Lecture 10

Benefits of Threads

* Higher throughput, though in some pathetic scenarios it is possible to have the overhead of context switching among threads steal away any throughput gains and result in worse performance than a single-threaded scenario. However such cases are unlikely and an exception, rather than the norm.
* Responsive applications that give the illusion of multi-tasking.
* Efficient utilization of resources. Note that thread creation is light-weight in comparison to spawning a brand new process. Web servers that use threads instead of creating new processes when fielding web requests, consume far fewer resources.

Problems with Threads

However, as it is said, there's no free lunch in life. The premium for using threads manifests in the following forms:

* Usually very hard to find bugs, some that may only rear head in production environments
* Higher cost of code maintenance since the code inherently becomes harder to reason about
* Increased utilization of system resources. Creation of each thread consumes additional memory, CPU cycles for book-keeping and waste of time in context switches.
* Programs may experience slowdown as coordination amongst threads comes at a price. Acquiring and releasing locks adds to program execution time. Threads fighting over acquiring locks cause lock contention.

Program

A program is a set of instructions and associated data that resides on the disk and is loaded by the operating system to perform some task. An executable file or a python script file are examples of programs. In order to run a program, the operating system's kernel is first asked to create a new process, which is an environment in which a program executes.

Process

A process is a program in execution. A process is an execution environment that consists of instructions, user-data, and system-data segments, as well as lots of other resources such as CPU, memory, address-space, disk and network I/O acquired at runtime. A program can have several copies of it running at the same time but a process necessarily belongs to only one program.

Thread

Thread is the smallest unit of execution in a process. A thread simply executes instructions serially. A process can have multiple threads running as part of it. Usually, there would be some state associated with the process that is shared among all the threads and in turn each thread would have some state private to itself. The globally shared state amongst the threads of a process is visible and accessible to all the threads, and special attention needs to be paid when any thread tries to read or write to this global shared state.

Serial Execution

When programs are serially executed, they are scheduled one at a time on the CPU. Once a task gets completed, the next one gets a chance to run. Each task is run from the beginning to the end without interruption. The analogy for serial execution is a circus juggler who can only juggle one ball at a time. Definitely not very entertaining!

Concurrency

A concurrent program is one that can be decomposed into constituent parts and each part can be executed out of order or in partial order without affecting the final outcome. A system capable of running several distinct programs or more than one independent unit of the same program in overlapping time intervals is called a concurrent system. The execution of two programs or units of the same program may not happen simultaneously.

A concurrent system can have two programs in progress at the same time where progress doesn't imply execution. One program can be suspended while the other executes. Both programs are able to make progress as their execution is interleaved. In concurrent systems, the goal is to maximize throughput and minimize latency. For example, a browser running on a single core machine has to be responsive to user clicks but also be able to render HTML on screen as quickly as possible. Concurrent systems achieve lower latency and higher throughput when programs running on the system require frequent network or disk I/O.

The classic example of a concurrent system is that of an operating system running on a single core machine. Such an operating system is concurrent but not parallel. It can only process one task at any given point in time but all the tasks being managed by the operating system appear to make progress because the operating system is designed for concurrency. Each task gets a slice of the CPU time to execute and move forward.

Going back to our circus analogy, a concurrent juggler is one who can juggle several balls at the same time. However, at any one point in time, he can only have a single ball in his hand while the rest are in flight. Each ball gets a time slice during which it lands in the juggler's hand and then is thrown back up. A concurrent system is in a similar sense juggling several processes at the same time.

Parallelism

A parallel system is one which necessarily has the ability to execute multiple programs at the same time. Usually, this capability is aided by hardware in the form of multicore processors on individual machines or as computing clusters where several machines are hooked up to solve independent pieces of a problem simultaneously. Remember an individual problem has to be concurrent in nature, that is portions of it can be worked on independently without affecting the final outcome before it can be executed in parallel.

In parallel systems the emphasis is on increasing throughput and optimizing usage of hardware resources. The goal is to extract out as much computation speedup as possible. Example problems include matrix multiplication, 3D rendering, data analysis, and particle simulation.

Revisiting our juggler analogy, a parallel system would map to at least two or more jugglers juggling one or more balls. In the case of an operating system, if it runs on a machine with say four CPUs then the operating system can execute four tasks at the same time, making execution parallel. Either a single (large) problem can be executed in parallel or distinct programs can be executed in parallel on a system supporting parallel execution.

Concurrency vs Parallelism

From the above discussion it should be apparent that a concurrent system need not be parallel, whereas a parallel system is indeed concurrent. Additionally, a system can be both concurrent and parallel e.g. a multitasking operating system running on a multicore machine.

Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once. Last but not the least, you'll find literature describing concurrency as a property of a program or a system whereas parallelism as a runtime behaviour of executing multiple tasks.

Preemptive Multitasking

In preemptive multitasking, the operating system preempts a program to allow another waiting task to run on the CPU. Programs or threads can't decide how long for or when they can use the CPU. The operating system’s scheduler decides which thread or program gets to use the CPU next and for how much time. Furthermore, scheduling of programs or threads on the CPU isn’t predictable. A thread or program once taken off of the CPU by the scheduler can't determine when it will get on the CPU next. As a consequence, if a malicious program initiates an infinite loop, it only hurts itself without affecting other programs or threads. Lastly, the programmer isn't burdened to decide when to give up control back to the CPU in code.

Cooperative Multitasking

Cooperative Multitasking involves well-behaved programs to voluntarily give up control back to the scheduler so that another program can run. A program or thread may give up control after a period of time has expired or if it becomes idle or logically blocked. The operating system’s scheduler has no say in how long a program or thread runs for. A malicious program can bring the entire system to a halt by busy waiting or running an infinite loop and not giving up control. The process scheduler for an operating system implementing cooperative multitasking is called a cooperative scheduler. As the name implies, the participating programs or threads are required to cooperate to make the scheduling scheme work.

Synchronous

Synchronous execution refers to line-by-line execution of code. If a function is invoked, the program execution waits until the function call is completed. Synchronous execution blocks at each method call before proceeding to the next line of code. A program executes in the same sequence as the code in the source code file. Synchronous execution is synonymous to serial execution.

Asynchronous

Asynchronous (or async) execution refers to execution that doesn't block when invoking subroutines. Or if you prefer the more fancy Wikipedia definition: Asynchronous programming is a means of parallel programming in which a unit of work runs separately from the main application thread and notifies the calling thread of its completion, failure or progress. An asynchronous program doesn’t wait for a task to complete and can move on to the next task.

In contrast to synchronous execution, asynchronous execution doesn't necessarily execute code line by line, that is instructions may not run in the sequence they appear in the code. Async execution can invoke a method and move onto the next line of code without waiting for the invoked function to complete or receive its result. Usually, such methods return an entity sometimes called a future or promise that is a representation of an in-progress computation. The program can query for the status of the computation via the returned future or promise and retrieve the result once completed. Asynchronous programming is an excellent choice for applications that do extensive network or disk I/O and spend most of their time waiting. As an example, Javascript enables concurrency using AJAX library's asynchronous method calls. In non-threaded environments, asynchronous programming provides an alternative to threads in order to achieve concurrency and fall under the cooperative multitasking model.

CPU Bound

Programs which are compute-intensive i.e. program execution requires very high utilization of the CPU (close to 100%) are called CPU bound programs. Such programs primarily depend on improving CPU speed to decrease program completion time. This could include programs such as data crunching, image processing, matrix multiplication etc.

If a CPU bound program is provided a more powerful CPU it can potentially complete faster. Eventually, there is a limit on how powerful a single CPU can be. At this point, the recourse is to harness the computing power of multiple CPUs and structure your program code in a way that can take advantage of the multiple CPU units available. Say we are trying to sum up the first 1 million natural numbers. A single-threaded program would sum in a single loop from 1 to 1000000. To cut down on execution time, we can create two threads and divide the range into two halves. The first thread sums the numbers from 1 to 500000 and the second sums the numbers from 500001 to 1000000. If there are two processors available on the machine, then each thread can independently run on a single CPU in parallel. In the end, we sum the results from the two threads to get the final result. Theoretically, the multithreaded program should finish in half the time that it takes for the single-threaded program. However, there will be a slight overhead of creating the two threads and merging the results from the two threads.

Multithreaded programs can improve performance in cases where the problem lends itself to being divided into smaller pieces that different threads can work on independently. This may not always be true though.

I/O Bound

I/O bound programs are the opposite of CPU bound programs. Such programs spend most of their time waiting for input or output operations to complete while the CPU sits idle. I/O operations can consist of operations that write or read from main memory or network interfaces. Because the CPU and main memory are physically separate a data bus exists between the two to transfer bits to and fro. Similarly, data needs to be moved between network interfaces and CPU/memory. Even though the physical distances are tiny, the time taken to move the data across is big enough for several thousand CPU cycles to go waste. This is why I/O bound programs would show relatively lower CPU utilization than CPU bound programs.

Throughput

Throughput is defined as the rate of doing work or how much work gets done per unit of time. If you are an Instagram user, you could define throughput as the number of images your phone or browser downloads per unit of time.

Latency

Latency is defined as the time required to complete a task or produce a result. Latency is also referred to as response time. The time it takes for a web browser to download Instagram images from the internet is the latency for downloading the images.

Critical Section

Critical section is any piece of code that has the possibility of being executed concurrently by more than one thread of the application and exposes any shared data or resources used by the application for access.

Race Condition

Race conditions happen when threads run through critical sections without thread synchronization. The threads "race" through the critical section to write or read shared resources and depending on the order in which threads finish the "race", the program output changes. In a race condition, threads access shared resources or program variables that might be worked on by other threads at the same time causing the application data to be inconsistent.

As an example consider a thread that tests for a state/condition, called a predicate, and then based on the condition takes subsequent action. This sequence is called test-then-act. The pitfall here is that the state can be mutated by the second thread just after the test by the first thread and before the first thread takes action based on the test. A different thread changes the predicate in between the test and act. In this case, action by the first thread is not justified since the predicate doesn't hold when the action is executed.

Consider the snippet below. We have two threads working on the same variable randInt. The modifier thread perpetually updates the value of randInt in a loop while the printer thread prints the value of randInt only if randInt is divisible by 5. If you let this program run, you'll notice some values get printed even though they aren't divisible by 5 demonstrating a thread unsafe version of test-then-act.

Mutex

Mutex as the name hints implies mutual exclusion. A mutex is used to guard shared data such as a linked-list, an array or any primitive type. A mutex allows only a single thread to access a resource or critical section.

Once a thread acquires a mutex, all other threads attempting to acquire the same mutex are blocked until the first thread releases the mutex. Once released, most implementations arbitrarily chose one of the waiting threads to acquire the mutex and make progress.

Semaphore

Semaphore, on the other hand, is used for limiting access to a collection of resources. Think of semaphore as having a limited number of permits to give out. If a semaphore has given out all the permits it has, then any new thread that comes along requesting for a permit will be blocked, till an earlier thread with a permit returns it to the semaphore. A typical example would be a pool of database connections that can be handed out to requesting threads. Say there are ten available connections but 50 requesting threads. In such a scenario, a semaphore can only give out ten permits or connections at any given point in time.

A semaphore with a single permit is called a binary semaphore