【OS】Day30(2)

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

【Ch26】Concurrency: An Introduction(2)

26.3 Why It Gets Worse: Shared Data

Let us imagine a simple example where two threads wish to update a global shared variable. The code we'll study is in the figure below:

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

static volatile int counter = 0;

// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    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: done with both (counter = %d)\n",
           counter);
    return 0;
}
```

Figure 26.6: **Sharing Data: Uh Oh (`t1.c`)**

We now compile and run the program. Sometimes, everything works how we might expect:

```
prompt> gcc -o main main.c -Wall -pthread; ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

Unfortunately, when we run this code, even on a single processor, we don't necessarily get the desired result. Sometimes, we get:

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

Let's try it one more time.

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

Why does this happen?

26.4 The Heart of The Problem: Uncontrolled Scheduling

To understand why this happens, we must understand the code sequence that the compiler generates. It looks something like this:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Let us imagine one of our two threads enters this region of code, and is thus about to increment counter by one.

1. It loads the value(`val = 50`) to `%eax`.
2. Add one to the register

Now, a timer interrupt goes off, thus, the OS saves the state ot the currently running thread to the thread's TCB.

Now, thread 2 is chosen to run. It enters the same piece of code and does the following:

1. It loads the value(`val = 50`) to `%eax`.
2. Add one to the register
3. Store the value back to memory

Thus, the global variable counter now has the value 51.

Finally, another context switch occurs, and thread 1 resumes running. It just executed the `mov` and add, and is now about to perform the final `mov` instruction. Recall also that `eax = 51`. Thus, the final `mov` instruction executes, and saves the value to memory; the counter is set to 51 again.

A "correct" version of this program should have resulted in the variable counter equal to 52.

The execution trace is of following:

OS	Thread 1	Thread 2	(after instruction) PC	eax	counter
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

What we have demonstrated here is called a race condition(or more specifically, a data race): the results depend on the timing execution of the code.

With some bad luck(i.e. context switches that occur at untimely points), we get the wrong result.

In fact, we may get a different result each time; thus, instead of a nice deterministic computation, we call this result indeterminate.

We call this code a critical section, which is a piece of code that accesses a shared variable.

What we want for this code is mutual exclusion. This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

26.5 The Wish for Atomicity

One way to solve this problem would be to have more powerful instructions that, in a single step, did exactly whatever we needed done and thus removed the possibility of an untimely interrupt.

For example, what if we had a super instruction that looked like this:

```
memory-add 0x8049a1c, $0x1
```

The hardware guarantees that it executes atomically, meaning as a unit. Thus, this instruction cannot be interrupted mid-execution.

However, such instructions are not possible. Thus, what we will instead do is ask the hardware for a few useful instructions upon which we can build a general set of what we call synchronization primitives.

By using this hardware support, in combination with some help from the operating system, we will be able to build multi-threaded code that accesses critical sections in a synchronized and controlled manner, and thus reliably produces the correct result despite the challenging nature of concurrent execution.

Key Point Terms

ASIDE: KEY CONCURRENCY TERMS
CRITICAL SECTION, RACE CONDITION,
INDETERMINATE, MUTUAL EXCLUSION

These four terms are so central to concurrent code that we thought it worth while to call them out explicitly. See some of Dijkstra's early work [D65,D68] for more details.

- A **critical section** is a piece of code that accesses a *shared* resource, usually a variable or data structure.
- A **race condition** (or **data race** [NM92]) arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- An **indeterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not **deterministic**, something we usually expect from computer systems.
- To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.