A thread can be in one of the following states:

* [NEW](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.State.html#NEW)  
  A thread that has not yet started is in this state. No system resources have been allocated to this thread yet.
* [RUNNABLE](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.State.html#RUNNABLE)  
  A thread executing in the Java virtual machine is in this state. Resources have been allocated to the thread.
* [BLOCKED](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.State.html#BLOCKED)  
  A thread that is blocked waiting for a monitor lock is in this state.
* [WAITING](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.State.html#WAITING)  
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
* [TIMED\_WAITING](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.State.html#TIMED_WAITING)  
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
* [TERMINATED](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.State.html#TERMINATED)  
  A thread that has exited is in this state.

The start() method invokes the run() method

The time unit of the sleep method is miliseconds

**What are the advantages / reasons of using threads instead of processes?**

You'd prefer multiple threads over multiple processes for two reasons:

1. Inter-thread communication (sharing data etc.) is significantly simpler to program than inter-process communication.
2. Context switches between threads are faster than between processes. That is, it's quicker for the OS to stop one thread and start running another than do the same with two processes.

Two ways of providing the run( ) method for a thread:

1. ***Subclassing the thread class and overriding the run( ) method.***
2. **Class A extends Thread {**
3. **Public void run( ) {**
4. **//code**
5. **}**
6. **}**
7. ***Implementing the Runnable interface.***
8. **CSCI 340**
9. **Lecturer: Dr. Simina Fluture**
10. Revised 3/17/06
11. **Class B implements Runnable {**
12. **Public void run( ) {**
13. **//code**
14. **}**
15. **}**

Java uses a preemptive priority CPU scheduling algorithm.

The range of Thread priority in [java](http://www.javaranch.com/) is 1-10. The minimum priority is 1 and the maximum is 10. The default priority of any thread in java is 5.

For the *wait( )* and *sleep( )* methods, if the thread that is interrupted is blocked, the method that blocked

the thread throws an InterruptException object.

**First-Come First-Served (FCFS)**

The ready queue contains the PCB of the process and it is treated like a FIFO queue.

Newly created processes and those becoming ready are added to the rear of the queue. The next process

to be run is the one in front of the queue.

+) easy to understand and implement

-) the waiting time might be too long

possibility of the convoy effect

**Shortest-Job-First-Scheduling (SJF)**

Suppose that the service time is known in advance.

The processes in the ready list, at the time when the CPU becomes available, are sorted in increasing

order by the service time (next CPU burst) and scheduled for the CPU in that order.

The SJF may be either preemptive or non preemptive. A nonpreemptive algorithm will allow the currently

running process to finish its CPU burst.

**Example:**

+) the SJF is proved to be the optimal scheduling algorithm with respect to the *average waiting time* of

the ready processes.

SJF algorithm can be used for long term scheduling in a batch system by using as the length of the next

CPU request the process' time limit that a user specifies when the job is submited.

-) SJF may penalize processes with high service time requests - starvation

SJF cannot be implemented at the level of short-term CPU scheduling. It is hard to predict the length of

the next CPU burst.

One approach is to approximate SJF scheduling by predicting the value of the next CPU burst time.

**Priority scheduling algorithms** (6.3.3 from the book)

The processes are allocated to the CPU on the basis of their assigned priorities.

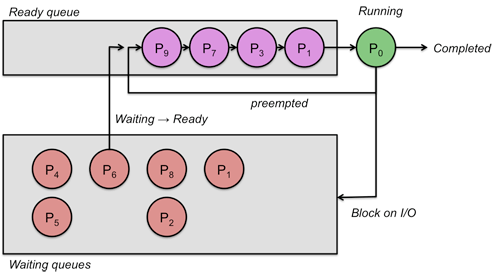
Priority scheduling can be either **preemptive** on **nonpreemptive**.

-) indefinite blocking or starvation.

Solution for the starvation of low-priority processes is *aging*.

**Round robin scheduling**

Round robin scheduling is a preemptive version of first-come, first-served scheduling. Processes are dispatched in a first-in-first-out sequence but each process is allowed to run for only a limited amount of time. This time interval is known as a time-slice or quantum. If a process does not complete or get blocked because of an I/O operation within the time slice, the time slice expires and the process is preempted. process gets blocked because of an I/O operation), it is then preempted. This preempted process is placed at the back of the ready list where it must wait for the processes that were already on the list to cycle through the CPU.



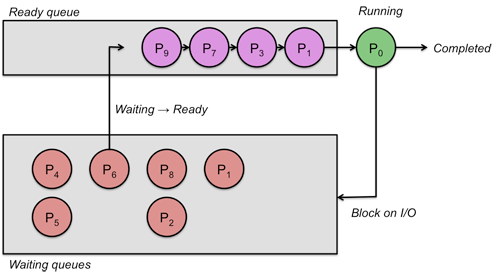
With round robin scheduling, interactive performance depends on the length of the quantum and the number of processes in the ready list (run queue). A very long quantum makes the algorithm behave very much like first come, first served scheduling since it's very likely that a process with block or complete before the time slice is up. A small quantum lets the system cycle through processes quickly. This is wonderful for interactive processes. Unfortunately, there is an overhead to context switching and having to do so frequently increases the percentage of system time that is used on context switching rather than real work.

Advantage: Round robin scheduling is fair in that every process gets an equal share of the CPU. It is easy to implement and, if we know the number of processes on the ready list, we can know the worst-case response time for a process.

Disadvantage: Giving every process an equal share of the CPU is not always a good idea. For instance, highly interactive processes will get scheduled no more frequently than CPU-bound processes.

**First-Come, First-Served Scheduling**

Possibly the most straightforward approach to scheduling processes is to maintain a FIFO (first-in, first-out) ready queue. New processes go to the end of the queue. When the scheduler needs to run a process, it picks the process that is at the head of the queue. The scheduler is non-preemptive. If the process has to block on I/O, it enters the *waiting* state and the scheduler picks the process from the head of the queue. When I/O is complete and the process is ready to run again, it gets put at the end of the queue.



With first-come, first-served scheduling, a process with a long CPU burst will hold up other processes. Moreover, it can hurt overall throughput since I/O on processes in the *waiting* state may complete while the CPU bound process is still running. Now devices are not being used effectively. For increasing throughput, it would have been great if the scheduler instead could have briefly run some I/O bound process that could request some I/O and wait. Because CPU bound processes don't get preempted, they hurt interactive performance because the interactive process won't get scheduled until the CPU bound one has completed.

Advantage: FIFO scheduling is simple to implement. It is also intuitively fair (the first one in line gets to run first).

Disadvantage: The greatest drawback of first-come, first-served scheduling is that it is not preemptive. Because of this, it is not suitable for interactive jobs. Another drawback is that a long-running process will delay all jobs behind it.

[**http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html**](http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html)

**Joins**

The join method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing,

t.join();

causes the current thread to pause execution until t's thread terminates. Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.

Like sleep, join responds to an interrupt by exiting with an InterruptedException.

Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system. The sleep method can also be used for pacing, as shown in the example that follows, and waiting for another thread with duties that are understood to have time requirements, as with the SimpleThreads example in a later section.

### Windows

In the Hotspot implementation, the way that Thread.yield() works has changed between Java 5 and Java 6.

In Java 5, Thread.yield() calls the Windows API call Sleep(0). This has the special effect of **clearing the current thread's quantum** and putting it to the **end of the queue for its priority level**. In other words, all runnable threads of the same priority (and those of greater priority) will get a chance to run before the yielded thread is next given CPU time. When it is eventually re-scheduled, it will come back with a full [full quantum](http://www.javamex.com/tutorials/threads/thread_scheduling.shtml#quantum), but doesn't "carry over" any of the remaining quantum from the time of yielding. This behaviour is a little different from a non-zero sleep where the sleeping thread generally loses 1 quantum value (in effect, 1/3 of a 10 or 15ms tick).

In Java 6, this behaviour was changed. The Hotspot VM now implements Thread.yield() using the Windows SwitchToThread() API call. This call makes the current thread **give up its current timeslice**, but not its entire quantum. This means that depending on the priorities of other threads, the yielding thread can be **scheduled back in one interrupt period later**. (See the section on [thread scheduling](http://www.javamex.com/tutorials/threads/thread_scheduling.shtml) for more information on timeslices.)

http://users.cis.fiu.edu/~ssood001/os/FinalReportOS.pdf