Lecture 08—Race Conditions; More Synchronization

January 21, 2015

Roadmap

Last Time: Race Conditions, Locking (mutexes)

Now: More Synchronization Mechanisms

Part I

More Synchronization

Mutexes Recap

Our focus is on how to use mutexes correctly:

- Call `lock` on mutex `m1`. Upon return from `lock`, you have exclusive access to `m1` until you `unlock` it.
- Other calls to `lock` `m1` will not return until `m1` is available.

For background on selection algorithms, look at Lamport's bakery algorithm.
(Not in scope for this course.)

More on Mutexes

Can also “try-lock”: grab lock if available, else return to caller (and do something else).

Excessive use of locks can serialize programs.

- Linux kernel used to rely on a Big Kernel Lock protecting lots of resources in the 2.0 era.
- Linux 2.2 improved performance on SMPs by cutting down on the use of the BKL.

Note: in Windows, “mutex” is an inter-process communication mechanism. Windows “critical sections” are our mutexes.

Spinlocks

Functionally equivalent to `mutex`.

- `pthread_spinlock_t`, `pthread_spin_lock`, `pthread_spin_trylock` and friends

Implementation difference: spinlocks will repeatedly try the lock and will not put the thread to sleep.

Good if your protected code is short.

Mutexes may be implemented as a combination between spinning and sleeping (spin for a short time, then sleep).

Read-Write Locks

Two observations:

- If there are only reads, there's no datarace.
- Often, writes are relatively rare.

With mutexes/spinlocks, you have to lock the data, even for a read, since a write could happen.

But, most of the time, reads can happen in parallel, as long as there's no write.

Solution: Multiple threads can hold a read lock
    (`pthread_rwlock_rdlock`)
but only one thread may hold the associated write lock
    (`pthread_rwlock_wrlock`);
grabbing the write waits until current readers are done.

Semaphores

Semaphores have a `value`. You specify initial `value`.

Semaphores allow sharing of a # of instances of a resource.

Two fundamental operations: `wait` and `post`.

- `wait` is like `lock`; reserves the resource and decrements the value.
  - If value is 0, sleep until value is greater than 0.
- `post` is like `unlock`; releases the resource and increments the value.

Barriers

Allows you to ensure that (some subset of) a collection of threads all reach the barrier before finishing.

Pthreads: A barrier is a `pthread_barrier_t`.

Functions: `_init()` (parameter: how many threads the barrier should wait for) and `_destroy()`.

Also `_wait()`: similar to `pthread_join()`, but waits for the specified number of threads to arrive at the barrier

Lock-Free Algorithms

We'll talk more about this in a few weeks.

Modern CPUs support atomic operations, such as compare-and-swap, which enable experts to write lock-free code.

Lock-free implementations are extremely complicated and must still contain certain synchronization constructs.

Semaphores Usage

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

- Also must link with `-pthread` (or `-lrt` on Solaris).
- All functions return 0 on success.
- Same usage as mutexes in terms of passing pointers.

How could you use as `semaphore` as a `mutex`?

Semaphores Usage

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

- Also must link with `-pthread` (or `-lrt` on Solaris).
- All functions return 0 on success.
- Same usage as mutexes in terms of passing pointers.

How could you use as `semaphore` as a `mutex`?

- If the initial `value` is 1 and you use `wait` to lock and `post` to unlock, it's equivalent to a `mutex`.

Semaphores for Signalling

Here's an example from the book. How would you make this always print "Thread 1" then "Thread 2" using semaphores?

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#include <stdlib.h>

void* p1 (void* arg) { printf("Thread 1\n"); }

void* p2 (void* arg) { printf("Thread 2\n"); }

int main(int argc, char *argv[])
{
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, p1, NULL);
    pthread_create(&thread[1], NULL, p2, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    return EXIT_SUCCESS;
}
```

Semaphores for Signalling

Here's their solution. Is it actually correct?

```
sem_t sem;
void* p1 (void* arg) {
  printf("Thread 1\n");
  sem_post(&sem);
}
void* p2 (void* arg) {
  sem_wait(&sem);
  printf("Thread 2\n");
}

int main(int argc, char *argv[])
{
    pthread_t thread[2];
    sem_init(&sem, 0, /* value: */ 1);
    pthread_create(&thread[0], NULL, p1, NULL);
    pthread_create(&thread[1], NULL, p2, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    sem_destroy(&sem);
}
```

Semaphores for Signalling

1. `value` is initially 1.
2. Say `p2` hits its `sem_wait` first and succeeds.
3. `value` is now 0 and `p2` prints “Thread 2” first.

- If `p1` happens first, it would just increase `value` to 2.

Semaphores for Signalling

1. `value` is initially 1.

2. Say `p2` hits its `sem_wait` first and succeeds.

3. `value` is now 0 and `p2` prints "Thread 2" first.

- If `p1` happens first, it would just increase `value` to 2.

- Fix: set the initial `value` to 0.

  Then, if `p2` hits its `sem_wait` first, it will not print until `p1` posts (and prints "Thread 1") first.

C++ atomics

Coming soon.

Part II

Making C Compilers Work For You

Three Address Code

- An intermediate code used by compilers for analysis and optimization.
- Statements represent one fundamental operation—we can consider each operation atomic.
- Statements have the form:

  $$\mathit{result} := \mathit{operand}_1 \ \mathit{operator} \ \mathit{operand}_2$$

- Useful for reasoning about data races, and easier to read than assembly.
  (separates out memory reads/writes).

GIMPLE

- GIMPLE is the three address code used by `gcc`.
- To see the GIMPLE representation of your code use the `-fdump-tree-gimple` flag.
- To see all of the three address code generated by the compiler use `-fdump-tree-all`. You'll probably just be interested in the optimized version.
- Use GIMPLE to reason about your code at a low level without having to read assembly.

Live Coding Demo: GIMPLE

volatile Keyword

- Used to notify the compiler that the variable may be changed by “external forces”. For instance,

```
int i = 0;

while (i != 255) {
  ...
```

volatile prevents this from being optimized to:

```
int i = 0;

while (true) {
  ...
```

- Variable will not actually be volatile in the critical section and only prevents useful optimizations.
- Usually wrong unless there is a **very** good reason for it.

Branch Prediction Hints

As seen earlier in class, gcc allows you to give branch prediction hints by calling this builtin function:

```
long __builtin_expect (long exp, long c)
```

The expected result is that `exp` equals `c`.

Compiler reorders code & tells CPU the prediction.

The `restrict` Keyword

A new feature of C99: “The restrict type qualifier allows programs to be written so that translators can produce significantly faster executables.”

- To request C99 in `gcc`, use the `-std=c99` flag.

`restrict` means: you are promising the compiler that the pointer will never alias (another pointer will not point to the same data) for the lifetime of the pointer.

Example of `restrict` (1)

Pointers declared with `restrict` must never point to the same data.

From Wikipedia:

```
void updatePtrs(int* ptrA, int* ptrB, int* val) {
    *ptrA += *val;
    *ptrB += *val;
}
```

Would declaring all these pointers as `restrict` generate better code?

Example of restrict (2)

Let's look at the GIMPLE:

```
void updatePtrs(int* ptrA, int* ptrB, int* val) {
 D.1609 = *ptrA;
 D.1610 = *val;
 D.1611 = D.1609 + D.1610;
 *ptrA = D.1611;
 D.1612 = *ptrB;
 D.1610 = *val;
 D.1613 = D.1612 + D.1610;
 *ptrB = D.1613;
}
```

- Could any operation be left out if all the pointers didn't overlap?

Example of restrict (3)

```
1  void updatePtrs(int* ptrA, int* ptrB, int* val) {
2   D.1609 = *ptrA;
3   D.1610 = *val;
4   D.1611 = D.1609 + D.1610;
5   *ptrA = D.1611;
6   D.1612 = *ptrB;
7   D.1610 = *val;
8   D.1613 = D.1612 + D.1610;
9   *ptrB = D.1613;
10 }
```

- If `ptrA` and `val` are not equal, you don't have to reload the data on **line 7**.
- Otherwise, you would: there might be a call

  `updatePtrs(&x, &y, &x);`

Example of `restrict` (4)

Hence, this markup allows optimization:

```
void updatePtrs(int* restrict ptrA,
                int* restrict ptrB,
                int* restrict val)
```

Note: you can get the optimization by just declaring `ptrA` and `val` as `restrict`; `ptrB` isn't needed for this optimization

Summary of `restrict`

- Use `restrict` whenever you know the pointer will not alias another pointer (also declared `restrict`)

It's hard for the compiler to infer pointer aliasing information; it's easier for you to specify it.

$\Rightarrow$ compiler can better optimize your code (more perf!)

Caveat: don't lie to the compiler, or you will get undefined behaviour.

Aside: `restrict` is not the same as `const`. `const` data can still be changed through an alias.

Next topic: Dependencies

Dependencies are the main limitation to parallelization.

Example: computation must be evaulated as `XY` and not `YX`.

Not synchronization

Assume (for now) no synchronization problems.

Only trying to identify code that is safe to run in parallel.

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42
x = x + 1
```

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42
x = x + 1
```

No.

- Assume x initially 1. What are possible outcomes?

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42
x = x + 1
```

No.

- Assume x initially 1. What are possible outcomes?
  x = 43 or x = 42

Next, we'll classify dependencies.

Read After Read (RAR)

Can we execute these 2 lines in parallel? (initially x is 2)

```
y = x + 1
z = x + 5
```

Read After Read (RAR)

Can we execute these 2 lines in parallel? (initially x is 2)

```
y = x + 1
z = x + 5
```

Yes.

- Variables y and z are independent.
- Variable x is only read.

RAR dependency allows parallelization.

Read After Write (RAW)

What about these 2 lines? (again, initially x is 2):

```
x = 37
z = x + 5
```

Read After Write (RAW)

What about these 2 lines? (again, initially x is 2):

```
x = 37
z = x + 5
```

No, $z = 42$ or $z = 7$.

RAW inhibits parallelization: can't change ordering.
Also known as a true dependency.

Write After Read (WAR)

What if we change the order now? (again, initially x is 2)

```
z = x + 5
x = 37
```

Write After Read (WAR)

What if we change the order now? (again, initially x is 2)

```
z = x + 5
x = 37
```

No. Again, $z = 42$ or $z = 7$.

- WAR is also known as a anti-dependency.
- But, we can modify this code to enable parallelization.

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x
z = x_copy + 5
x = 37
```

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x
z = x_copy + 5
x = 37
```

We can now run the last 2 lines in parallel.

- Induced a true dependency (RAW) between first 2 lines.
- Isn't that bad?

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x
z = x_copy + 5
x = 37
```

We can now run the last 2 lines in parallel.

- Induced a true dependency (RAW) between first 2 lines.
- Isn't that bad?

Not always:

```
z = very_long_function(x) + 5
x = very_long_calculation()
```

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5
z = x + 40
```

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5
z = x + 40
```

Nope, $z = 42$ or $z = 7$.

- WAW is also known as an output dependency.
- We can remove this dependency (like WAR):

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5
z = x + 40
```

Nope, z = 42 or z = 7.

- WAW is also known as an output dependency.
- We can remove this dependency (like WAR):

```
z_copy = x + 5
z = x + 40
```

Summary of Memory-carried Dependencies

		Second Access	
		Read	**Write**
First Access	**Read**	No Dependency Read After Read (RAR)	Anti-dependency Write After Read (WAR)
	Write	True Dependency Read After Write (RAW)	Output Dependency Write After Write (WAW)