CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Thread Issues

Effectively using threads requires care

Implicit threading is transferring responsibilty for thread creation and management from programmers to compilers or libraries

Offers the efficiency of parallel threads without some of the coding details

We look briefly at *thread pools* and *OpenMP*, but these are not the only ways of doing implicit threading

For example, `java.util.concurrent` supports implicit threading in Java

Thread pool

We discussed servers that accept a user request and spawn a thread to handle it

This leads to a lot of overhead due to thread creation and can result in too many threads being produced

Instead, create some predefined number of threads, called the *thread pool*, and assign tasks to the pool

Threads will take work as it becomes available and will go back to the pool when finished instead of being deleted

Pthreads do not have built-in mechanism for thread pools

Windows threads do have a thread pool API. For example, to request that a thread from a pool call `PoolFunction`:

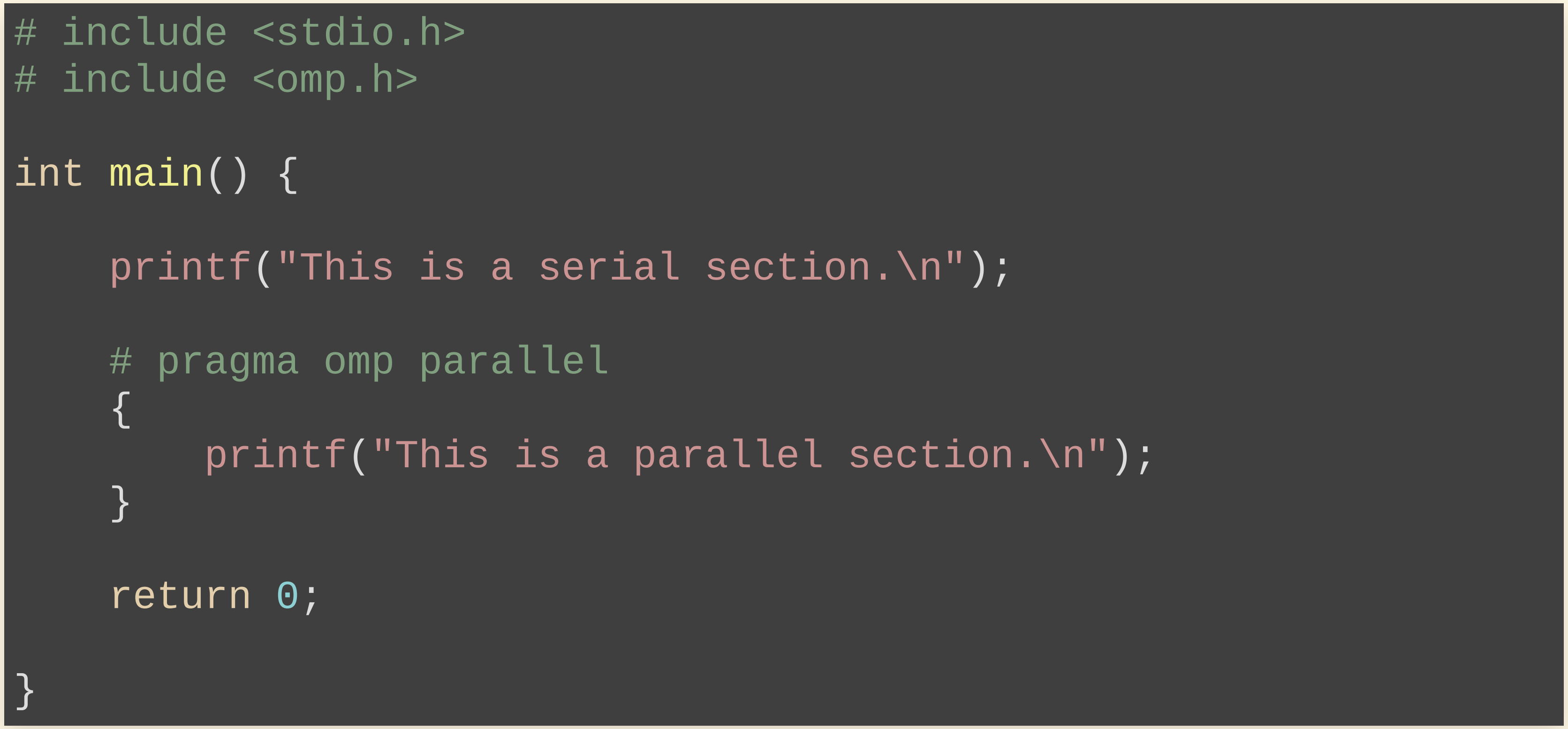
```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

`java.util.concurrent` also provides thread pools

OpenMP

OpenMP provides compiler directives that programmers use to specify parallel regions

OpenMP runtime handles details of thread creation and management



```c
# include <stdio.h>
# include <omp.h>

int main() {

    printf("This is a serial section.\n");

    # pragma omp parallel
    {
        printf("This is a parallel section.\n");
    }

    return 0;

}
```

Programmer need not wory about threads. Threads are created at the entry to the parallel region and terminated at the end of the region

OpenMP defaults to trying to maximize system resource utilization by creating one thread per processing core

This may not seem all that useful, but OpenMP also helps parallelize loops

```c
# pragma omp parallel for
for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

`fork()` and `exec()` with threads

Need to decide what happens when multi-threaded process calls `fork` -- does not process have all threads, or only thread that called `fork`?

In Linux, child process has just one thread

However, forking from multi-threaded process is generally considered a bad idea, except possibly if you `exec` immediately after

When thread calls exec, what happens to other threads?

Previously said that memory of a process is overwritten when exec is called, so other threads would not be able to continue running

In Linux, all other threads are terminated and new process is single-threaded

Cancelling threads

If a thread is no longer needed, another thread can cancel it. Cancellation can be either *asynchronous* or *deferred*

Asynchronous cancellation happens immediately and does not leave the thread an opportunity to clean up its resources. Therefore, asynchronous cancellation is not recommended

With deferred cancellation, a thread notifies the target thread that it should cancel, but it is up to the target

Pthreads handles both types of cancellation with `pthread_cancel`

Whether cancellation is deferred or asynchronous (or not allowed at all) depends on configuration of target thread

With deferred cancellation, target thread can occasionally call `pthread_testcancel` to determine whether it has been cancelled

If a thread goes through deferred cancellation, a cleanup handler is invoked that allows resources to be managed correctly before the cancellation happens

In Linux, there are no "processes" and "threads" -- everything is a "task"

Tasks can share varying amounts of data with other tasks, essentially allowing us to treat them as processes **or** threads depending on what sharing is done

`clone` system call is similar to `fork`, but takes arguments that allow children to share memory with parent rather than copy it, essentially making them threads rather than processes

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Recall the `task_struct` that we previously said was similar to a PCB. Elements of `task_struct` are pointers that can either point to copy of parent's memory (making a "process") or point directly to parent's memory (making a "thread")