CPU Scheduling

These slides were compiled from the Applied OSC textbook slides (Silberschatz, Galvin, and Gagne) and the instructor’s class materials.

“Clean diagrams lead to clean thinking which in turn leads to clean designs”

-- Unknown Dynamics Professor (CSUS)

CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
  - FCFS
  - SJF
  - Priority
  - Round Robin
  - Multi-level Feedback Queue
- OS Examples
- Algorithm Evaluation

- Excellent re-statement of the book's chapter: http://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html

Glossary

Term	Definition
PCB	Process Control Blocks
FCFS	First-Come First-Served scheduling algorithm
SJF	Smallest-Job-First scheduling algorithm
RR	Round Robin
FIFO	First In First Out (Queue)
Quantum	schedulable time interval (“time slice”)
Foreground	interactive processes
Background	batch (or system) processes
TAT	Turnaround Time

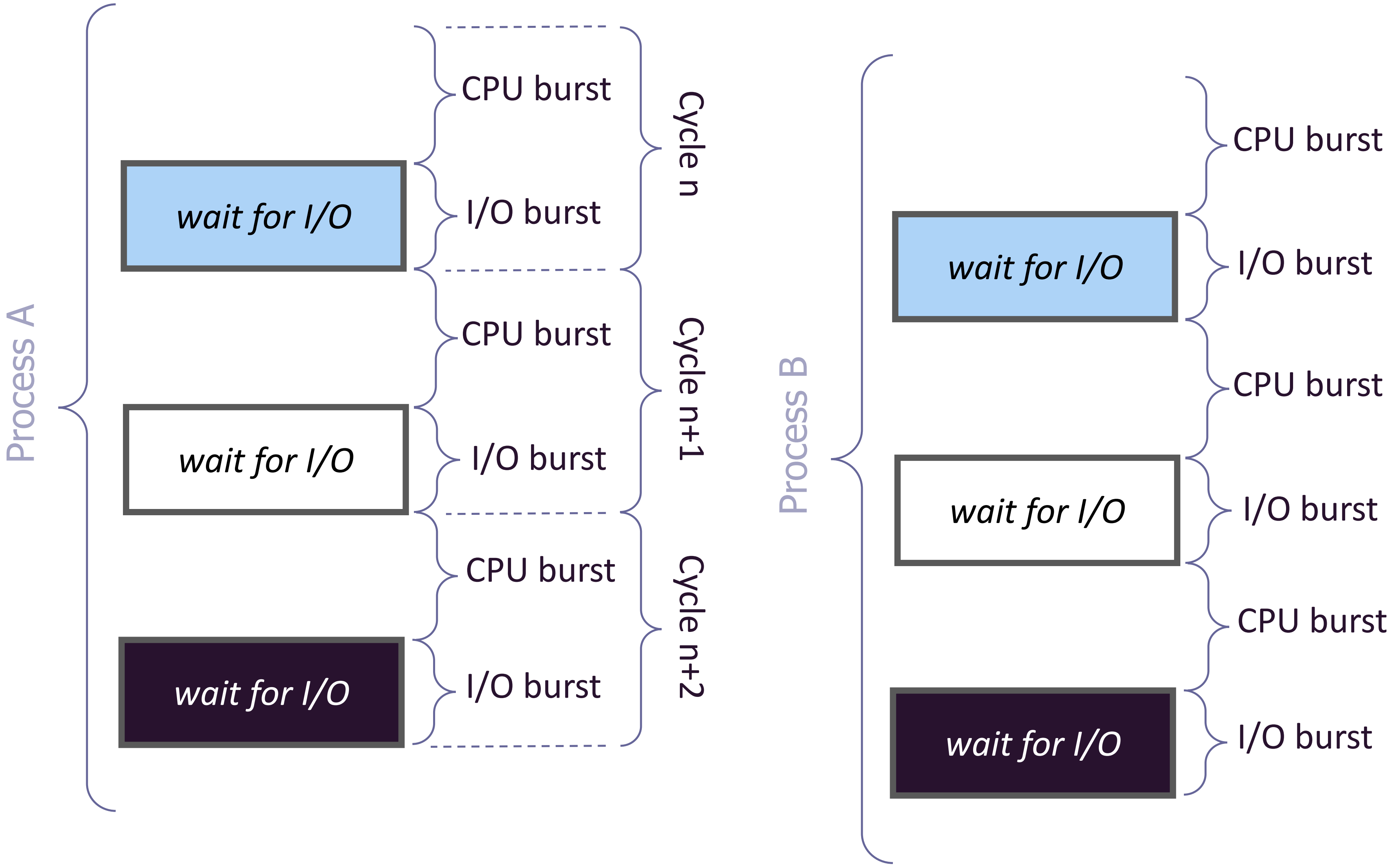
Processes vs. Threads

- Based on last class
  - One or many threads execute in the context of a process
  - There exists a nuanced distinction between processes and threads in Linux (e.g., tasks)

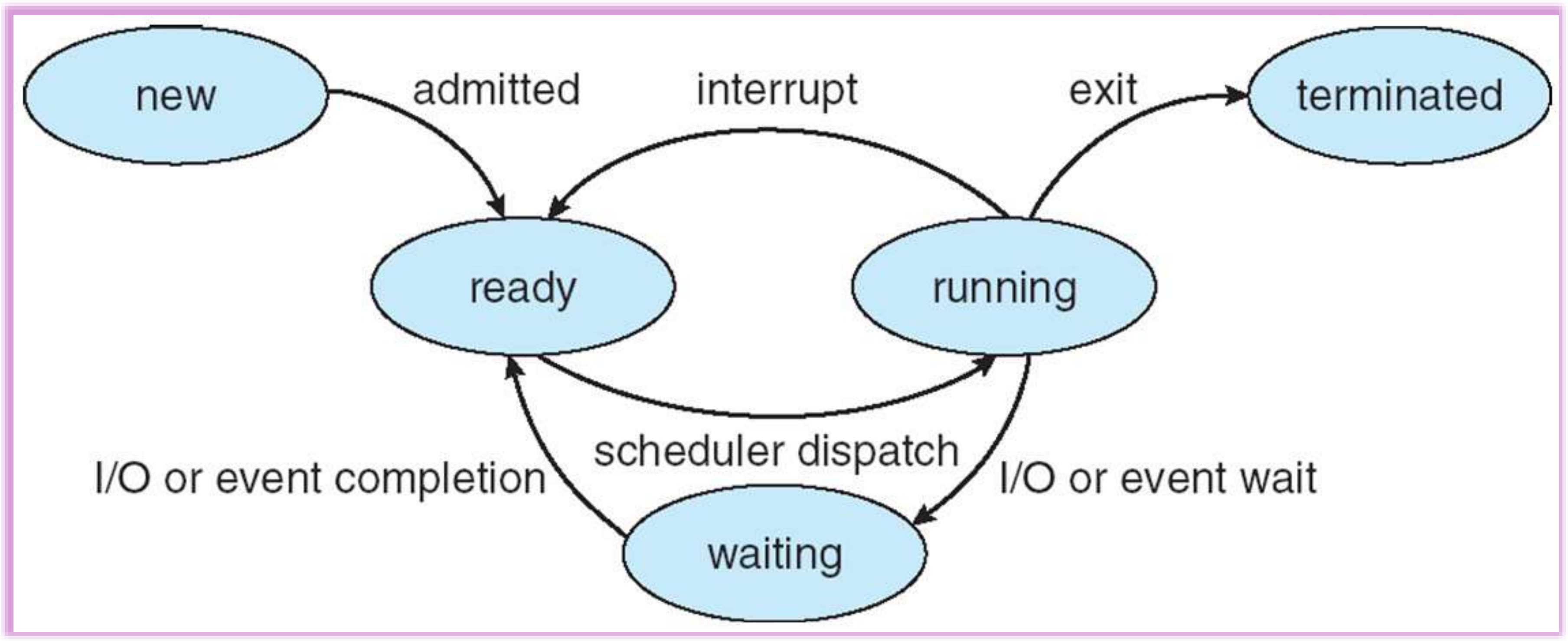
- In Chapter 5, following terms are used interchangeably
  - Process scheduling
  - Thread scheduling

- In reality: Kernel level threads being scheduled – not processes.

CPU-I/O Burst Cycles



Remember Process States



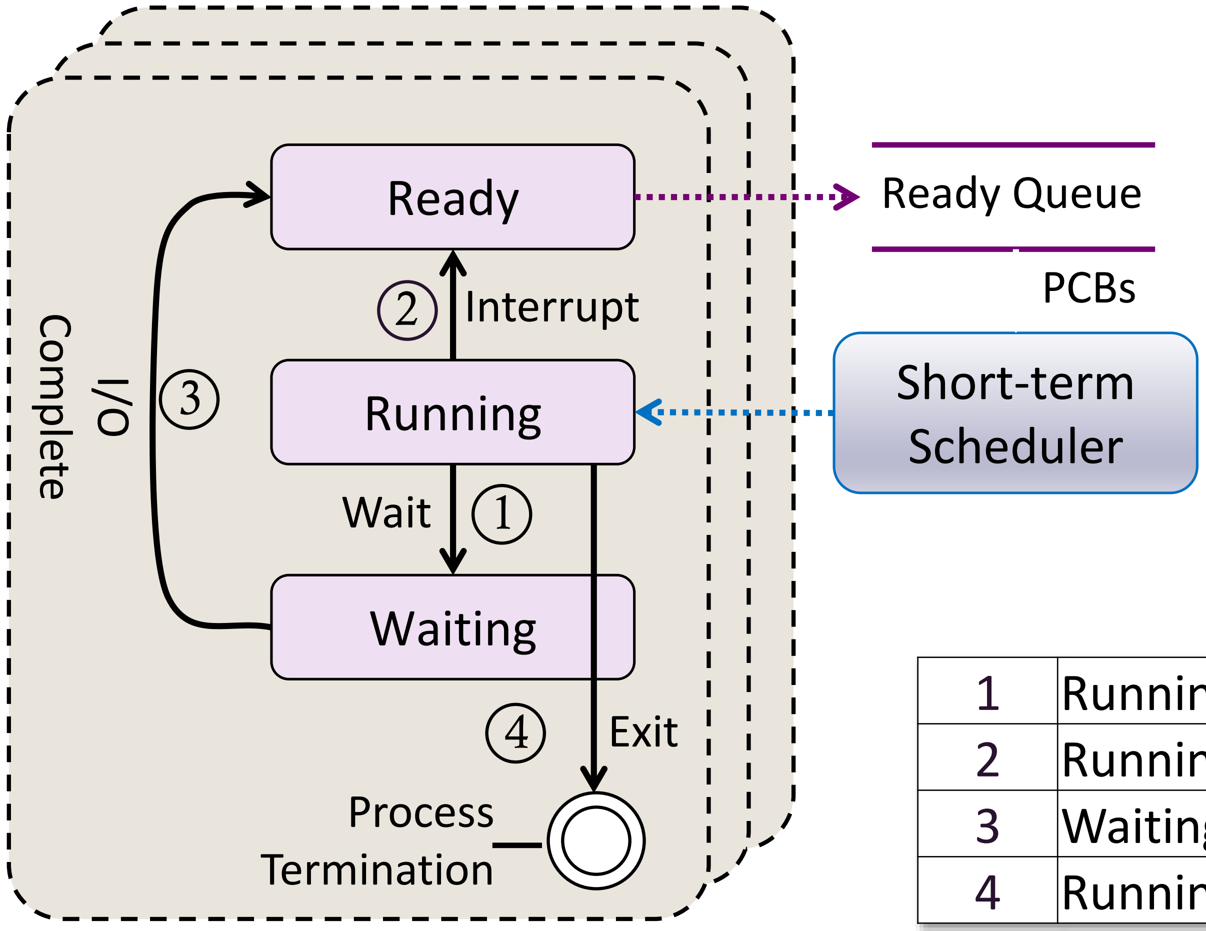
CPU Scheduler

- **Selects** from among the processes **in memory** that are ready to execute, and **allocates** the CPU to one of them. (Short-term scheduler)

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (by sleep).
  2. Switches from running to ready state (by yield).
  3. Switches from waiting to ready (by an interrupt).
  4. Terminates (by exit).
- Scheduling under 1 and 4 is *non preemptive*.

- All other scheduling is *preemptive*.

CPU Scheduler



UML Process State Diagram

1	Running → Waiting
2	Running → Ready
3	Waiting → Ready
4	Running → Termination

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible, from 40%-90%
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time (TAT)** – 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)

Discussion 1

Can an OS satisfy all these metrics at the same time?

- CPU utilization
- Throughput
- TAT
- Waiting time
- Response time?

Scheduling Strategies

- FCFS
- SJF
- Priority
- Round Robin

- Preemptive vs. non-preemtive

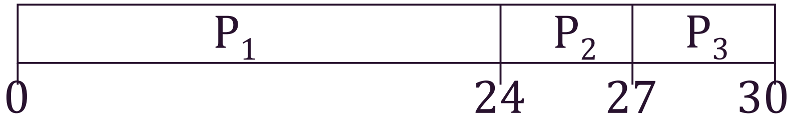
First Come First Served (FCFS) Scheduling

The process that requests the CPU first is allocated the CPU first.

- Example:

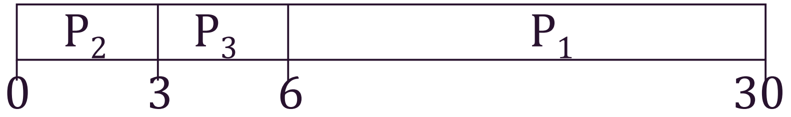
Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3



Waiting time for: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

- Suppose that the processes arrive in the order: P_2 , P_3 , P_1 .



Waiting time for: $P_1 = 6$; $P_2 = 0$; $P_3 = 3$

- Non-preemptive scheduling
- ***Troublesome*** for time-sharing systems
  - *Convoy effect* short process behind long process

Discussion 1

Let's say that the following processes are created in the following order (P_1, P_2, and then P_3), and execute for the following lengths of time.

Process	CPU Burst Time (msec)
P1	34
P2	16
P3	37

1. Calculate the following (assuming that the processes are scheduled using the FCFS algorithm):

a) The wait time for each process

b) The average wait time

c) Throughput

d) TAT for each process

e) Why are you not being asked to calculate response time or CPU utilization?

2. Which of the above measures would be 'best' to focus on, and why?

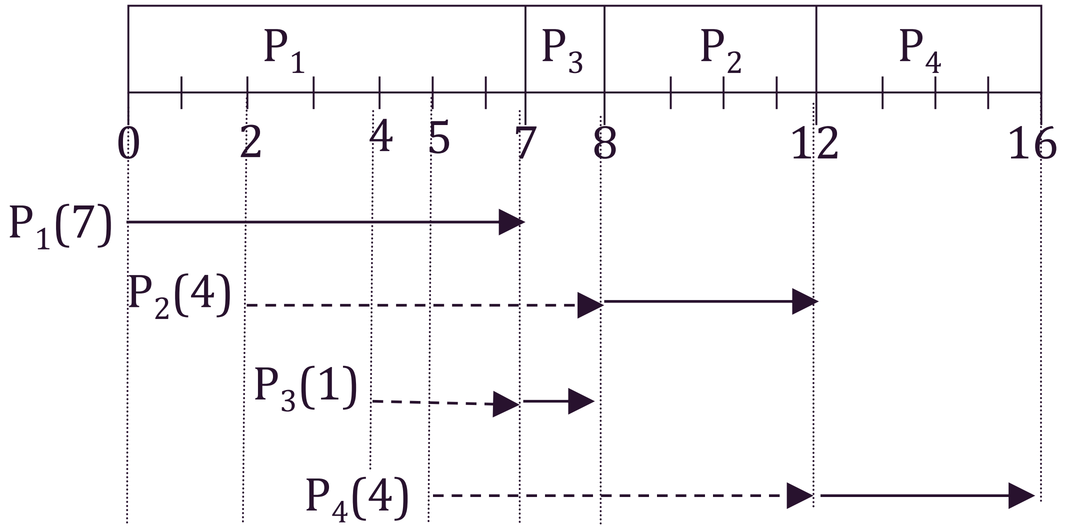
Shortest Job First (SJF) Scheduling

- Example:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Non preemptive

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



P_1's waiting time = 0

P_2's waiting time = 6

P_3's waiting time = 3

P_4's waiting time = 7

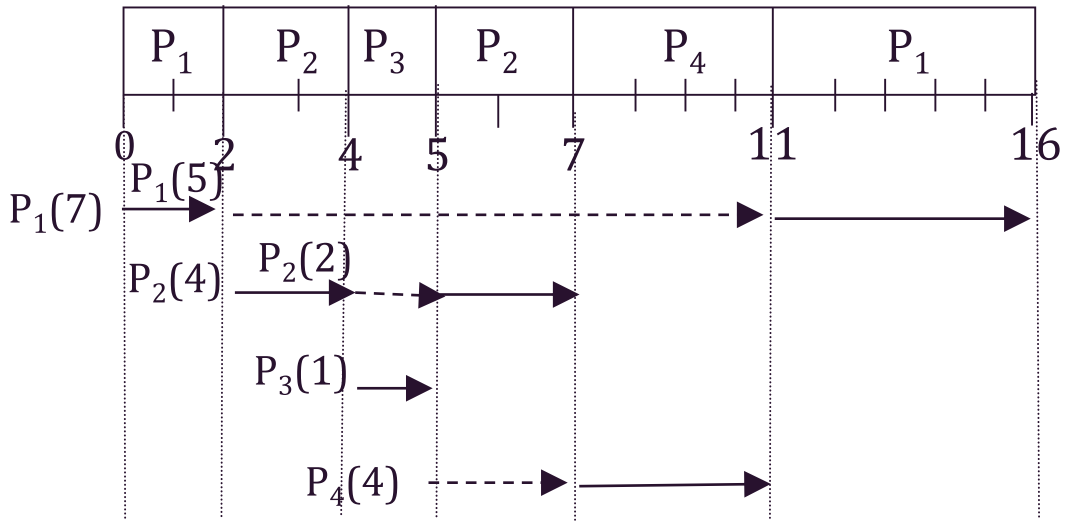
Shortest Job First Scheduling Cont'd

- Example:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Preemptive SJF

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



P_1's waiting time = 9

P_2's waiting time = 1

P_3's waiting time = 0

P_4's waiting time = 2

CLASSWORK

Process	Arrival Time	CPU Burst Time
P1	0	6
P2	2	6
P3	4	1
P4	5	2

Let's say that these processes are created at the times specified in the 'Arrival Time' column and then execute for the length of time listed in the "CPU Burst Time" column

1. Draw a Gantt chart if that work is scheduled using SJF_{NP}
   (or at least, a Gantt-esque chart – one of those bar things that shows what he CPU is executing)

   Calculate the wait time for each process, and the average wait time

2. Repeat both parts of step #1 again, this time for SJF_{P}

Shortest Job First Scheduling Cont'd

- Preemptive SJF is superior to non preemptive SJF.
- However, there are no accurate estimations to know the length of the next CPU burst
- **Estimation** introduced in Textbook:
  - Exponential Moving Average

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

Break

Priority Scheduling

- An algorithm that gives **preferential treatment to important jobs**
  - Each process is associated with a priority and **the one with the highest priority is granted the CPU**
  - **Equal priority** processes are scheduled in **FCFS order**
    - SJF is a special case of the general priority scheduling algorithm

- Priorities can be assigned to processes by a system administrator (e.g. staff processes have higher priority than student ones) or determined by the Processor Manager on characteristics such as:
  - Memory requirements
  - Peripheral devices required
  - Total CPU time
  - Amount of time already spent processing

Priority Scheduling

- A **priority** number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority in Unix but lowest in Java).
  - Preemptive
  - Non-preemptive

- Problem ≡ **Starvation** – low priority processes may never execute.
- Solution ≡ ***Aging*** – as time progresses increase the

priority of the process.

CLASSWORK

Process	Arrival Time	CPU Burst Time	Priority
P1	0	6	3
P2	2	6	2
P3	4	1	2
P4	5	2	1

Let's say that these processes are created at the times specified in the 'Arrival Time' column and then execute for the length of time listed in the "CPU Burst Time" column.

1. Draw a Gantt chart if that work is scheduled using ***non***-preemptive priority scheduling

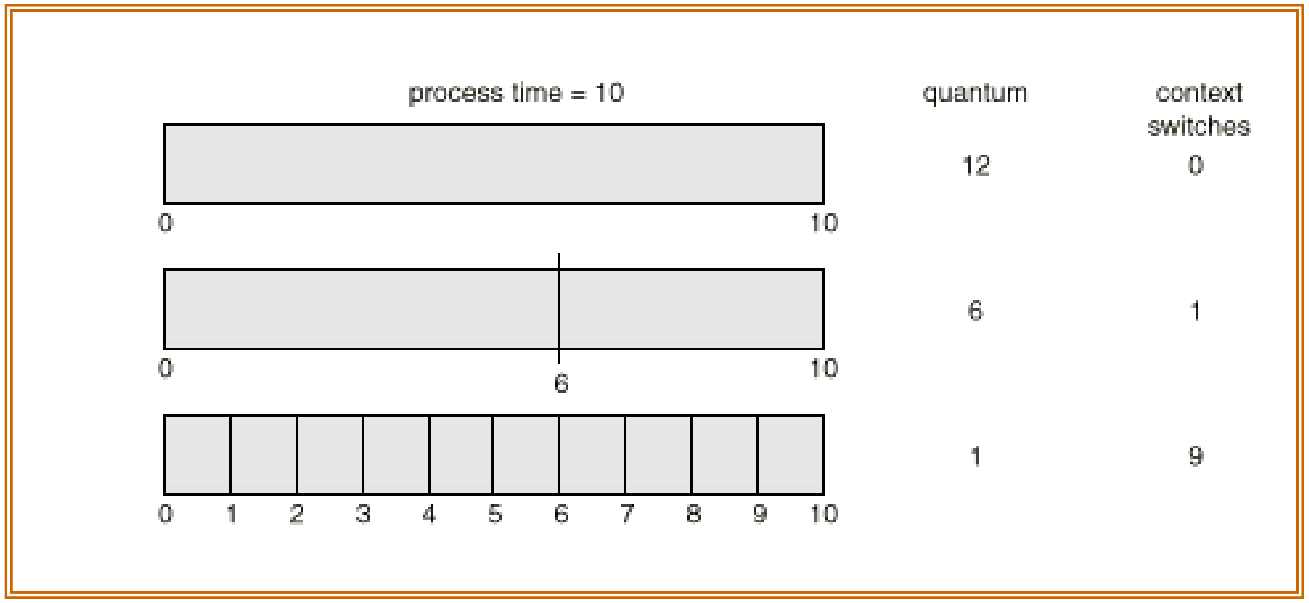
   Calculate the wait time for each process, and the average wait time.
2. Repeat both parts of step #1 again, this time for ***preemptive*** priority scheduling

Round Robin Scheduling Rule

- **A preemptive algorithm** that gives a set CPU time **to all active processes**
  - Similar to FCFS, but allows for preemption by switching between processes
  - **Ready queue** is treated as a ***circular queue*** where CPU goes round the queue, allocating each process a pre-determined amount of time
- Time is defined by a time quantum: a small unit of time (typically varying anywhere between 10 and 100 milliseconds)
  - Ready queue treated as a First-In-First-Out (FIFO) queue
    - **New processes joining the queue are added to the back of it**
- CPU scheduler **selects** the process **at the front of the queue**, sets the timer to the time quantum and grants the CPU to this process

Round Robin Scheduling

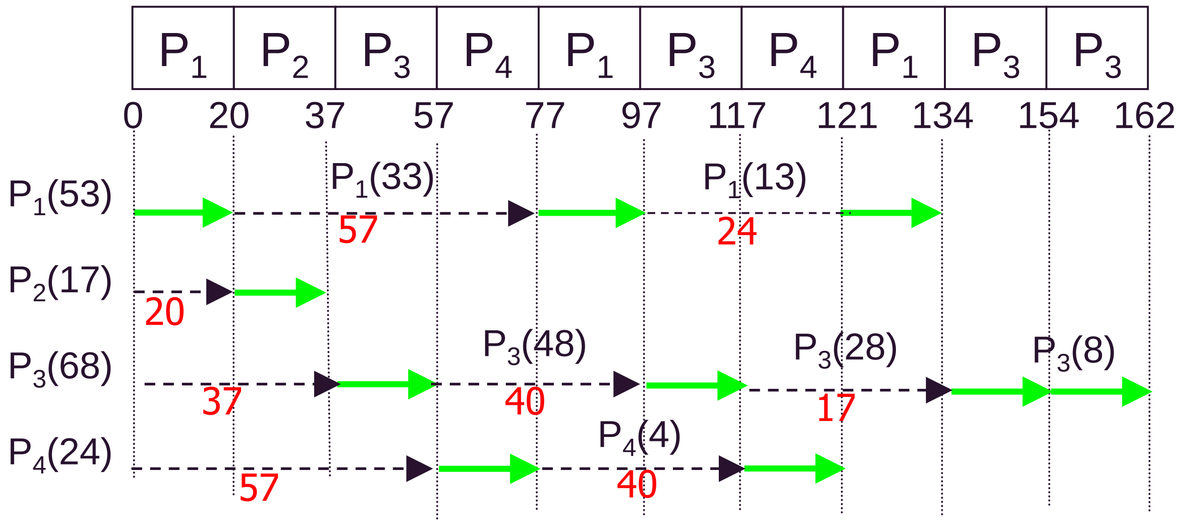
- Typically, higher average turnaround than SJF, but better *response*.
- Performance (*q* = *quantum*)
  - large *q* ⇒ FCFS
  - small *q* ⇒ *q* must be large with respect to **context switch**, otherwise the overhead is too high.
  - Trade-off: Throughput (large *q*) v. Response Time



Round Robin Scheduling

- Each process is given CPU time in turn, (i.e. time quantum: usually 10-100ms)
- …thus waits no longer than (n – 1) * time quantum
- time quantum = 20ms

Process	CPU Burst Time (ms)	Wait Time (ms)	TAT (ms)
P_1	53	57+24 = 81	134
P_2	17	20	37
P_3	68	37 + 40 + 17= 94	162
P_4	24	57 + 40 = 97	121
AVERAGE	41ms	73ms	114ms



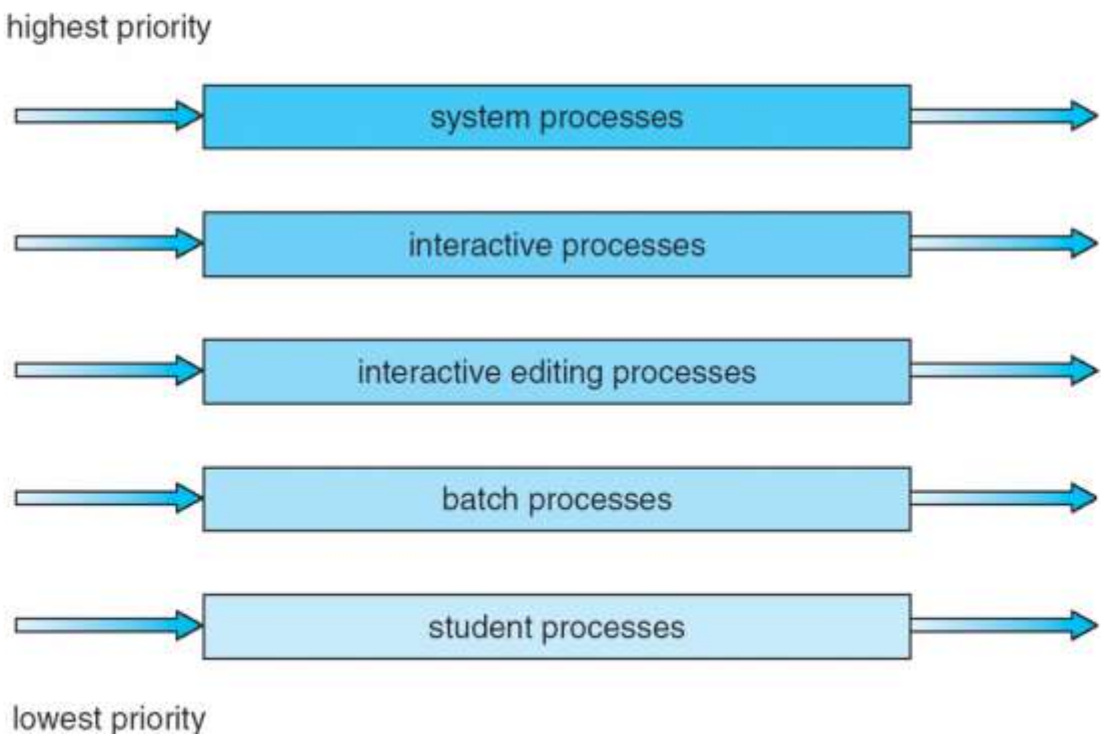
CLASSWORK

Process	Arrival Time	CPU Burst Time
P1	0	9
P2	4	4
P3	6	1
P4	8	6

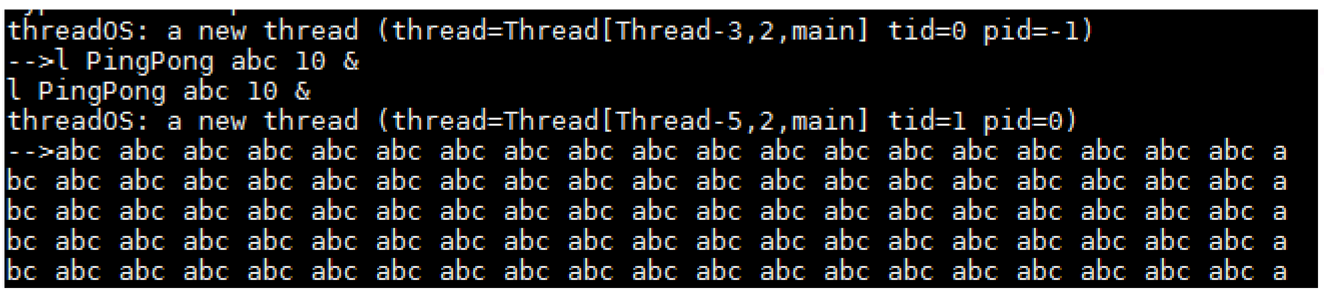
For each of the following algorithms, draw a Gantt chart and calculate the average wait time and turn around time.

1. Round-robin with quantum of one time-unit
2. Round-robin with quantum of three time-units
3. SJF w/ preemption (Shortest Remaining Time First)
4. SJF w/o preemption
5. FCFS

Multilevel Queue Scheduling



- Each queue has its own scheduling algorithm
- Inter-queue scheduling:
  - Time-slicing example:
    - foreground (interactive): RR, 80% CPU time
    - background (batch): FCFS, 20% CPU time
  - Or, priority scheduling amongst queues (lower queues ignored until higher queues empty)

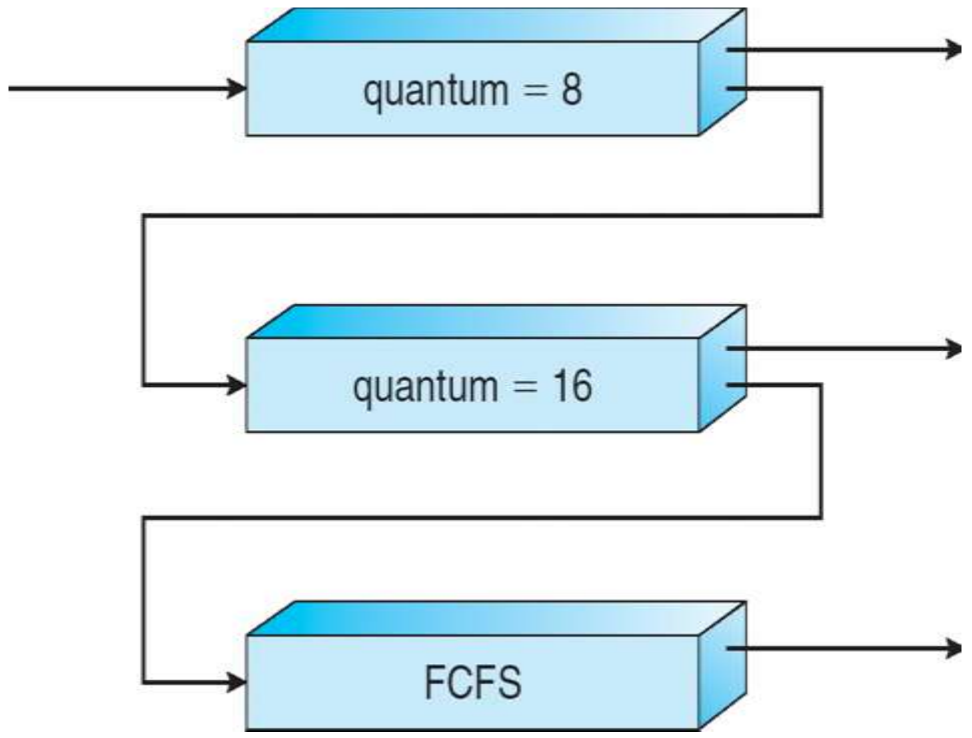


```
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l PingPong abc 10 &
l PingPong abc 10 &
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)
-->abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc a
bc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc a
bc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc a
bc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc a
bc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc a
```

Multilevel Feedback-Queue Scheduling

Variable based on consumption

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS



- A new job enters queue Q_0 which is scheduled via FCFS. When it gains CPU, the current job receives 8 milliseconds. If it does not finish in 8 milliseconds, the job is moved to queue Q_1.
- The remaining jobs in Q_0 are processed before we look at Q_1.
- At Q_1 job is again scheduled via FCFS and receives 16 additional milliseconds.
- Any jobs in Q_1, and then the remaining jobs in Q_0 are processed before we look at Q_2.
- If it still does not complete, it is preempted and moved to queue Q_2 (In this picture Q_2 follows FCFS but it could follow something else, like RR)

Class work

Process	Arrival Time	CPU Burst #1 Time	CPU Burst #2 Time
P1	0	9	9
P2	4	4	4
P3	6	1	1
P4	8	3	6

Draw a Gantt chart and calculate the average wait time and turn around time using a non-preemptive MFQS.

- The scheduler should use 3 queues.
    - The (highest priority) queue has a time slice of 2 units (of time)
    - The second queue has a time slice of 4 units
        - Both the first and second queues are scheduled using RR
    - The third (lowest priority) queue has a time slice of 8 units and is scheduled using the RR scheduling algorithm

- Remember that a process is only moved down to a lower queue when it still has time remaining in it's current burst at the end of it's turn.

Discussion

- What is the main problem in MFQS?

- How can you address this problem?

Think about your own scheduling algorithm, briefly discuss how it would solve the MFQS problem

Scheduling Summary

- Algorithms used to optimize criteria to allow for **fairness**, **responsiveness**, and **throughput**

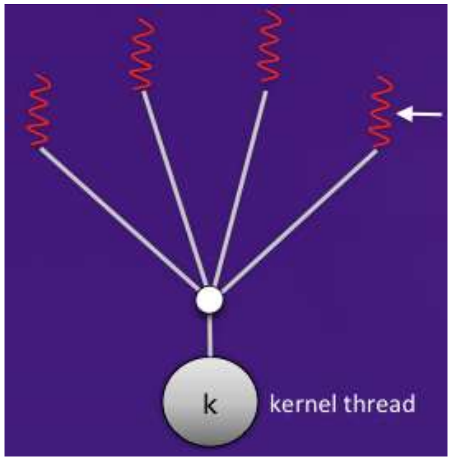
- Todays OS'es take the best characteristics of all of these:
  - **Preemptive Vs. Non-preemptive**
  - **First Come First Serve**
  - **Shortest Job First** (IO Boost, Quantum boost)
  - **Priority**: real time, interactive jobs
  - **Round-robin**/quantum (fairness)

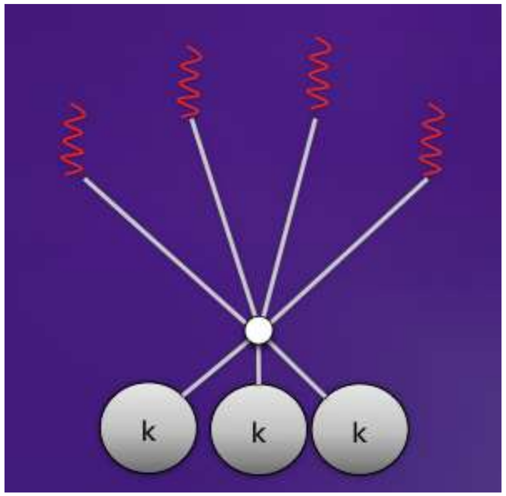
Read through the reset of slides
Finish the exercise

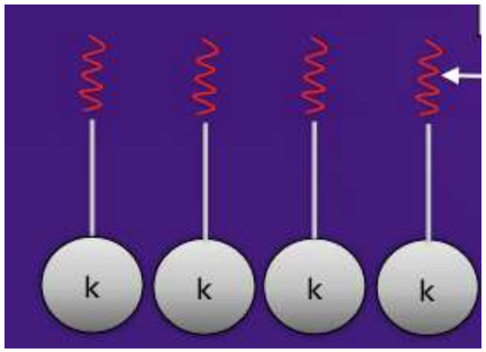
Scheduling Threads

- Who handles the scheduling? Kernel or thread Library?
  - Kernel in the end controls resources (CPU)
  - Library needs to manage user-thread scheduling if multiple

- Pthreads library allows you to pick between user threads or kernel threads
  - Assuming that the platform supports both ☺

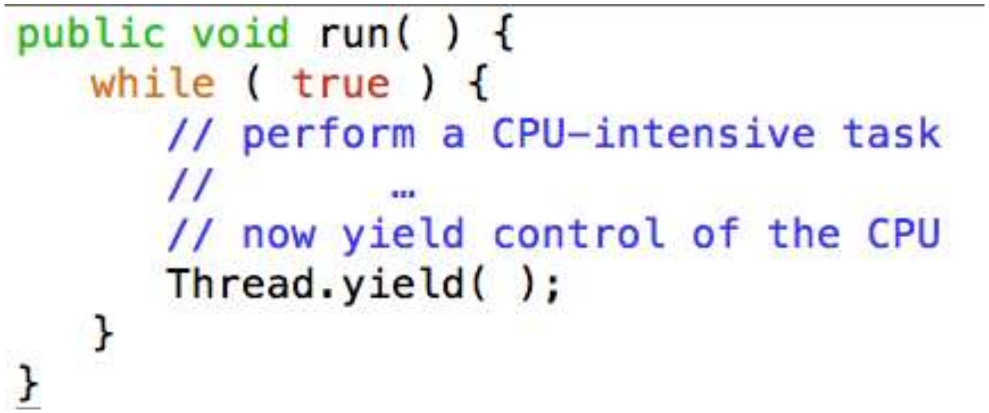






Cooperative multithreading in Java

- A thread voluntary **yielding** control of the CPU is called **cooperative** multitasking.



```
public void run( ) {
    while ( true ) {
        // perform a CPU-intensive task
        //          …
        // now yield control of the CPU
        Thread.yield( );
    }
}
```

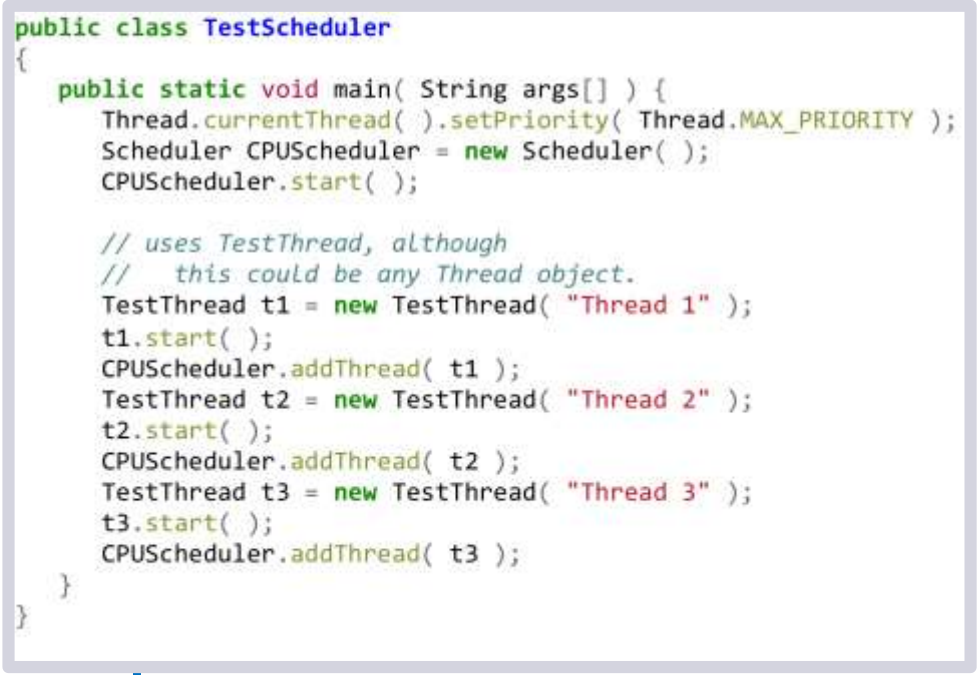
- `Thread.yield( )` is a "hint" for the OS – no guarantee
    - Linux does not care about it
    - Solaris allows a thread to yield its execution when the corresponding LWP exhausts a given time quantum

- Conclusion: Don't write a program based on **`yield( )`**

Java-Based Round-Robin Scheduler (Naïve Example)

```java
public class Scheduler extends Thread {
   public Scheduler( ) {
      timeSlice = DEFAULTSLICE;
      queue = new CircularList( );
   }
   public Scheduler( int quantum ) {
      timeSlice = quantum;
      queue = new CircularList( );
   }
   public void addThread( Thread t ) {
      t.setPriority( 2 );
      queue.addItem( t );
   }
   private void schedulerSleep( ) {
      try {
         Thread.sleep( timeSlice );
      } catch ( InterruptedException e ) { };
   }
   public void run( ) {
      private CircularList queue;
      private int timeSlice;
      private static final int DEFAULT_TIME_SLICE=1000;
      Thread current;

      this.setPriority( 6 );
      while ( true ) {
         // get the next thread
         current = ( Thread )queue.getNext( );
         if ( ( current != null ) && ( current.isAlive( ) ) ) {
            current.setPriority( 4 );
            schedulerSleep( );
            current.setPriority( 2 );
         }
      }
   }
}
```



```java
public class TestScheduler
{
   public static void main( String args[] ) {
      Thread.currentThread( ).setPriority( Thread.MAX_PRIORITY );
      Scheduler CPUScheduler = new Scheduler( );
      CPUScheduler.start( );

      // uses TestThread, although
      //   this could be any Thread object.
      TestThread t1 = new TestThread( "Thread 1" );
      t1.start( );
      CPUScheduler.addThread( t1 );
      TestThread t2 = new TestThread( "Thread 2" );
      t2.start( );
      CPUScheduler.addThread( t2 );
      TestThread t3 = new TestThread( "Thread 3" );
      t3.start( );
      CPUScheduler.addThread( t3 );
   }
}
```

Why does this not work?

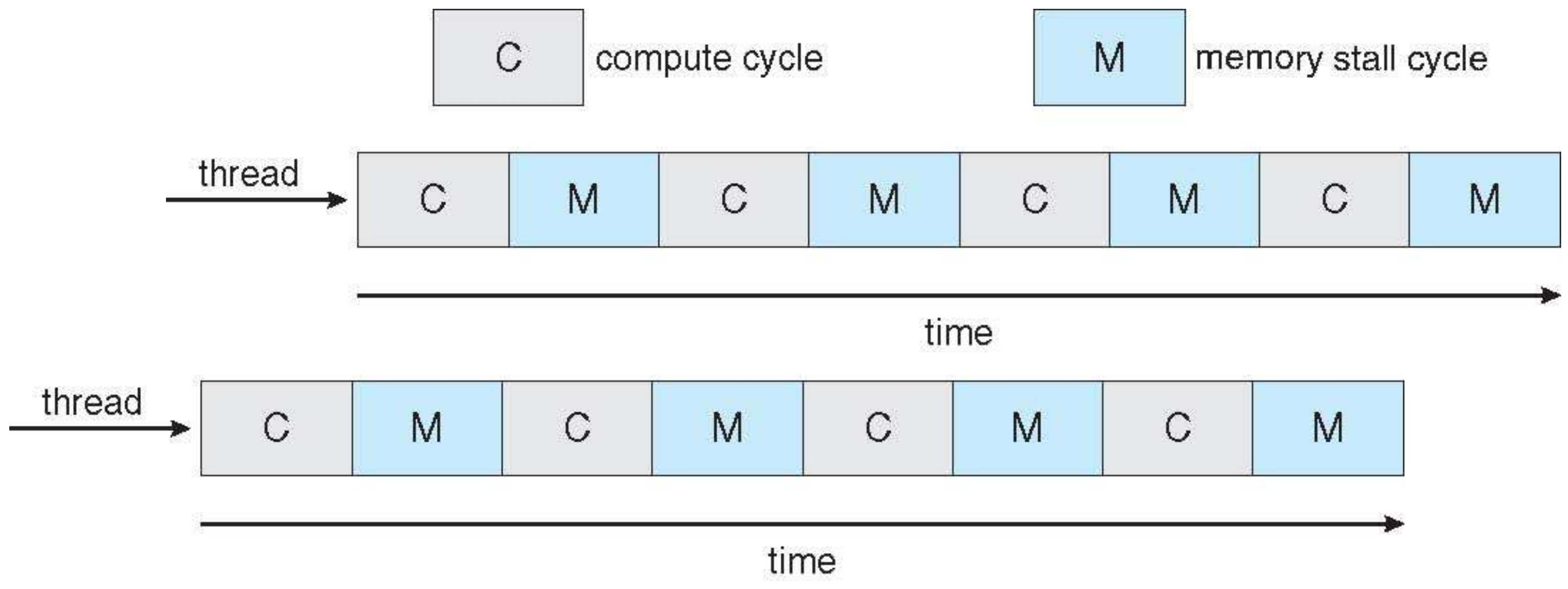
- Java: runs threads with a higher priority until they are blocked --Threads with a lower priority cannot run concurrently
  - ✓ This may cause starvation or deadlock.
- Java's strict priority scheduling was implemented in Java green threads.
- In Java 2, thread's priority is typically mapped to the underlying OS-native thread's priority. (Note we are at Java7 or 8 now)
- OS: gives more CPU time to threads with a higher priority. (Threads with a lower priority can still run concurrently.)
- If we do not want to depend on OS scheduling, we have to use:
  - ✓ Thread.suspend( )
  - ✓ Thread.resume( )

Multiprocessor Scheduling

- OS code and user process mapping:
  - Asymmetric multiprocessing
  - Symmetric multiprocessing (SMP)
- Scheduling criteria:
  - Processor affinity
  - Load balancing
- Thread context switch:
  - Coarse-grained software thread
  - Fine-grained hardware thread

Multicore Processors

- Multiple processor cores on same physical chip: Faster and consume less power
- Multiple Fine-grained hardware threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
  - vs. coarse-grained software thread



Algorithm Evaluation

- Define the selection criteria (i.e.):
  - Optimize CPU Utilization OR
  - Maximize throughput

- Evaluation Methods
  - Analytic
  - Deterministic Modeling
    - Queueing Models (see Little's formula next)
    - Simulations, with data from
      - Industry Standard Benchmarks
      - Real workloads
  - Implementation

Little's Formula

$$n = l ´ W$$

$$l = \frac{n}{W}$$

$$W = \frac{n}{l}$$

n	Average queue length
λ	Average arrival rate for new processes in queue
W	Average waiting time in queue

Little's Formula - Examples

Ex:	n	λ (processes/sec)	W (sec)	Narrative
1	256	1000		What's the average wait time for 1000 p/sec in a 256 element queue?
2		15000	0.1	How deep should a queue be for a desired wait time of 1/10 sec if processes arrive every 67usec?
3	64		3	If average wait time is 3 sec and queue depth is 64 what is the average throughput of processes that can be accommodated?

Exercises

- Programming Assignment 2:
  - Check the syllabus for its due date

- No-turn-in problems:
  1. Solve Exercise 5.5
  2. Solve Exercise 5.12
  3. Solve Exercise 5.14