Lecture 05 - `restrict` Keyword, Race Conditions and More Synchronization

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

January 13, 2012

Previously

- Conditions where you would make multiple processes instead of threads
- How to create, exit and join POSIX threads
- Remember, they are 1:1 with kernel threads and can run in parallel on multiple CPUs
- The difference between joinable/detached threads
- Mutex usage

Quick Blurb on Mutexes

- Mutexes simply ensure that if you succeed in calling `lock` with a certain mutex, `m1`, you will have exclusive access to `m1` until you `unlock` it
- Other calls to `lock` with the same mutex, `m1`, will wait until it's available
- If you want background on selection algorithms, look at Lamport's bakery algorithm, but you don't have to know them for this course
- Our focus is on how to use them correctly

Three Address Code

- A representation of intermediate code used by compilers, mostly used for analysis and optimization
- Statements represent one fundamental operation (for the most part, we can consider each operation atomic)
- Useful to reason about data races and easier to read than assembly (as long as you seperate out memory reads/writes)
- Statements have the form:
  $result := operand_1 \; operator \; operand_2$

GIMPLE

- GIMPLE is the three address code used by `gcc`
- To see the GIMPLE representation of your compilation 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 this if you want to reason about your code at a low-level without having to read assembly

Overview of `restrict`

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

- For C99 standard in `gcc` use the `-std=c99` flag

- If you declare a pointer with `restrict`, you are ensuring to 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)

- If you have a bunch of pointers declared with `restrict`, you are saying that these will never point to the same data
- Below is the Wikipedia example, would declaring all these pointers as `restrict` generate better code?

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

Example of restrict (2)

- Let's look at the GIMPLE instead

```
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;
```

- Is there any operation here that could be left out if all the pointers represent different data?

Example of `restrict` (3)

- If `ptrA` and `val` are different pointers, you don't have to reload the data on **line 6**
- Otherwise you would since you could call `updatePtrs(&x, &y, &x);`
- If you change the arguments to, you will get the optimized version

```
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 declare as `restrict`)
- The compiler is not able to know whether pointers alias, so you must provide this
- This allows the compiler to do better optimization for your code (and therefore run faster)
- Caveat: don't lie to the compiler, or else you will get undefined behaviour
- Aside: this not the same as `const`, `const` data can still be changed through a different pointer

Race Conditions

- Recall, a race happens when you have two concurrent accesses to the same state, at least one of which is a **write**
- This is a problem because the final state will not be the same as running one access to completion and then the other
- We should be worried about race conditions between any variables which are shared between threads

Example Data Race (1)

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

void* run1(void* arg)
{
    int* x = (int*) arg;
    *x += 1;
}

void* run2(void* arg)
{
    int* x = (int*) arg;
    *x += 2;
}
```

Example Data Race (2)

```
int main(int argc, char *argv[])
{
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_join(t1, NULL);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}
```

- Do we have a data race? Why or why not?

Example Data Race (2)

```
int main(int argc, char *argv[])
{
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_join(t1, NULL);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}
```

- Do we have a data race? Why or why not?
- No, we don't. Only one thread is active at a time

Example Data Race (3)

```
int main(int argc, char *argv[])
{
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}
```

- Do we have a data race now? Why or why not?

Example Data Race (3)

```
int main(int argc, char *argv[])
{
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}
```

- Do we have a data race now? Why or why not?
- Yes, we do. We have 2 threads trying to access the same data

Consequence of Example Data Race?

- What are the possible outputs? (initially `*x` is 1)

```
run1                     run2
D.1 = *x;                D.1 = *x;
D.2 = D.1 + 1;           D.2 = D.1 + 2
*x = D.2;                *x = D.2;
```

- Again, the important times to worry about in a data race are the memory reads and writes

Outcome of Example Data Race

- Let's call the read and write from `run1` R1 and W1 (R2 and W2 from `run2`)
- The read, in a function, has to come before it's write

All possible orderings:

Order				`*x`
R1	W1	R2	W2	4
R1	R2	W1	W2	3
R1	R2	W2	W1	2
R2	W2	R1	W1	4
R2	R1	W2	W1	2
R2	R1	W1	W2	3

Detecting Data Races Automatically

- There are also tools to help you find data races in your program
- `helgrind` is one such tool, it runs your program on top of it and analyzes it (it will however, cause a large slowdown)
- Run with `valgrind --tool=helgrind <prog>`
- It will warn you of possible data races along with locations
- For useful debugging locations, compile with debugging information `-g` flag for `gcc`

Helgrind Output for Example

```
==5036== Possible data race during read of size 4 at
         0x53F2040 by thread #3
==5036== Locks held: none
==5036==    at 0x400710: run2 (in datarace.c:14)
...
==5036==
==5036== This conflicts with a previous write of size 4 by
         thread #2
==5036== Locks held: none
==5036==    at 0x400700: run1 (in datarace.c:8)
...
==5036==
==5036== Address 0x53F2040 is 0 bytes inside a block of size
         4 alloc'd
...
==5036==    by 0x4005AE: main (in datarace.c:19)
```

Spinlocks

- Functionally equivalent to `mutex`
- To use in Pthread's, use `pthread_spinlock_t`, `pthread_spin_lock`/`pthread_spin_trylock` and friends
- Until mutexes, spinlocks will repeatedly try the lock and will not put the thread to sleep (so it can be used for another task)
- Good to use if your protected code is short
- Mutexes may be implemented as a combination between spinning/sleeping (spin for a short time, then sleep)

Read-Write Locks

- If there are only reads, there's no datarace
- It might be the case that writes are rare
- With mutexes/spinlocks, you have to lock the data, even for a read since you don't know if a write could happen
- But, most of the time, reads can happen in parallel, as long as there's no write
- Multiple threads can hold a read lock (`pthread_rwlock_rdlock`), but only one thread may hold a write lock (`pthread_rwlock_wrlock`) and will wait until the current readers are done

Semaphores

- Semaphores have a `value` and can be used for signalling between threads (initially set to any specified value)
- There may be as many threads with the semaphore as `value` allows
- Two fundamental operations `wait` and `post`
- `wait` is like `lock`, it decrements the value
  - If the value is 0, it will wait until the value is greater than 0
- `post` is like `unlock`, it increments the value

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 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 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 this 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, 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

- `value` is initially 1
- `p2` hits it's `sem_wait` first and succeeds
- `value` is now 0 and `p2` prints "Thread 2"
- It doesn't matter if `p1` happens first, it would just increase `value` to 2

Semaphores for Signalling

- `value` is initially 1
- `p2` hits it's `sem_wait` first and succeeds
- `value` is now 0 and `p2` prints "Thread 2"
- It doesn't matter if `p1` happens first, it would just increase `value` to 2
- The solution is to set the initial `value` to 0
- In this case, if `p2` hits it's `sem_wait` first it will wait until `p1` posts, after it prints "Thread 1"