EYNTK - Multithreading

Multithreading

README

- Multithreaded Programming (POSIX pthreads Tutorial) (/rltoken/CrNXhLx5hLYEBHFOtUMLRg)
- Multithreading in C (/rltoken/bsmGdbxgo3tbxNjeJVU0bA)

Process vs. Thread

As described in the resources above, the difference between a process and a thread could be summarized as follows:

A process can have sub-processes. Threads are part of a process. Threads (of the same process) run in a shared memory space, while processes run in separate memory spaces. Threads share things like the code and data sections, and processes have their own.

What are threads good for

Threads are very useful when used properly. In fact, parallelism is a great way to separate logic components in some cases:

- In a browser, multiple tabs can be different threads
- A text editor can use multiple threads, one thread to format the text, other thread to process inputs, etc.
- Video games ugely benefit from multithreading. A thread can be used for handling t, another to handle the rendering, another to run a clock, etc.
- In Image/Video rendering/editing, different threads can handle different chuncks of the image, or in the case of videos, a thread could handle the audio while another could handle the video.

There are 2 main cases for turning to the use threads:

Something is time sensitive

By time sensitive, understand anything that cannot afford to spend time on a task, whther this task is time consuming or not. The perfect example is the web browser: Each tab is handled by a thread. This allows the main thread (basically the browser window) to be able to handle clicks from the user, even when pages are loading. Whether a page load in a nanosecond or 15 seconds, it's better to have a thread handle it.

The same goes for a web server. When the server receives a request, it makes sense to have this request handled in a thread so that the flow of the server is not interupted and it can receive more requests.

Something is time consuming

Now this one is pretty straight forward. A task that takes a considerable amount of time can be run in a thread, or even a pool of threads.

There are many examples of the use of threads to handle time consuming tasks. Let's say you are rendering an image, pixel by pixel. You could divide your image into chuncks and have a thread per chunck. If you have 4 chuncks, you could render them simultaneously.

In other words, if you have many steps that are supposed to run sequentially, in some cases you can run those steps simultaneously using threads.

Wait a minute

If a thread is much faster than a process to create, why would we use `fork()` to create a subprocess in our shell project? Why not use a thread?

And the answer is a simple as "because a thread is not what you needed.". What you needed an isolated and independant `process` to run `execve`, which would override said sub-process with the program to execute.

When you create a thread in a process, this thread belongs to the same process, and threads share theur address space. Using `execve` would just erase the current process, and you'd end your program.

Limitations (Amdahl's Law)

It could sound obvious to say that a process cannot just spwan as many threads as they want without experiencing any issue. Threads are powerful tools when used correctly. In fact, the only a certain number of simultaneous threads your computer can run before it gets inneffi your CPU can handle 16 simultaneous threads and you try to run 30, you will not benefit from simultaneous execution.

And for some use cases it's okay! Again if we take a web browser as an example. We don't really care whether a page in a tab is loaded simultaneously from other tabs. What we want is just that it loads.

On the other hand, it becomes important for time consuming operations. If your goal is to save time, you have to consider the number of threads you want to use. Using too many threads can become as inneficient as using just two or three.

Shared resources and Mutual Exclusion

This is described in the resources attached to this concept page, but I wanted to emphasize on this aspect.

Threads within a process share the same address space, meaning that threads can access the same data. And this can become an issue. Threads need a mechanism to allow safe access to data.

Mutual exclusion

Mutual exclusion (`mutex`) is a method for serializing access to shared data or resources. A mutex is basically a lock that can be attached (virtually attached) to a resource. When a thread wants to modify/access the resource, it must first gain access to the lock (the mutex). Once it has gained access to the lock, the thread may operate on the resource without worrying that another thread might be using it. When the thread is done with the resource, it can release the mutex for other threads to use.

Limitations - Deadlock

One of the most common problem with multithreading and the use of mutex is the creation of a `deadlock` . This happens when a programs stops execution or hangs indefinitely due to conflicting mutex locking.

For example, a simple deadlock situation: thread 1 locks lock A, thread 2 locks lock B, thread 1 wants lock B and thread 2 wants lock A. Instant deadlock. You can prevent this from happening by making sure threads acquire locks in an agreed order (i.e. preservation of lock ordering). Deadlock can also happen if threads do not unlock mutexes properly.

Limitations - Race condition

Race conditions are … a pain. Our human brain is good at understading sequential programs. Execute instruction A, then B, then C, etc. When it comes to parallel execution, our brain can only imagine possibilities:

- Execute T1-A, then T2-A, then T1-B, etc OR
- Execute T1-A, then T1-B, then T2-A, etc.

Let's take 2 set of instructions executed in parallel. At some point in their execution, the two sets must access a resource. Who is going to access it first? Impossible to know, or to assume. That's what is called a `race condition` .

Race conditions are particuly tough to debug as they are most of the time specific to YOUR case. The best way to avoid race conditions is to define some sort of sequential protocol to access specific resource.

Conclusion

We know covered everything that needed to be covered, so let's dive in with the project!