CMPUT 379 Lab

ETLC E1003: Tuesday, 5:00 – 7:50 PM.

Tianyu Zhang, Peiran Yao

CAB 311: Thursday, 2:00 – 4:50 PM.

Max Ellis, Aidan Bush

Last Week...

- Review on Threads and Thread safe data structures

- Assignment 2: Thread pool and MapReduce

Today's Lab

- Threading examples
- Fine and coarse grained locks

Threading examples credited to **Operating Systems: Three Easy Pieces**. Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
Arpaci-Dusseau Books. August, 2018 (Version 1.00)
https://github.com/remzi-arpacidusseau/ostep-code

Example: joining

- Suppose we'd like to wait until a thread finishes.
- And there is no **pthread_wait()**

Joining - Spin Lock

- Simple solution: use global variable and while loop
- But it wastes CPU

```c
void *child(void *arg) {
    printf("child\n");
    sleep(5);
    done = 1;
    return NULL;
}
```

```c
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    Pthread_create(&p, NULL, child,
NULL);
    while (done == 0)
    ; // spin
    printf("parent: end\n");
    return 0;
}
```

Joining - Condition Variable

- Fix: use condition variable to send signals

```
void *child(void *arg) {
    printf("child\n");
    sleep(1);
    Mutex_lock(&m);
    done = 1;
    Cond_signal(&c);
    Mutex_unlock(&m);
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    Pthread_create(&p, NULL, child,
NULL);
    Mutex_lock(&m);
    while (done == 0)
    Cond_wait(&c, &m); // releases
lock when going to sleep
    Mutex_unlock(&m);
    printf("parent: end\n");
    return 0;
}
```

Joining - Why Use Lock?

- **done** might be changed to 1, and the main thread will miss the signal before it starts waiting

```c
void *child(void *arg) {
    printf("child: begin\n");
    sleep(1);
    done = 1;                  // 3
    printf("child: signal\n");
    Cond_signal(&c);           // 4
    return NULL;
}
```

```c
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    Pthread_create(&p, NULL, child, NULL);   // 1
    Mutex_lock(&m);
    printf("parent: check condition\n");
    while (done == 0) {                       // 2
    sleep(2);
    printf("parent: wait to be
signalled...\n");
    Cond_wait(&c, &m);                        // 5
    }
    Mutex_unlock(&m);
    printf("parent: end\n");
    return 0;
}
```

Joining - Why Use **done**?

- The main thread may also miss the signal before waiting

```
void *child(void *arg) {
    printf("child: begin\n");
    Mutex_lock(&m);
    printf("child: signal\n");
    Cond_signal(&c);        2
    Mutex_unlock(&m);
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    Pthread_create(&p, NULL, child, NULL);
    sleep(2);                   1
    printf("parent: wait to be
signalled...\n");
    Mutex_lock(&m);
    Cond_wait(&c, &m);          3
    Mutex_unlock(&m);
    printf("parent: end\n");
    return 0;
}
```

Example: Producer-Consumer

- Some producers produce items and put them in a queue
- Some consumers take items out of queue and process
- The queue has a limited size

Producer-Consumer Challenges

- Some producers produce items and put them in a queue
- Some consumers take items out of queue and process
- The queue has a limited size

- Concurrent access to a data structure
- What to do when the queue is full / empty?

Producer-Consumer Solutions

- Concurrent access to a data structure
  - Use locks to protect it
- What to do when the queue is full?
  - Producers use condition variable to wait for the queue have spaces
  - Consumers notify producers when they removed items from a full queue
- What to do when the queue is empty?
  - Consumers use condition variable to wait for the queue to be filled
  - Producers notify consumers if there is a new item

Common Bugs - Atomicity Failure

- Check-then-use should be protected by locks.

Common Bugs - Atomicity Failure

- Check-then-use should be protected by locks.

```
void *thread1(void *arg) {
    printf("t1: before check\n");
    if (thd->proc_info) {
        printf("t1: after check\n");
        sleep(2);
        printf("t1: use!\n");
        printf("%d\n", thd->proc_info->pid);
    }
    return NULL;
}
```

Common Bugs - Atomicity Failure

- Check-then-use should be protected by locks.

```
void *thread2(void *arg) {
    printf("                    t2: begin\n");
    sleep(1); // change to 5 to make the code "work"...
    printf("                    t2: set to NULL\n");
    thd->proc_info = NULL;
    return NULL;
}
```

Common Bugs - Atomicity Failure

- Fix: add a lock to **proc_info**

Common Bugs - Ordering

- Two threads may have different order of execution

```
int main(int argc, char *argv[]) {
    printf("ordering: begin\n");
    mThread =
PR_CreateThread(mMain);
    PR_WaitThread(mThread);
    printf("ordering: end\n");
    return 0;
}
```

3

```
pr_thread_t *PR_CreateThread(void
*(*start_routine)(void *)) {
    pr_thread_t *p =
malloc(sizeof(pr_thread_t));
    p->State = PR_STATE_INIT;
    Pthread_create(&p->Tid, NULL,
start_routine, NULL);
    sleep(1);
    return p;
}
```

2

```
void *mMain(void *arg) {
    printf("mMain: begin\n");
    int mState = mThread->State;
    printf("mMain: state is %d\n", mState);
    return NULL;
}
```

1

Common Bugs - Ordering

- Two threads may have different order of execution
- Fix: use condition variables

```
void *mMain(void *arg) {
    printf("mMain: begin\n");
    // wait for thread structure to be
initialized
    Pthread_mutex_lock(&mtLock);
    while (mtInit == 0)
    Pthread_cond_wait(&mtCond, &mtLock);
    Pthread_mutex_unlock(&mtLock);

    int mState = mThread->State;
    printf("mMain: state is %d\n", mState);
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    printf("ordering: begin\n");
    mThread = PR_CreateThread(mMain);
    // signal: thread has been created,
and mThread initialized
    Pthread_mutex_lock(&mtLock);
    mtInit = 1;
    Pthread_cond_signal(&mtCond);
    Pthread_mutex_unlock(&mtLock)
    // ……….
}
```

Common Bugs - Deadlock

- When both threads are waiting for a resource that the other is holding

```c
void *thread1(void *arg) {
    printf("t1: begin\n");
    printf("t1: try to acquire L1...\n");
    Pthread_mutex_lock(&L1);   1
    printf("t1: L1 acquired\n");
    printf("t1: try to acquire L2...\n");
    Pthread_mutex_lock(&L2);   3
    printf("t1: L2 acquired\n");
    Pthread_mutex_unlock(&L1);
    Pthread_mutex_unlock(&L2);
    return NULL;
}
```

```c
void *thread2(void *arg) {
    printf("t2: begin\n");
    printf("t2: try to acquire L2...\n");
    Pthread_mutex_lock(&L2);   2
    printf("t2: L2 acquired\n");
    printf("t2: try to acquire L1...\n");
    Pthread_mutex_lock(&L1);   4
    printf("t2: L1 acquired\n");
    Pthread_mutex_unlock(&L1);
    Pthread_mutex_unlock(&L2);
    return NULL;
}
```

Common Bugs - Deadlock

- Fix: order of resource acquisition matters

Locking - Fine and Coarse Grained

- Fine grained
  - Locking small sections of a data structure
  - Allows only locking what needs to be
  - Usually faster

- Coarse grained
  - Locking larger sections of a data structure
  - Locking large sections of a data structure that are not modified
  - Usually slower

Advantages of Fine Grained Locks

- Allows multiple threads to access shared data more freely

- Improved efficiency from allowing more threads to access data in parallel

Example - Concurrent Array

- Suppose we wish to share an array across threads
- There are possible design choices
  - 1 global mutex
  - 1 local mutex per element
  - 1 mutex per block of element

Example - Concurrent Array

- Each thread randomly replaces an element

```
void* random_acccess(void * args) { // thread function
    unsigned seed = (unsigned) time(NULL); // rand() has a global lock. Use 
rand_r() instead.
    for (int i = 0; i < 1024000; i++) {
        replace(rand_r(&seed) % ARRAY_SIZE, rand_r(&seed));  // randomly 
replace an element
    }
    return NULL;
}
```

Example - Concurrent Array

- Lock the entire array (coarse)

```
int shared_array[ARRAY_SIZE];
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  // 1 mutex for the entire array
int replace(unsigned int index, int new_val) {
    int old_val;
    pthread_mutex_lock(&mutex);   // lock the whole array
    old_val = shared_array[index];
    shared_array[index] = new_val;
    pthread_mutex_unlock(&mutex);
    return old_val;
}
```

Example - Concurrent Array

- Lock single element (most fine-grained)

```
int shared_array[ARRAY_SIZE];
pthread_mutex_t mutex[ARRAY_SIZE]; // one mutex per element


int replace(unsigned int index, int new_val) {
    int old_val;
    pthread_mutex_lock(&mutex[index]);  // only lock the element
    old_val = shared_array[index];
    shared_array[index] = new_val;
    pthread_mutex_unlock(&mutex[index]);
    return old_val;
}
```

Example - Concurrent Array

- Comparing the efficiency
- 32 elements
- 16 threads

```
$ time ./fine
./fine  1.16s user 0.67s system 353% cpu 0.518 total

$ time ./coarse
./coarse  1.46s user 5.45s system 343% cpu 2.008 total
```

FYI

- When you use a lab computer (using SSH or physically be there), please make sure to clean-up your session before logging out!

- Check any running processes by command “***ps***”, and kill all processes you created.

- We received a warning from the department regarding this. Lots of people put “dragonshell” to the background by “Ctrl+Z” and then log out with no cleanup.