Shared Memory Programming

Reading: Chapter 12 of *Computer Systems: A Programmers Perspective*

Shared memory

Consider our concurrent dot-product algorithm.

concurrently for $k \in \{0, ..p\}$
  $sum[k] := 0$
  for $i \in \{ \left\lfloor \frac{k \times n}{p} \right\rfloor , .. \left\lfloor \frac{(k+1) \times n}{p} \right\rfloor \}$
    $sum[k] := sum[k] + a[i] \times b[i]$
$result := 0$
for $k \in \{0, ..p\}$ $result := result + sum[k]$

There are $p + 1$ threads. There are p 'worker' threads calculating sum and the 'master thread' that adds items of sum.

We assume that arrays a, b, and sum can be accessed by all threads.

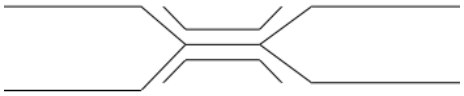
I.e. they represent shared memory.

In this case the worker threads access disjoint sets of locations and the master thread does not do anything until the worker threads are finished. It will work.

What happens when two threads access the same memory location at about the same time?

Race conditions

Consider two parallel train tracks that merge where there is a bridge.



When two trains both attempt to cross the bridge from west to east at about the same time several possibilities exist.

- The north train crosses first; then the south train.
- The south train crosses first; then the north train.
- The trains collide.

[See animation.]

Two agents using the same resource at the same time can lead to problems.

A program has a *race condition* when, because of concurrency, multiple behaviours are possible and some behaviours are incorrect.

So race conditions are bugs, but they may not be revealed by testing.

Race conditions while sharing memory

Consider what can happen when two threads attempt to write to shared memory at the same time

concurrently

...

$x := x + 1$

...

||

...

$x := x + 1$

...

end concurrently

$print(x)$

We consider the program correct if for an input of $x = y$ the final state is $x = y + 2$.

Suppose $x = 40$ to start.

What will it print?

- If the first assignment executes first: 42
- If the second assignment executes first: 42

But storing and fetching are separate memory operations:

- If both threads fetch 40, and then they both store, the result will be 41.

Thread 0			Thread 1			Memory
	$r1$	$r2$		$r1$	$r2$	x
						40
$r1 \longleftarrow fetch(x)$						
	40					
$r2 \longleftarrow r1 + 1$						
		41				
			$r1 \longleftarrow fetch(x)$			
				40		
			$r2 \longleftarrow r1 + 1$			
					41	
			$x \longleftarrow store(r2)$			
						41
$x \longleftarrow store(r2)$						
						41

Testing of the above program will likely not reveal the bug.

This is an example of a race condition.

Mutual exclusion

The solution to many race conditions is *mutual exclusion*.

This means only one thread uses a resource at a time.

Let's go back to the train example.

Two trains share a bridge. They use the following convention.

- We have a bowl with a token in it.
- Before crossing the bridge the train driver must remove the token from the bowl.
  - If the bowl is empty the driver must wait until it is not.
- The driver with the token may cross the bridge.
- After reaching the other side, the driver places the token back in the bowl.

[See animation.]

In programming we call the bowl/token a “mutex”.

[Language note: “mutex” is a contraction of “mutual exclusion”]

Each mutex m has two states: locked and unlocked.

There are two operations on a mutex m

- lock(m) changes the state from unlocked to locked.
  - If m is unlocked, it immediately becomes locked. And the call returns.
  - If m is locked, the thread is blocked; the thread waits.
  - If more than one thread tries to lock at the same time only one succeeds; the others wait.
- unlock(lm)
  - Should only be executed by the thread that locked the mutex.
  - Immediately changes the state to unlocked.
  - If there are any waiting threads, one (and only one) waiting thread will then lock the mutex and return from lock.

For example we can protect the variable x like this.

var $l : Mutex$
concurrently
    ...
    lock(l)
    $x := x + 1$
    unlock(l)
    ...
||
    ...
    lock(l)
    $x := x + 1$
    unlock(l)
    ...
end concurrently
$print(x)$

Three threads try to use a resource protected by mutex l. Each thread does the following

lock(l)
use resource
unlock(l)

A possible scenario.

Thread A	Thread B	Thread C	l
			unlocked
calls lock(l)			
lock returns			locked
	calls lock(l)		
		calls lock(l)	
uses resource			
calls unlock(l)			
unlock returns	lock returns		
	uses resource		
	calls unlock(l)		
	unlock returns	lock returns	
		uses resource	
		calls unlock(l)	
		unlock returns	unlocked

Another example

Here is another example. Let's remove the sum array from the dot product calculation

var $result := 0$
var $l : Mutex$ // Used to protect *result*.
concurrently for $k \in \{0, ..p\}$
  var $sum := 0$
  for $i \in \{ \left\lfloor \frac{k \times n}{p} \right\rfloor , .. \left\lfloor \frac{(k+1) \times n}{p} \right\rfloor \}$
    $sum := sum + a[i] \times b[i]$
  lock l
  $result := result + sum$
  unlock l

Here each sum variable is local to its thread.

Using a mutex to protect a data structure.

Threads in C and UNIX

Making a new thread with fork.

We can use the UNIX fork system call to create a new process. Since each process in UNIX has its own thread, this will create a new thread. Unfortunately for us, each process has it's own memory space. So our new process will not share memory.

Here is an example program that illustrates that processes do not (by default) share memory.

Program forkExample.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <sys/wait.h>
 4 #include <stdlib.h>
 5
 6 static const int ITERATIONS = 100000 ;
 7
 8 int x = 0 ;
 9
10
11 int main(int argc, char **argv) {
12
13     pid_t parent_pid = getpid() ;
14     printf( "The parent process has started. Its PID is %lu\n",
15             (unsigned long) parent_pid ) ;
16
17
18     pid_t child_pid = fork() ;
19     if( child_pid == 0 ) {
20         pid_t my_pid = getpid() ;
21         printf("The child process has started. Its PID is %lu.\n",
22                (unsigned long) my_pid ) ;
23         printf("In the child process, the address of x is %lx and its value is %d.\n",
24                (unsigned long) &x, x ) ;
25         for( int i=0 ; i < ITERATIONS ; ++i ) {
26             ++x ; }
27         printf("In the child process, the final value of x is %d.\n", x ) ;
28         exit(0) ; }
29     printf("In the parent process, the Child's pid is %lu.\n",
30            (unsigned long) child_pid ) ;
31     printf("In the parent process, the address of x is %lx and its value is %d.\n",
32            (unsigned long) &x, x ) ;
33     for( int i=0 ; i < ITERATIONS ; ++i ) {
34         --x ; }
35     printf("In the parent process, the final value of x is %d.\n", x ) ;
36     pid_t result = waitpid( child_pid, NULL, 0 ) ;
37     return 0 ; }
```

How it works:

- On line 18 the call to fork creates a new process.
- Both processes run the same program.
- Both processes return from the fork call.
- For the child process the result of fork is 0, so lines 20-28 are executed only by the child.
- For the original (parent) process, the result of fork is >

0 and is the process identifier of the child.

- Lines 29–37 are executed only by the parent process.
- The exit call on line 28 terminates the child process.
- The waitpid call on line 36 waits until the child process has terminated.

Running the program we get the following output

```
The parent process has started. Its PID is 17127
In the parent process, the Child's pid is 17128.
In the parent process, the address of x is 10d05d038 and its value is 0.
The child process has started. Its PID is 17128.
In the child process, the address of x is 10d05d038 and its value is 0.
In the parent process, the final value of x is -100000.
In the child process, the final value of x is 100000.
```

Notice that outputs from the parent and child are interleaved: We have concurrency.

Notice that the child and parent have different process identifiers.

Both the child and parent processes have variable x at the same address 0x10d05d038.

But since the child and parent have different virtual memory spaces, this address does not map to the same physical address.

After the fork there are 2 variables x — one for each process.

Creating new threads with pthreads.

Unix supports “Posix threads”. There are also implementations of Posix threads for other operating systems (e.g., Windows).

Hence forth pthreads = Posix threads.

With pthreads we can make a new thread that is part of the same process and hence shares memory with the parent thread.

Here is an example with pthreads

Program twoThreads.c

```
 1 #include <stdio.h>
 2 #include <pthread.h>
 3
 4 static const int ITERATIONS = 1000 ;
 5
 6 void *childThread(void *arg) {
 7     for( int i=0 ; i < ITERATIONS ; ++i ) {
 8         printf("HELLO FROM CHILD\n") ; }
 9     return NULL ; }
10
11 int main(int argc, char **argv) {
12
13     pthread_t childID ;
14
15     int err = pthread_create( &childID, NULL, childThread, NULL ) ;
16     for( int i=0 ; i < ITERATIONS ; ++i ) {
17         printf("hello from parent\n" ) ; }
18
19     pthread_join( childID, NULL ) ;
20     return 0 ;
21 }
```

Notes:

- On line 15 we create a new thread using pthread_create.
- The third argument is a pointer to a function. This is the code the child thread will execute.
- Each thread says hello 1000 times.

- On line 19, the main thread waits until the child thread had finished.

The output looks like this.

```
hello from parent
HELLO FROM CHILD
hello from parent
hello from parent
HELLO FROM CHILD
hello from parent
HELLO FROM CHILD
HELLO FROM CHILD
...
```

You can see that there are two threads of control executing at the same time.

The two threads share memory as you can see from this example

Program raceCondition.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <pthread.h>
 4
 5 static const int ITERATIONS = 100 ;
 6
 7 int x = 0 ;
 8
 9 void *childThread(void *arg) {
10     pid_t child_pid = getpid() ;
11     printf( "The child thread has started. Its PID is %lu\n",
12             (unsigned long) child_pid ) ;
13
14     for( int i=0 ; i < ITERATIONS ; ++i ) {
15         ++x ; }
16     return NULL ; }
17
18 int main(int argc, char **argv) {
19
20     pid_t parent_pid = getpid() ;
21     printf( "The parent thread has started. Its PID is %lu\n",
22             (unsigned long) parent_pid ) ;
23
24     pthread_t childID ;
25     int err = pthread_create( &childID, NULL, childThread, NULL ) ;
26     for( int i=0 ; i < ITERATIONS ; ++i ) {
27         --x ; }
28
29     pthread_join( childID, NULL ) ;
30
31     printf( "The final value of x is %d\n", x ) ;
32     return 0 ; }
```

The output is

The parent thread has started. Its PID is 17316
The child thread has started. Its PID is 17316
The final value of x is 0

Both threads exist in the same process and they share the same variable x.

But! Doesn't this program have a race condition?

Let's increase the number of iterations from 100 to 1,000.

The final value of x is 0

Let's increase the number of iterations to 10,000.

The final value of x is 0

Let's increase the number of iterations to 100,000

The final value of x is -70564

Run it again

The final value of x is -1163

And again

The final value of x is 80629

Now the bug is apparent.

There is a race condition.

Notice that naive testing did not find this bug. This is the nature of race conditions.

My motto "Testing concurrent programs doesn't!"

Using a Mutex

PThreads has a type for mutexes.

Program noRaceCondition.c

```
 1 #include <stdio.h>
 2 #include <pthread.h>
 3
 4 static const int ITERATIONS = 10000000 ;
 5
 6 pthread_mutex_t xMutex ;
 7 int x = 0 ;
 8
 9 void *childThread(void *arg) {
10     for( int i=0 ; i < ITERATIONS ; ++i ) {
11         pthread_mutex_lock( &xMutex ) ;
12         ++x ;
13         pthread_mutex_unlock( &xMutex ) ; }
14     return NULL ; }
15
16 int main(int argc, char **argv) {
17
18     pthread_mutex_init( &xMutex, NULL ) ;
19
20     pthread_t childID ;
21     int err = pthread_create( &childID, NULL, childThread, NULL ) ;
22
23     for( int i=0 ; i < ITERATIONS ; ++i ) {
24         pthread_mutex_lock( &xMutex ) ;
25         --x ;
26         pthread_mutex_unlock( &xMutex ) ; }
27
28     pthread_join( childID, NULL ) ;
29     printf( "The final value of x is %d\n", x ) ;
30     return 0 ; }
```

Notes:

- On line, 6 a mutex is declared. Its type is pthread_mutex_t.
- On line, 18 it is initialized.
- On line, 11 the thread locks the mutex.
- On line, 12 the thread uses x
- On line 13, the thread unlocks the mutex.
- On lines 24–26, the parent surrounds its use of x with

lock and unlock

The output of the program after 10,000,000 iterations is

The final value of x is 0