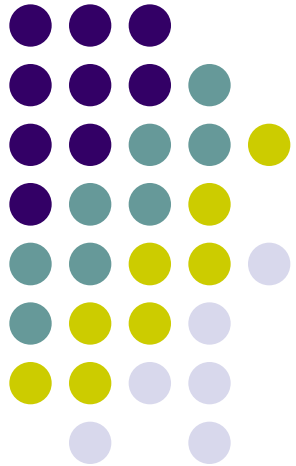
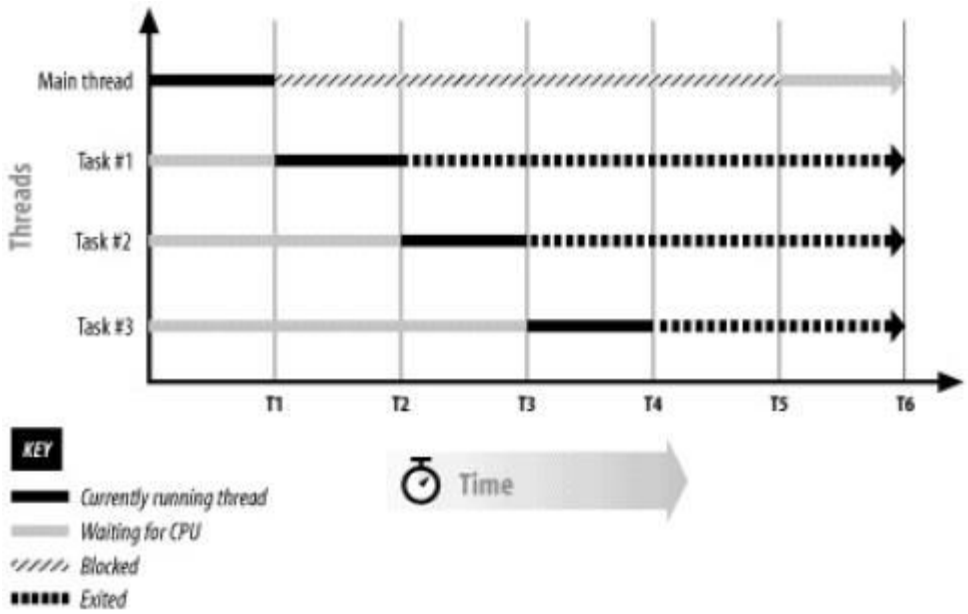
ITMC403 Parallel and Distributed Computing

Scheduling Threading Implementations



Priority-based and Preemptive Scheduling

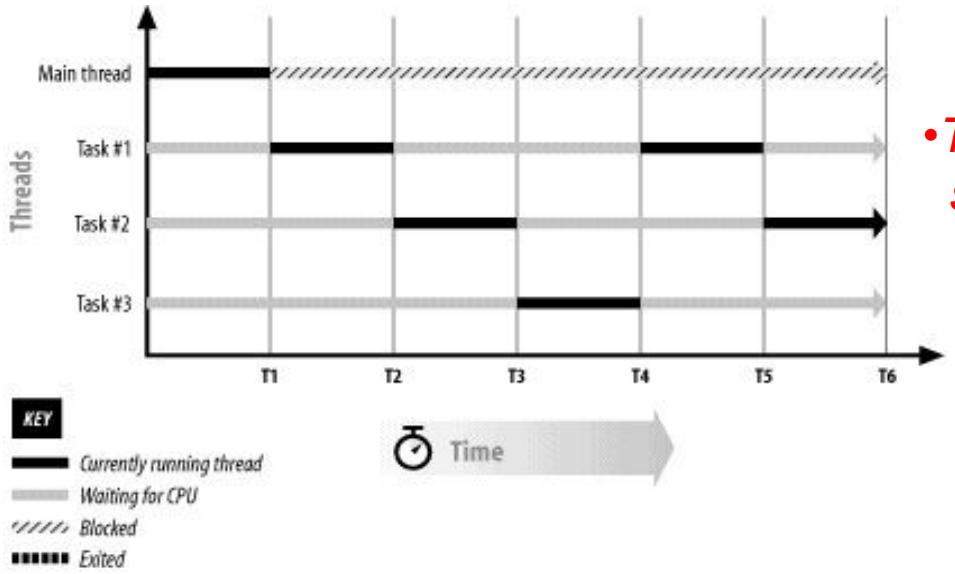


- *When we run the program for a second time:*
- *A simple thread-state diagram (proceeds sequentially)*

The main thread is the currently running thread until it blocks at time T1. At that point, one of the task threads becomes the currently running thread; it remains the currently running thread until time T2 when it finishes and transitions to the exiting state. Another task thread becomes the currently running thread, and the cycle continues until all threads have completed.

So why is the output different the first time we run the example?

Time-slicing Scheduling

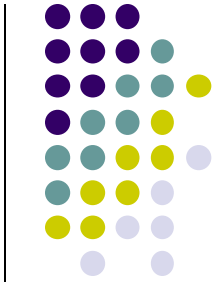


- *Thread states with OS scheduling*

The thread scheduler on that OS, in addition to being priority-based and preemptive, is also time-slicing.

That means when threads are waiting for the CPU, the operating system allows one of them to run for a very short time. It then interrupts that thread and allows a second thread to run for a very short time, and so on.

Priority Exceptions



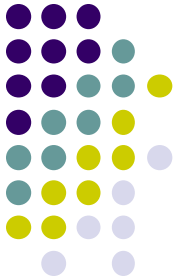
- When an operating system schedules Java threads, it may choose to run a lower-priority thread instead of a higher-priority thread in two instances.

1- Priority inversion

In a typical priority-based threading system, something unusual occurs when a thread attempts to acquire a lock that is held by a lower-priority thread: because the higher-priority thread becomes blocked, it temporarily runs with an effective priority of the lower-priority thread.

Priority Exceptions (Cont.)

Example of Priority inversion



A thread with a priority of 8 that wants to acquire a lock that is held by a thread with a priority of 2. Because the priority 8 thread is waiting for the priority 2 thread to release the lock, it ends up running with an Effective priority of 2. This is known as priority inversion.

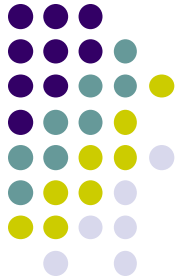
Solution: using priority inheritance

With priority inheritance, a thread that holds a lock that is wanted by a thread with a higher priority has its priority temporarily and silently raised: its new priority becomes the same as the priority of the thread that it is causing to block. When the thread releases the lock, its priority is lowered to its original value.

Note: The goal of priority inheritance is to allow the high-priority thread to run as soon as possible.

Priority Exceptions (Cont.)

2- Complex priorities



- It is a new priority as a result of combining of Java's and OS's priorities.
  - Java has11 priority levels (10 of which are available to developers).
  - The second case involves the priority assigned to threads by OS.

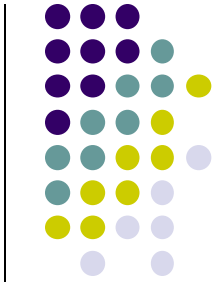
- OS usually have many more priorities. More important, though, is that the priority that the OS assigns to a thread is a complex formula that takes many pieces of information into account.

- A simple version of this formula might be this:

  RealPriority = JavaPriority + SecondsWaitingForCPU

Notes: Complex priorities are advantageous because they help to prevent thread starvation.

Scheduling Threading Implementations

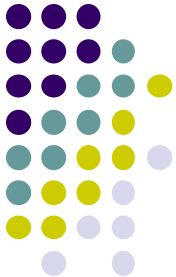


- **Green Threads method**
- Refers to a model in which the JVM creates, schedules and manages Java threads without the OS threads library. Green thread model used in Java 1.1 as it is faster to process.

- Supports the fact that the code is emulating many different threads is unknown outside of the virtual machine.

- The threads in this method are often called ***user-level threads*** because they exist only within the user level of the application: no calls into the OS are required to handle any thread details.
- It is platform independent.

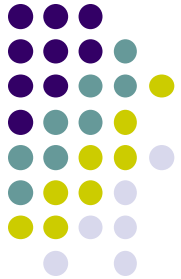
The green thread model is completely deterministic with respect to scheduling.



Starting task Task 5 at 07:23:12:074
Ending task Task 5 at 07:23:12:995 after 921 milliseconds
Starting task Task 4 at 07:23:13:111
Starting task Task 6 at 07:23:13:281
Ending task Task 6 at 07:23:14:256 after 975 milliseconds
Starting task Task 7 at 07:23:14:386
Ending task Task 7 at 07:23:15:398 after 1012 milliseconds
Starting task Task 8 at 07:23:15:504
Ending task Task 8 at 07:23:16:567 after 963 milliseconds
Starting task Task 9 at 07:23:16:624
Ending task Task 9 at 07:23:17:699 after 1075 milliseconds
Ending task Task 4 at 07:23:18:912 after 5801 milliseconds
Starting task Task 3 at 07:23:19:114
Ending task Task 3 at 07:23:20:177 after 1063 milliseconds
Starting task Task 2 at 07:23:20:301
Ending task Task 2 at 07:23:21:305 after 1004 milliseconds
Starting task Task 1 at 07:23:21:486
Ending task Task 1 at 07:23:22:449 after 963 milliseconds

Running our priority calculation above, we see this output:

Scheduling Threading Implementations (Cont.)

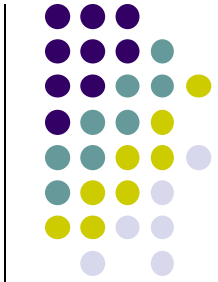


- **Native Threads method**
- Refers to a model in which the JVM creates, schedules and manages Java threads using OS threads library. Current java API uses Native threads model due to Green threads.

- There is one to one mapping between Java threads and OS threads.
- The OS scheduler makes no real distinction in this case between a **process** and **a thread**: it treats each thread like a process. Of course, there are still other differences in the OS between a thread and a process, but not as far as the scheduler is concerned.

- The threads in this method are often called ***System-level threads.***

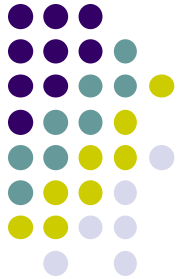
Scheduling Threading Implementations



Mapping of Java thread priorities on Win32 platforms

Java priority	Win32 priority
0	*THREAD_PRIORITY_IDLE*
1	*(Thread.MIN_PRIORITY) THREAD_PRIORITY_LOWEST*
2	*THREAD_PRIORITY_LOWEST*
3	*THREAD_PRIORITY_BELOW_NORMAL*
4	*THREAD_PRIORITY_BELOW_NORMAL*
5	*(Thread.NORM_PRIORITY) THREAD_PRIORITY_NORMAL*
6	*THREAD_PRIORITY_ABOVE_NORMAL*
7	*THREAD_PRIORITY_ABOVE_NORMAL*
8	*THREAD_PRIORITY_HIGHEST*
9	*THREAD_PRIORITY_HIGHEST*
10	*(Thread.MAX_PRIORITY) THREAD_PRIORITY_TIME_CRITICAL*

USER- AND SYSTEM-LEVEL THREADS (OS review)

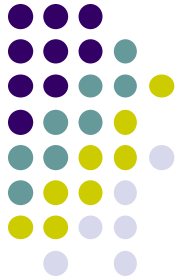


In most OSs, the OS is logically divided into two pieces:

user level and **system level**.

The OS itself — that is, the OS kernel — lies at the system level. The kernel is responsible for handling system calls on behalf of programs run at the user level. When a program running at user level wants to read a file; for example, it must call (or trap) into the OS kernel, which reads the file and returns the data to the program. This separation has many advantages, not the least of which is that it allows for a more robust system: if a program performs an illegal operation, it can be terminated without affecting other programs or the kernel itself. Only when the kernel executes an illegal operation does the entire machine crash. Because of this separation, it is possible to have support for threads at the **user level**, the **system level**, or at **both levels** independently.

Scheduling Threading Implementations (Cont.)



- **Native Threads complex priority calculation**
- Threads are subject to priority inheritance.
- The actual priority of a thread is based on its programmed (or inverted) priority minus a value that indicates how recently the thread has actually run. This value is subject to continual adjustment: the more time passes, the closer to zero that value becomes. This primarily distinguishes between threads of the same priority, and it leads to ***round-robin scheduling*** of threads of the same priority.
- On another level, a thread that has not run for a very long time is given a temporary priority boost. The value of this boost decays over time as the thread has a chance to run. This prevents threads from absolute starvation while still giving preference to higher-priority threads over lower-priority threads. The effect of this priority boost depends on the original priority of the thread. Threads running in a program that has keyboard and mouse focus are given a priority boost over threads in other programs.

What is round-robin scheduling?

- **Answer:**

It is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way. It is basically the preemptive scheduling of First come First Serve CPU Scheduling algorithm.