Chapter 5: CPU Scheduling

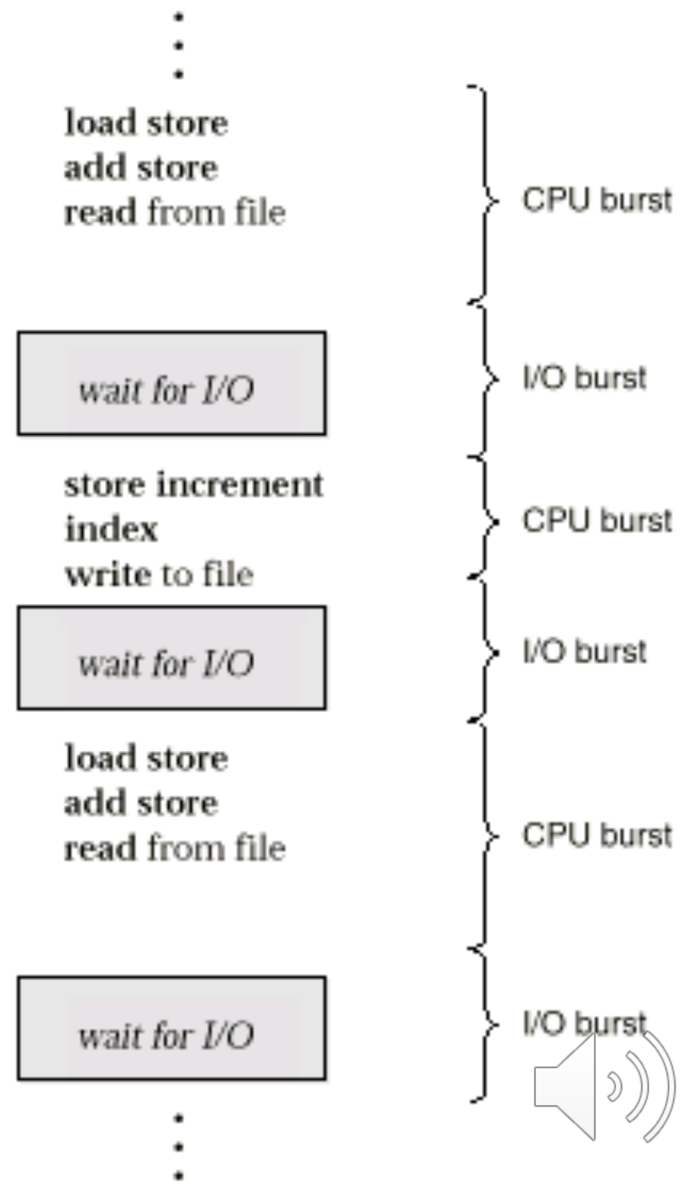
肖 卿 俊
办公室：江宁区无线谷6号楼226办公室
电邮：csqjxiao@seu.edu.cn
主页： https://csqjxiao.github.io/PersonalPage
电话：025-52091022

Chapter 5: CPU Scheduling

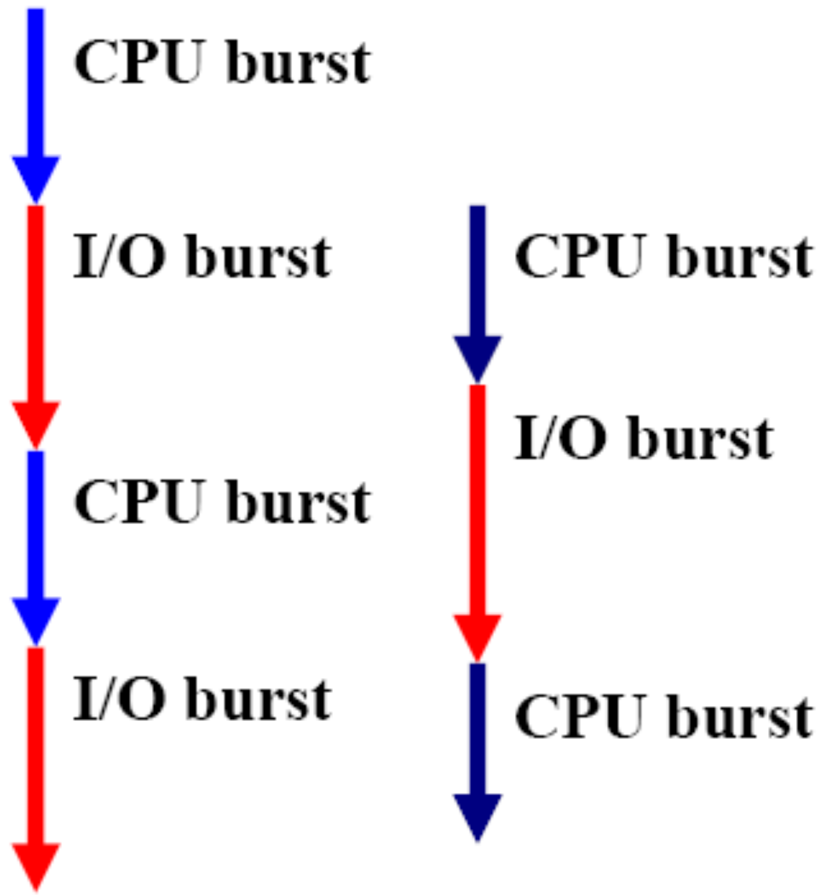
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- A Fact: Process execution consists of an alternating sequence of CPU execution and I/O wait, called *CPU–I/O Burst Cycle*



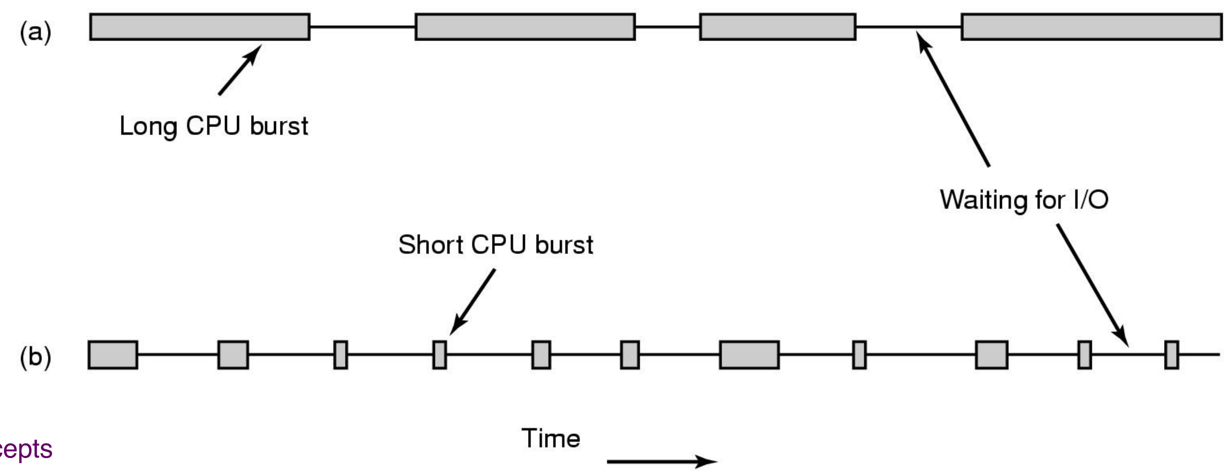
CPU-I/O Burst Cycle



- Process execution repeats the CPU burst and I/O burst cycle.
- When a process begins an I/O burst, another process can use the CPU for a CPU burst.

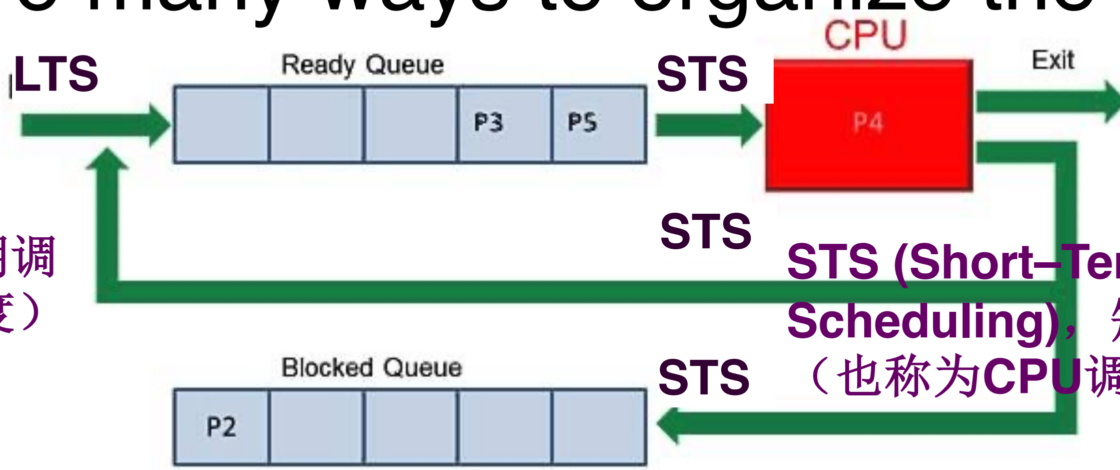
CPU-bound and I/O-bound

- A process is CPU-bound if it generates I/O requests infrequently, using more of its time doing computation.
- A process is I/O-bound if it spends more of its time to do I/O than it spends doing computation
- A CPU-bound process might have a few very long CPU bursts, while an I/O-bound process typically has many short CPU bursts



CPU Scheduler

- When the CPU is idle, the OS must select another process to run.
- This selection process is carried out by the *short-term scheduler* (or *CPU scheduler*).
- The CPU scheduler selects a process from the ready queue and allocates the CPU to it.
- The ready queue does not have to be a FIFO one. There are many ways to organize the ready queue.

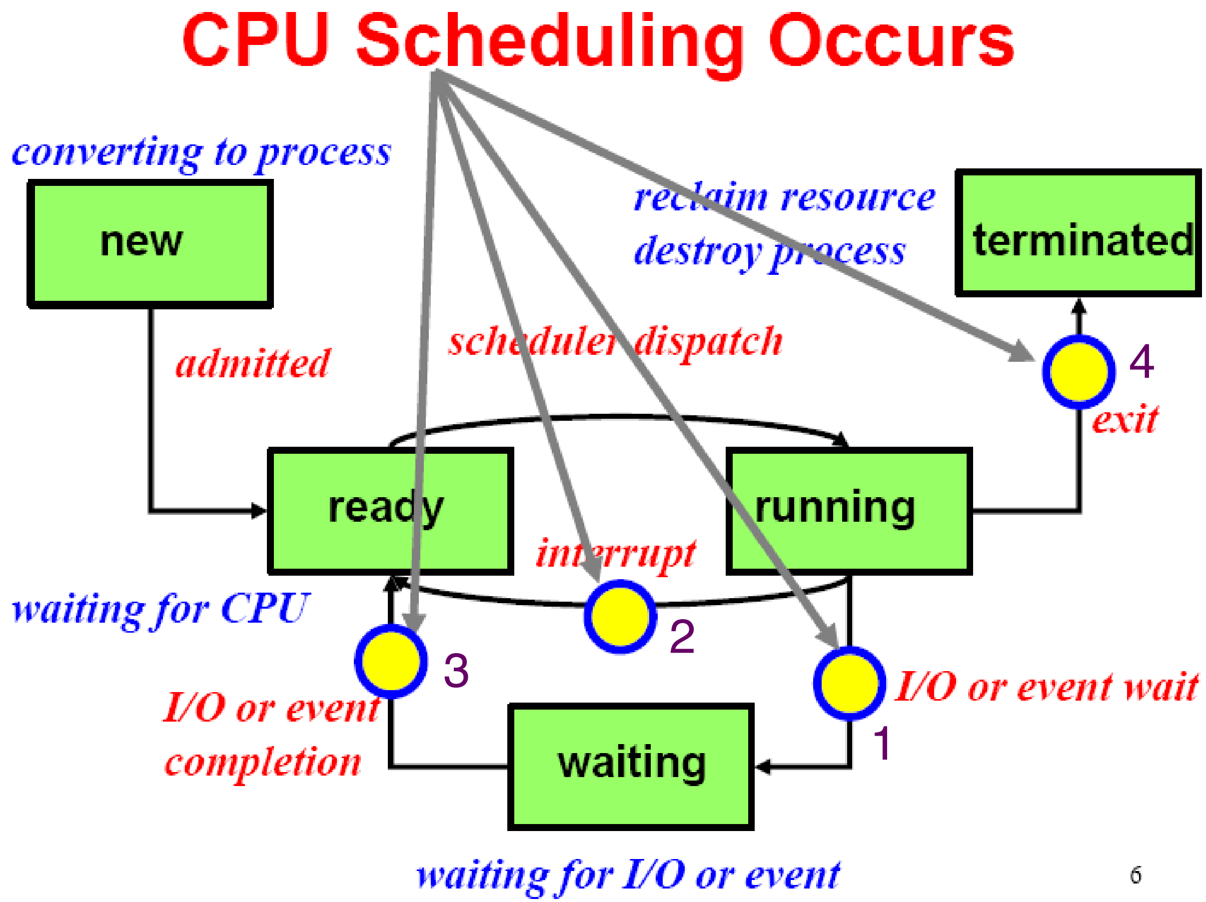


LTS (Long-Term Scheduling)，长期调度（也称为Job调度）

STS (Short–Term Scheduling)，短期调度（也称为CPU调度）

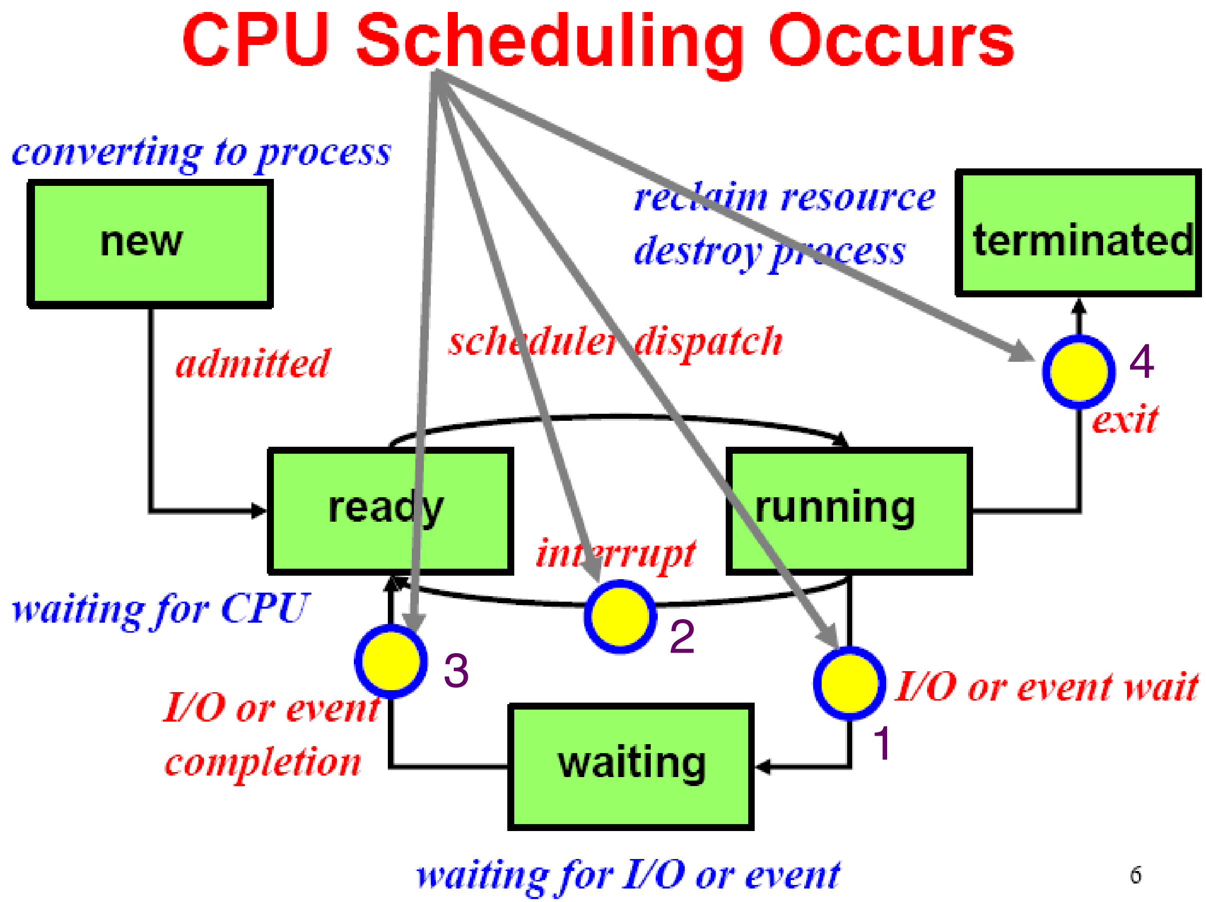
Circumstances that scheduling may take place

1. A process switches from the running state to the waiting state (*e.g.*, doing for I/O)



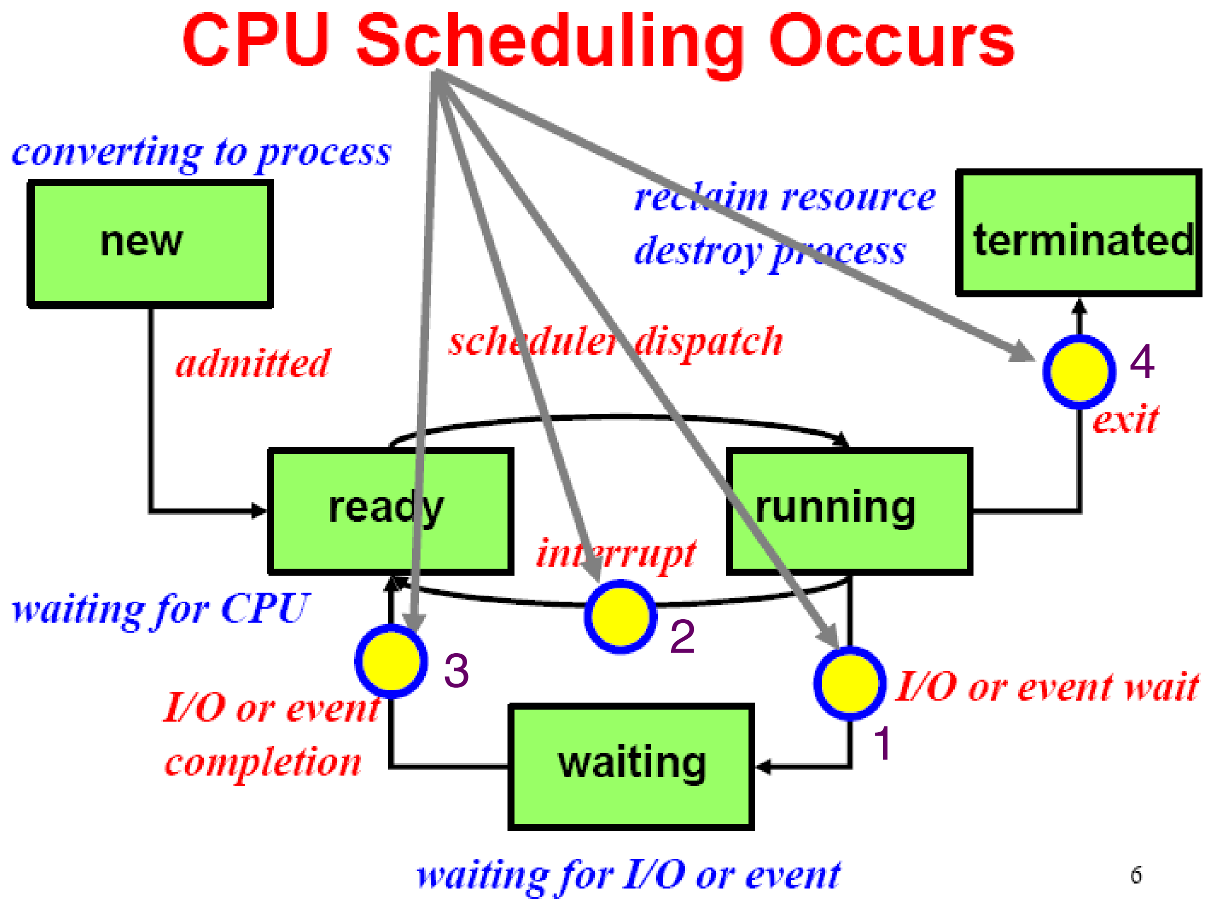
Circumstances that scheduling may take place

2. A process switches from the running state to the ready state (*e.g.*, an interrupt occurs)



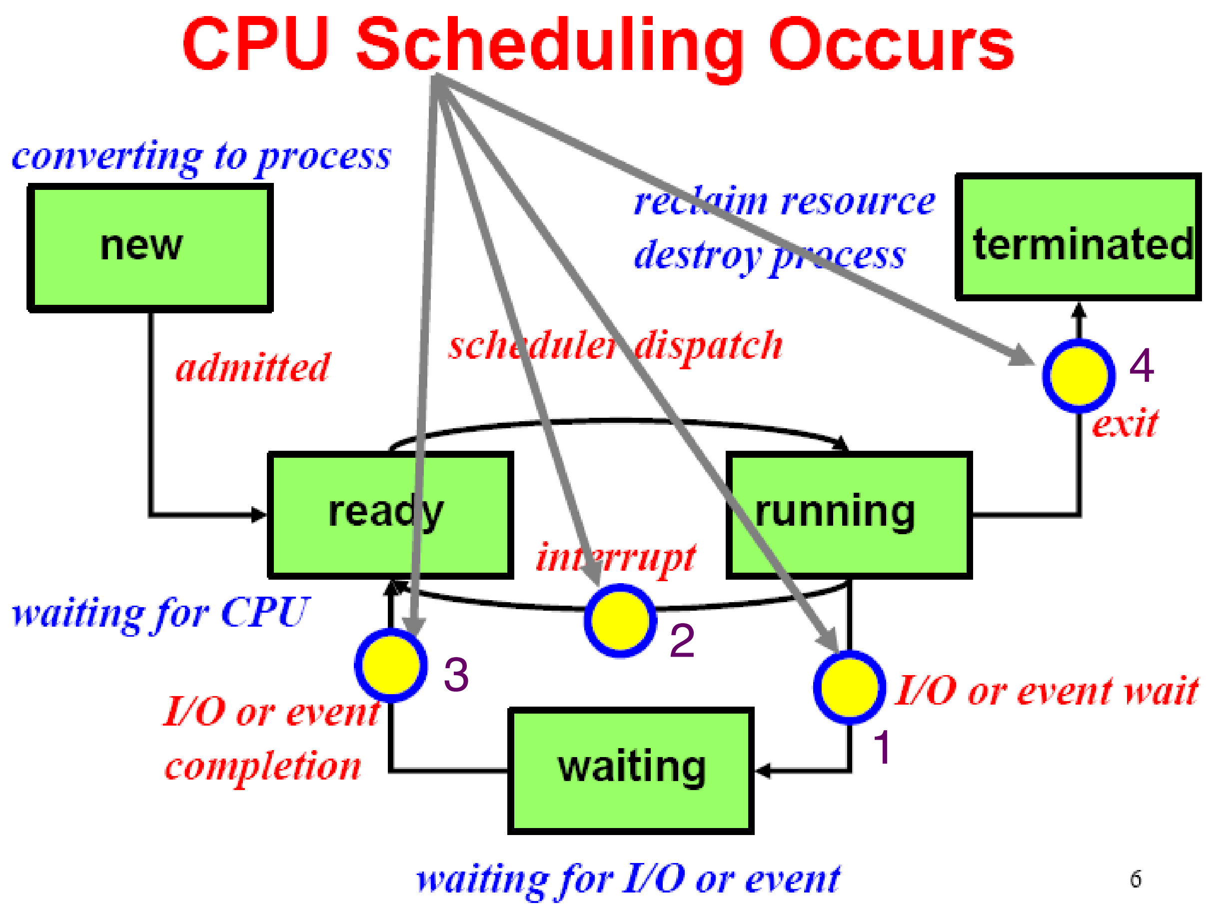
Circumstances that scheduling may take place

3. A process switches from the waiting state to the ready state (*e.g.*, I/O completion)



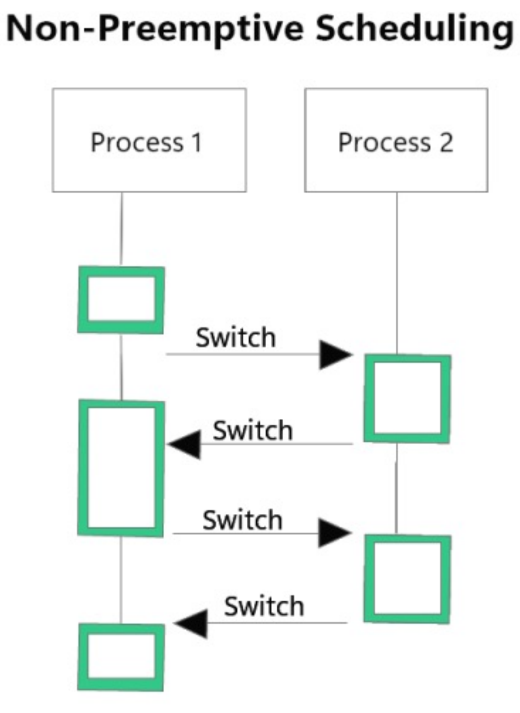
Circumstances that scheduling may take place

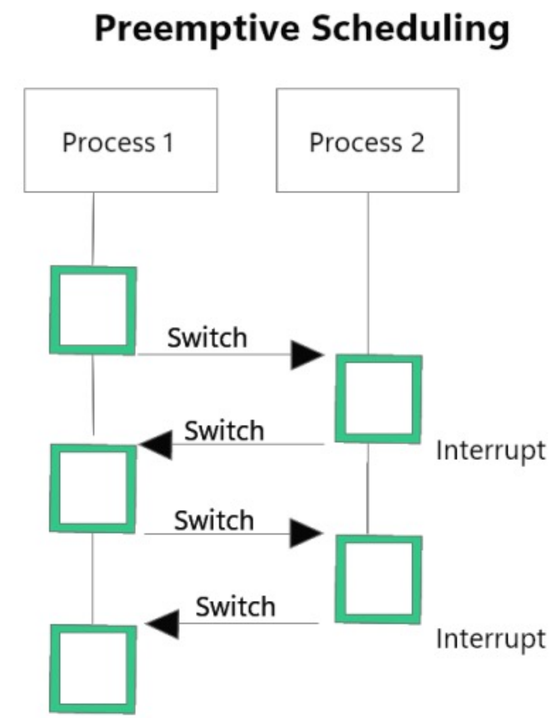
4. A process terminates

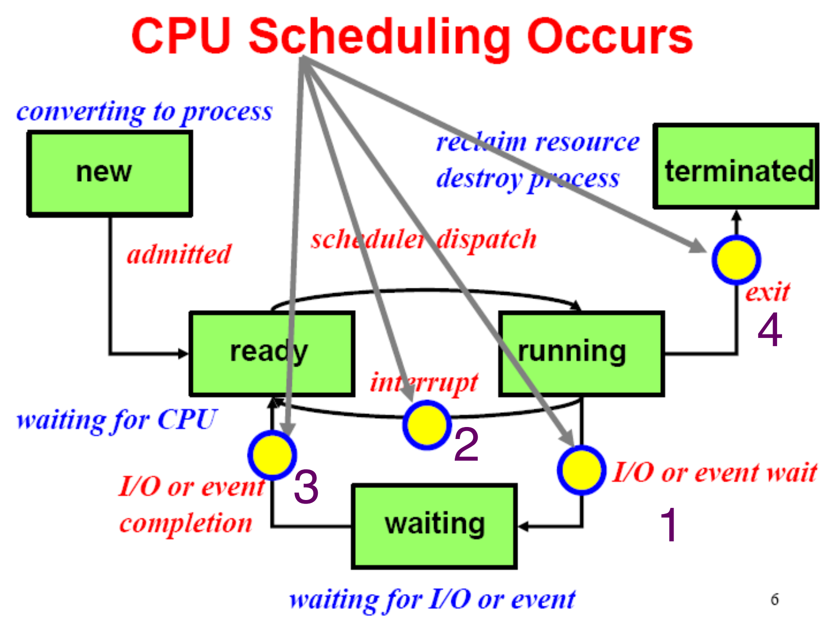


Non-preemptive vs. Preemptive

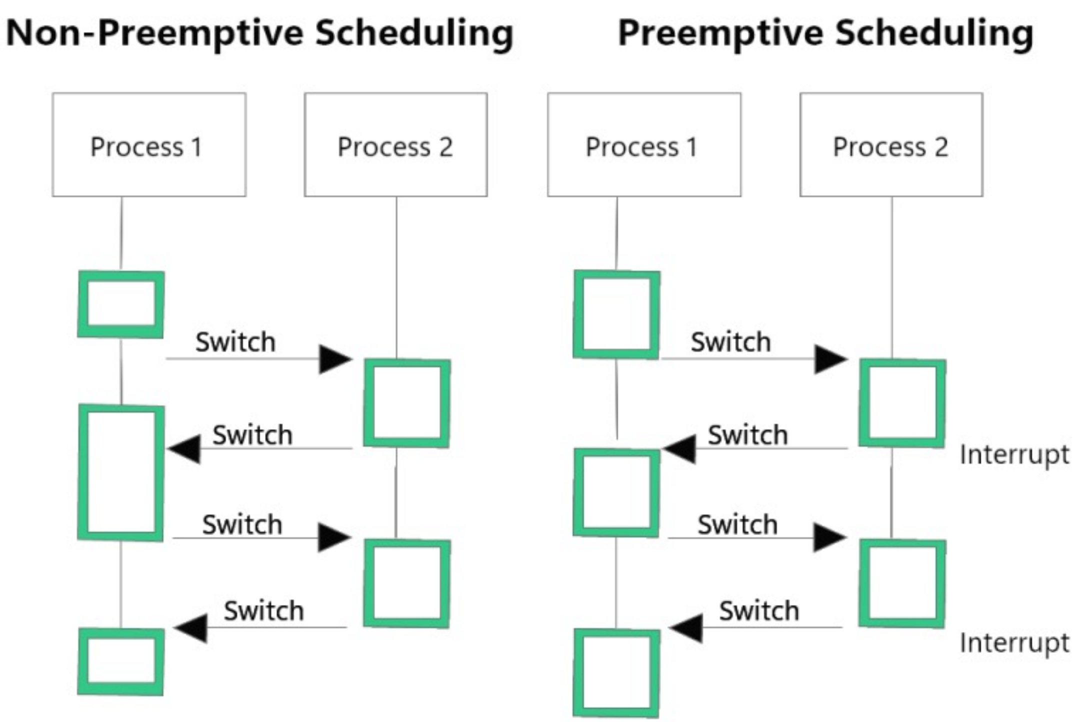
- **Non-preemptive scheduling**: scheduling occurs when a process voluntarily leave the CPU resource. It either enters the waiting state (case 1) or terminates (case 4).
  - Simple, but very efficient with less context switch
  - 对应以前提过的多道系统（multi-programed OS）

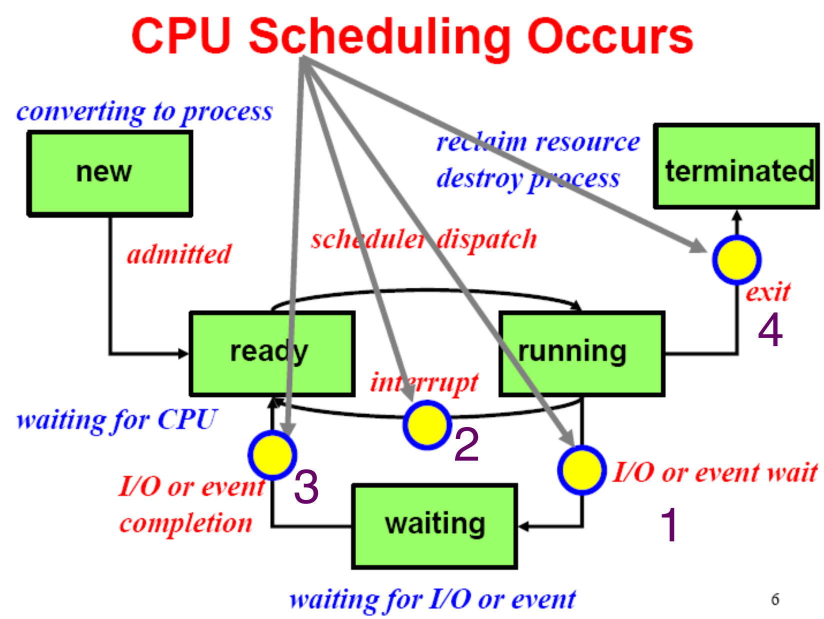






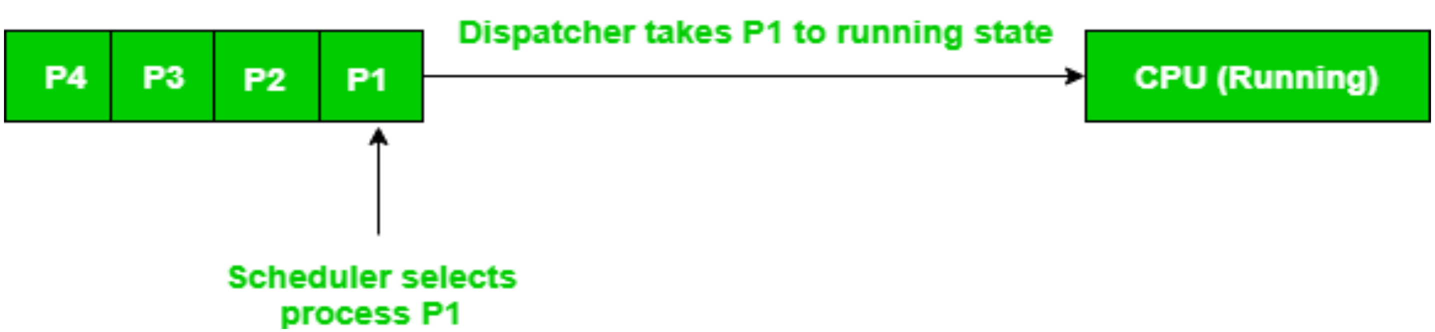
- **Preemptive scheduling (抢占式调度)**: scheduling occurs in all possible cases.
  - What if the running process is in critical section modifying some shared data? There is a possibility of race condition. Mutual exclusion of accessing critical section may be violated.
  - The kernel must pay special attention to this situation and, hence, is more complex





Dispatcher

- Dispatcher module (分配器) gives control of the CPU to the process selected by the short-term scheduler (调度器); this involves:
  - switching execution context (save & reload)
  - switching to user mode
  - jumping to the proper location in the user program to restart that program



- *Dispatch latency* – time it takes for dispatcher to stop one process and start another running.

Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling

Separation of Policy and Mechanism

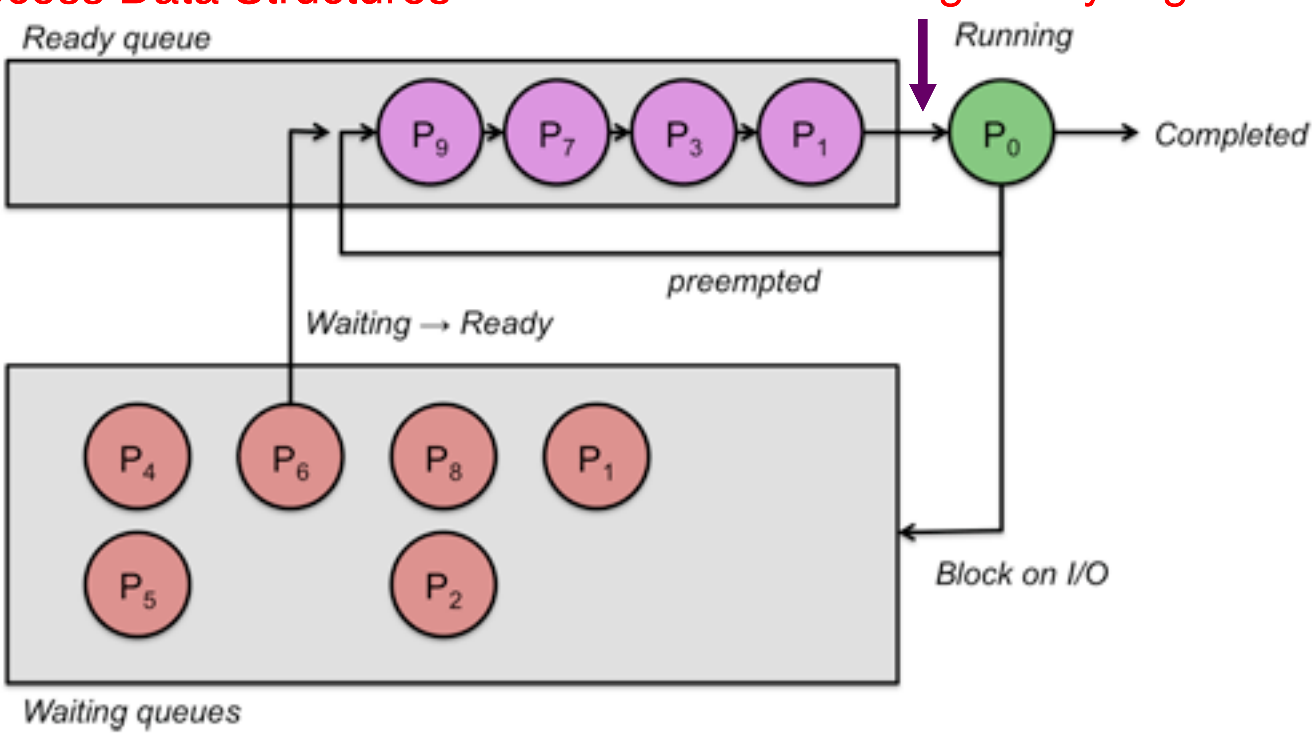
- "Why and What" vs. "How"
- Objectives and strategies vs. data structures
  - hardware and software implementation issues.
- Process abstraction vs. Process machinery

Process Control Block

Process Control Block
Pointer
Process State
Process Number
Process Counter
Registers
Memory Limit
List of Open Files
. . .

Process Data Structures

CPU Scheduling Policy/Algorithm

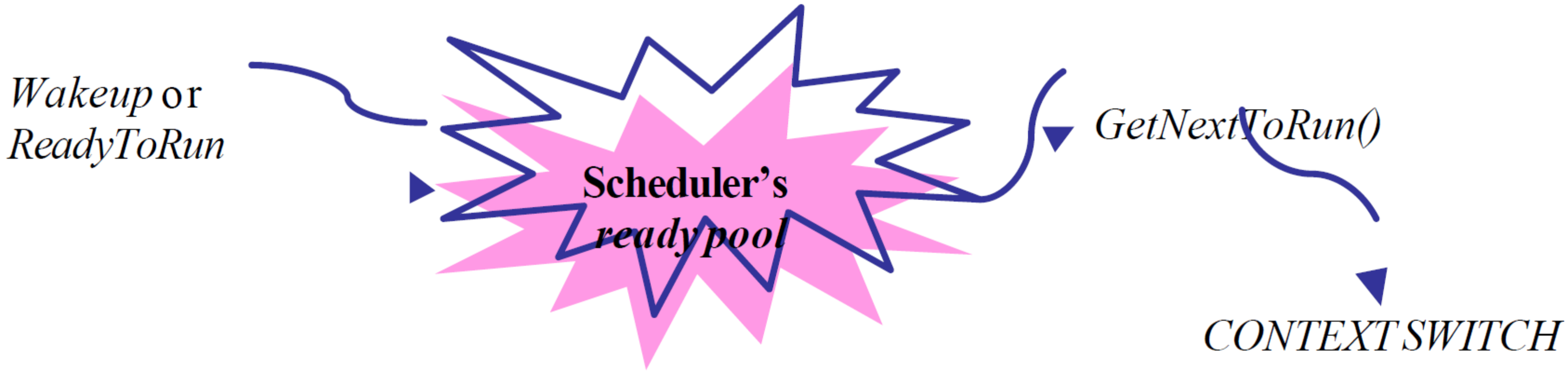


Scheduling: Policy and Mechanism

- Scheduling policy answers the question:
  - Which process/thread, among all those ready to run, should be given the chance to run next? In what order do the processes/threads get to run? For how long?
- Mechanisms are the tools for supporting the process/thread abstractions and affect how the scheduling policy can be implemented (this is review)
  - How the process or thread is represented to the system - process or thread control blocks.
  - What happens on a context switch.
  - When do we get the chance to make these scheduling decisions (timer interrupts, thread operations that yield or block, user program system calls)

CPU Scheduling Policy

- The scheduler's moves are dictated by a *scheduling policy*



- The CPU scheduler makes a sequence of "moves" that determines the interleaving of processes
  - Programs use process synchronization to prevent "bad moves"
  - …but otherwise scheduling choices appear (to the program) to be nondeterministic.

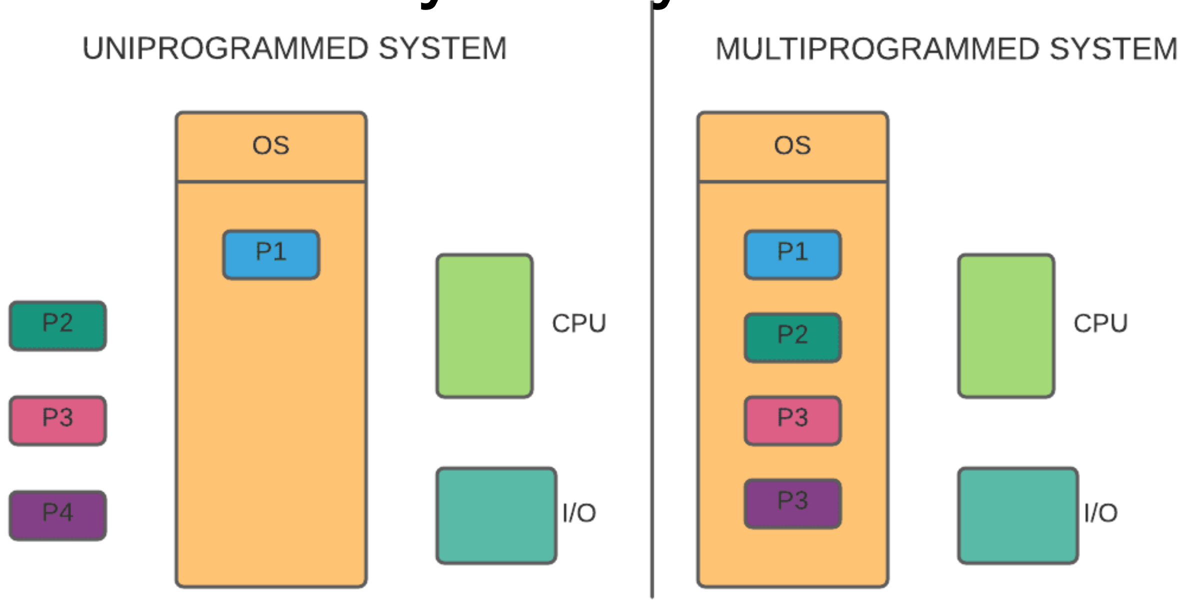
Scheduling Criteria

- Before presenting detailed scheduling policies, we discuss how to evaluate the “goodness” of a scheduling policy.

- There are many criteria for comparing different scheduling policies. Here are five common ones
  - CPU utilization (CPU利用率)
  - Throughput (执行任务的吞吐量)
  - Turnaround time (进程的周转时间)
  - Waiting time (进程的等待时间)
  - Response time (对用户的响应时间)

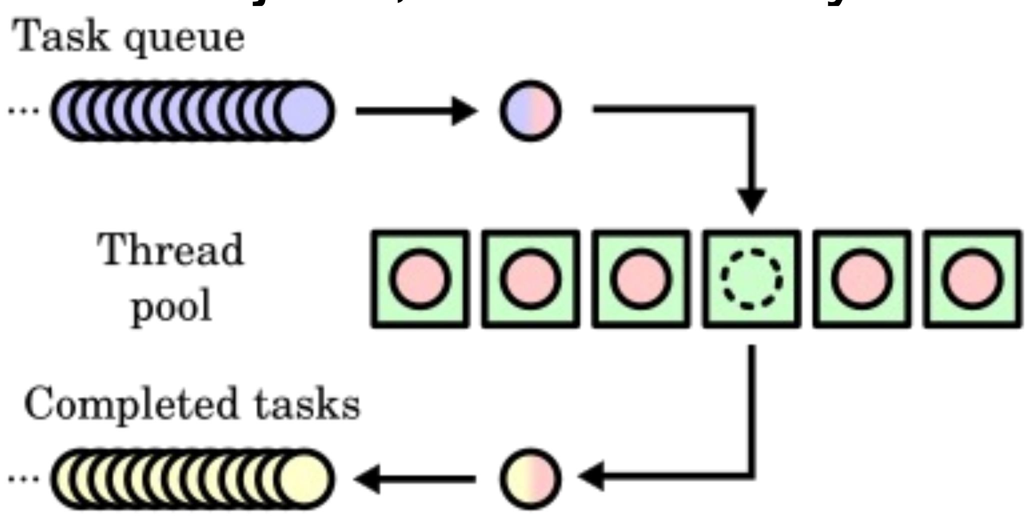
CPU Utilization

- We want to keep the CPU as busy as possible.
- CPU utilization ranges from 0 to 100 percent. Normally 40% is lightly loaded and 90% or higher is heavily loaded.
- You can bring up a CPU usage meter to see CPU utilization on your system.



Throughput

- The number of processes completed per time unit is called *throughput*.
- Higher throughput means more jobs get done.
- However, this criteria is affected by the characteristics of processes.
  - For long processes, this rate may be one job per hour, and, for short jobs, this rate may be 10 per minute.



Turnaround Time

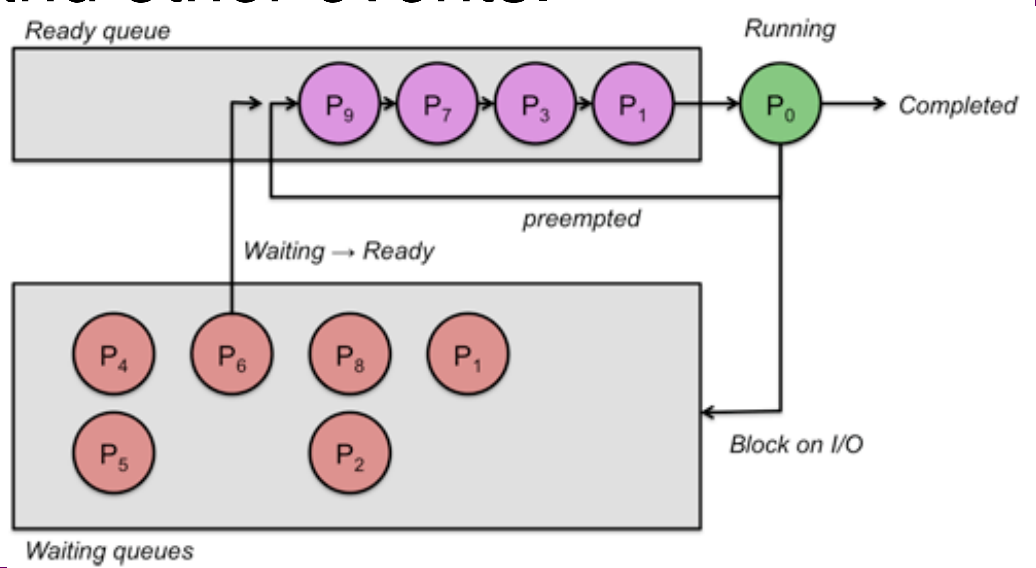
- The time period between job submission to completion is the turnaround time.
- From a user's point of view, turnaround time is more important than CPU utilization and throughput.
- Turnaround time is the sum of
  - Waiting time before entering the system
  - Waiting time in the ready queue
  - Waiting time in all other events (e.g., I/O)
  - Burst time, i.e., the process actually running on the CPU

Turn Around Time (TAT) = CT - AT
OR
Turn Around Time (TAT) = WT + BT

So,
CT - AT = WT + BT
Waiting Time (WT) = TAT - BT

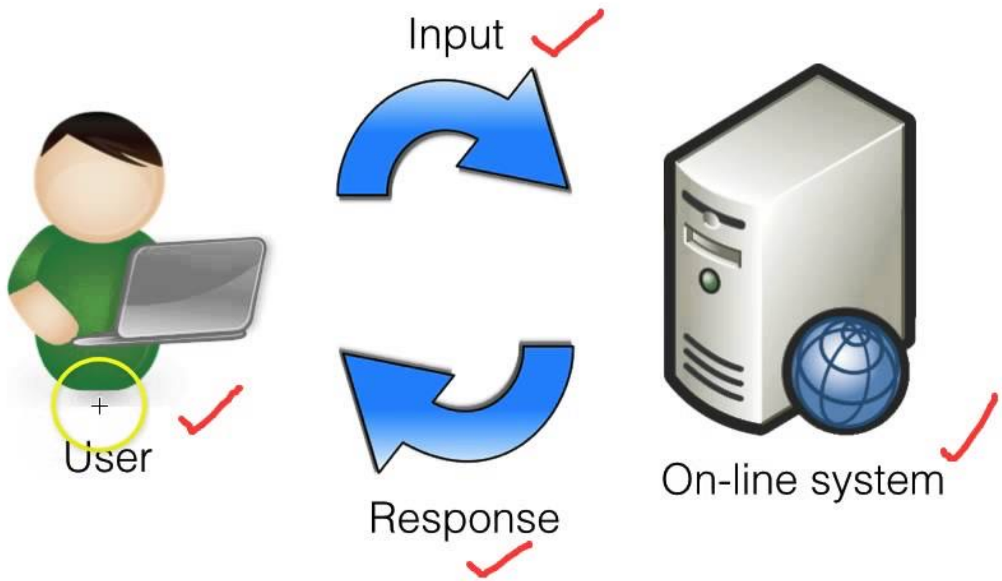
CPU Waiting Time

- CPU waiting time (or waiting time for short) is the sum of the periods that a process spends waiting in the ready queue.
- Why only ready queue?
  - CPU scheduling algorithms do not affect the amount of the waiting time during which a process waits for I/O and other events.
  - However, CPU scheduling algorithms do affect the time that a process stays in the ready queue



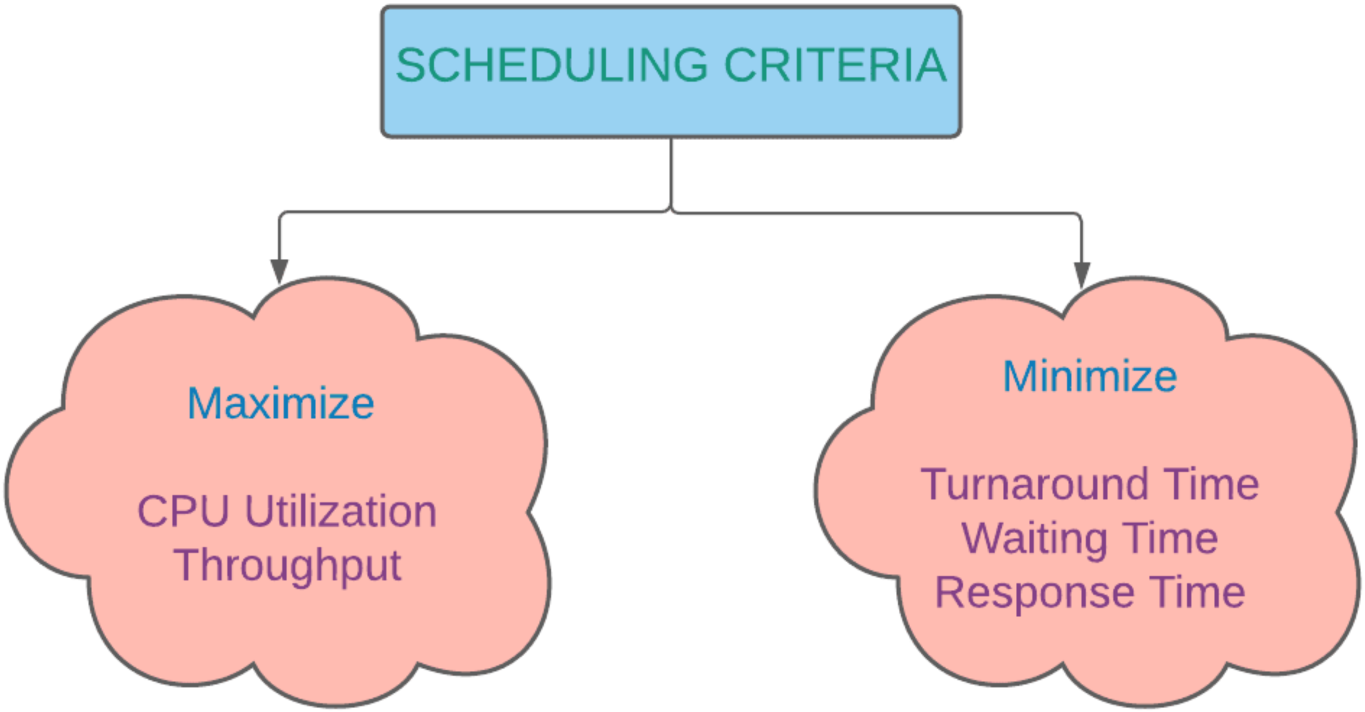
Response Time

- The time from the submission of a request (in an interactive system) to the first response is called response time. It does not include the time that it takes to output the response.
  - For example, in front of your workstation, you perhaps care more about the time between hitting the Return key and getting your first output (e.g., response time) than the time from hittingthe Return key to the completion of your program (e.g., turnaround time).



CPU Scheduling Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time



Chapter 6: CPU Scheduling

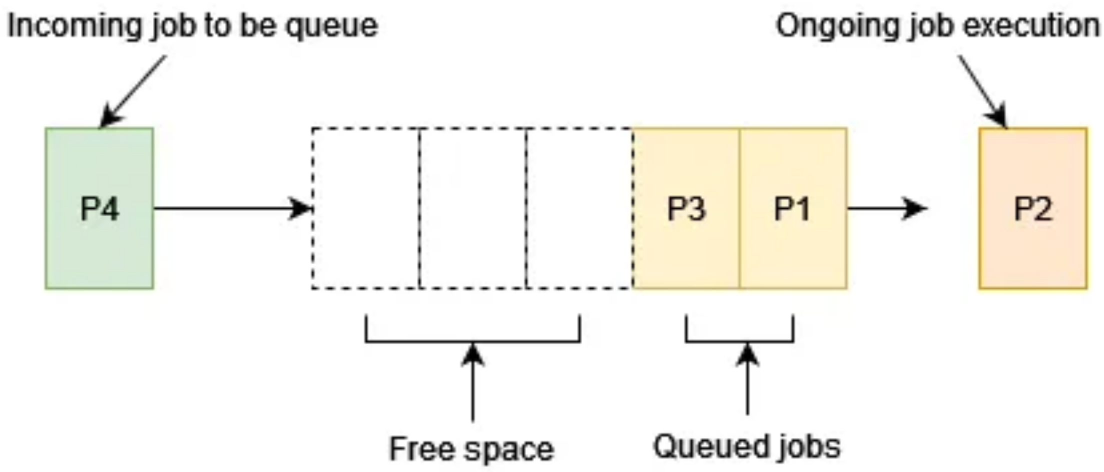
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling

Scheduling Algorithms

- We will discuss a number of scheduling algorithms (or scheduling policies):
  - First-Come, First-Served (FCFS)
  - Round-Robin
  - Lottery Scheduling (with demonstration code)
  - Shortest-Job-First (SJF)
  - Priority
  - Multilevel Queue
  - Multilevel Feedback Queue

First-Come, First-Served (FCFS) Scheduling

- The process that requests the CPU first is allocated the CPU first.
- This can easily be implemented using a FIFO (First In First Out) queue.
- FCFS is not preemptive. Once a process has the CPU, it will occupy the CPU until the process completes or voluntarily enters the wait state.



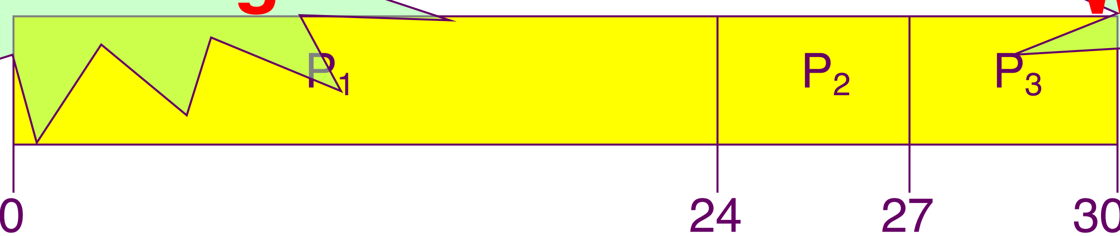
FCFS Scheduling (Cont.)

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:

Average waiting time?

Waiting time?



- Waiting time for P1 = 0; P2 = 24; P3 = 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

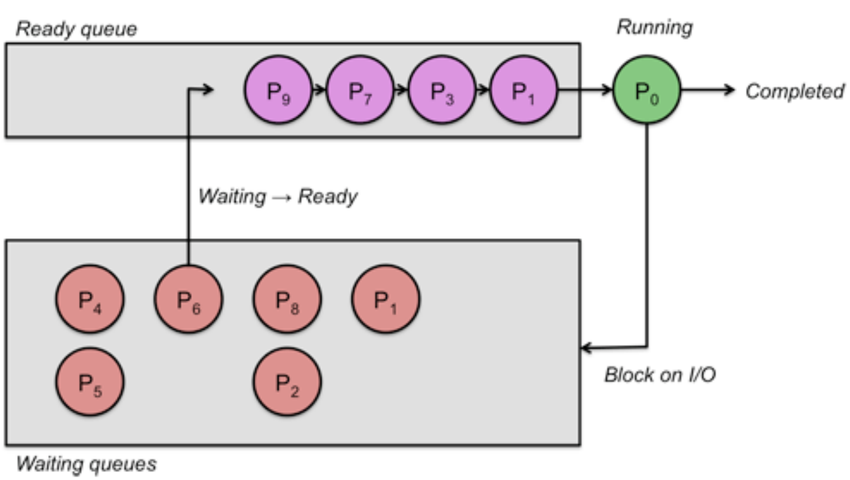
FCFS Problems

- It is easy to have the *convoy effect*: all the processes wait for the one big process to get off the CPU.
- Consider a CPU-bound process running with many I/O-bound process.
- It is in favor of long processes and may not be fair to those short ones. What if your 1-minute job is behind a 10-hour job?
- It is troublesome for time-sharing systems, where each user needs to get a share of the CPU at regular intervals.

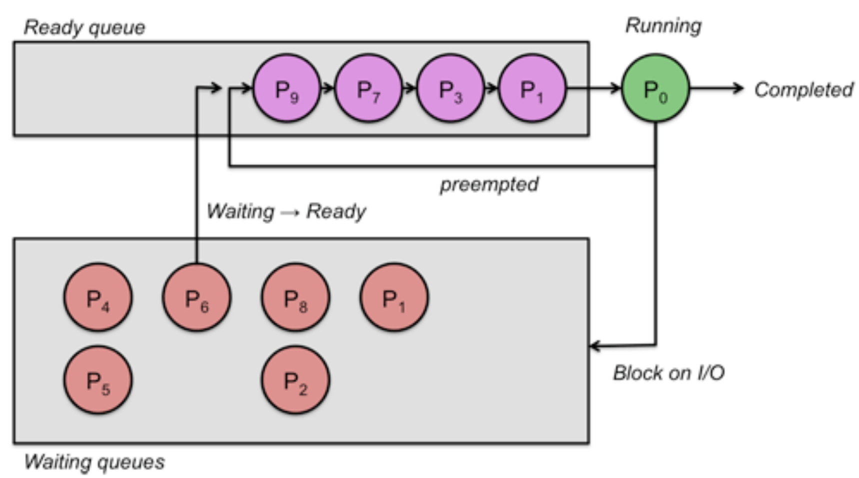
Round Robin (RR) (1)

- RR is similar to FCFS, except that each process is assigned a time quantum.
- All processes in the ready queue is a FIFO list.
- When CPU is free, the scheduler picks the first and lets it run for one time quantum (or slice)

First-Come, First-Served Scheduling



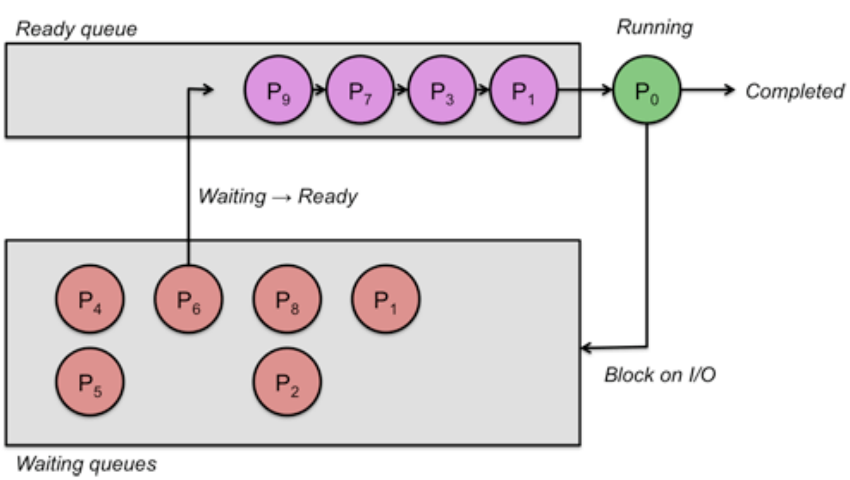
Round Robin Scheduling



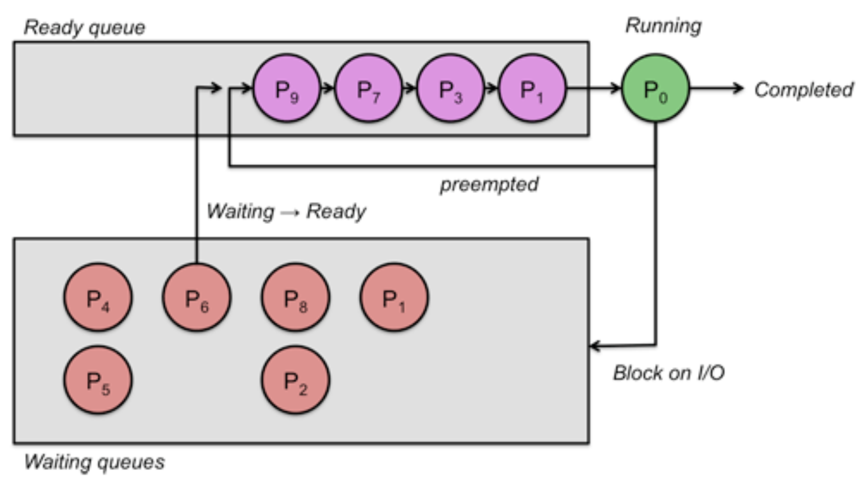
Round Robin (RR) (2)

- If that process uses CPU for less than one time quantum, it is moved to the of the waiting list.
- Otherwise, when one time quantum is up, that process is preempted by the scheduler and moved to the tail of the ready queue, a FIFO list

First-Come, First-Served Scheduling



Round Robin Scheduling



Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

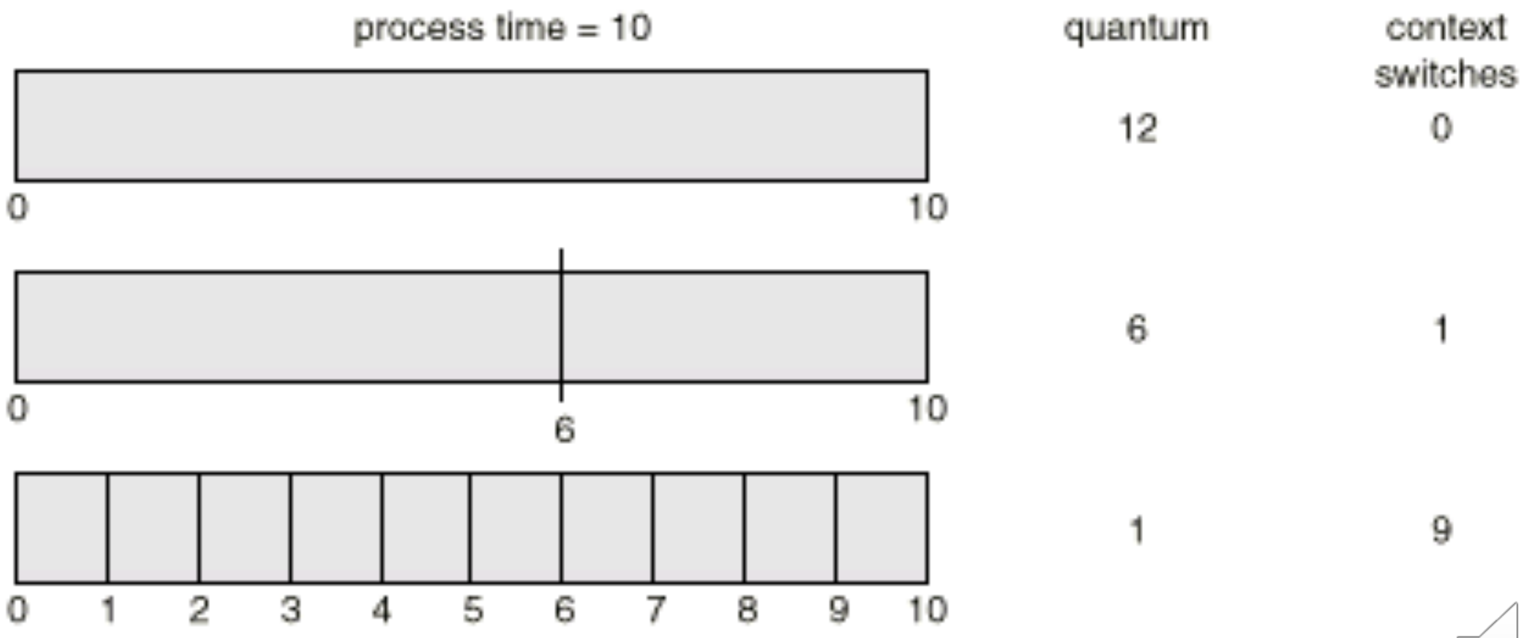
0 20 37 57 77 97 117 121 134 154 162

RR Scheduling: Some Issues

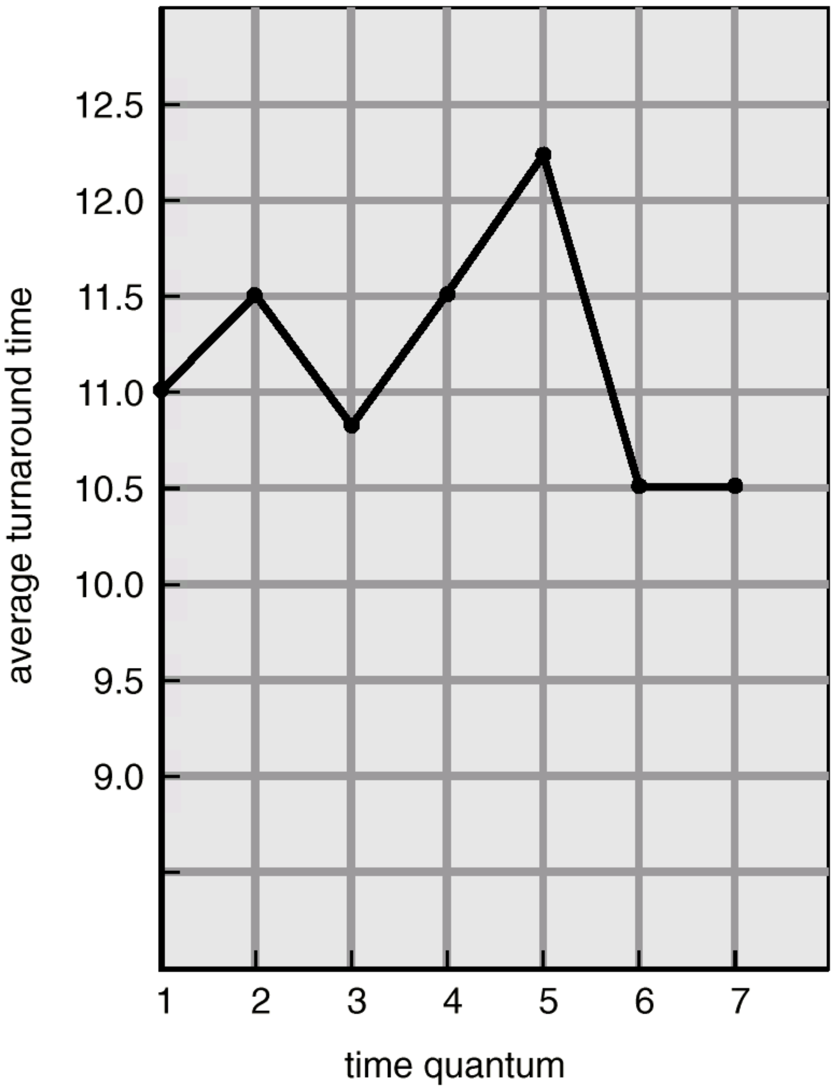
- If time quantum is too large (i.e., larger than all the CPU bursts), RR reduces to FCFS
- If time quantum is too small (smaller than all the CPU bursts), RR becomes processor sharing
- Context switching may affect RR's performance
  - Shorter time quantum means more context switches
- Turnaround time also depends on the size of time quantum.
- In general, 80% of the CPU bursts should be shorter than the time quantum

Time Quantum and Context Switch Time

- Context switching may affect RR's performance
  - Shorter time quantum means more context switches



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

- When time quantum = 1,
  Turnaround of P_1 = 15
  Turnaround of P_2 = 9
  Turnaround of P_3 = 3
  Turnaround of P_4 = 17
  Average Turnaround = 11

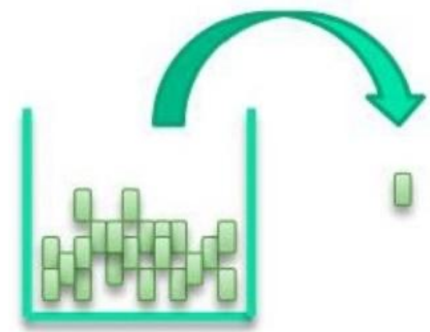
- If using SJF (P3, P2, P1, P4)
  Turnaround of P_1 = 10
  Turnaround of P_2 = 4
  Turnaround of P_3 = 1
  Turnaround of P_4 = 17
  Average Turnaround = 8

Lottery Scheduling

- RR gives a roughly equal share of CPU to all ready processes
- Lottery scheduler is a proportional-share scheduler (fair-share scheduler)

Issue 100 Lottery Tickets

P_0 30%	P_2 15%	P_3 25%	P_4 30%
T=30	T=15	T=25	T=30

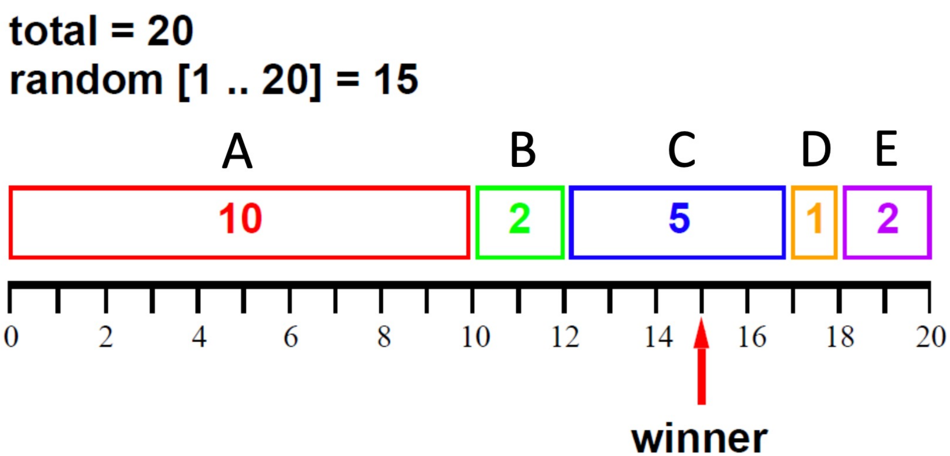


Ticket holder gets CPU until next drawing

- Instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time

Lottery Scheduling (cont.)

- Basic idea
  - Every so often, hold a lottery to determine which process should get to run next;
  - Processes that should run more often should be given more chances to win the lottery.
- Tickets
  - are used to represent the share of a resource that a process (or user or whatever) should receive.



List-based lottery (winner = 15)

A Simple Unfairness Metric

- Suppose:
  - Two jobs competing against one another, each with the same number of tickets and the same run time.
- An unfairness metric U:
  - The time the first job completes divided by the time that the second job completes.
  - With a perfect fair scheduler, two jobs should finish at roughly the same time, i.e., U=1.

A Simple Unfairness Metric

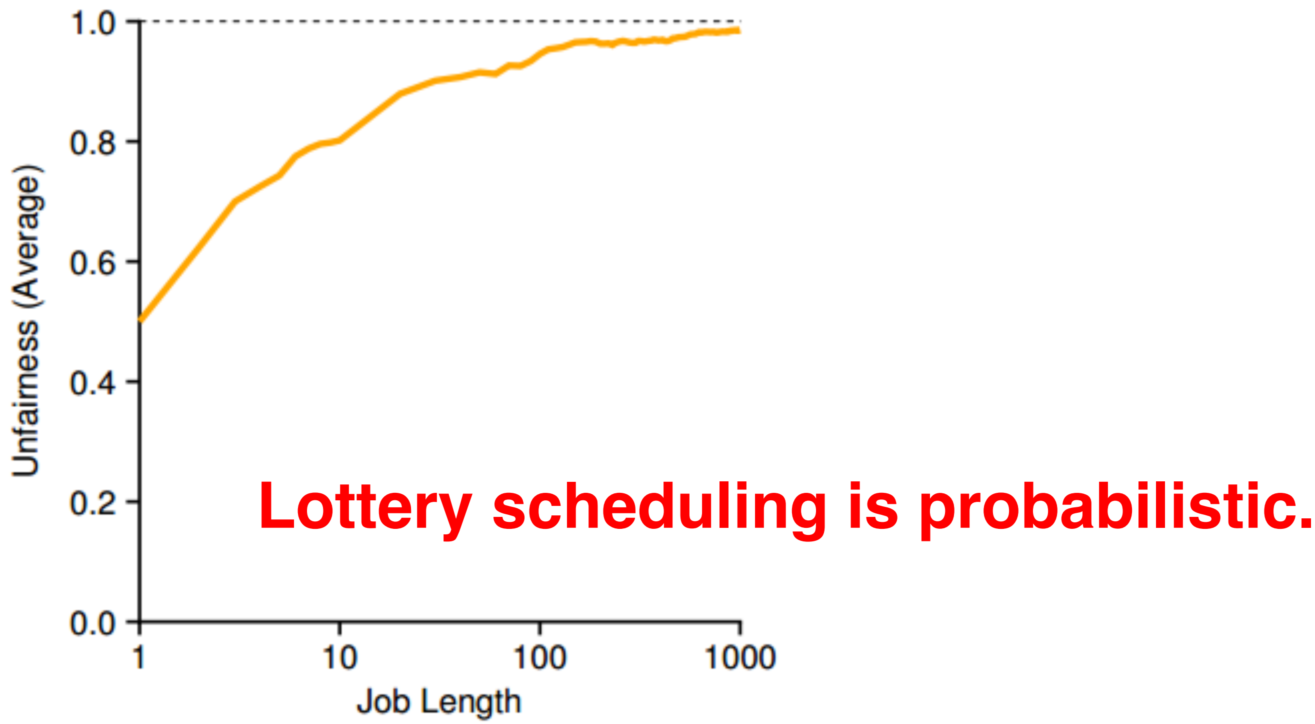


Figure 9.2: **Lottery Fairness Study**

Only as the jobs run for a significant number of time slices does the lottery scheduler approach the desired outcome.

Lottery Scheduling: Summary

- Lottery scheduling has not achieved wide-spread adoption as CPU schedulers.
  - Ticket assignment is a hard problem.

- However, it is useful in domains where this problem is relatively easy to solve.
  - VMWare: You might like to assign one-quarter of your CPU cycles to the Windows VM and the rest to your base Linux installation

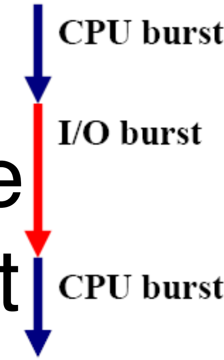
Lottery Demonstration Code

```
int gtickets = 0; // global ticket count
struct node_t {
    int          tickets;
    struct node_t *next;
};
struct node_t *head = NULL;
void insert(int tickets) {
    struct node_t *tmp =
            malloc(sizeof(struct node_t));
    assert(tmp != NULL);
    tmp->tickets = tickets;
    tmp->next    = head;
    head         = tmp;
    gtickets    += tickets;
}
```

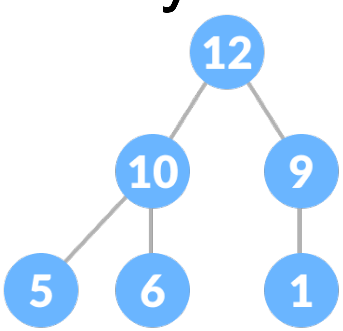
```
int main(int argc, char *argv[]) {
     ……
     // populate list with some number of jobs
    insert(50);   insert(100);   insert(25);
    for (int i = 0; i < loops; i++) {
            int counter = 0;
            int winner  = random() % gtickets;
            struct node_t *current = head;
            while (current) {
               counter = counter + current->tickets;
               if (counter > winner)  break;
               current = current->next;
            }
            printf("winner: %d %d\n\n", winner,
current->tickets);
    }
```

Shortest-Job-First (SJF) Scheduling

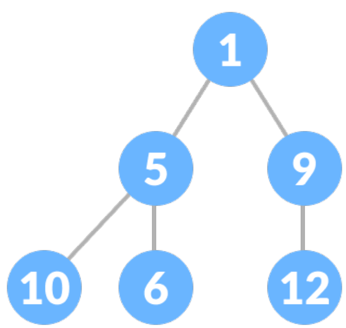
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.



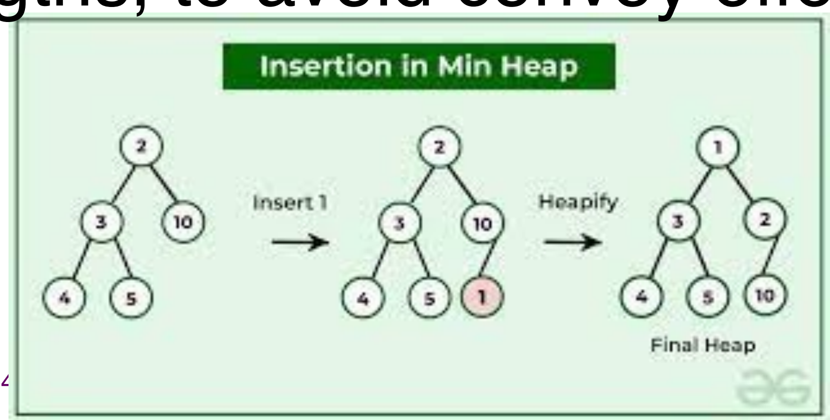
- When a process must be selected from the ready queue, the process with the smallest next CPU burst is selected.
  - Thus, organize the ready queue as a min heap, so that the processes in the ready queue are sorted by their CPU burst lengths, to avoid convoy effect.



Max Heap

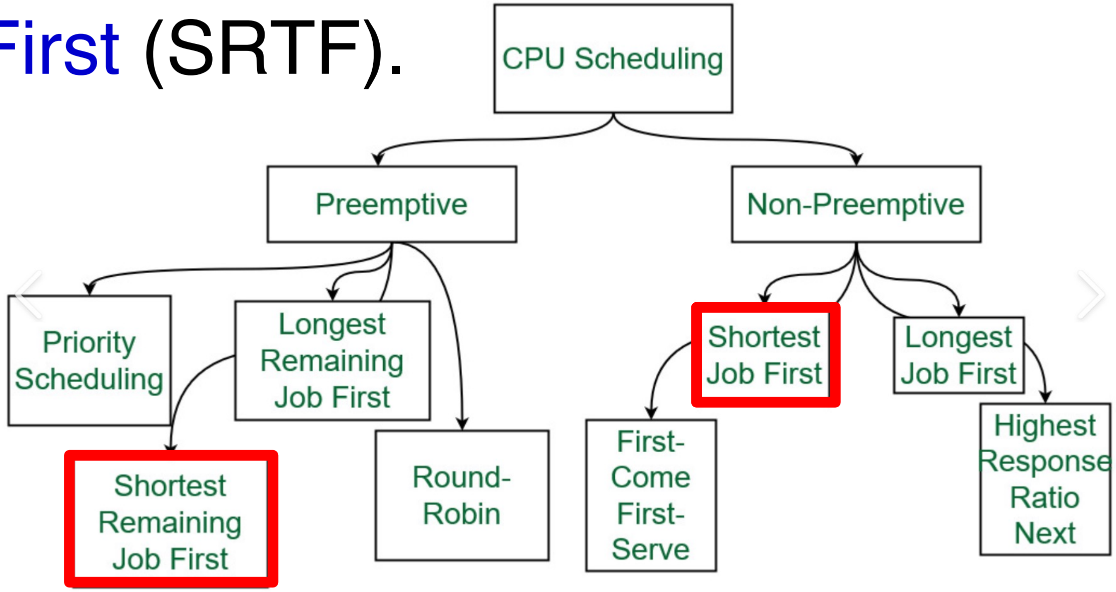


Min Heap

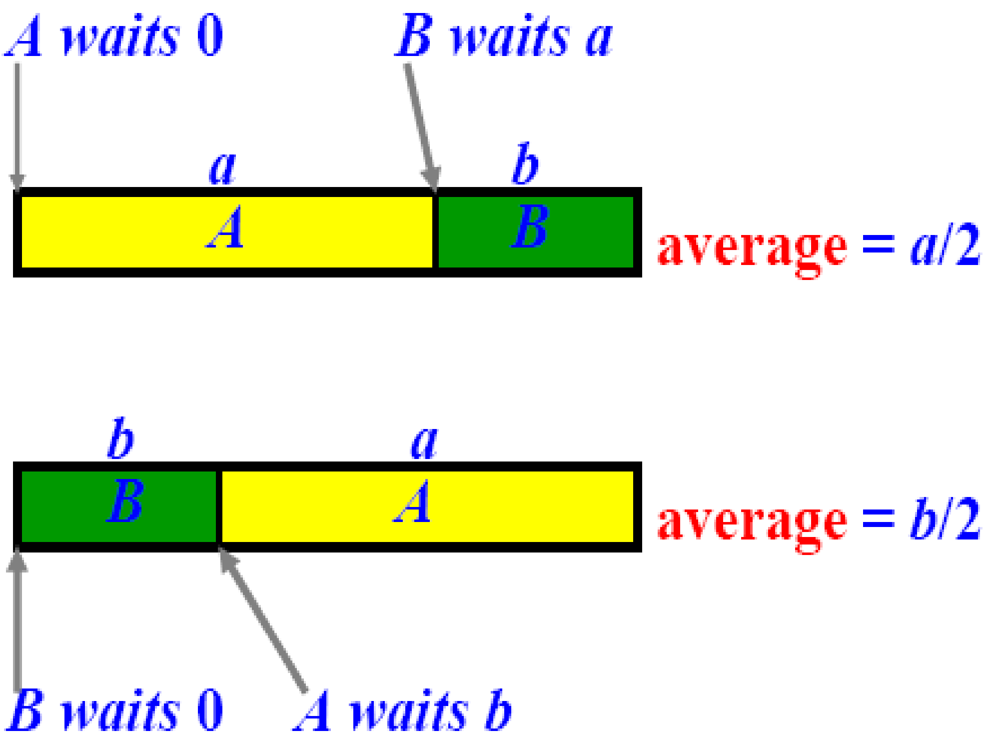


Shortest-Job-First Scheduling (Cont.)

- SJF can be non-preemptive or preemptive.
  - Non-preemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
  - Preemptive – if a new process arrives (or enters the ready queue) with CPU burst length less than remaining time of current executing process, preemp
  - This scheme is known as the Shortest-Remaining -Time-First (SRTF).



SJF can be proved optimal – It gives minimum average waiting time for a given set of processes.

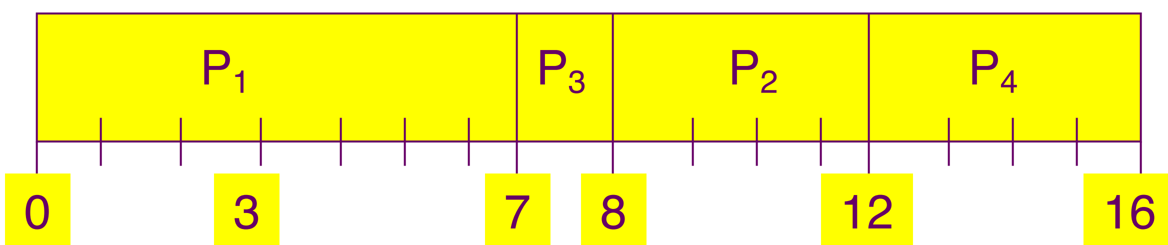


- Every time we make a short job before a long job, we reduce average waiting time.
- We may switch out of order jobs until all jobs are in order.
- If all the jobs are sorted, job switching is impossible.

An Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

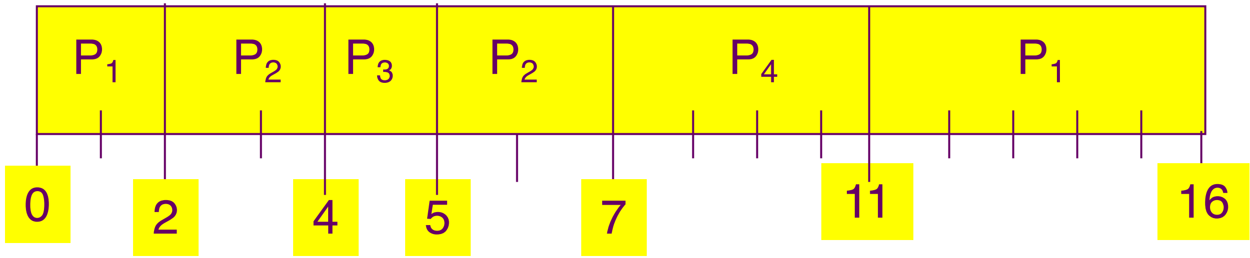


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

An Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 + 2)/4 = 3

But How Do We Know the Next CPU Burst of a Process?

- Without a good answer to this question, SJF cannot be used for CPU scheduling.

- We try to predict the next CPU burst!

- Can be done by using the length of previous CPU bursts, using exponential averaging.

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

Two Extreme Examples of Exponential Averaging

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

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

Expand Exponential Averaging Formula

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

- If we expand the formula, we get:

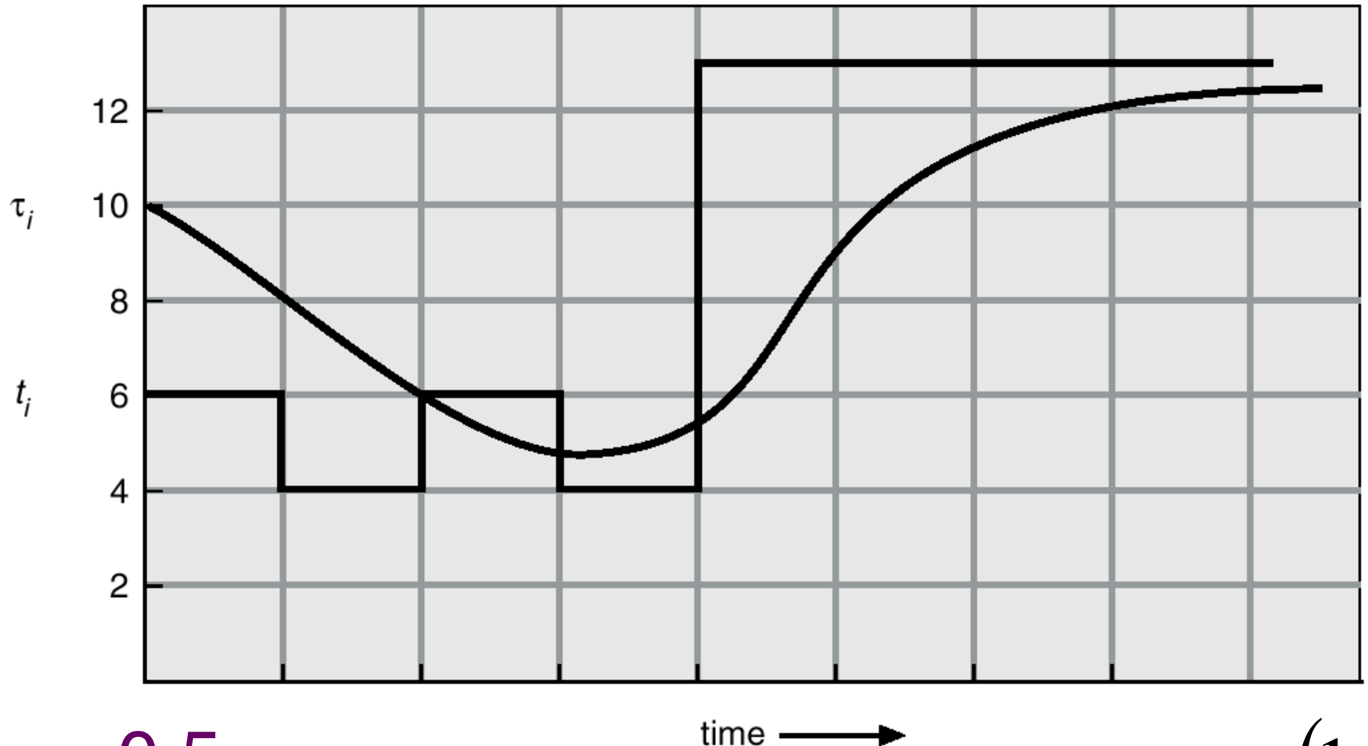
  $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + \dots + (1 - \alpha)^j \alpha\, t_{n-j} + \dots$

  $+ (1 - \alpha)^{n+1} \tau_0$

  - Then, τ_{n+1} is a linear combination of τ_0, t_1, t_2, …, t_n

- Since both α and $(1 - \alpha)$ are no more than 1, the $(n- j)^{th}$ term has the weight $(1 - \alpha)^j \alpha$, which decreases exponentially as the index j grows

An Example of Predicting the Length of the Next CPU Burst

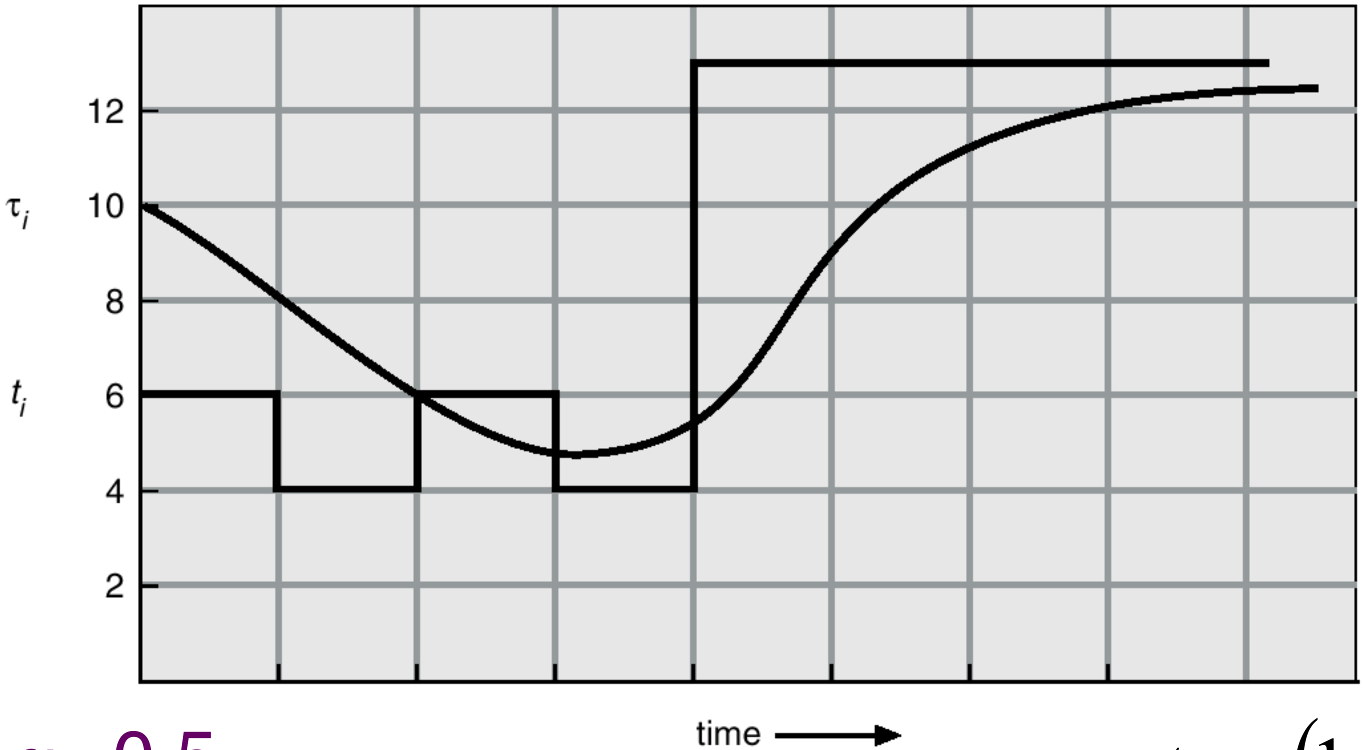


Assume $\alpha = 0.5$

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

CPU burst (t_i)		6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	What are predicted values?						. . .

An Example of Predicting the Length of the Next CPU Burst



Assume $\alpha = 0.5$

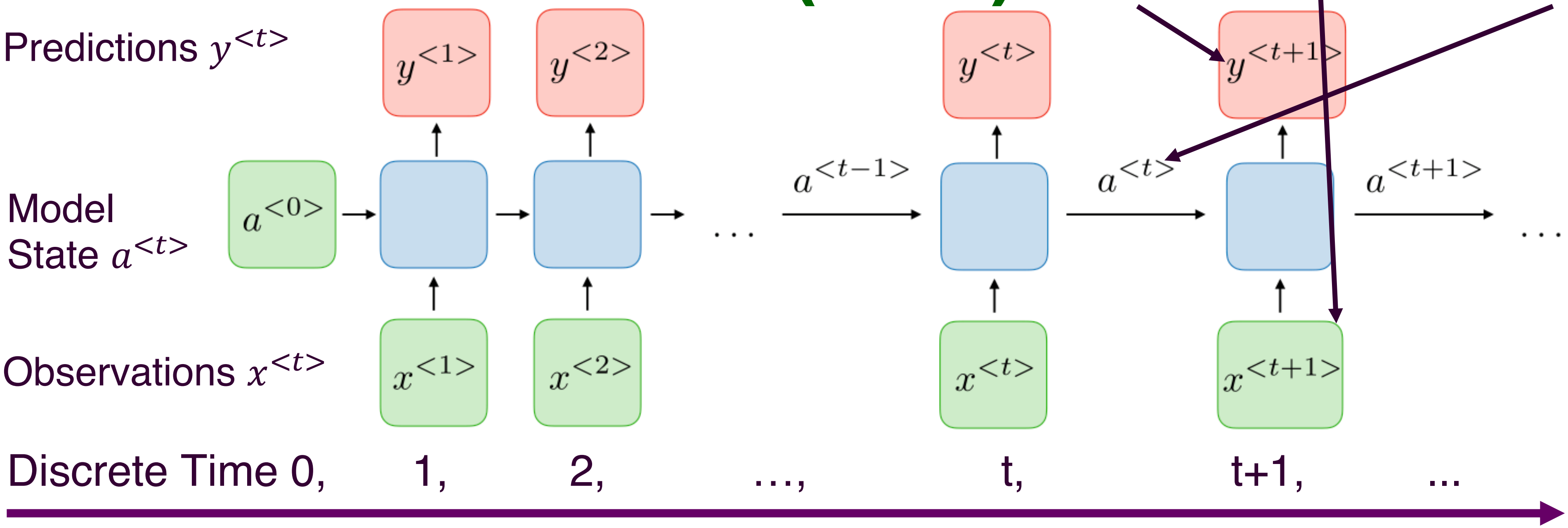
$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$

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

Question: How to train the model parameter α?

Exponential averaging is just one time series prediction tool. There are many others, e.g., Recurrent Neural Networks (RNN)

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



- The seq2seq loss function L is defined based on the sum of prediction errors of all time steps

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}(\hat{y}^{<t>}, y^{<t>})$$

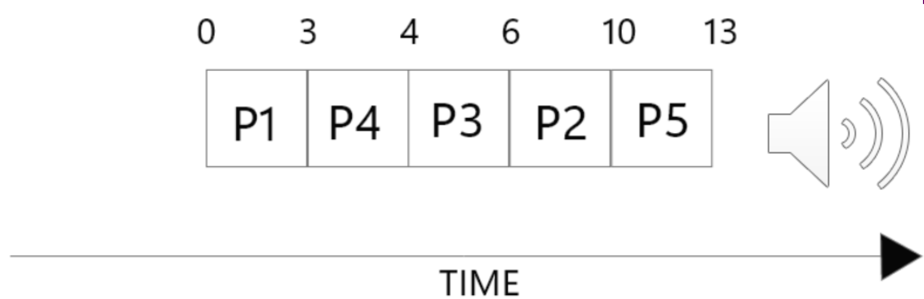
SJF Problems

- It is difficult to estimate the next burst time value accurately.

- SJF is in favor of short jobs. As a result, some long jobs may not have a chance to run at all. This is called *starvation*.

Priority Scheduling

- Each process has a *priority*.
- Priority may be decided internally or externally:
  - internal priority: determined by time limits, memory requirement, # of files, and so on.
  - external priority: not controlled by the OS (*e.g.*, importance of the process)
- The scheduler always picks the process (in ready queue) with the highest priority to run.
  - The lesser the numeric value of priority, the higher the priority of the process

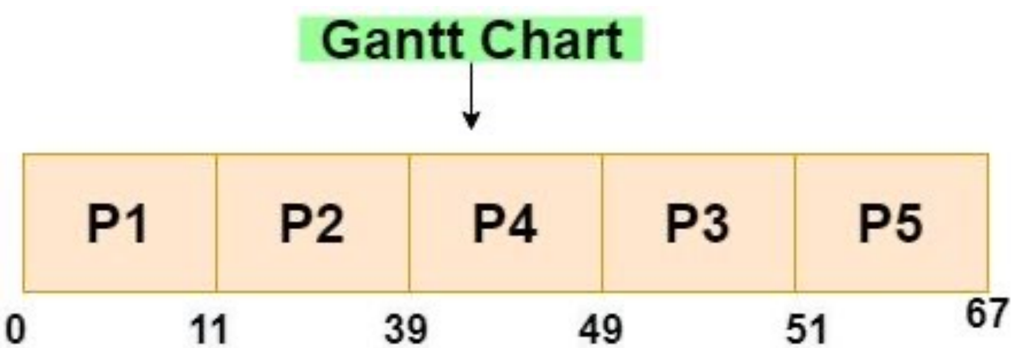


Process	P1	P2	P3	P4	P5
Burst Time	3	4	2	1	3
Priority	1	3	2	1	4

Priority Scheduling (Cont.)

- FCFS and SJF can be regarded as special cases of priority scheduling. (Why?)
- Priority scheduling can be non-preemptive or preemptive. An example of non-preemptive SJF:
  - The lesser the numeric value of priority, the higher the priority of the process
    - P1 has the lowest arrival time so it is scheduled first.
    - Next process P4 arrives at time=2….

Process ID	Priority	Arrival Time	Burst Time
P1	2	0	11
P2	0	5	28
P3	3	12	2
P4	1	2	10
P5	4	9	16



Priority Scheduling (Cont.)

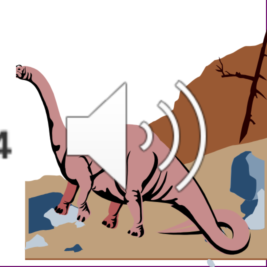
- With preemptive priority scheduling, if the newly arrived process has a higher priority than the running one, the latter is preempted.
- An example:

Process	Arrival Time	Priority	CPU Burst time
P0	0	4	5
P1	1	1	2
P2	3	3	4
P3	4	2	3

- Gantt chat:

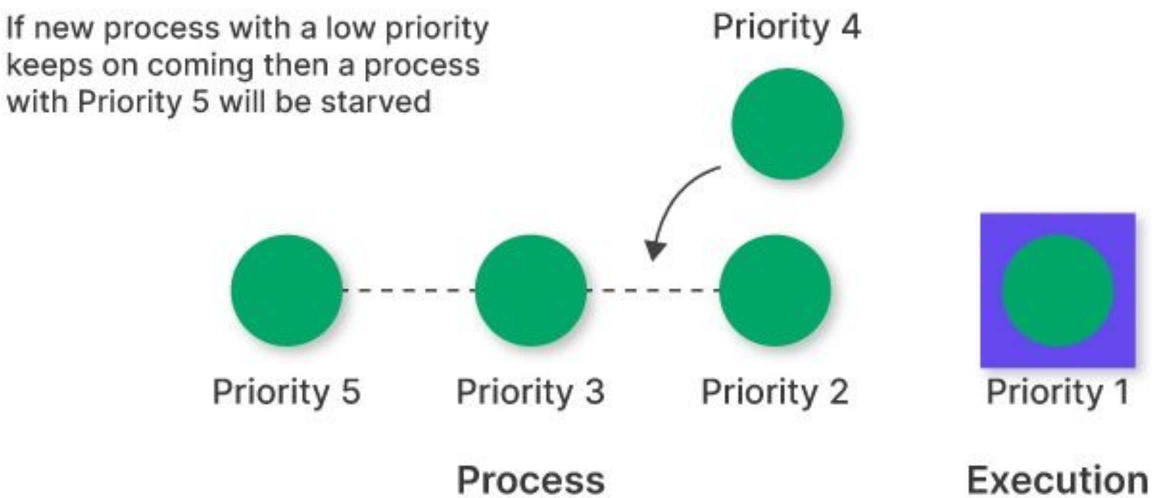
P0	P1	P2	P3	P2	P0

0 1 3 4 7 10 14



Aging

- Indefinite block (or starvation) may occur: a low priority process may never have a chance to run



- *Aging* (gradually increases the priority of processes that wait in system for a long time) is a technique to overcome starvation problem.
  - Example: If 0 is the highest (*resp*., lowest) priority, we could decrease (*resp*., increase) the priority of a waiting process by 1 each fixed period (e.g. minute)

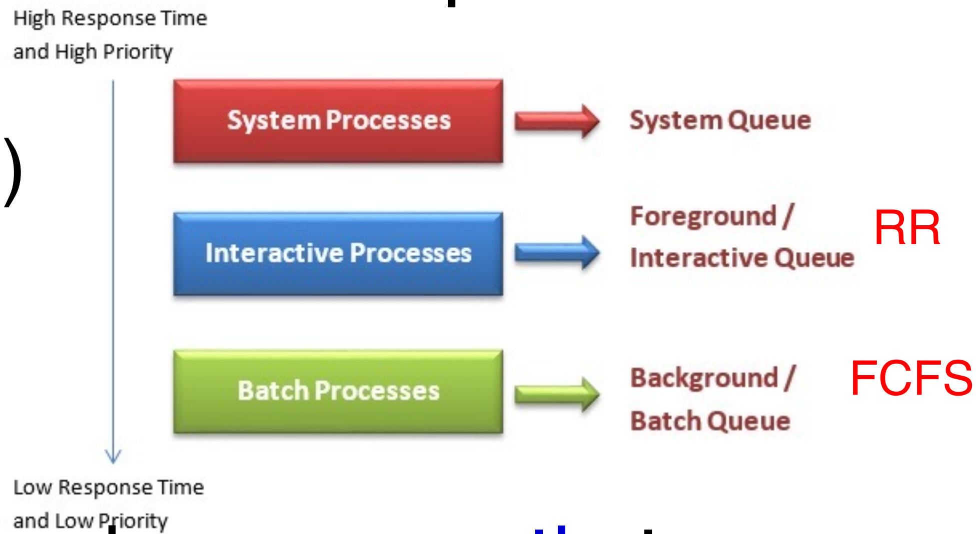
A Short Recap

	Average Turnaround Time	Response Time	Fairness
FCFS	Bad, Convoy effect	Bad, convoy effect	Bad
Round Robin	Bad, change with time quantum	Good	Good
Lottery	Bad, any policy that seeks fairness is bad on performance	Probabilistic, so no guarantee on the worst case	Better and more flexible, but ticket assignment is hard
SJF	Provably optimal	Bad	Bad, essentially a priority scheduler that favors short jobs
Priority Scheduling (I/O bound > CPU bound)	Could be good, if higher priority is given to processes with shorter CPU bursts	Bad, a low-priority process may not be executed after a long time	Bad, have starvation problem, can be mitigated by aging

Can we combine the advantages of SJF and Round Robin?

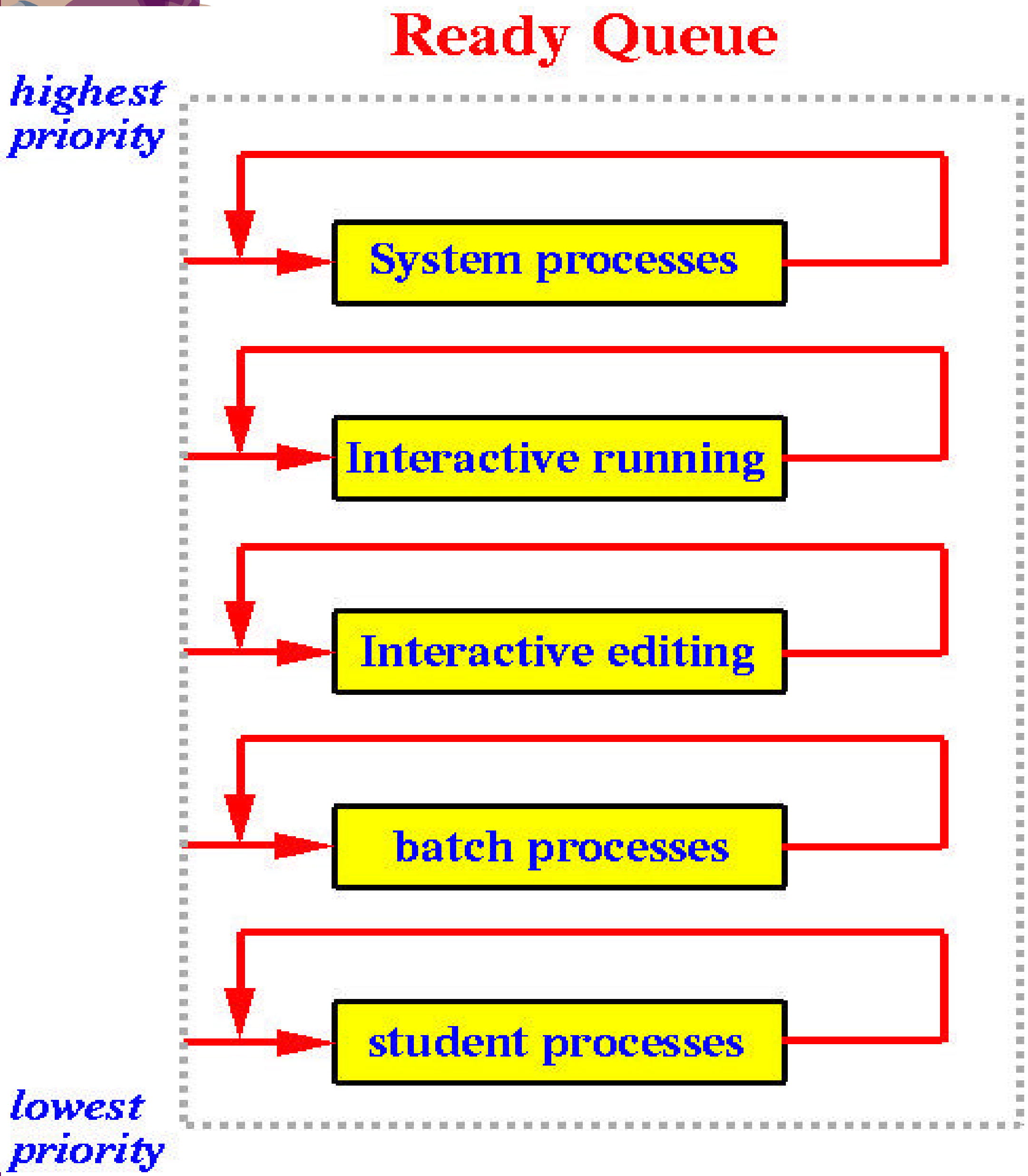
Multilevel Queue

- Ready queue is partitioned into separate queues:
  - foreground (interactive)
  - background (batch)



- Each process is assigned permanently to one queue based on some properties of the process (*e.g.*, memory usage, priority, process type)

- Each queue has its own scheduling algorithm,
  - foreground: RR for good fairness and response time
  - background: FCFS for simplicity



- A process P can run only if all queues above the queue that contains P are empty.
- When a process is running and a process in a higher priority queue comes in, the running process is preempted.

Multilevel Queue (Cont.)

- Scheduling must be done between the queues.
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Lottery Scheduling – 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

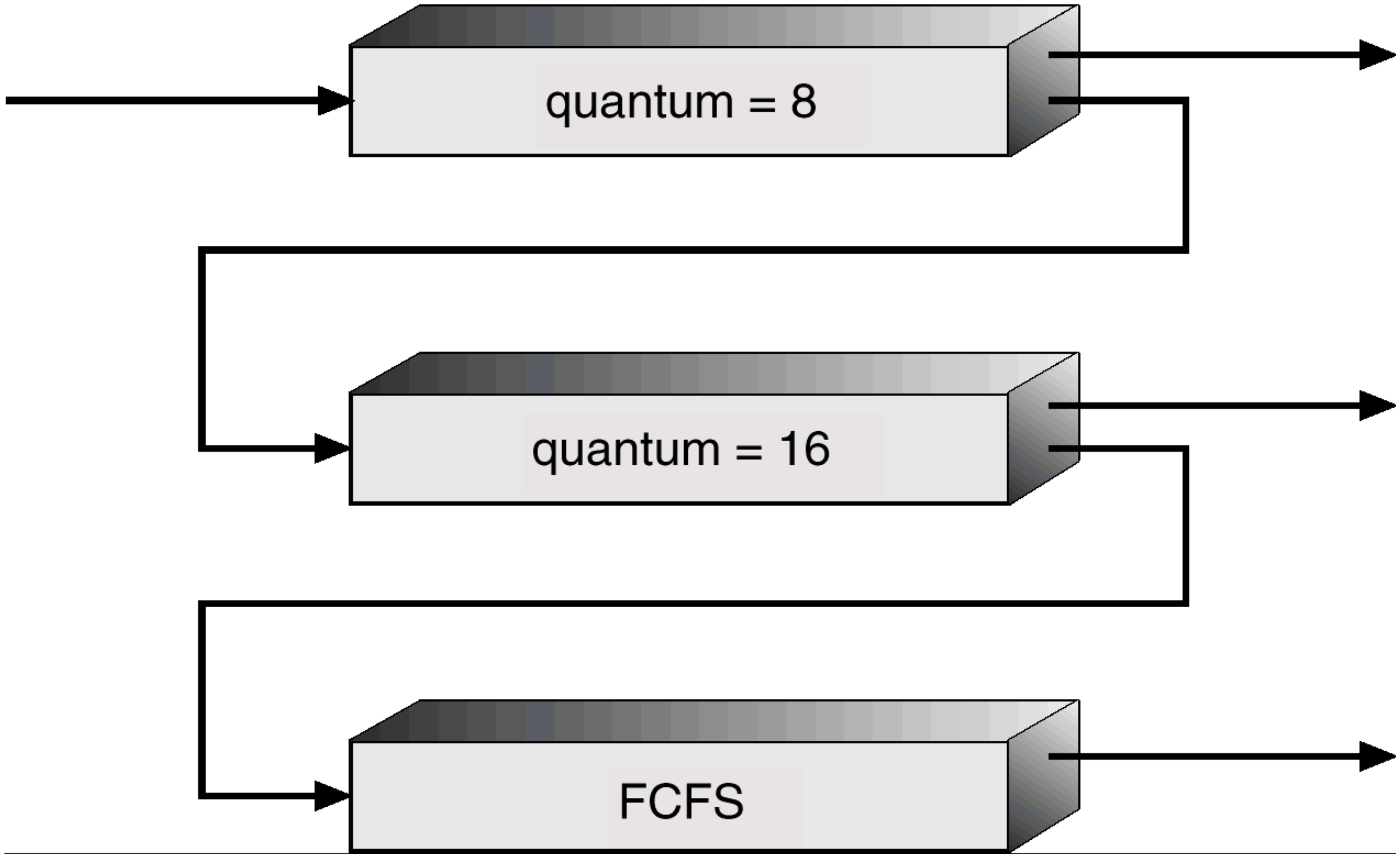
Multilevel Feedback Queue

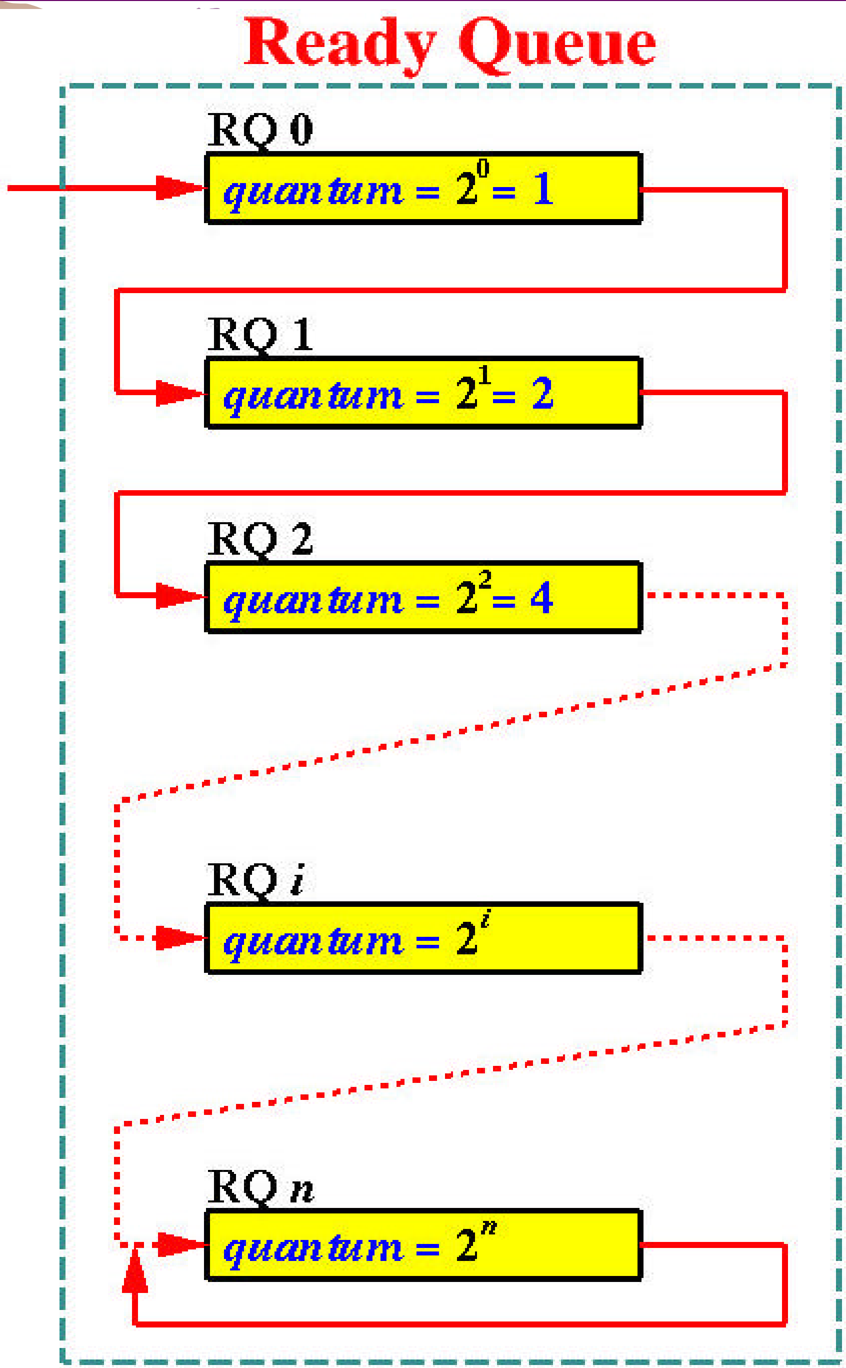
- *Multilevel queue with feedback scheduling* is similar to multilevel queue; however, it allows processes to move between queues.
  - Aging can be implemented by this way
- Basic Idea: Processes with shorter (longer) CPU bursts are given higher (lower) priority.
- If a process uses more (less) CPU time, it is moved to a queue of lower (higher) priority. As a result, I/O-bound (CPU-bound) processes will be in higher (lower) priority queues.
  - Example: if a process didn’t finish (or finish) in its allocated time quantum, it will be demoted to a lower-priority queue (or promoted to a higher-priority queue).

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR with time quantum 16 milliseconds
  - Q_2 – FCFS (equivalently, RR with ∞ time quantum)
- An example of demotion to low-priority queue
  - A new job enters Q_0 which is served by RR. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to Q_1.
  - At Q_1 job is again served RR and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues





- Processes in queue i have time quantum 2^i
- When a process' behavior changes, it may be placed (*i.e.*, promoted or demoted) into a difference queue.
- Thus, when an I/O-bound process starts to use more CPU, it may be demoted to a lower queue

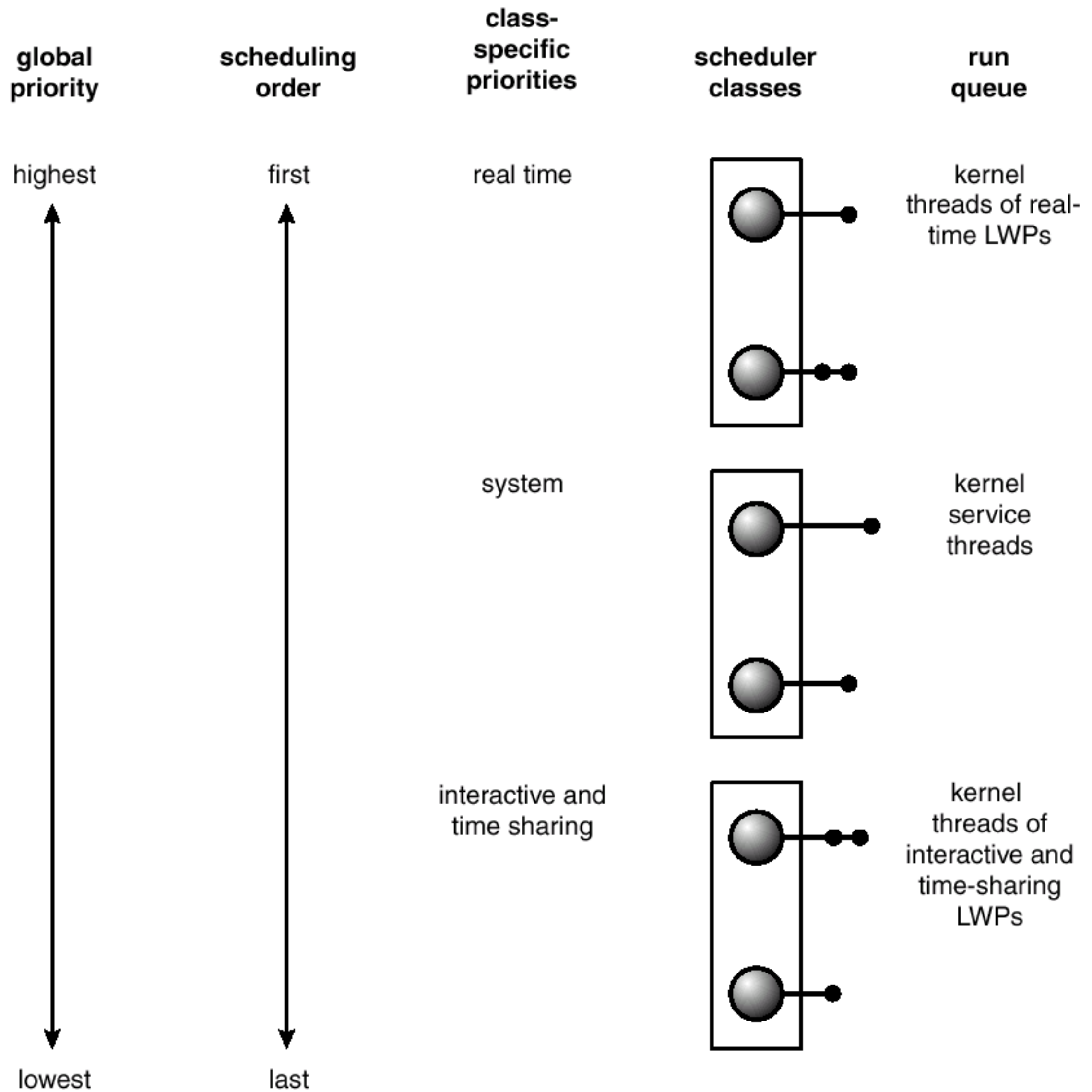
Multilevel Feedback Queue (Cont.)

- 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

Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling

Solaris 2 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

Lowest priority (priority 0)

Highest priority (priority 59)

Demoted to lower priority

Windows 2000 (XP) Priorities

Thread priority level

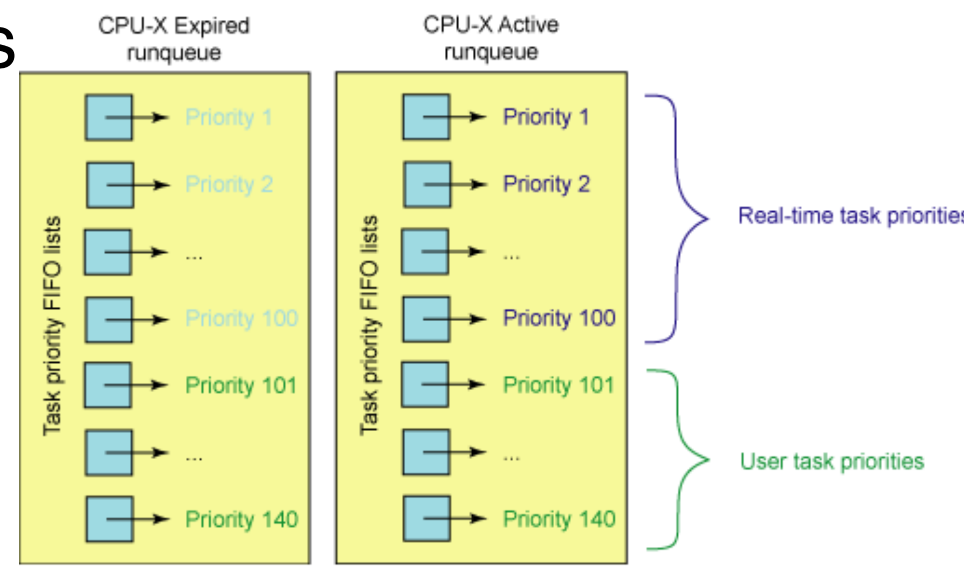
	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Process priority class

https://msdn.microsoft.com/en-us/library/windows/desktop/ms685100(v=vs.85).aspx

Linux Scheduling

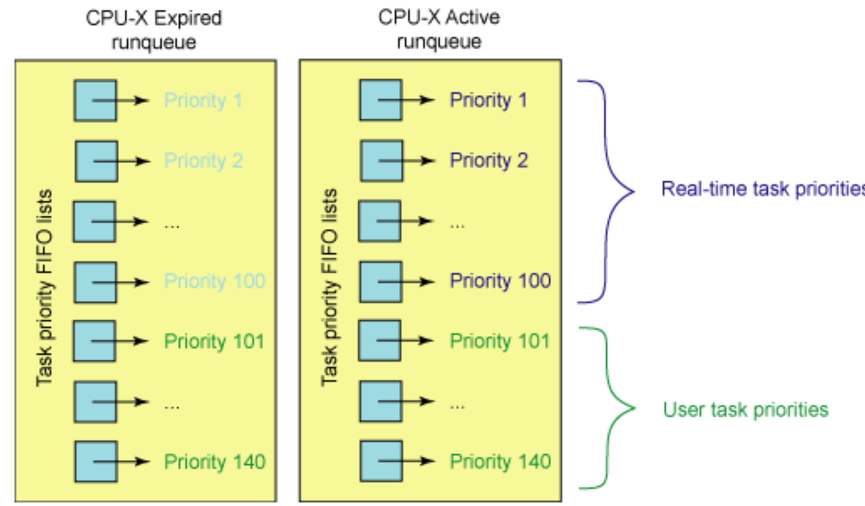
- Each CPU has a runqueue made up of 140 priority lists that are serviced in FIFO order.
- Tasks that are scheduled to execute are added to the end of respective runqueue's priority list
- Two scheduling algorithms
  - time-sharing algorithms for user tasks, and
  - real-time scheduling algorithms



The Linux 2.6 scheduler runqueue structure

Linux Scheduling (Cont.)

- Real-time
  - Posix.1b compliant
    - ✓two classes: FCFS and RR
    - ✓Highest priority process always runs first
  - Soft real-time

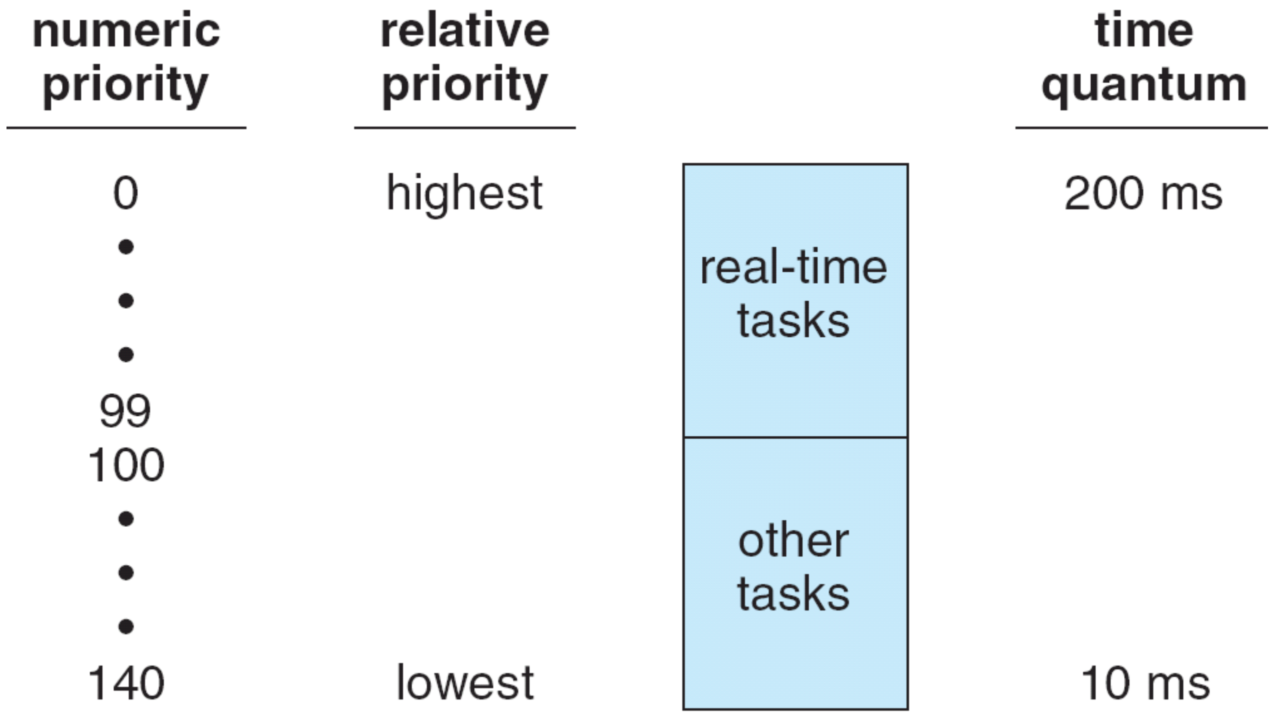


The Linux 2.6 scheduler runqueue structure

- Time-sharing
  - Prioritized credit-based (优先级化的基于信用值的调度): process with most credits is scheduled next
  - Credit subtracted when timer interrupt occurs
  - When credit = 0, another process chosen
  - When all runnable processes have credit = 0, recrediting occurs
    - ✓Based on factors including priority and history

The Relationship Between Priorities and Time-slice length

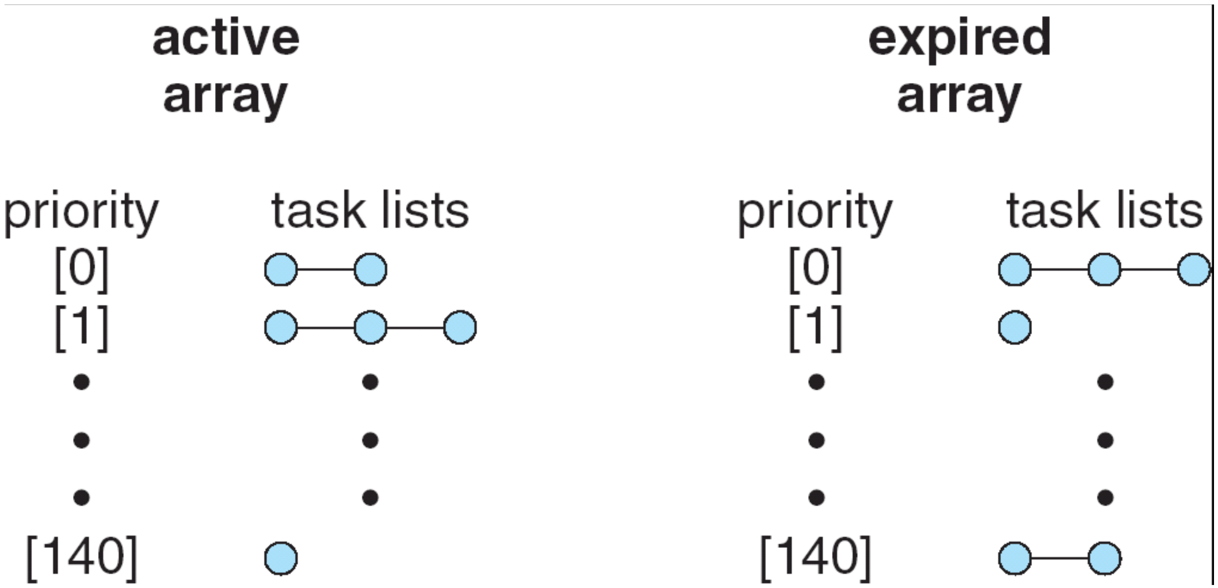
- The first 100 priority lists of the runqueue are reserved for real-time tasks, and the last 40 are used for user tasks (MAX_RT_PRIO=100 and MAX_PRIO=140)



http://www.cs.montana.edu/~chandrima.sarkar/AdvancedOS/CSCI560_Proj_main/index.html

List of Tasks Indexed According to Priorities

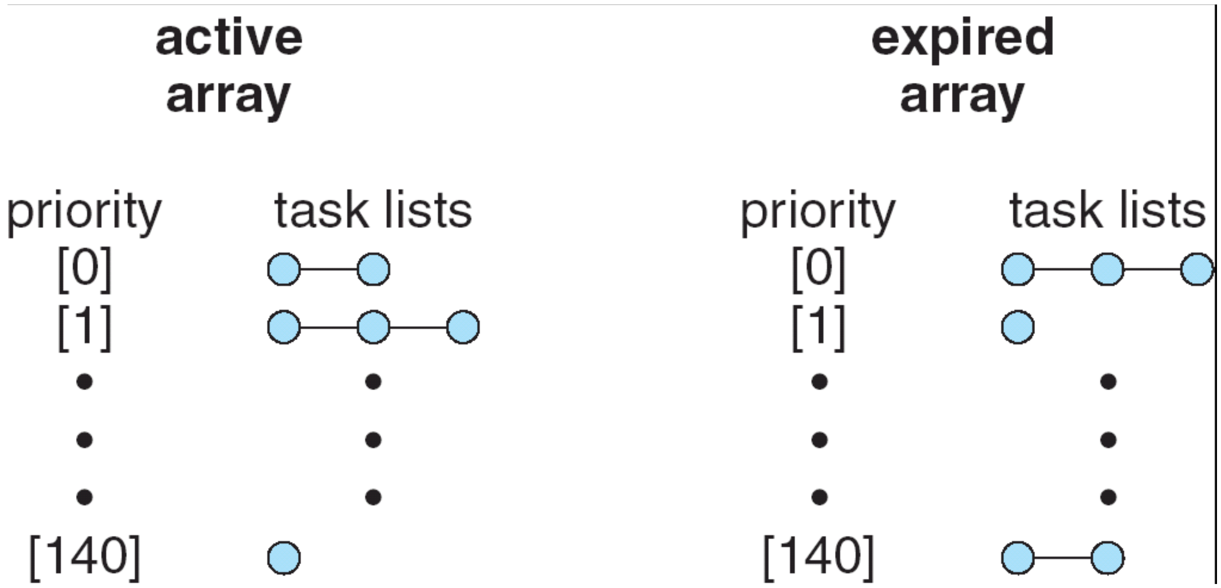
- In addition to the CPU's runqueue, which is called the active runqueue, there's also an expired runqueue



- When a task on the active runqueue uses all of its time slice, it's moved to the expired runqueue. During the move, its time slice is recalculated (and so is its priority)

List of Tasks Indexed According to Priorities (cont.)

- If no tasks exist on the active runqueue for a given priority, the pointers for the active and expired runqueues are swapped, thus making the expired priority list the active one



Scheduler Policy

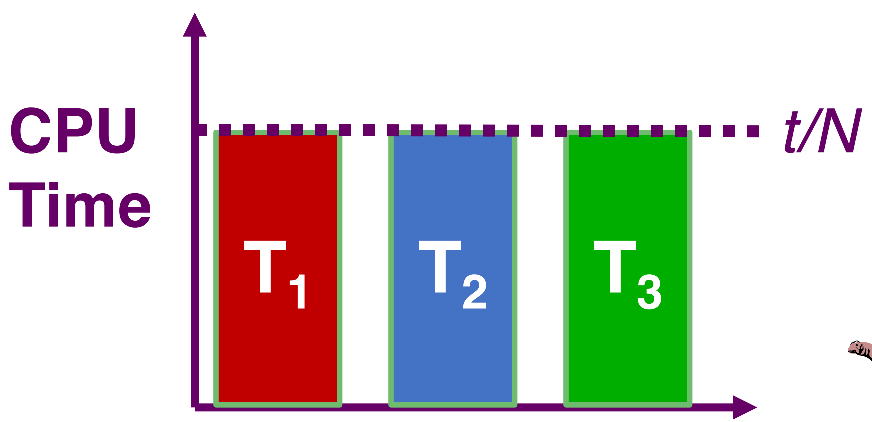
- Each process has an associated scheduling policy and a static scheduling priority
  - SCHED_FIFO - A First-In, First-Out **real-time process**
  - SCHED_RR - A Round Robin **real-time process**
  - SCHED_NORMAL: A conventional, time-shared process (used to be called SCHED_OTHER) for **normal tasks**
  - SCHED_BATCH - for "batch" style execution of processes; for **computing-intensive tasks**
  - SCHED_IDLE - for running very low priority **background job**

http://linux.die.net/man/2/sched_setscheduler

Linux Completely Fair Scheduler

- Linux CFS was a process scheduler that was merged into the 2.6.23 (October 2007) release of the Linux kernel. It was the default scheduler of the tasks of the SCHED_NORMAL class
- Goal: Each process gets an equal share of CPU
- *N* threads "simultaneously" execute on $1/N^{th}$ of CPU

At *any* time *t* we would observe:

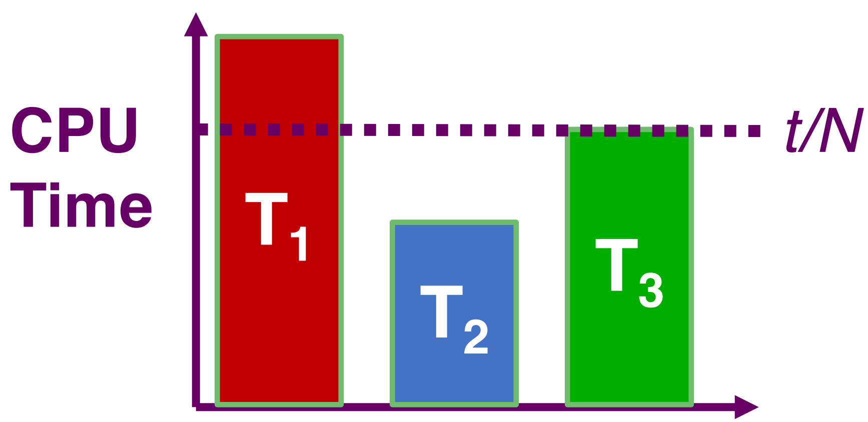


Linux Completely Fair Scheduler

- Can't do this with real hardware
  - Still need to give out full CPU in time slices
- Instead: track CPU time given to a thread so far

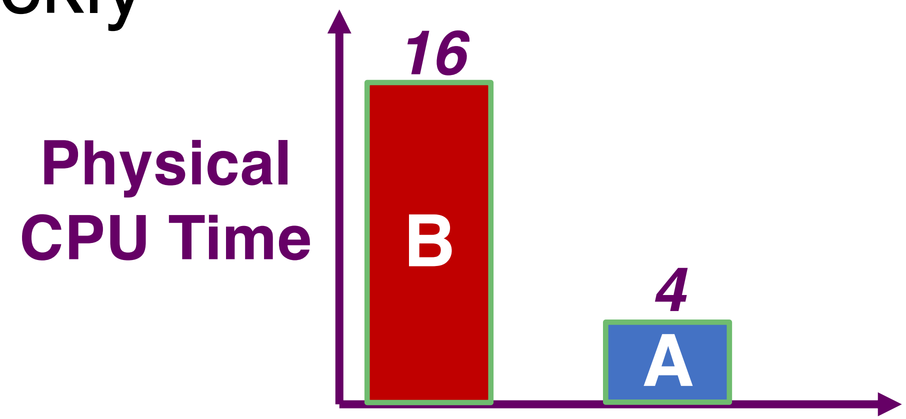
Scheduling Decision:

- "Repair" illusion of complete fairness
- Choose thread with minimum CPU time



Linux CFS

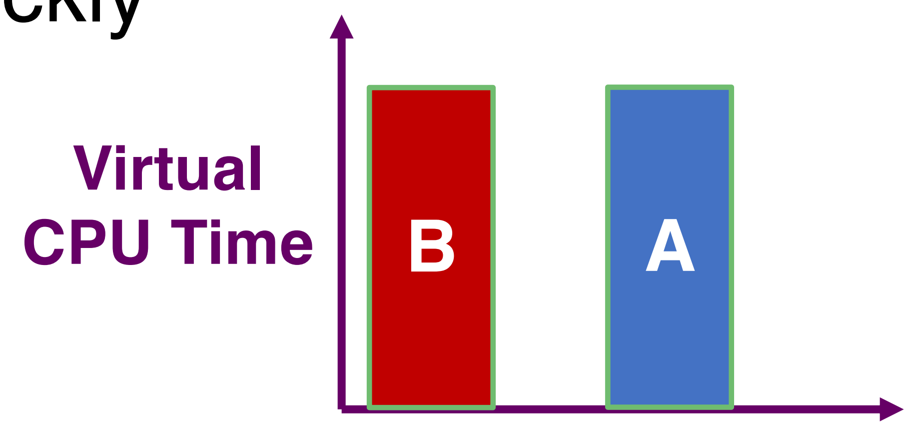
- Track a thread's *virtual* runtime rather than its true physical runtime
- Higher weight: Virtual runtime increases more slowly
- Lower weight: Virtual runtime increases more quickly



Linux CFS

- Track a thread's *virtual* runtime rather than its true physical runtime
- Higher weight: Virtual runtime increases more slowly
- Lower weight: Virtual runtime increases more quickly

Actually Used for Decisions

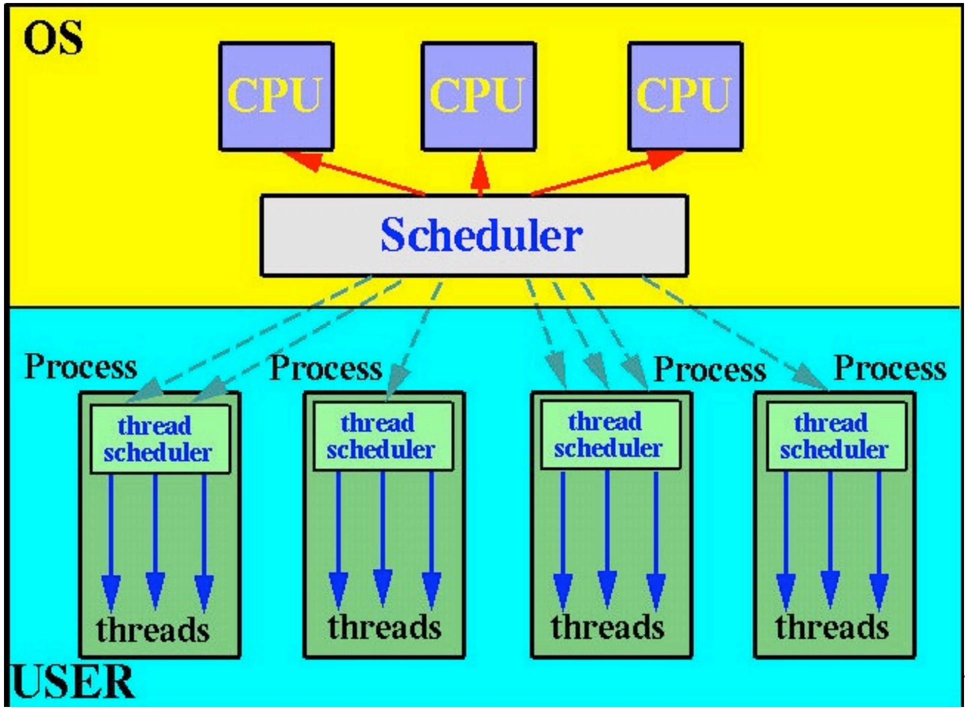


Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling

Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP
- Global Scheduling – How the kernel decides which kernel thread to run next



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);
}
```

SCHED_OTHER is the standard Linux time-sharing scheduler that is intended for all processes that do not require the special real-time mechanisms.

http://linux.die.net/man/2/sched_setscheduler

Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling

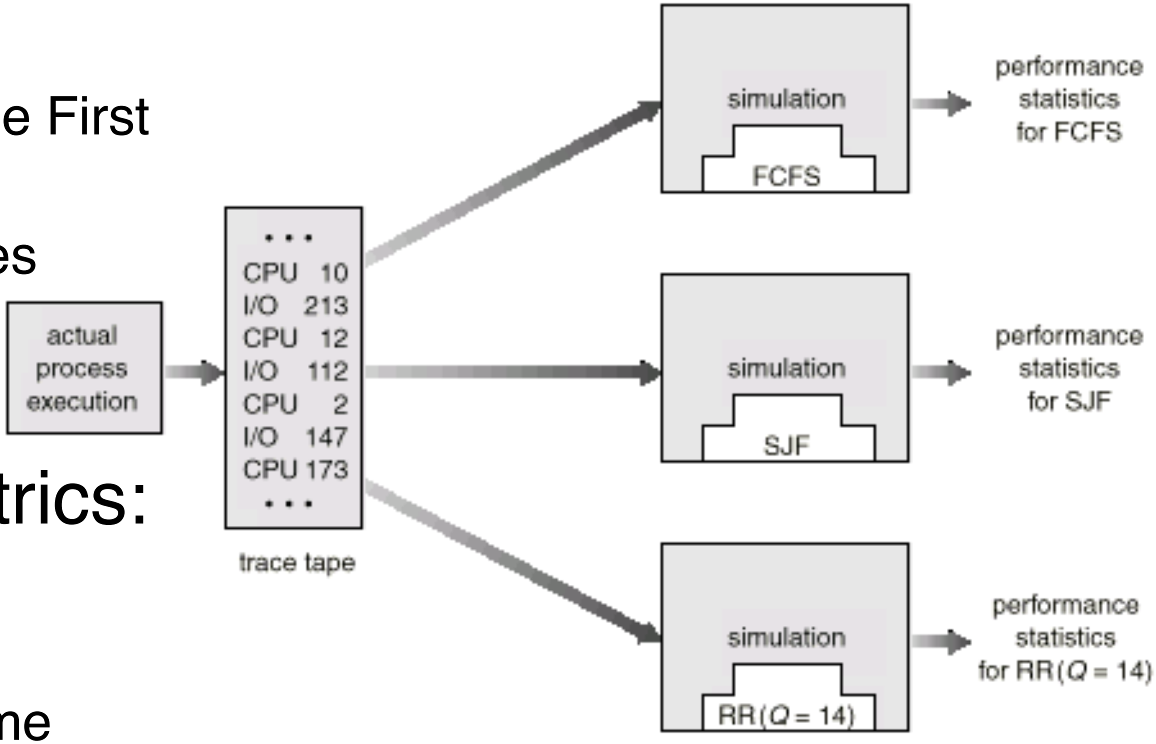
Scheduling Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each scheduling algorithm for that workload.
- Queuing models
- Simulations
- Implementation

Evaluation of CPU Schedulers by Simulation

- This simulator looks at the following scheduling algorithms:
  - First Come First Served
  - Shortest Job First
  - Shortest Remaining Time First
  - Round Robin
  - POSIX Dynamic Priorities Scheduling
- We will observe the following output metrics:
  - Job Throughput
  - CPU Utilization
  - Average Turnaround Time
  - Average Response Time
  - Average Waiting Time

Simulation and Performance Evaluation of CPU Scheduling Algorithms
https://github.com/jasmarc/scheduler
https://github.com/joedodson/cpu-scheduling-sim



Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real System Examples
- Thread Scheduling
- Algorithm Evaluation
- Multiple-Processor Scheduling

Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available.
- *Homogeneous processors* within a multiprocessor system.
  - *Load sharing*
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing.