CISC 372: Parallel Computing
Threads, part 2: data races, mutexes, and critical sections

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

Next example: `add_pthread.c`

- should sum integers from 1 to n
  - where n is the number of threads created
  - n is the command line arg
- result should be $n(n+1)/2$

add_pthread.c

```
int nthreads; // number of threads to create
int sum = 0;
void* hello(void* arg) {
  int * tidp = (int*)arg;
  sum += (*tidp)+1;
  return NULL;
}
int main(int argc, char *argv[]) {
  nthreads = atoi(argv[1]);
  pthread_t threads[nthreads];
  int tids[nthreads];
  for (int i=0; i<nthreads; i++) tids[i] = i;
  for (int i=0; i<nthreads; i++)
    pthread_create(threads + i, NULL, hello, tids + i);
  for (int i=0; i<nthreads; i++)
    pthread_join(threads[i], NULL);
  printf("The sum is %d\n", sum);
}
```

Testing add_pthread.c

```
basie:add siegel$ ./add_pthread.exec 20
The sum is 210
basie:add siegel$ ./add_pthread.exec 20
The sum is 210
basie:add siegel$ ./add_pthread.exec 20
The sum is 210
basie:add siegel$ ./add_pthread.exec 20
The sum is 210
```

The program must be correct, right?

Testing add_pthread.c, cont.

Try it 1000 times. . .

```
basie:add siegel$ for i in {1..1000}; do ./add_pthread.exec 20; done
The sum is 210
The sum is 186
The sum is 210
The sum is 210
The sum is 208
   ...
```

Hmmm...

Testing `add_pthread.c`, cont.

Better yet, collate the results:

```
for i in {1..1000}; do ./add_pthread.exec 20; done | sort -n | uniq -c
```

```
      1 The sum is 176
      1 The sum is 178
      1 The sum is 179
      1 The sum is 184
      2 The sum is 188
      3 The sum is 189
      7 The sum is 190
     11 The sum is 191
     12 The sum is 192
      7 The sum is 193
      8 The sum is 194
     11 The sum is 195
      7 The sum is 196
```

```
      7 The sum is 197
     13 The sum is 198
      9 The sum is 199
     11 The sum is 200
      9 The sum is 201
     11 The sum is 202
      9 The sum is 203
     10 The sum is 204
     10 The sum is 205
      1 The sum is 206
    158 The sum is 208
     91 The sum is 209
    589 The sum is 210
```

What went wrong?

- a data race
- `x+=y` really consists of several machine-level steps:
  - read `x` into a register
  - read `y` into a register
  - compute the sum and store it in `x`
- if two threads are executing concurrently, this might happen:
  1. thread 1: read `x`
  2. thread 2: read `x`
  3. thread 1: read `y`
  4. thread 2: read `y`
  5. thread 1: compute sum and store it in `x`
  6. thread 2: compute sum and store it in `x`
- the contribution from thread 1 is overwritten!
- worse:
  - total garbage could be written to `x`
  - compiler could change code in some unpredictable way based on assumption there is no race

Data races

A *data race* occurs whenever

- two threads can access the same memory location concurrently, and
- at least one of the accesses is a write.

Two kinds of data races:

- `read-write`: one thread reads and the other writes, or
- `write-write`: both threads write

A data race in a Pthreads program results in undefined behavior.

The program could do "anything" (crash, return weird results,...)

You can not assume the value written will be one of the two possible "reasonable" values.

- it is the programmer's responsibility to avoid all data races

Mutexes

- mutex = "mutual exclusion lock"
- used to guarantee that at most one thread can access a shared object at any time
- many variations possible; for now, use default settings
- supports "lock" and "unlock" operations
- in this example, a single mutex is used to control access to `sum`
- each thread obtains the lock before reading and modifying `sum`
- ...and releases lock when it is done
- a thread will block when trying to obtain the lock if another thread owns the lock

add_pthread_fix.c

```
int nthreads, sum = 0;
pthread_mutex_t mutexsum;
void* hello(void* arg) {
  int * tidp = (int*)arg;
  pthread_mutex_lock(&mutexsum);
  sum += (*tidp)+1;
  pthread_mutex_unlock(&mutexsum);
  return NULL;
}
int main (int argc, char *argv[]) {
  nthreads = atoi(argv[1]);
  pthread_t threads[nthreads];
  int tids[nthreads];
  pthread_mutex_init(&mutexsum, NULL);
  for (int i=0; i<nthreads; i++) tids[i] = i;
  for (int i=0; i<nthreads; i++) pthread_create(threads + i, NULL, hello, tids + i);
  for (int i=0; i<nthreads; i++) pthread_join(threads[i], NULL);
  pthread_mutex_destroy(&mutexsum);
  printf("The sum is %d\n", sum);
}
```

Test add_pthread_fix.c

```
for i in {1..1000}; do ./add_pthread_fix.exec 20; done | sort -n | uniq -c
```

```
1000 The sum is 210
```

Mutex semantics

- the semantics of a concurrency primitive can be specified by
  1. a model of the state, and
  2. specifying the atomic operations that change the state
- mutex state
  - the state is either a reference to one thread or `NULL`
  - the thread that "owns" the locked lock, or the lock is open
- atomic actions
  - `lock`
    - if the state is `NULL`, a thread t may execute this action and the state becomes t
    - if the state is non-null, t will block
  - `unlock`: if the state is t then t may execute this action and state becomes `NULL`
- all other actions: undefined
  - a thread that does not own the lock attempts to unlock it
  - a thread that owns the lock attempts to lock it
- this is all for basic mutexes; other variations are more lenient

Using mutexes

- a mutex is typically used to control access to some shared data
- this is purely a programming convention
  - no formal relationship between the mutex and the data
  - programmer should document the relationship clearly
- typical control flow:
  1. obtain lock;
  2. access the shared data;
  3. release the lock;
- do this wherever the data is accessed!
  - if you miss one case, all bets are off

Pthreads mutex interface

- type
  - `pthread_mutex_t` : opaque handle to a mutex
- functions

```
int pthread_mutex_init    ( pthread_mutex_t * mutex,
                            pthread_mutexattr_t * attr );
int pthread_mutex_destroy ( pthread_mutex_t * mutex    );
int pthread_mutex_lock    ( pthread_mutex_t * mutex    );
int pthread_mutex_unlock  ( pthread_mutex_t * mutex    );
```

- use `NULL` for the attribute argument for now
- all functions return error code (0=success)

Mutexes and memory

- different fields of a struct occupy distinct memory locations
- different cells of an array are different memory locations
- a mutex is not required if each thread is accessing its own section of the array/struct
- however performance problems are possible
  - the cache system may have to constantly reload the line containing your cell
  - if another thread is accessing a nearby cell in the same line

The Critical Section Problem

- another common concurrent program pattern:

```
while (true) {
  enter critical section
  CRITICAL SECTION: only one thread at a time
  exit critical section
  NON-CRITICAL SECTION: any number of threads can execute
}
```

- the problem: design entrance/exit protocols (and appropriate state) such that
  1. deadlock-free
  2. no unnecessary delay
     - a thread that is trying to enter the critical section when no one else is in the critical section will enter without delay
  3. mutual exclusion: at most one thread in critical section at any time
  4. fairness (or, no starvation)
     - if a thread is trying to enter the critical section then eventually it will succeed (after some finite delay)

Critical section: desired properties

1. deadlock-free
2. no unnecessary delay
   - a thread that is trying to enter the critical section when no one else is in the critical section will enter without delay
3. mutual exclusion: at most one thread in critical section at any time
4. fairness (or, no starvation)
   - if a thread is trying to enter the critical section then eventually it will succeed (after some finite delay)

Solution #1: use a mutex

- see `crit_sec_mutex.c`
- which properties hold?
- 1–3 : yes
- What about 4? *Not necessarily.* This is a hard problem.
  - famous solutions: Lamport's bakery algorithm, Peterson's mutual exclusion algorithm