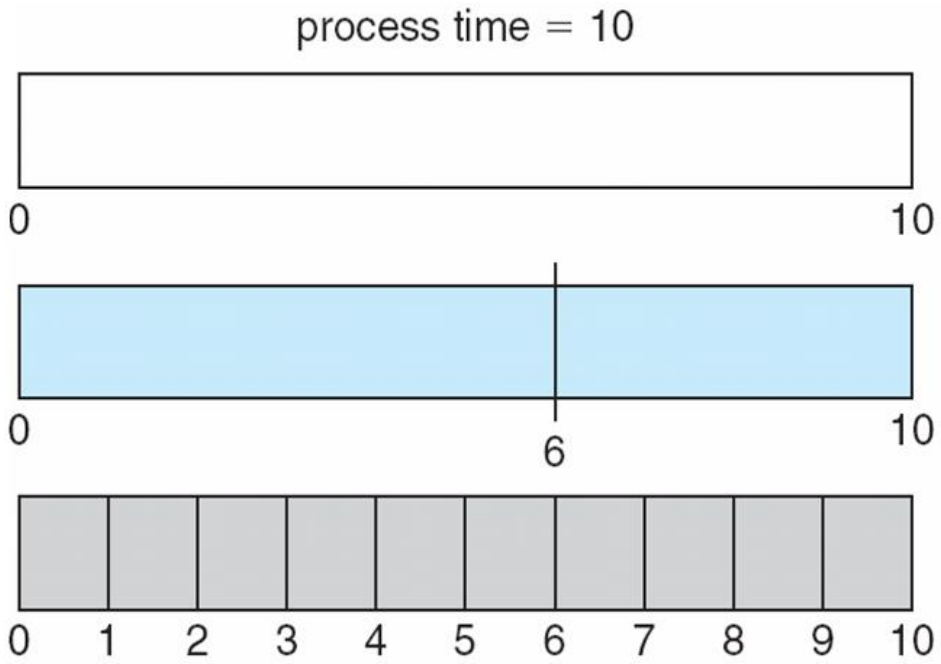
- With a time quantum of 4, the Gantt chart is:



average waiting time: ((10-4) + 4 + 7)/3 = 5.66

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

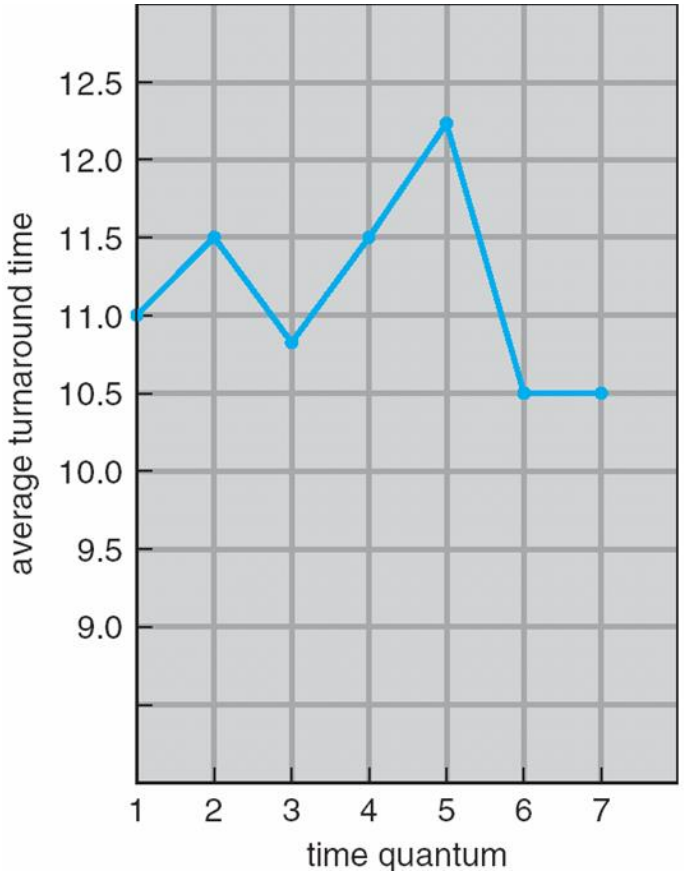
Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

- typical numbers on modern systems:
  - time quantum: 10 to 100 milliseconds
  - context switch: < 10 microseconds

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

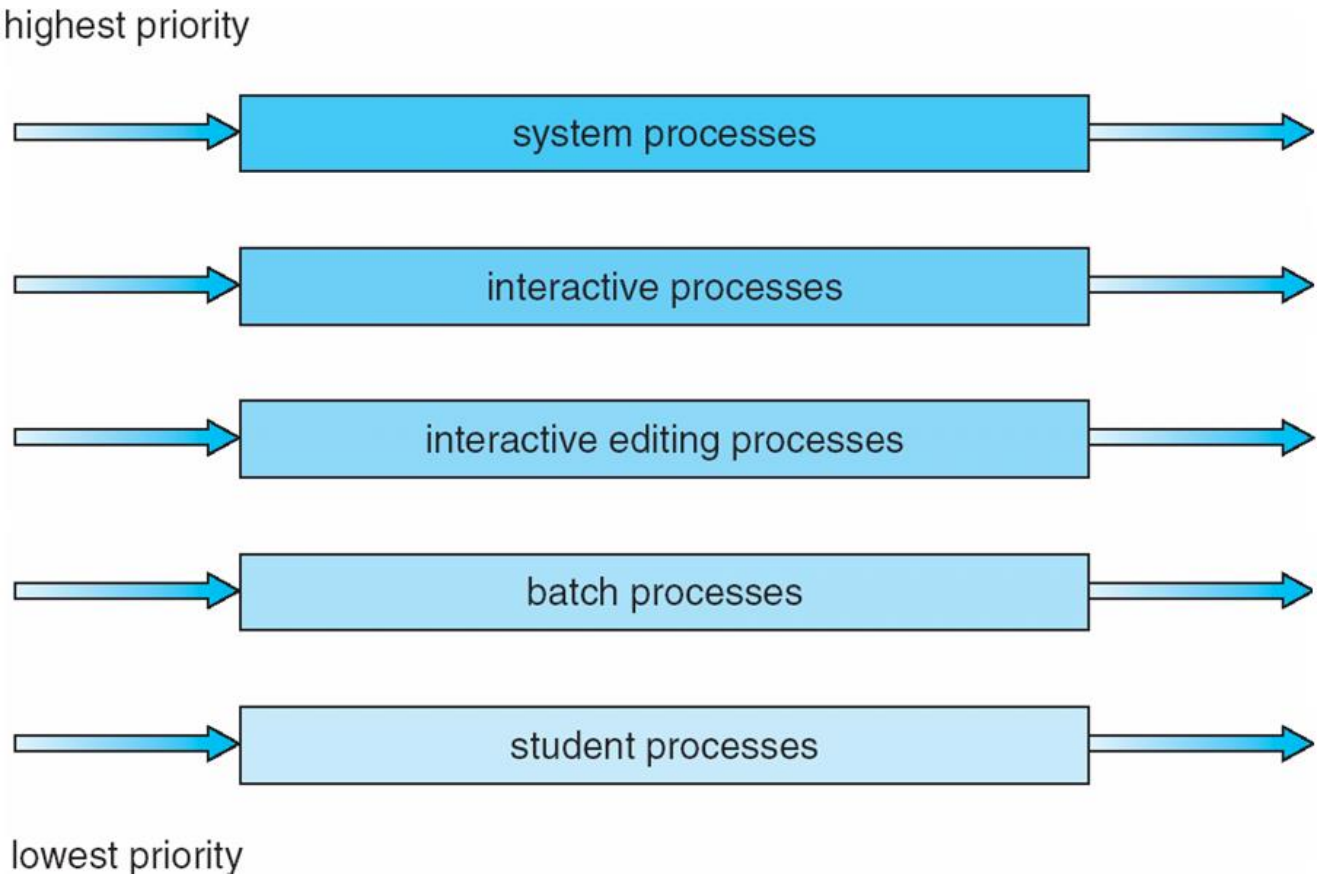
Multilevel Queue

- Ready queue is partitioned into separate queues:
  e.g., foreground (interactive) and background (batch)
  - processes are assigned to the queues based on some criterion

- Each queue can have 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



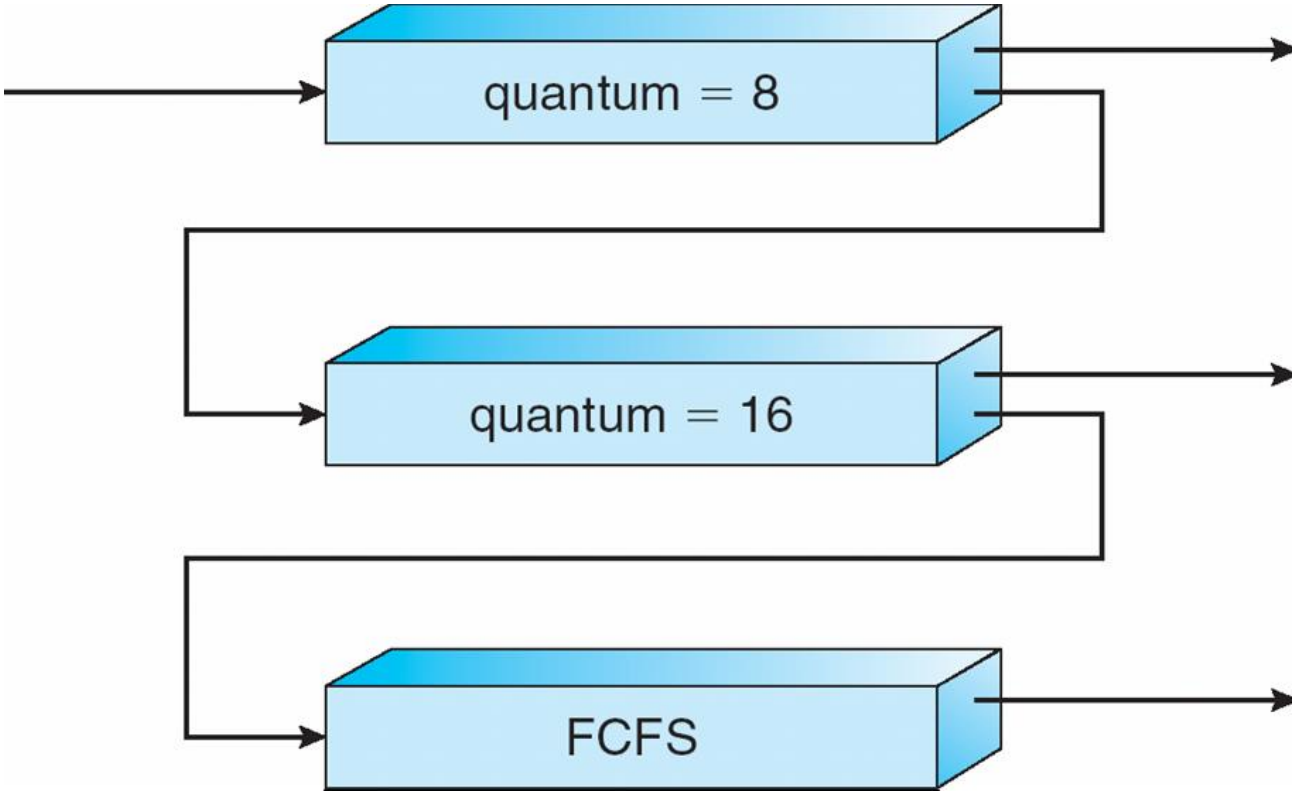
Multilevel Feedback Queue

- Most general CPU scheduling algorithm
- Like multilevel feedback queue with the addition that a process can move between the various queues
  - for example, 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



Thread Scheduling

- Distinction between user-level and kernel-level threads
- In the many-to-one and many-to-many models, the thread library schedules user-level threads to run on 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
- PCS typically uses a priority scheduler
  (priority of user-level thread set by the programmer)

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.

```
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(pthread_attr_t *attr, int *scope);
```

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, 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);
```

Pthread Scheduling API

```
    /* 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);
}
```

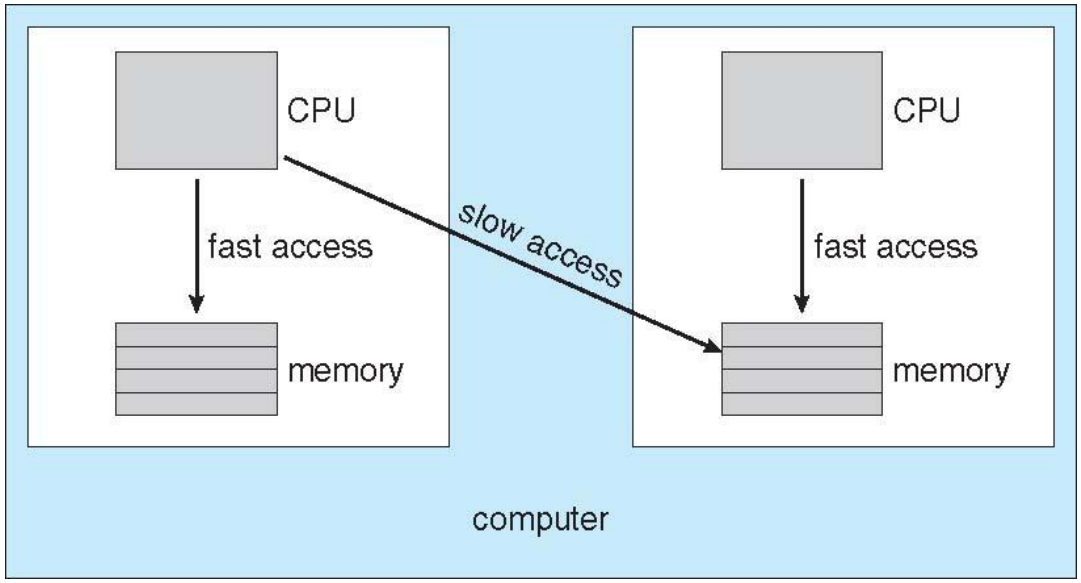
Multiple-Processor Scheduling

- We assume **homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor (the master server) 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

Multiple-Processor Scheduling

- **Processor affinity** – process has affinity for processor on which it is currently running
  - soft affinity
  - hard affinity

- Main memory architecture can affect processor affinity



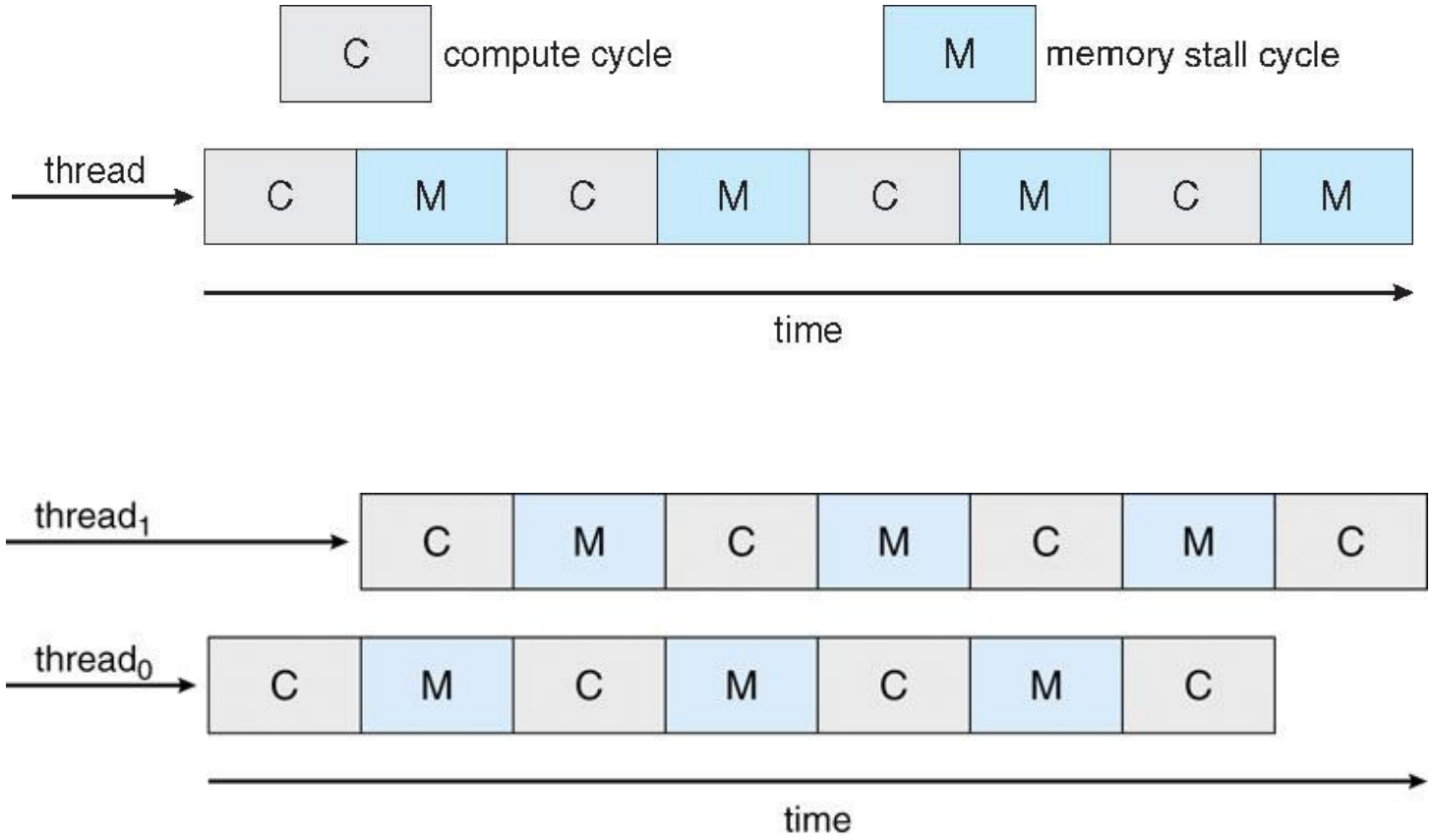
Multiple-Processor Scheduling

- **Load balancing** – balance load in private ready queues
  - push migration
  - pull migration
  - often, a combination of both is implemented

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
  - coarse-grained vs fine grained multithreading
    - coarse-grained: long-latency event triggers switching
    - fine-grained: switch frequently (e.g., at instruction boundaries)

Multithreaded Multicore System

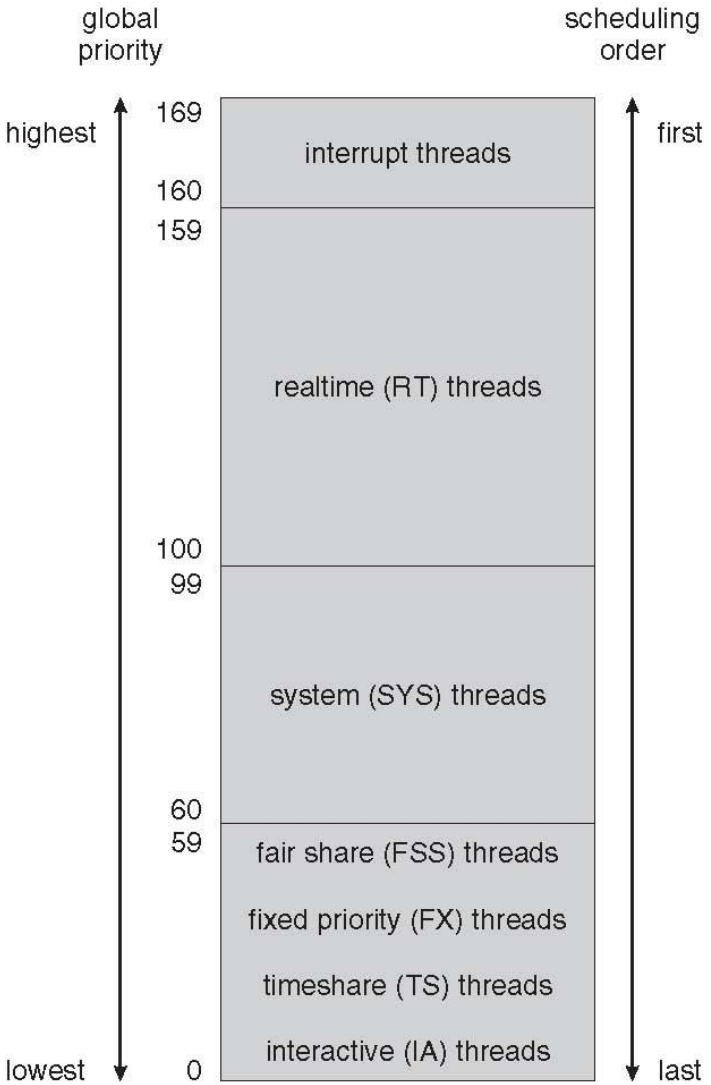


Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Solaris Scheduling

- Priority-based thread scheduling
- Threads are classified into
  - Interrupt threads
  - RT: realtime
  - SYS: system
  - TS: time-share
  - IA: interactive
  - FSS: fair-share
  - FX: fixed priority



Solaris Dispatch Table

- Priorities for threads in the TS/IA class are adjusted dynamically and scheduled using a multilevel feedback queue scheduler.

- Inverse relationship between time quantum and priority:

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

lower priority for CPU-bound threads

boost priority for interactive threads

Windows XP Scheduling

- Priority-based preemptive scheduler

- Two classes: real-time (16~31) and variable (1~15), with relative priorities.
- Priorities for threads in the variable class change dynamically
  - lower when quantum runs out, never below ‘normal’
  - boost when becoming ready; boost depends on I/O event

- Distinction between foreground and background process

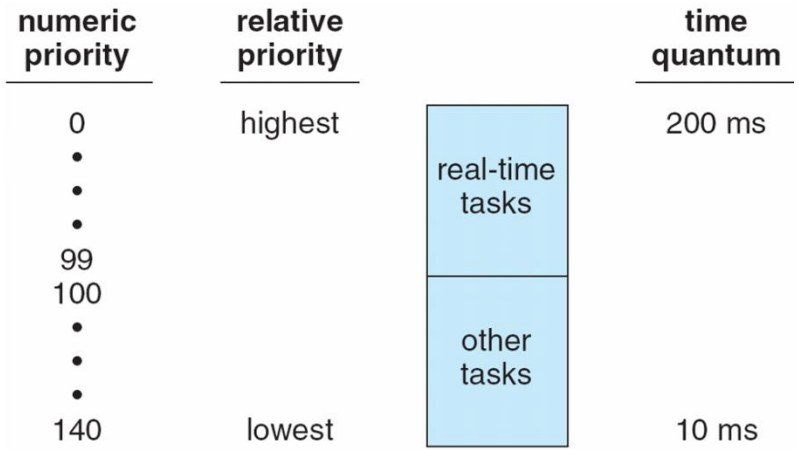
	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Linux Scheduling

- Scheduler depends on the kernel version
  - before 2.5: UNIX variant
  - 2.5 ~ 2.6.23: Constant order O(1) scheduling time
  - >= 2.6.23: Completely Fair Scheduler

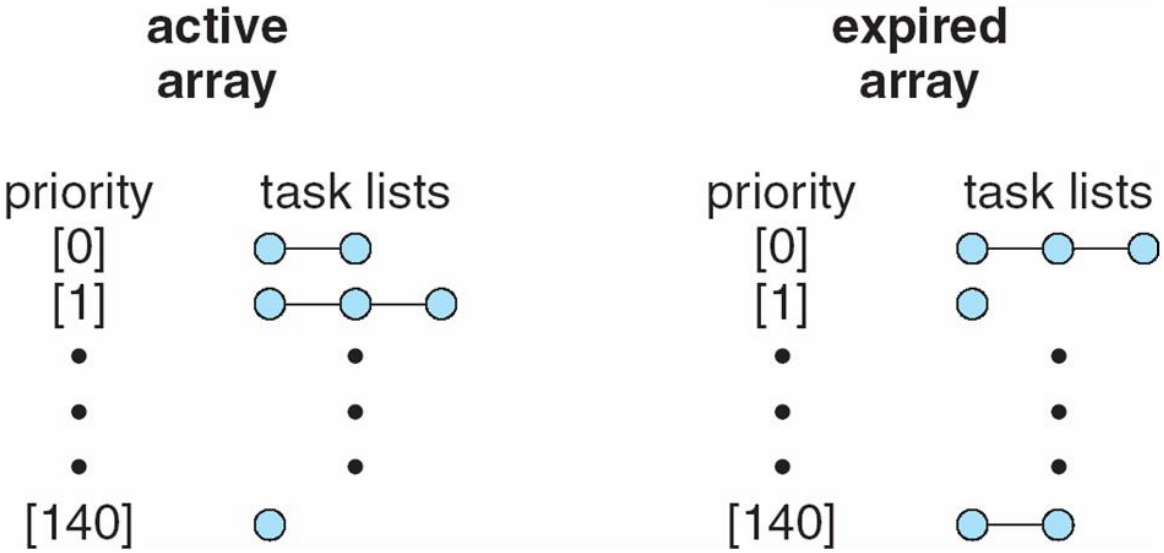
- UNIX scheduler:
  slow for many processes, inadequate SMP-support

- O(1) scheduler:
  preemptive, priority-based
  - Two priority ranges: time-sharing and real-time
  - Real-time range from 0 to 99 and nice value from 100 to 140



Linux Scheduling

- per-core runqueue:
  - Active array: tasks that have not used up all CPU time in their quantum
  - Expired array: tasks that have used up the whole quantum
- Only tasks in the active array are scheduled and moved to the expired array once they completely exhaust their quantum.
- When the active array is empty, switch.



Algorithm Evaluation

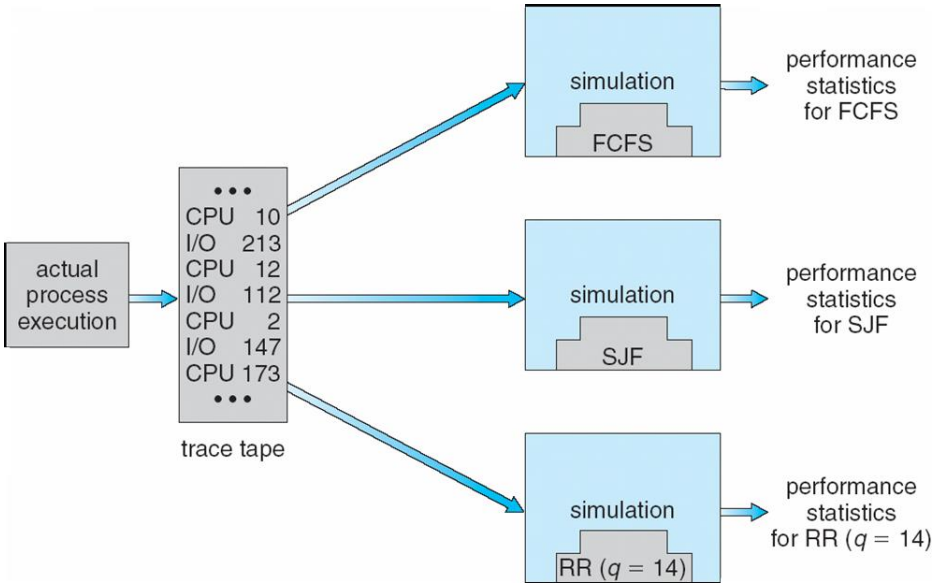
- Which algorithm is best for a system?
  → select measures, define measurement criteria

- Deterministic modeling
  Take a particular predetermined workload and compute the performance of each algorithm for that workload
  - task queue must be available beforehand

- Queuing models
  From a running system, build a distribution of CPU and I/O bursts (typically exponential distribution). Mathematical analysis is used to compare algorithms
  - often, the models are not realistic enough

Algorithm Evaluation

- Simulation
  Use randomly generated task lists (or trace tapes) to simulate the scheduling performance on a programmed model of the computer system.



- Implementation
  Implement the new algorithm in a real system