USER LEVEL SCHEDULING

Submitted to

Ahmedabad University

BY

DHARIT PARIKH 121010
DHRUVI PANDYA 121011
HEMIL MODI 121016

UNDER THE GUIDANCE OF
PROF. HEMAL SHAH

INFORMATION AND COMMUNICATION TECHNOLOGY
INSTITUTE OF ENGINEERING AND TECHNOLOGY

2014-2015

Acknowledgements

We are profoundly grateful to **Prof. HEMAL SHAH** for his expert guidance and continuous encouragement throughout to see that this project rights its target since its commencement to its completion.

We would like to express deepest appreciation towards INSTITUTE OF ENGINEERING AND TECHNOLOGY. **Ms. POORNIMA SHAH**, Project Coordinator whose invaluable guidance supported us in completing this project.

At last we must express our sincere heartfelt gratitude to all the staff members of Computer Engineering Department who helped me directly or indirectly during this course of work.

DHRUVI PANDYA
DHARIT PARIKH
HEMIL MODI

Contents

Chapter 1

Threads

1.1 What are threads?

A thread is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers and a thread ID. A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism. Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others. This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.

For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.

1.2 Process vs Threads

Few of the basic differences between thread and process are:

1. that if one process is blocked then no other process can execute until the first process is unblocked While in the case of threads, if one thread is blocked and waiting, second thread in the same task can run.
2. Multiple processes without using threads use more resources. While multiple threaded processes use fewer resources.
3. Process switching needs interaction with operating system. While thread switching does not need to interact with operating system.

4. In multiple processes each process operates independently of the others. While One thread can read, write or change another thread's data.

.

1.3 Advantages of threads

1. Within a process use of threads provide concurrency.
2. More economical generate and context switch then in the case of process.
3. Using threads you can maximize the usage of multicore processor.

1.4 Types of threads

Threads are created in the following two ways:

1. User Level Thread: The threads that are managed by user(application programmer)
2. Kernel Level Thread: Operating System managed threads acting on kernel, an operating system core.

1.4.1 User level threads

These are the threads which are managed by the application, the Operating system is not aware about such threads. These threads are mapped to the kernel threads. One by one using some scheduling algorithm these threads are passed on to kernel level thread and kernel thread does it's job. Thus, in the process operating system treats each thread as a process but in fact the process is a collection of multiple user level threads which are mapped onto kernel level thread one after the other.
Advantages of User level threads:

1. These type of threads are not operating system specific. they can run on any operating system.
2. These threads can be created and managed very easily by the application programmer.
3. These threads are application specific. Thus, different application can have different number of threads. Also, the scheduling algorithm can be different for an application.

4. Thread switching does not require kernel mode privilege.

1.4.2 Kernel Level Threads

In this case, thread management done by the Kernel. Code for the thread management resides in the kernel. Operating system directly supports the kernel level threads. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.
Advantages of kernel level threads:

1. If any one thread in a process is blocked, another thread of the same process can be schedule by kernel

2. kernel can simultaneously schedule multiple threads from the same process on multiple processes.

Chapter 2

Scheduling

2.1 User level thread scheduling

Simple and a very basic threading approach towards making multithreaded system is that we map each user level thread with each kernel level thread. In this case the scheduling of kernel thread will be managed by operating system which in-turn will schedule user level thread which are mapped on those kernel level threads.

Now, We take this solution to a step further by mapping two user level threads to one kernel level threads. Here in this case you will require some scheduling algorithm using which user level thread will be mapped on to kernel level thread. On each yield() call user thread saves registers, switch stacks just like kernel does. Operating system schedules the one kernel thread but the two user level thread are to be scheduled by us.

Now, we extend our idea of mapping two user level thread to a kernel thread.The OS kernel has no knowledge of user-level threads. From its perspective, a process is an opaque black box that occasionally makes system calls. Consequently, if that program has 100,000 user-level threads but only one kernel thread, then the process can only run one user-level thread at a time because there is only one kernel-level thread associated with it. On the other hand, if a process has multiple kernel-level threads, then it can execute multiple commands in parallel if there is a multicore machine.

2.2 Why bother?

1. Context switching overheads:
   Assuming the example of two thread and one CPU. If I can run the context switching code locally (avoiding trap overheads,etc), my threads get to run slightly longer! with few changes Stack switching code works in user space rather than switching at kernel level.

2. Finer-Grained Scheduling Control: Consider an example, were there are threads. Thread 1 has got a lock while thread 2 is waiting for a lock. Now, let's say quantum of thread 1 has expired and thread 2 is just spinning until it's quantum expires. This is the time when processor remains idle. So, in such scenario wouldn't it be nice if thread 2's quantum is given to thread 1 so that the processor does not remain idle.

3. Blocking I/O: Again Let's say we have two threads they each get half of the applications quantum. If A blocks on I/O and B is using the CPU.B gets half the CPU time. As quantum is lost.

2.3 Solution

A common compromise between these is to have a program request some fixed number of kernel-level threads, then have its own thread scheduler divvy up the user-level threads onto these kernel-level threads as appropriate. That way, multiple ULTs can execute in parallel, and the program can have fine-grained control over how threads execute.

As for how this mapping works - there are a bunch of different schemes. You could imagine that the user program uses any one of multiple different scheduling systems. In fact, if you do this substitution:
Kernel thread $<---->$ Processor core
User thread $<---->$ Kernel thread
Then in any scheme the OS could use to map kernel threads onto cores could also be used to map user-level threads onto kernel-level threads.

Combination of ULT's and KLT's

To understand the importance (what Wikipedia labels hybrid threading), consider the following examples:

Consider a multi-threaded program (multiple KLT's) where there are more KLT's than available logical cores. In order to efficiently use every core, as you mentioned, you want the scheduler to switch out KLT's that are blocking with ones that in a ready state and not blocking. This ensures the core is reducing its amount of idle time. Unfortunately, switching KLT's is expensive for the scheduler and it consumes a relatively large amount of CPU time.

This is one area where hybrid threading can be helpful. Consider a multi-threaded program with multiple KLT's and ULT's. only one ULT can be running at one time for each KLT. If a ULT is blocking, we still want to switch it out for one which is not blocking. Fortunately, ULT's are much more lightweight than KLT's, in the sense that there less resources assigned to a ULT and they require no interaction with the kernel scheduler. Essentially, it is almost always quicker to switch out ULT's than it is to switch out KLT's. As a result, we are able to significantly reduce a cores idle time relative to the first example.

Now, of course, all of this depends on the threading library being used for implementing ULT's. There are two ways (which I can come up with) for "mapping" ULT's to KLT's.

1. A collection of ULT's for all KLT's:
   This situation is ideal on a shared memory system. There is essentially a "pool" of ULT's to which each KLT has access. Ideally, the threading library scheduler would assign ULT's to each KLT upon request as opposed to the KLT's accessing the pool individually. The later could cause race conditions or deadlocks if not implemented with locks or something similar.

2. A collection of ULT's for each KLT (Qthreads):
   This situation is ideal on a distributed memory system. Each KLT would have a collection of ULT's to run. The draw back is that the user (or the threading library) would have to divide the ULT's between the KLT's. This could result in load imbalance since it is not guaranteed that all ULT's will have the same

amount of work to complete and complete roughly the same amount of time. The solution to this is allowing for ULT migration; that is, migrating ULT's between KLT's.

Chapter 3

How JAVA manages threads

3.1 API/Tools available in JAVA

ALL Java programs use Threads - even ”common” single-threaded ones. The creation of new Threads requires Objects that implement the Runnable Interface, which means they contain a method ”public void run( )” . Any descendant of the Thread class will naturally contain such a method. (In practice the run( ) method must be overridden / provided for the thread to have any practical functionality.) Creating a Thread Object does not start the thread running - To do that the program must call the Thread’s ”start( )” method. Start( ) allocates and initializes memory for the Thread, and then calls the run( ) method. (Programmers do not call run( ) directly.) Because Java does not support global variables, Threads must be passed a reference to a shared Object in order to share data, in this example the ”Sum” Object. Note that the JVM runs on top of a native OS, and that the JVM specification does not specify what model to use for mapping Java threads to kernel threads. This decision is JVM implementation dependant, and may be one-to-one, many-to-many, or many to one.. (On a UNIX system the JVM normally uses PThreads and on a Windows system it normally uses windows threads.)

Chapter 4

Implementation

4.1 About implementation

So, far we have implemented User Level Threads in the form of tasks of Producer Consumer Problem. The Language of implementation is java. We have not used the inbuilt libraries of Java. Rather, we were not able to use them as the implementation required some of functionality and access, which was not provided in inbuilt libraries.Our Package name is ULThreads.It has in total of 7 java files.

1. MyThread.java:
   It is our main class that creates thread for us.It is an extended Thread class, which has other private datas like the Thread name and the link to the ThreadQueue. As, it is an extended(inherited) class, it cause the methods of it superclass as well. MyThread has methods like, makenewThread(),getMyThreadId(), MyThreadStart(),MyThreadSuspend(), MyThreadResume() and MyThreadDelay().

2. ThreadQueue.java:
   It is our main Queue class for threads.This class takes MyThreads as an input and create a queue for it. It has functions like queuee(), dqueue(), dequeueSuccessful() and getEnd(). This class was designed keeping in mind, the application as Producer Consumer Problem.

3. MyConsumer.java and MyProducer.java:
   They are the class files which implements Runnables, hence this files have run() functions in them which gets executed by a thread.Hence, in the Main.java file, there are threads which asynchronously gets executed, as a result, the produce and consume task gets asynchronously executed.

4. inputBuffer.java:
   This class has the Buffer required for the Producer and Consumer code and the methods for adding to eating from the buffer, for the producers and the consumers.This class writes in a file, the logs of exchange occurring on the inputBuffer, along with the time-stamp to compare it with the activities of the scheduler.
   Example: adding to buffer: 14 Status:14 2015-04-23 11:43:34.0000000105

5. Main.java :
   It is the file which starts the execution as it contains the main thread.It creates the objects to the other class like inputbuffer,MyProducer, MyConsumer and ThreadQueue.It also creates Threads to asynchronously execute Producer, Consumer and the Scheduler.And at the end it calls the Schedulers to schedule the MyThreads which are stored in the ThreadQueue.

6. FCFS.java, RoundRobbin.cpp and ShortestProcessNext.cpp:
   We have implemented 3 Scheduling Policies, which act upon the ThreadQueue i.e. it takes the MyThread object stored in the ThreadQueue according to the algorithm and then it starts the MyThread. As a result, it goes off to the respective runnable i.e. either producer or Consumer and run it.The scheduler files also write its dispatch in the file with a time-stamp and the task it is doing which is producer or consumer along with the amount to be added or consumed.
   Example: Producer 14 2015-04-23 11:43:34.0000000080

References

[1] http://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

[2] www.tutorialspoint.com/operating_system/os_multi_threading.htm

[3] http://www3.cs.stonybrook.edu/~porter/courses/cse506/f11/slides/user-threading.pdf

[4] http://www.cs.utexas.edu/users/dahlin/Classes/UGOS/labs/labULT/proj-ULT.html

[5] https://www.udacity.com/wiki/ud702/project-1-user-level-threads

[6] http://tutorials.jenkov.com/java-concurrency/index.html

[7] http://www.javatpoint.com/multithreading-in-java