Locks

Glenn Bruns
CSUMB

Lecture Objectives

After this lecture, you should be able to:

- Explain why locks are used
- Define race condition, mutual exclusion, and critical section
- Add pthread locks to C code to get mutual exclusion
- Explain why special hardware instructions are often used for implementing locks

Recap: threaded counter code

```
static volatile int counter = 0;

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;
}

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;
}
```

shared variable

each thread runs mythread()

Compiling and running

```
$ gcc -o t1 t1.c -lpthread
$ ./t1
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 9087710)
$
$ ./t1
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 11928697)
$
```

Race conditions and mutual exclusion

A **critical section** is a piece of code that accesses a shared resource, such as a shared variable.

Mutual exclusion is when at most one thread at a time can be in a critical section.

A **race condition** exists if the output of a multi-threaded program depends on the scheduling or relative speed of the threads.

You avoid race conditions by preventing more than one thread from reading or writing shared data at the same time

Locks

Locks are used to prevent more than one thread from executing a section of code

- A lock can be in one of two states: busy or free
- A lock is initially free
- Use operation **lock** to get the lock
- Use operation **unlock** to release the lock
- Only one thread at a time can hold a particular lock.

‘lock’ is sometimes called ‘acquire’
‘unlock’ is sometimes called ‘free’

(some text here from *Operating Systems: Principle and Practice*, Anderson and Dahlin)

lock() is a blocking operation

lock() is a very strange function call

- it doesn't compute anything
- it doesn't necessarily return right away
- a thread that calls lock() might be "blocked" as it waits for lock() to return
- the whole point of using lock() is to control whether a thread can proceed or not

How is blocking a thread related to thread scheduling?

Pthread locks

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

mutex: a pointer to a pthread_mutex_t structure

"mutex" is short for "mutual exclusion"

Mutual exclusion using locks

```
pthread_mutex_t lock;                                    ← declare lock

void *mythread(void *arg) {
   int i;
   for (i = 0; i < 1e7; i++) {
      Pthread_mutex_lock(&lock);                         ← acquire lock
      counter = counter + 1;
      Pthread_mutex_unlock(&lock);                       ← release lock
   }
   return NULL;
}

int main(int argc, char *argv[]) {
   Pthread_mutex_init(&lock, NULL);                      ← initialize lock
   pthread_t p1, p2;
   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;
}
```

Running counter with pthreads lock

```
$ gcc -o t1-with-lock t1-with-lock.c –lpthread
$
$ ./t1-with-lock
main: begin (counter = 0)
main: done with both (counter = 20000000)
$
```

How are locks implemented?

```
typedef struct __lock_t {
  int flag;
} lock_t;

void init(lock_t *mutex) {
  mutex->flag = 0;
}

void lock(lock_t *mutex) {
  // what goes here?
}

void unlock(lock_t *mutex) {
  mutex->flag = 0;
}
```

Here's idea for a lock implementation.

Remember: lock() is a blocking operation.

Does this work?

```
typedef struct __lock_t {
  int flag;
} lock_t;

void init(lock_t *mutex) {
  mutex->flag = 0;
}

void lock(lock_t *mutex) {
  while (mutex->flag == 1)
      ;
  mutex->flag = 1;
}

void unlock(lock_t *mutex) {
  mutex->flag = 0;
}
```

How locks are implemented

- Pure software implementations
  - examples: Dekker’s algorithm, Peterson’s algorithm
  - problems:
    - very tricky algorithms
    - many classic algorithms don’t work on modern hardware
- Software + hardware support
  - special CPU instructions
    - test-and-set, compare-and-swap, fetch-and-add, …

Evaluating lock implementations

- **Correctness** – provide mutual exclusion
- **Fairness** – if a thread requests the lock, it will eventually get it
- **Performance** – overhead of locking should be minimized

<u>Execution times:</u>
counter code without locks: 0.33 s
counter code with pthreads locks: 4.11 s

Summary

- Key concepts in concurrency are race condition, critical section, and mutual exclusion
- Locks are a basic ingredient of multi-thread programming
- pthreads API: `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions
- Hard to build locks without hardware support
- We want lock implementations to be correct, fair, and have good performance

Bonus content

A seminal paper on concurrency:

Edsger Dijkstra, “Cooperating Sequential Processes”, 1968.

Available at:

cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF