Operating Systems

No. 4

ศรัณย์ อินทโกสุม

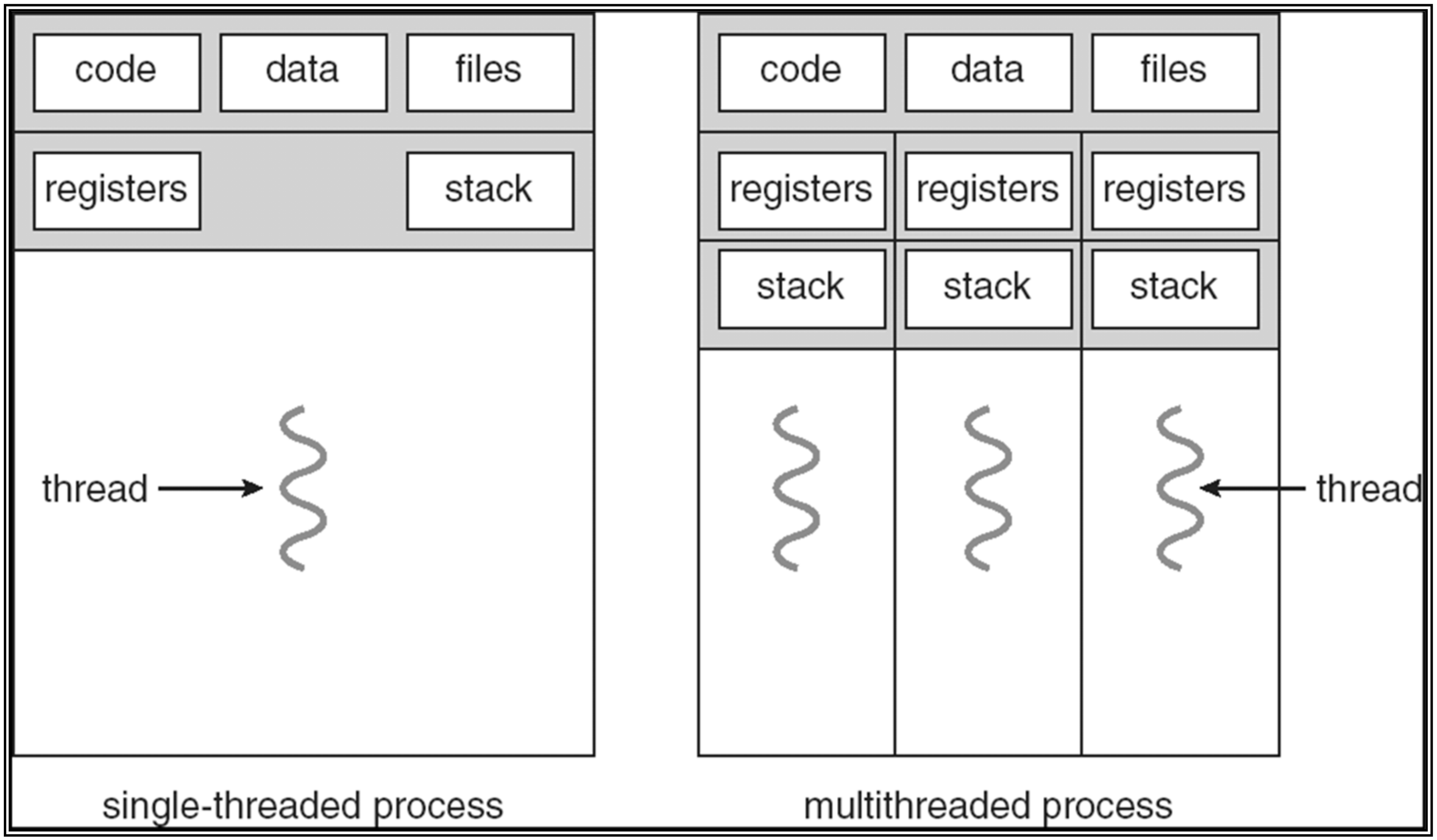
Sarun Intakosum

Thread

Threads Concepts

- **Thread:** A portion of a program that can run independently of other portions.
  - A thread is a flow of execution through the process's code with its own program counter, system registers, and stack.
  - The **heavyweight process** which owns the resources becomes a more passive element. Thread is also called **lightweight process.**
  - Thread becomes the element that uses the CPU and is scheduled for execution
  - Swapping threads is less time consuming than swapping processes
- Multithreaded applications programs can have several threads running at one time with the same or different priorities

Single and Multithreaded Processes



Categories of OS

- Single-process single-threaded
  - DOS
- Single-Process multi-threaded
  - JVM
- Multiprocessing single-threaded
  - UNIX
- Multiprocessing multi-threaded
  - Windows XP, Solaris

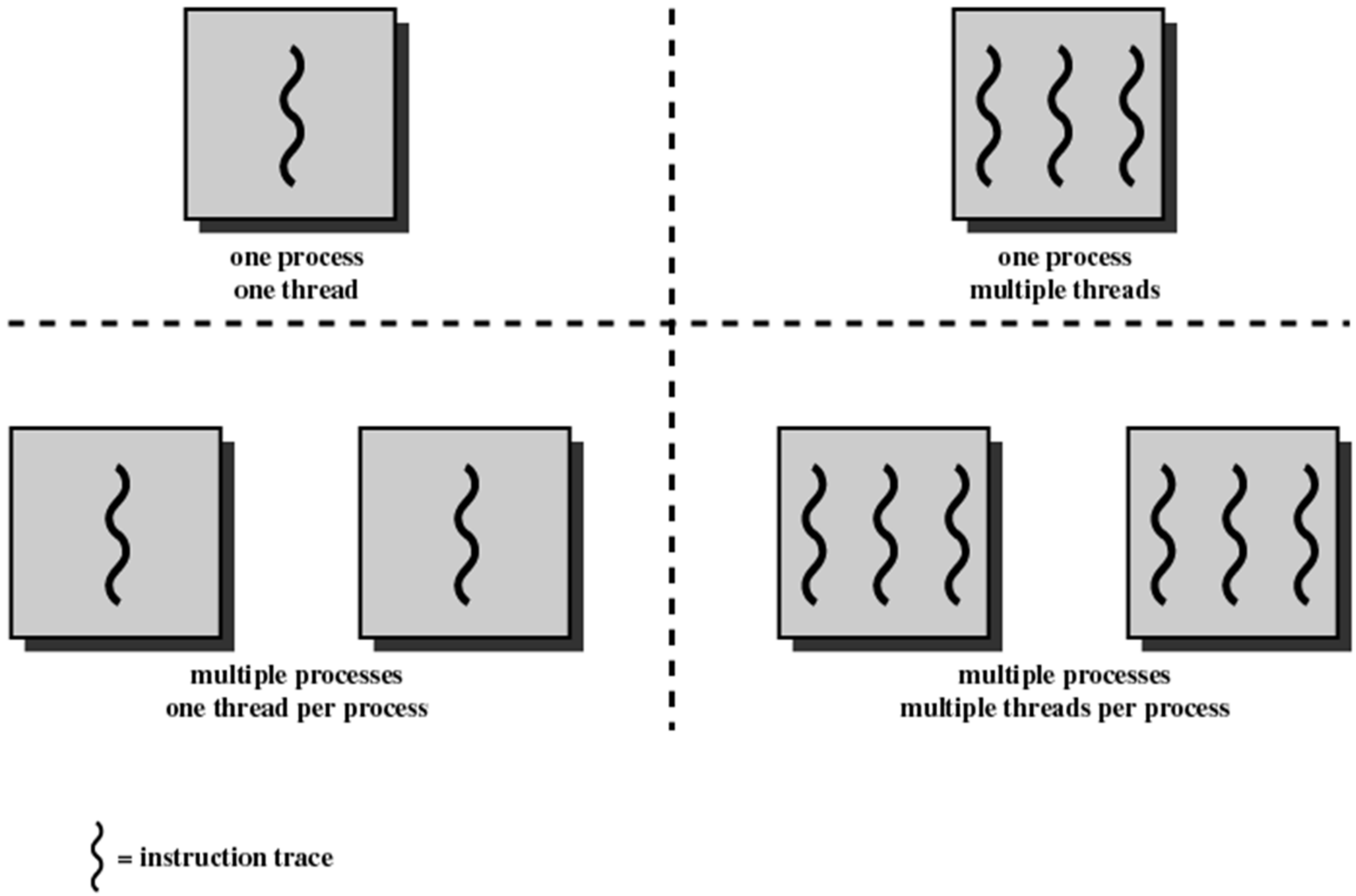
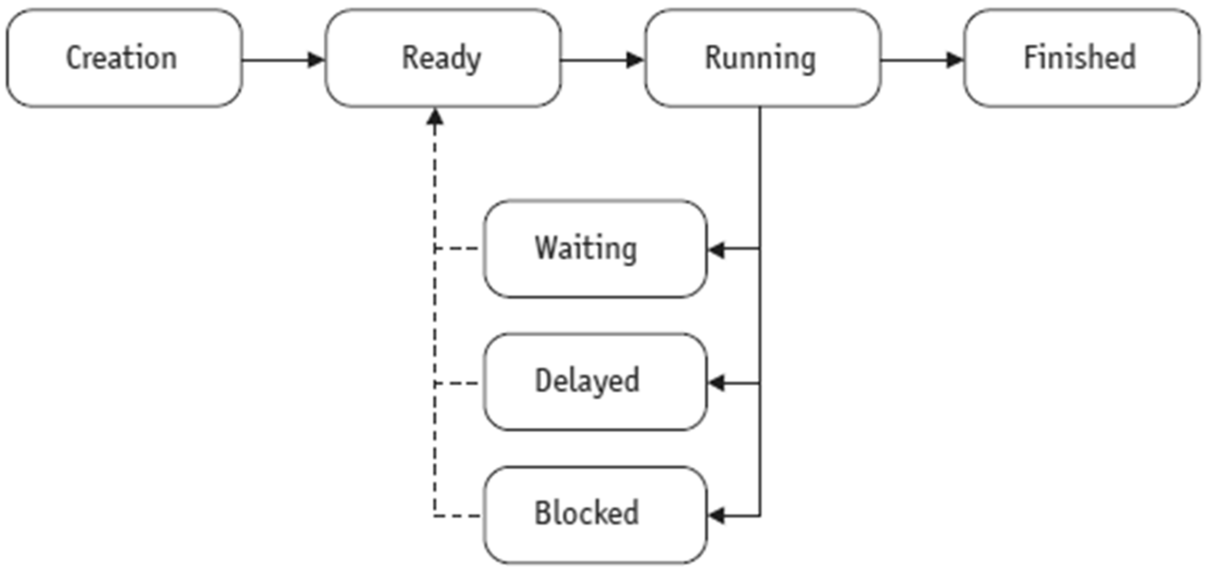


Figure 4.1 Threads and Processes [ANDE97]

Thread States



(figure 6.6)

A typical thread changes states from READY to FINISHED as it moves through the system.

Thread Control Block

(figure 6.7)

Comparison of a typical Thread Control Block (TCB) vs. a Process Control Block (PCB) from Chapter 4.

Thread Control Block	Process Control Block

Thread identification
Thread state
CPU information:
    Program counter
    Register contents
Thread priority
Pointer to process that created this thread
Pointers to all other threads created by this thread
	Process identification
	Process status
	Process state:
	    Process status word
	    Register contents
	    Main memory
	    Resources
	    Process priority
	Accounting

Types of Threads

There are two general types of threads:

- User-level threads (ULT)
- Kernel-level threads (KLT).

User Threads

- Thread management done by user-level threads library

- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads
    - Generally implemented using a thread library available on host OS.

- Using this library, the application invokes the appropriate functions for the various thread management tasks such as: creating, suspending, and terminating threads.

Kernel Threads

- Supported by the Kernel

- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

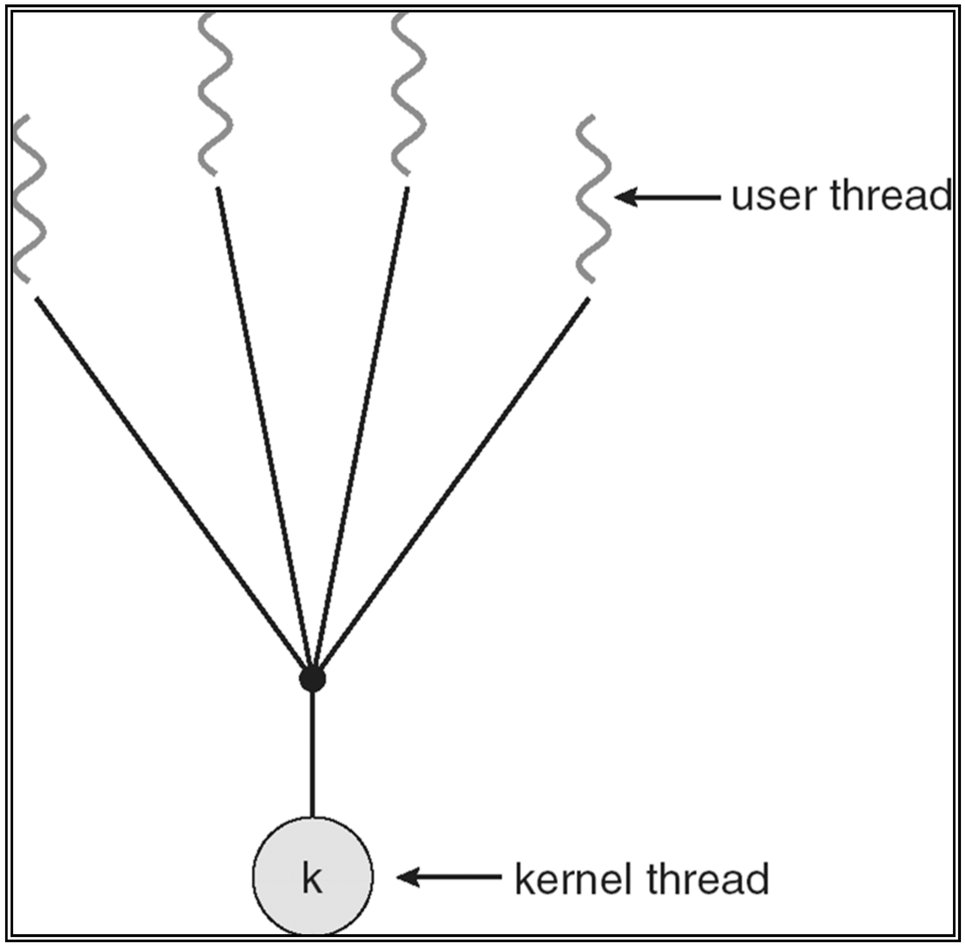
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

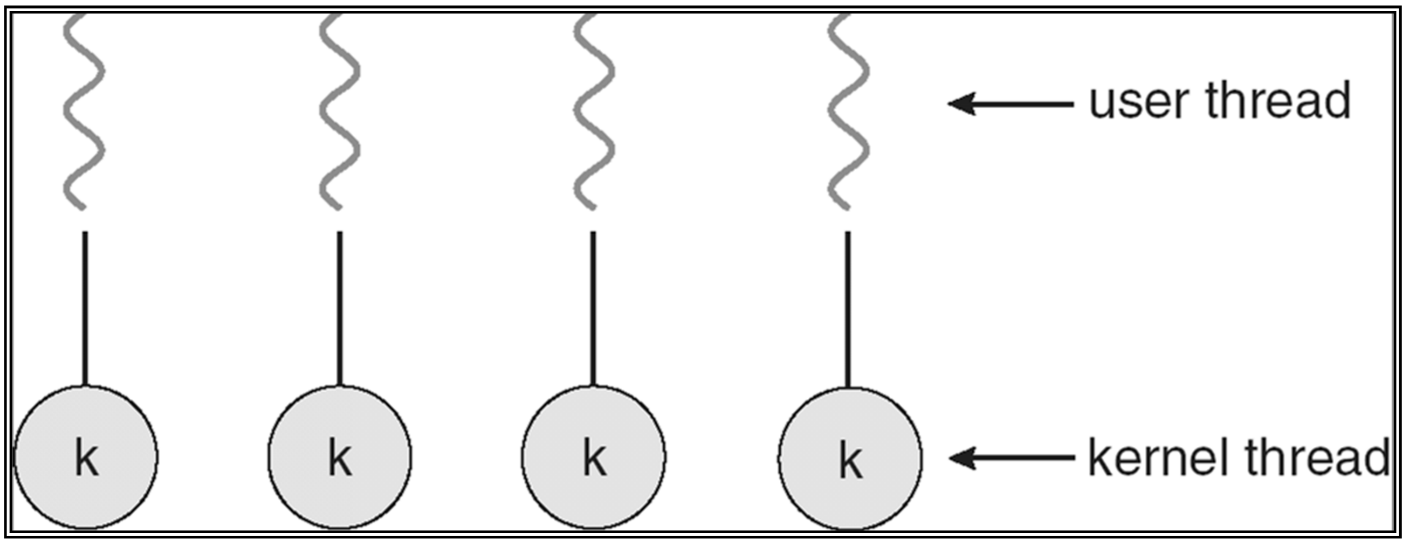
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

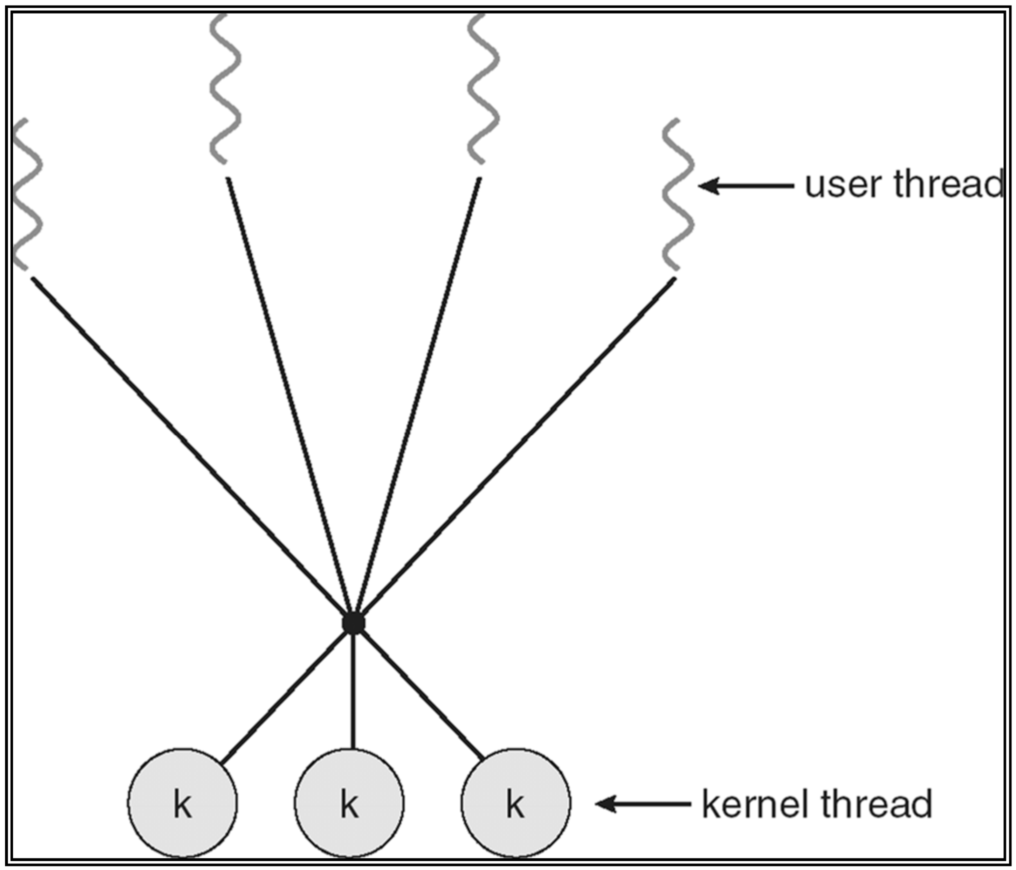
One-to-one Model



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

Many-to-Many Model



Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads

```
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */
main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of attributes for the thread */
  pthread_attr_init(&attr); /* get the default attributes */
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* now wait for the thread to exit */
  pthread_join(tid,NULL);
  printf("sum = %d\n",sum);
}
void *runner(void *param) {
  int upper = atoi(param);
  int i;
  sum = 0;
  if (upper > 0) {
    for (i = 1; i <= upper; i++)
      sum += i;
  }
  pthread_exit(0);
}
```

Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Extending the Thread Class

```
class Worker1 extends Thread
{
  public void run() {
    System.out.println("I Am a Worker Thread");
  }
}

public class First
{
  public static void main(String args[]) {
    Worker1 runner = new Worker1();
    runner.start();

    System.out.println("I Am The Main Thread");
  }
}
```

The Runnable Interface

```
public interface Runnable
{
  public abstract void run();
}
```

Implementing the Runnable Interface

```
class Worker2 implements Runnable
{
  public void run() {
    System.out.println("I Am a Worker Thread ");
  }
}
public class Second
{
  public static void main(String args[]) {
    Runnable runner = new Worker2();
    Thread thrd = new Thread(runner);
    thrd.start();

    System.out.println("I Am The Main Thread");
  }
}
```

Joining Threads

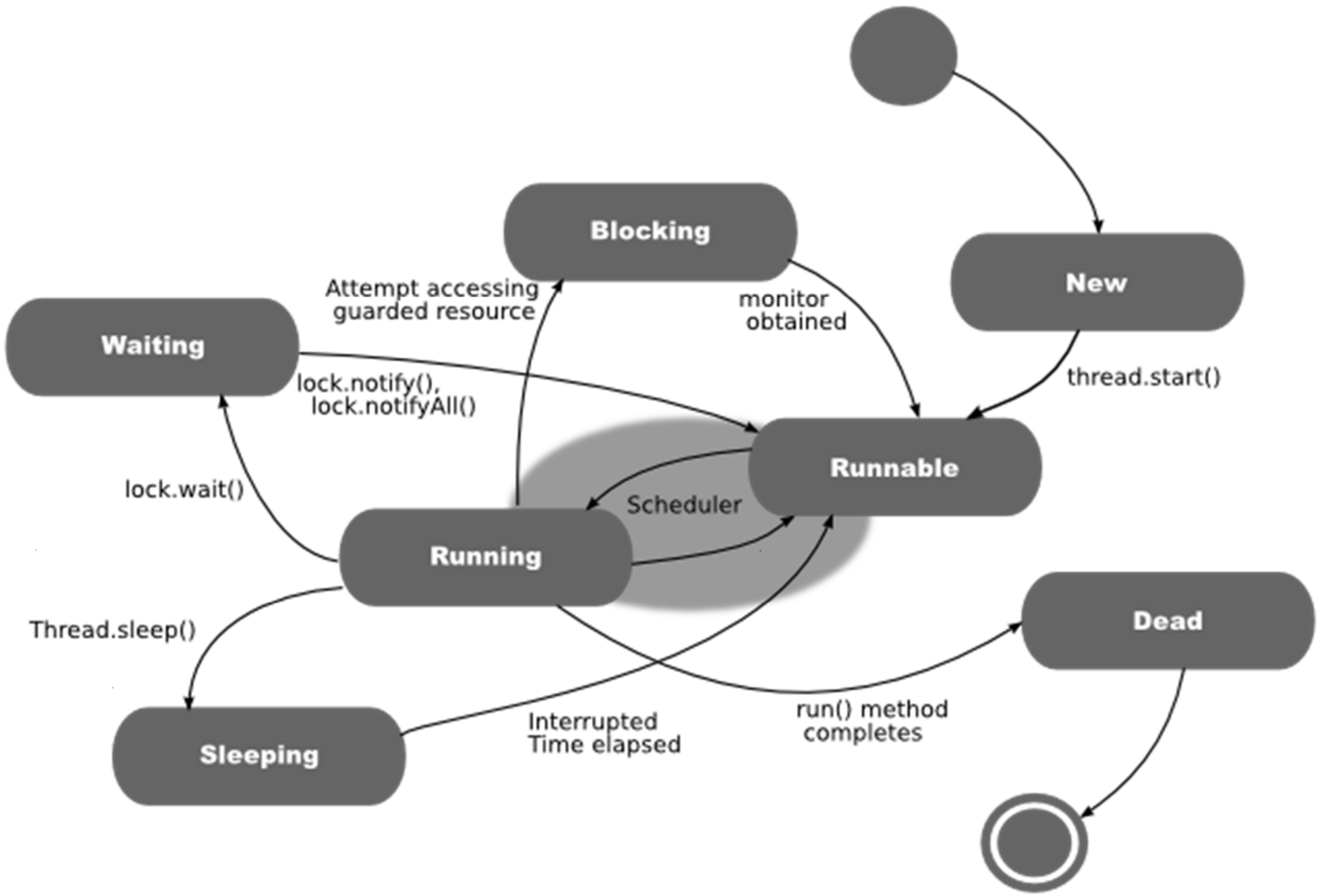
```
class JoinableWorker implements Runnable
{
  public void run() {
    System.out.println("Worker working");
  }
}

public class JoinExample
{
  public static void main(String[] args) {
    Thread task = new Thread(new JoinableWorker());
    task.start();

    try { task.join(); }
    catch (InterruptedException ie) { }

    System.out.println("Worker done");
  }
}
```

Java Thread States



fork(), exec() and Threads

- If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded?
  - Some UNIX systems have two versions of fork().
  - Which one is used depends on the application.
    - exec() is called immediately after fork().
    - exec() is not called immediately after fork().

Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Thread Cancellation

```
Thread thrd = new Thread (new InterruptibleThread());
Thrd.start();

. . .

// now interrupt it
Thrd.interrupt();
```

Thread Cancellation (Cont.)

```
public class InterruptibleThread implements Runnable
{
  public void run() {
    while (true) {
      /**
       * do some work for awhile
       */

      if (Thread.currentThread().isInterrupted()) {
        System.out.println("I'm interrupted!");
        break;
      }
    }

    // clean up and terminate
  }
}
```

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

Thread Pools

- Java provides 3 thread pool architectures:

  1. **Single thread executor** - pool of size 1.
     - `static ExecutorService newSingleThreadExecutor()`
  2. **Fixed thread executor** - pool of fixed size.
     - `static ExecutorService newFixedThreadPool(int nThreads)`
  3. **Cached thread pool** - pool of unbounded size
     - `static ExecutorService newCachedThreadPool()`

Thread Pools

A task to be serviced in a thread pool

- public class Task implements Runnable {
  public void run() {
  System.out.println("I am working on task");
  }
  }

Thread Pools

```
import java.util.concurrent.*;

public class TPExample {
  public static void main(String[] args) {
    int numTasks = Integer.parseInt(args[0].trim());
    ExecutorService pool =
        Executors.newCachedThreadPool();
    for(int i =0; i < numTasks; i++) {
      pool.execute(new Task());
    }
    pool.shutdown();
  }
}
```

CPU Scheduling

CPU Scheduler

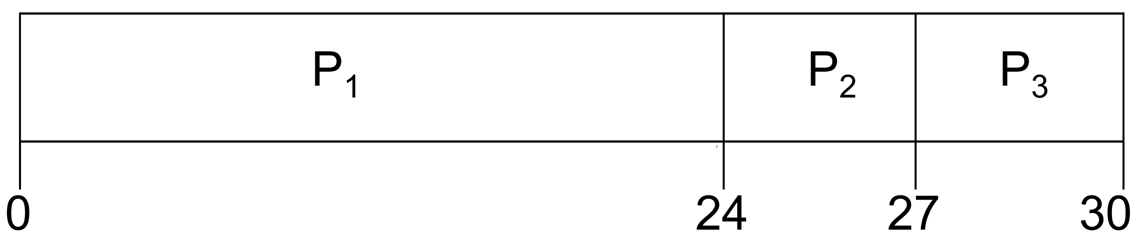
- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- **nonpreemptive**
- **preemptive**

Scheduling Algorithm Optimization Criteria

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

First-Come, First-Served (FCFS) Scheduling

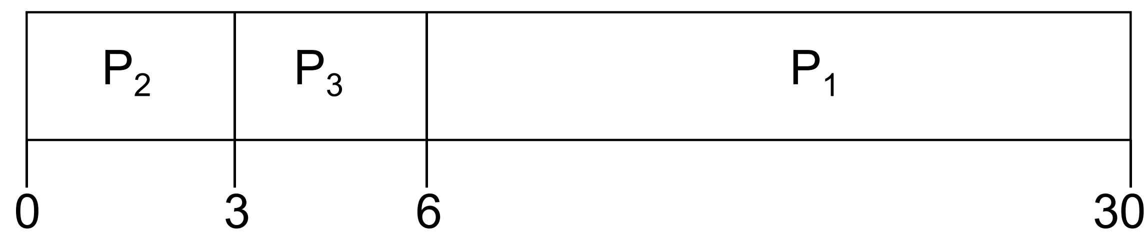
Process	Arrival Time	Burst Time
P_1	0.0	24
P_2	2.0	3
P_3	4.0	3



- Waiting time for P_1 = 0; P_2 = 22; P_3 = 23
- Average waiting time: (0 + 22 + 23)/3 = 15

FCFS Scheduling (Cont)

Process	Arrival Time	Burst Time
P_2	0.0	3
P_3	2.0	3
P_1	4.0	24



- Waiting time for $P_1 = 2; P_2 = 0; P_3 = 1$
- Average waiting time: (2 + 0 + 1)/3 = 1
- Much better than previous case
- *Convoy effect* short process behind long process

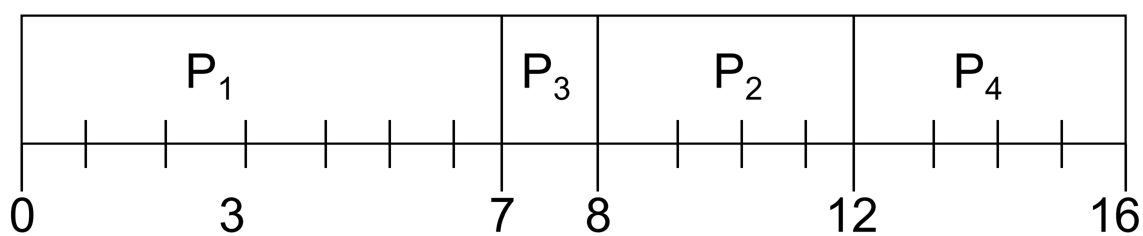
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request

Example of 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 scheduling chart



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

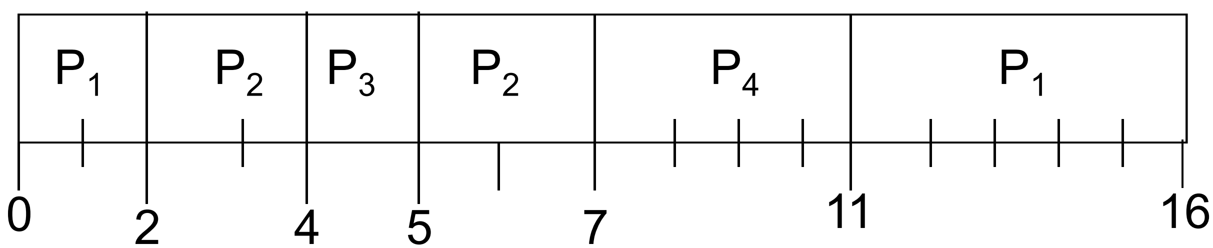
Shortest Remaining Time First (SRTF)

- Preemptive version of the SJF algorithm
- Processor allocated to job closest to completion
  - Current job can be preempted if newer job in READY queue has shorter time to completion
- Cannot be implemented in interactive system
  - Requires advance knowledge of the CPU time required to finish each job
- SRTF involves more overhead than SJF
  - OS monitors CPU time for all jobs in READY queue and performs context switching

Example of SRTF

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

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)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

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 $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Arrival Time	Burst Time
P_1	0.0	24
P_2	2.0	3
P_3	4.0	3

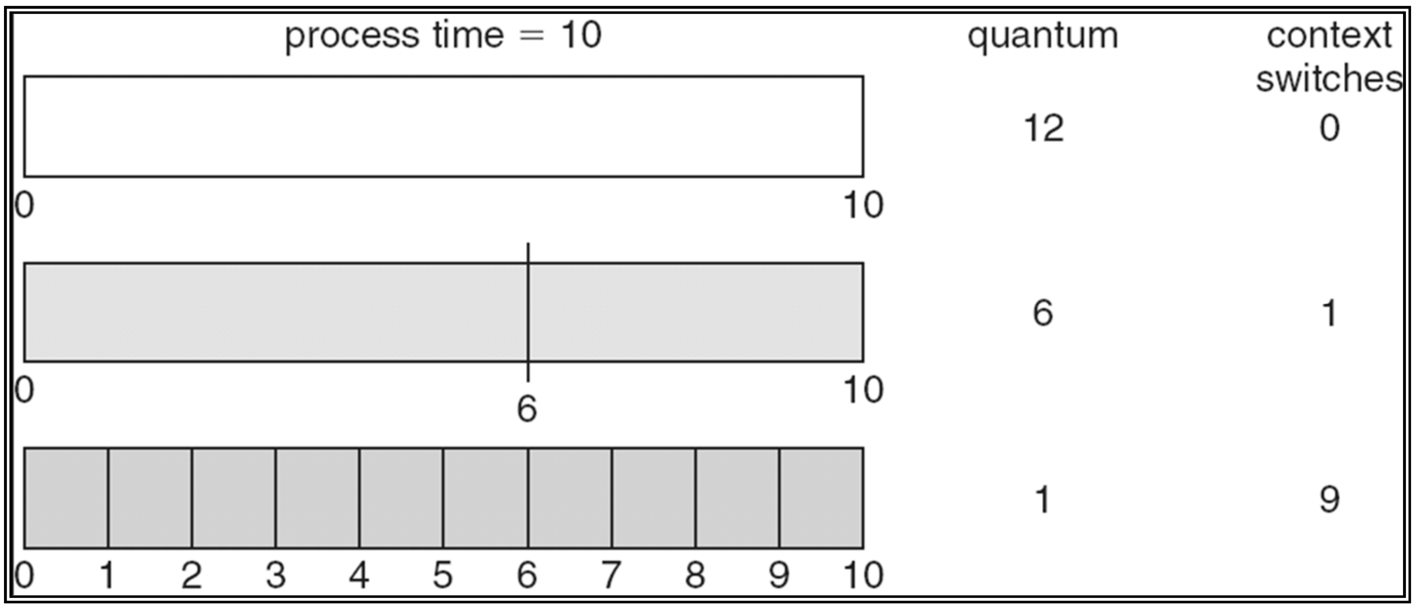
- The Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

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

Time Quantum and Context Switch Time



Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR 20% to background in FCFS

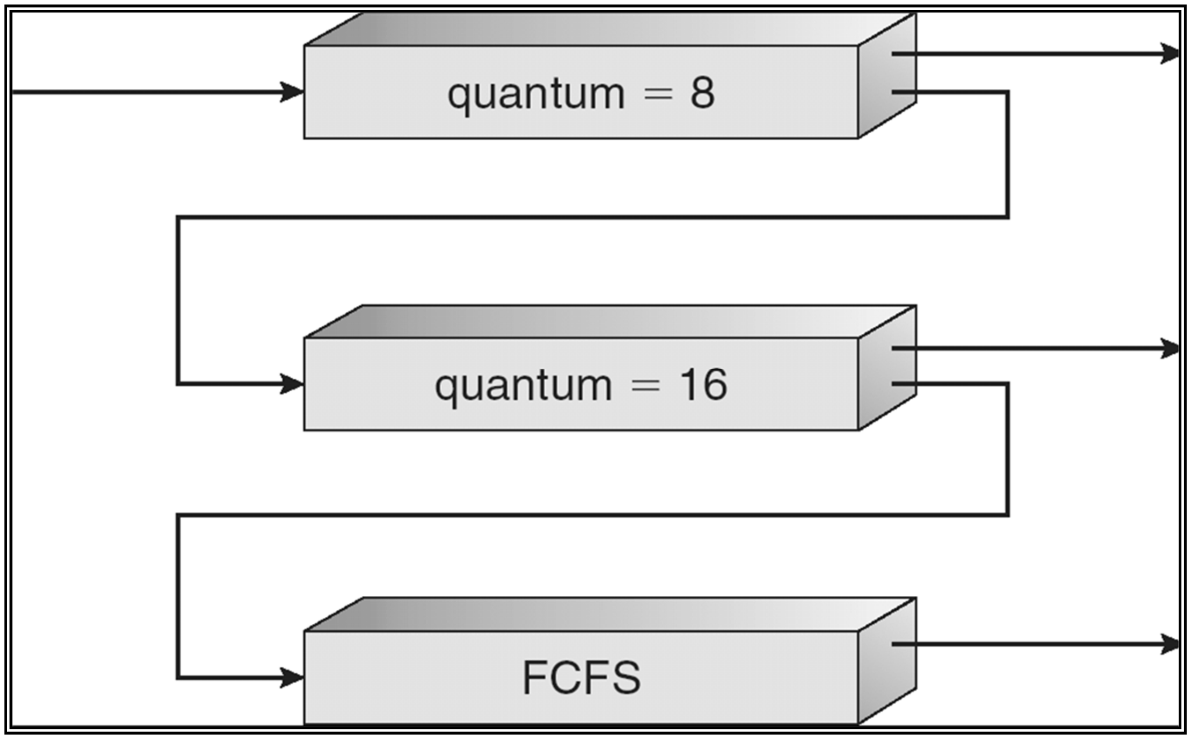
Multilevel Feedback Queue

- A process can move between the various queues; 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



Java Thread Scheduling

- Java specification does not specify the standard scheduling. It depends on the implementation of each JVM.
- Do not rely on the scheduling algorithm for the correctness of your program.

Java Threads Scheduling (Cont.)

- Normally, Java uses Preemptive and priority based scheduling algorithm.
- All Java threads have a priority. The thread with the highest priority is chosen first.
- If two threads of the same priority are waiting for the CPU, the scheduler arbitrarily chooses one of them to run.

Java Threads Scheduling (Cont.)

- The chosen thread runs until one of the following conditions is true:
  - A higher priority thread becomes runnable.
  - It yields, or its run method exits.
  - On systems that support time-slicing, its time allotment has expired.

Java Threads Scheduling (Cont.)

- Priority is in the range of 1-10.
- The default priority is 5.
- Method setPriority() is used to set a thread priority.
  - myThread.setPriority(6);
- A thread can yield control of the CPU using the yield() method.
  - Java can then choose another thread with the same priority to run.