【OS】Day30

Class	Operating System: Three Easy Pieces
Date	@February 1, 2022

【Ch26】Concurrency: An Introduction

In this note, we introduce a new abstraction for a single running process: that of a thread.

A multi-threaded program has more than one point of execution(i.e. multiple PCs, each of which is being fetched and executed from).

Perhaps another way to think of this is that each thread is very much like a separate process, except for one difference: they share the same address space and thus can access the same data.

The state of a single thread is thus very similar to that of a process. It has a program counter(PC) that tracks where the program is fetching instructions from.

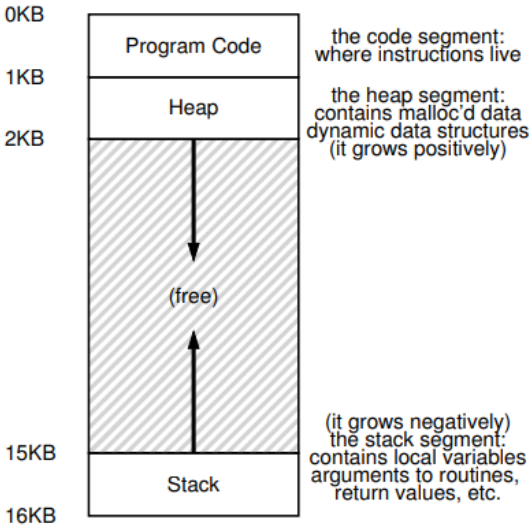
Each thread has its own private set or registers it uses for computation; thus, if there are two threads that are running on a single processor, when switching from running one(T1) to running the other(T2), a context switch must take place.

With processes, we saved state to a process control block(PCB); we'll need one or more thread control blocks(TCBs) to store that state of each thread of a process.

There is one major different, though, in the context switch we perform between threads as compared to processes: the address space remains the same(i.e. there is no need to switch which page table we are using).

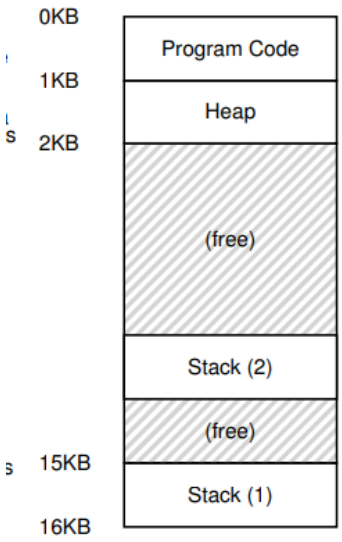
One other major difference between threads and processes concerns the stack.

In our simple model of the address space of a classic process(which we can now call a single-threaded process), there is a single stack, usually residing at the bottom of the address space.



However, in a multi-threaded process, each thread runs independently and of course may call into various routines to do whatever work it is doing. Instead of a single stack in the address space, there will be one per thread.

If we have a multi-threaded process that has two threads in it; the resulting address space looks like following:



Thus, any stack-allocated variables, parameters, return values, and other things that we put on the stack will be placed in what is sometimes called thread-local storage(i.e. the stack of the relevant thread).

26.1 Why Use Threads?

There are at least two major reasons we should use threads:

1. The first is parallelism.

   Imagine we are writing a program that performs operations on very large arrays, for example, incrementing the value of each element in the array by some amount.

   If we are running on just a single processor, the task is straightforward: just perform each operation and be done.

   However, if we are executing the program on a system with multiple processors, we have the potential of speeding up this process considerably by using the processors to each perform a portion of the work.

   The task of transforming your standard single-threaded program into a program that does this work on multiple CPUs is called parallelization, and using a thread per CPU to do this work is a natural and typical way to make programs run faster on modern hardware.

2. The second reason is to avoid blocking program progress due to slow I/O.

   Imagine that we are writing a program that performs different types of I/O. Instead of waiting, our program may wish to do something else, including utilizing the CPU to perform computation, or even issuing further I/O requests.

   Using threads is a natural way to avoid getting stuck; while one thread in your program waits, the CPU scheduler can switch to other threads, which are ready to run and do something useful.

   Threading enables overlap of I/O with other actives within a single program, much like multiprogramming did for processes across programs.

26.2 An Example: Thread Creation

Now say we want to run a program that creates two threads, each of which does some independent work, in this case printing “A” or “B”. The code is shown below:

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Figure 26.2: **Simple Thread Creation Code (`t0.c`)**

- The main function creates two threads, each of which will run the function `mythread()`.
- Once a thread is created, it may start running right away; alternately, it may be put in a “ready” but not “running” state and thus not run yet.
- After creating the two threads(T1 and T2), the main thread calls `pthread_join()`, which waits for a particular thread to complete. It does so twice, thus ensuring T1 and T2 will run and complete before finally allowing the main thread to run again. Overall, three threads were employed during this run: the main thread, T1, and T2.

Now let us examine the possible execution ordering of this little program:

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

Figure 26.3: **Thread Trace (1)**

However, this ordering is not the only possible ordering. In fact, given a sequence of instructions, there are quite a few, depending on which thread the scheduler decides to run at a given point.

For example, once a thread is created, it may run immediately, which would lead to the execution shown below:

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Figure 26.4: **Thread Trace (2)**

What threads run next is determined by the OS scheduler, and although the scheduler likely implements some sensible algorithm, it is hard to know what will run at any given moment in time.