Concurrency and Threading: Introduction

Hello, and Welcome to this introductory lesson on Concurrency and Threading.

We will start by exploring Concurrency by answering "why" concurrency is needed. We will start in a world where concurrency does not exist, and quickly realize that something has to change to improve performance. Starting from here, answering why concurrency is beneficial will be easy.

We will then answer "how" concurrency can be achieved. Advanced students in particular always think of concurrency in terms of multithreading. We want to keep the definition of concurrency a bit open, at least at the beginning, and talk about tasks, and how tasks are managed. We will distinguish serial management, where tasks are executed serially, from collaborative and preemptive task management.

The problem with concurrency is that the tasks, whether they are threads or other constructs, share a global state. We will compare our three task management approaches in terms of level of concurrency permitted and trade this off against difficulty of maintaining a consistent global state.

We will then briefly re-acquaint ourselves with threads as the preferred way to realize preemptive thread management.

Finally, we refresh our memory of POSIX pthreads as an example of a thread library.

In this lesson we will be looking at concurrency and threading from a user perspective. In future lessons we will take a more system and implementation oriented view.

Let's begin.

Concurrency: "Why?"

Let's answer the question of "why concurrency?" by looking at a very simple system of a server program, that executes, on behalf of clients, a particular task. If the server program does not support any form of concurrency, then only one copy of the task, or of any task for that manner, can run, and if a second client comes along with a request for the execution of a task, it is blocked.

The client is blocked until the task completes, and the server returns the response to the first client.

Only then, can the server start the task on behalf of the second client.

So, what could we do if we had concurrency, that is, when server were organized such that multiple tasks could run at the same time?

One immediate performance benefit would be "latency reduction", where tasks would take less time to complete.

Let's assume that we had more than one processor available, either as part of a multicore, or of a massively parallel machine.

The client issues a request to execute a task, and the server could split up the overall task into a set of shorter subtasks that could in turn be executed in parallel. The concurrent execution of the parallel subtask reduces the response time, the latency, of the overall task.

In many cases it is not possible to "reduce" the task completion latency, in particular when the task is not parallelizable, or when only one processor is available. It such cases it is often possible to hide the latency by taking advantage of multiprogramming.

The pattern for this is often that the client invokes a length task on the server.

Immediately after the first task is invoked, the client invokes a second.

An then a third.

While the latency of each individual task has not been reduced, this pipelining reduces the waiting time for the client after the return of the first task. The latency is in this way hidden without being reduced.

The most common benefit of concurrency is the increase of system utilization due to increased throughput, which in turn is due to the concurrent execution of tasks.

In this example, a first client invokes a task on the server, which is executed concurrently to a task that has been invoked by a second client.

Multiprogramming ensures that the executions of the two tasks overlap to some extent, and thus system utilization and throughput increase.

Concurrency: "How?"

How do we achieve this concurrency? Let's first make sure that we understand that we are talking about tasks, which is a control flow of some discrete unit of work that needs to be executed. Let's think in term of how we manage these tasks.

The simplest form is to execute each task to completion before starting a new task.

In this example, two clients issue a new task on the server, and only task gets to execute, and one of the two clients has to wait.

Only when the first task is completed, the second task can start, and then completes.

A slightly more efficient way is so-called "cooperative" management of tasks.

At any point in time, we can have multiple tasks ready to go, with of course only one running.

At some point, the running task voluntarily gives up the CPU to some other task, which then takes over.

That task in turn can voluntarily yield the CPU back, say to the first task.

Cooperative task management is rather simple to implement. For example, the Task can be split up into small function invocations, which are invoked by the server one at a time. For example, the last operation of such a function could be an asynchronous I/O operation, and the function returns after this operation is invoked but likely before it returns. While the I/O operation runs, the server calls some other function of some other task. We will talk more about this form of task management when we discuss event-based concurrency in a later lecture.

The last common way to implement concurrency is what we call "preemptive" task management, where the control flow of a task can be preempted and the CPU given to another task. As a result, the execution of tasks can interleave.

The execution of these tasks on the CPU is orchestrated by a scheduler, where this picture symbolically illustrates the scheduler and the dispatcher.

The most popular way to realize this form of task management is through threads.

There is an fly in the concurrency ointment of course, which is that tasks typically access a shared global state. In simple terms, tasks share data.

This of course leads to all kinds of complications.

Shared Global State

Let's have a closer look at what the problem is with sharing a global state, using an example.

Let's assume that multiple tasks access the state of an account, and we represent the account by the sum of all credits, the sum of all debits, and by the balance.

When we design systems, we rely, either explicitly or implicitly, on a set of so-called "invariants", which are expressions about the system that have to be true at all time.

In the example of the account, such an invariant would state that at all times the balance is equal to the difference between credits and debits.

In this example, both the sum of credits as well as the sum of all debits is 120, and the balance is zero. The invariant therefore holds.

If we modify the debits to 80 without updating the balance, then the invariant clearly does not hold anymore. We say that the invariant is violated.

While the invariant is violated, the system is in an inconsistent state. Tasks can temporarily violate invariants, but the system has to be brought back to a consistent state, where all invariants hold, before the global state becomes visible to anybody else.

If this is not guaranteed, the system has what are called "race conditions".

Concurrency: "How?"

We will briefly explore how our three task management approaches, namely serial, cooperative, and preemptive task management, deal with maintaining consistency of shared global state.

We will see that there is a tradeoff between ease of sharing data and performance. We will be comparing some pros and cons.

Serial Task Management

Let's look first at the case of serial task management.

When we have serial tasks, we may have multiple clients, but only one task is allowed to run at any given time. This task runs to completion before the next task can start.

Because of this, we don't have to worry about consistency of the data, as long as the task code does not have bugs, that is.

We say that we can define so-called "inter-task invariants", which only have to be valid when the task begins and when it terminates. If a task modifies the account in the previous example, it is no problem if the account invariant is violated while the task is running, as long as the invariant holds again when the task terminates.

We say that the invariant is "inter-task", and this is a bit of a technical term, because any task relying on such an invariant can do so safely because no other task can violate the invariant in the meantime. The reason for this is obvious because no other task can run.

Let's look at an example, where all the invariants initially hold. We mark this with a green data repository.

The first task accesses and modifies the data, which at some point may lead to a temporary violation of some invariant. We mark this with a red repository.

If the code for the task is implemented correctly, after some time the task will update the data such that the invariant holds again.

Maybe after some time the task completes. The data is left in a consistent state, with all the invariants satisfied.

When now the second task starts, it encounters the data in a consistent state.

The cons of serial task management is that there is "only one" task running at a given time.

As a result, there is no multiprogramming, and there is no support for multiprocessor parallelism.

Cooperative Task Management

Things are a bit better with cooperative task management. Again, we have two clients, with one task each ready to go. All invariants of the shared data hold. The data therefore is in "consistent" state.

As we illustrated earlier, cooperative task management allows for some controlled multiprogramming.

Let's see how this plays with shared data. The task accesses shared data.

After some modifications, the data may become inconsistent. This is ok as long as no other task accesses it.

Now the task gets ready to give up the CPU. Before it does so, it makes the data consistent again.

And hand over the CPU to another task. As we described earlier, this can be as simple as the task returning from a function because it has invoked an asynchronous IO operation and would have to wait anyway.

We notice that invariants must be ensured only at the points when the task yields the CPU. It is not necessary to ensure invariants at any other point.

Another task takes over, and is sure to encounter the data in a consistent state.

It may temporarily violate invariants, but before giving up the CPU again, the task makes the invariants valid again.

It may then give up the CPU. We notice that, of course, the invariants are not automatically enforced. If the task leaves the data in an inconsistent state, the system does not notice this, and the next task will have to deal with inconsistent data. (10)

The first task then resumes from the previous yield. It then may access the data again, and may temporarily violate some invariant, but before it completes, all data is consistent again.

The task completes, and the other task takes over again, possibly modifying the data and temporarily rendering the shared data inconsistent. When it completes, however, all invariants are satisfied again.

Of course the reality is a bit more complicated, as the local state of a task when it resumes after a yield may depend on invalid assumptions about portions of the global state that are not controllable.

For example, the task may have checked whether a file exists, then yields, and when it resumes it wants to open the file. In the meantime some other task may have deleted the file.

But for now let's not worry about this case.

Preemptive Task Management

As we will discover in detail later, in preemptive task management, where we have a preemptive scheduler orchestrating the execution of tasks, we get high levels of multiprogramming and support for parallelism if we have more than one CPU.

The access to shared data is complicated, however.

In this case the task starts modifying the data and temporarily violates one or more invariants, with the intention to satisfy them again later.

Unfortunately, the preemptive scheduler decides that it is time for another task to start or continue its execution.

The currently running task gets preempted, that is, has to "involuntarily" give up the CPU while the data is in an inconsistent state. (5)

When the other task either starts or resumes from a previous voluntary yield or a preemption, it encounters the data in an inconsistent state.

We have a potential race condition.

The major problem of preemptive task management therefore is that the programmer needs to ensure that all invariants (strictly speaking the "inter-task invariants") hold at all times.

The way to do this is to ensure that no other task ever encounters data that is inconsistent.

The mechanism to enforce this is synchronization of the tasks: in lock-based systems we ensure that tasks that may access inconsistent data are blocked from execution until the data is consistent again. We will learn in a later lecture about transaction based synchronization, where we can synchronize tasks without locks.

Our discussion until now has been intentionally somewhat abstract. We have talked about tasks, without specifying what these tasks are, other than that they are control flows, basically work that needs to be done. In practice, preemptive task management is mostly realized through threads, and we will focus on threads first.

Preemtive Task Management: "How?"

Threads are managed by a scheduler.

Let's start with a process, which we know well.

The address space of the process contains, among others, the executable, the data segment, and the file descriptor table.

There is of course the thread of control.

Which needs a set of registers and a stack.

We can not add a second thread of control.

In order for this to work, we need to give it its own register set and stack. Now we have two full-fledged threads.

Example: POSIX pthreads

Let's take the opportunity to look at a widely-used thread model, namely POSIX pthreads. Many other thread packages, abstractions, and implementations exist, but some are language-level concepts or not so widely used.

Here we see a table of operations that we can do with threads in POSIX. We can create a thread, terminate another thread, wait for another thread, and so on.

We notice that there is no operation controlling the actual execution of the thread. This is all largely hidden from the user and handled, using preemptive task management, by the scheduler.

Much of how threads are used becomes evident when we look at how threads are created.

The function pthread_create creates a new thread and hands it over to the scheduler for execution.

The arguments are as follows:

The thread argument is the task descriptor. Tasks are identified using this descriptor, and it is initialized as part of the pthread_create function.

The next argument is a set of attributes that define how the thread has to be controlled. We will look at them a bit more closely in a minute.

Next is a pointer to the function that defines what the thread actually does. When the function returns, the thread terminates. We call this function the "thread function".

If we look closely at the signature of this argument, we see that this is a function pointer that takes a void pointer as argument, and that returns a void pointer.

The last argument is the argument that is passed to the thread function when the thread starts execution. This argument is a void pointer, which is what the thread function expects. More complicated arguments can be passed to the function, for example structures, by passing pointers to them, cast as void pointers. Inside the thread function they are cast back to pointers to the original structure.

In this example we open a file and pass the file descriptor as argument when we create a new thread. When the thread starts executing it calls function "processfd", which is defined somewhere else, and which will be given as argument the pointer to the file descriptor "fd".

pthreads: Thread Attributes

We can define how the thread is controlled at run time through the thread attributes, which are passed as argument during thread creation.

We don't want to spend too much time with these attributes. We just want to get an idea what can be controlled.

pthreads: Thread Attributes - State

The first group of attributes defines how the thread runs in relation to its parent.

We say that a thread can run in attached or detached mode. In detached mode there is no connection to the parent thread.

If the thread is started in attached mode, it depends on the parent thread. For example, if the parent thread terminates, this thread terminates as well. On the other hand, if this thread terminates, it remains in limbo until the parent calls pthread_join to await this thread's completion.

pthreads: Thread Attributes - Stack

A second set of attributes controls the management of the thread stack.

For example, the programmer can define a location and a size of the stack for the thread, which will be used instead of the default stack that pthread create would allocate. This sometimes comes in handy because some implementations of pthreads allocate really large stacks by default.

The stack can grow, and this can be controlled by the attributed "setguardsize". The details are not important here.

pthreads: Thread Attributes - Scheduling

Another interesting set of parameters are those that control the scheduling of the thread.

For example, we can specify that we want to specifically inherit the parameters from the parent thread.

There are a number of scheduling policies, ranging from FIFO scheduling, to round robin, to sporadic scheduling, and to specifically tailored ones. We will discuss some of these policies when we talk about scheduling.

Finally, an important attribute is the contention scope of the thread, which defines whether the thread has to compete for CPU at system level or at process level. This attributed controls how the CPU is allocated across processes. If in a system with two processes the second process has 99 threads that compete at system level, then the first process gets only 1 percent of the CPU, because it has only one thread, which competes with the other 99. If the 99 threads compete at process level, then the first process, which has only one thread, gets 50 % of the CPU bandwidth.

Summary - Concurrency and Threading: Introduction

After this detailed description of attributes for pthreads we have come to the conclusion of this introductory lesson on Concurrency and Threading.

We have explored the benefits of concurrency, and have looked at way that concurrency can be realized, at least at conceptual level, by using a task abstraction and by describing how tasks can be managed.

We distinguished between serial, collaborative, and preemptive task management.

Preemptive task management allows for high levels of multiprogramming and parallelism in principle, but in practice it is hampered by the problem that the concurrent tasks have to access a global shared state. We learned about the need for invariants when accessing shared global state, and we have seen that these invariants are easier to ensure by programmers in systems that use serial or collaborative task management.

We briefly reminded ourselves of what threads are and how they are structured, and we looked at one widely used threading model, namely POSIX pthreads.

In this lesson we have explored Concurrency and Threading from a user perspective. In the following lessons we will revisit threads, and later events, from a system perspective, where we will learn, among others, how the OS switches between threads, that is, dispatches threads, and how the scheduler uses the dispatcher to allocate the CPU among thread.

We sincerely hope that you have enjoyed this introductory lesson on Concurrency and Threading. The intention of this lesson was to refresh your memory on concurrency in general and on threads in particular. This will make the following lessons easier to follow.

Thank you for watching.