15-213 Recitation 14: Threads and Synchronization

18 April 2016
Ralf Brown and the 15-213 staff
Agenda

- Reminders
- Threads Revisited
- Synchronization

These aren't the only threads that can get tangled....

Image: SilviaStoedter @ pixabay.com
Reminders

- Proxylab is due a week from tomorrow
- there are no grace days!
- make your code robust against unexpected inputs
Thread Life Cycle

Thread1 → pthread_create → Thread1

Thread2 → communication and synch → thread_func runs → Thread2

Thread1 → pthread_join → Thread1

Thread2 → return or pthread_exit → resources cleaned up
Thread Life Cycle

Thread1

pthread_create

Thread2

detach thread

Thread2

thread_func runs

communication and synch

Thread2

return or pthread_exit

if you don't need a return value

pthread_detach(pthread_self())

resources automatically cleaned up without join
Using Threads

- Let's sum the elements of an array

```c
int *nums;

int sum_array(size_t n) {
    int sum = 0;
    // iterate over the elements of the array
    for (int i = 0; i < n; i++)
        sum += nums[i];
    return sum;
}
```
Using Threads: Summing an Array

Multi-threaded

// the main thread function – sum a section of the array
void *thread_fun(void *vargp) {
    int myid = *((int*)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;
    int sum = 0;
    for (i = start; i < end; i++)
        sum += nums[i];
    psum[myid] = sum;
    return NULL;
}
Using Threads: Summing an Array

```c
int sum_array(int nelems) {
    int sum = 0;
    // figure out how big the sections should be
    nelems_per_thread = nelems / nthreads;
    // create threads
    for (i = 0; i < nthreads; i++) {
        myid[i] = i;
        Pthread_create(&tid[i], NULL, thread_fun, &myid[i]);
    }
    // wait for the threads to finish
    for (i = 0; i < nthreads; i++)
        Pthread_join(tid[i], NULL);
    // collect results
    for (i = 0; i < nthreads; i++)
        sum += psum[i];
    // add leftover elements
    for (e = nthreads * nelems_per_thread; e < nelems; e++)
        sum += nums[e];
    return sum;
}
```

why use an array for myid?
volatile int total = 0;
void *incr(void *ptr) {
    for (int i = 0; i < *((int*)ptr); i++)
        total++;
    return NULL;
}

int main() {
    pthread_t tids[NTHREADS];
    int y = NINCR;
    for (int i = 0; i < NTHREADS; i++)
        Pthread_create(&tids[i], NULL, incr, &y);
    for (int i = 0; i < NTHREADS; i++)
        Pthread_join(tids[i], NULL);
    printf("total is: %d\n", total);
    return 0;
}
Critical Sections and Shared Variables

```c
volatile int total = 0;
void *incr(void *ptr) {
    for (int i = 0; i < *((int*)ptr); i++)
        total++;
    return NULL;
}

int main() {
    pthread_t tids[NTHREADS];
    int y = NINCR;
    for (int i = 0; i < NTHREADS; i++)
        Pthread_create(&tids[i], NULL, incr, &y);
    for (int i = 0; i < NTHREADS; i++)
        Pthread_join(tids[i], NULL);
    printf("total is: %d\n", total);
    return 0;
}
```

We have NTHREADS incrementing the total NINCR times each but the total may be much less than NINCR*NTHREADS!
Critical Sections and Shared Variables

```c
volatile int total = 0;
void *incr(void *ptr) {
    for (int i = 0; i < *((int*)ptr); i++)
        total++;
    return NULL;
}

int main() {
    pthread_t tids[NTHREADS];
    int y = NINCR;
    for (int i = 0; i < NTHREADS; i++)
        Pthread_create(&tids[i], NULL, incr, &y);
    for (int i = 0; i < NTHREADS; i++)
        Pthread_join(tids[i], NULL);
    printf("total is: %d\n", total);
    return 0;
}
```
What Happened?

thread 1:
- Reads: 0
- Writes: 1

thread 2:
- Reads: 1
- Writes: 2

thread 3:
- Reads: 0
- Writes: 1

thread 1:
- Reads: 2
- Writes: 3

thread 1:
- Reads: 2
- Writes: 3

thread 2:
- Reads: 1
- Writes: 2

thread 3:
- Reads: 1
- Writes: 2
Threads: Mutual Exclusion

- Need to prevent multiple threads from accessing the same resource at the same time
- In our `sum_array` example, we managed to avoid simultaneous access by giving each thread a separate section of the array
- In general, we'll need a way to temporarily stop a thread while another is accessing the resource it wants to use
  - we use a semaphore or mutex
  - trying to lock a mutex while it is already locked blocks the thread until it is unlocked by the other thread that had already locked it
  - the code between a pair of lock and unlock calls is a critical section.
Semaphores

- Special counters with an invariant: their value is never negative
- two *atomic* operations
  - \( P(s) \) tries to decrement the counter \( s \) (locking the resource), and puts the thread to sleep if the counter is already zero
  - \( V(s) \) increments the counter (freeing the resource) and wakes any thread that may be waiting on \( s \)
- Mutexes are a subclass of semaphores
  - their value is always either 0 or 1
  - often faster than semaphores because the binary value permits optimizations
Using Semaphores

- **Limited resource**
  - initialize count to total number of items available
  - decrement just before starting to use one of the items
  - increment when done using the item

- **Producer-Consumer**
  - initialize count to zero
  - producer generates a new item, then increments semaphore
  - consumer decrements semaphore, then retrieves item
Protecting Shared Resources with a Semaphore

- Suspend execution of thread until resource is “acquired”

```c
volatile int total = 0;
sem_t sem;

void *incr(void *ptr) {
    for (int i = 0; i < *ptr; i++) {
        sem_wait(&sem);
        total++;
        // CRITICAL SECTION
        sem_post(&sem);
    }
    return NULL;
}

int main() {
    ...
    sem_init(&sem, 0, 1);
}
```

remember to initialize the semaphore first!
Protecting Shared Resources with a Mutex

- Can use a mutex just like a semaphore initialized to 1

```c
volatile int total = 0;
pthread_mutex_t M;

void *incr(void *ptr) {
    for (int i = 0; i < *ptr; i++) {
        pthread_mutex_lock(&M);
        total++;
        // CRITICAL SECTION
        pthread_mutex_unlock(&M);
    }
    return NULL;
}

int main() {
    ...  
pthread_mutex_init(&M);
    
    remember to initialize the mutex first!
}
Problem Solved?

- Sort of...
- Locks in threads are *slow*.
  - they involve OS calls and *uncached* memory accesses
- Only one instance of the critical section can run at once
  - your eight-core CPU effectively becomes a single-core CPU
Other Solutions

- avoid shared modifiable memory if at all possible
  - (shared read-only memory is OK)
- use a more sophisticated thread synchronization model
  - reader/writer
  - producer-consumer
Problem: Deadlock

- What's about to happen?

Process 1

```c
pthread_mutex_lock(&A);
...
pthread_mutex_lock(&B);
```

Process 2

```c
pthread_mutex_lock(&B);
...
pthread_mutex_lock(&A);
```
Problem: Deadlock

Process 1
pthread_mutex_lock(&A);
...  
pthread_mutex_lock(&B);

already locked, so we block until process 2 unlocks it

Mutex A

Mutex B

Process 2
pthread_mutex_lock(&B);
...  
pthread_mutex_lock(&A);

already locked, so we block until process 1 unlocks it
Readers and Writers

Diagram by
Readers and Writers

cache

reader

reader

reader

writer

reader
Readers and Writers

reader

reader

reader

writer

reader

reader
Readers and Writers

- cache
- writer
- reader

The diagram illustrates the interaction between readers and writers accessing a cache.
Readers and Writers

- Readers:
  - Reader 1
  - Reader 2
  - Reader 3

- Writer

- Cache

- Connections:
  - Writer to Cache
  - Cache to Readers
Starvation

- If there are many readers, they may keep writers from acquiring the resource because someone is always reading
  - the writer is being **starved** of the resource
- Important to minimize the amount of time you hold a lock
Read(ers)-Write(r) Lock

- allows a single writer xor multiple concurrent readers
- `int pthread_rwlock_init(pthread_rwlock *lock, const pthread_rwlockattr_t *attr);`
- lock for reading; blocks if someone is currently (attempting to) write
- `int pthread_rwlock_wrlock(pthread_rwlock *lock);`
- lock for writing; blocks until all current users finish
- `int pthread_rwlock_unlock(pthread_rwlock *lock);`
- `int pthread_rwlock_destroy(pthread_rwlock *lock);`
Problem: Livelock

- Much like deadlock, but the processes/threads spin indefinitely instead of hanging
- Think of two people trying to get past each other in a hallway
  - both move the same way
  - then both move the other way at the same time
  - awkward dance continues...
- Often the result of attempting to compensate for potential deadlock
  - spinning on a trylock() to avoid hanging on a lock()
Which Lock Do I Use?

- Consider **what** is shared and what type of **access** is desired
  - only one thread at a time allowed
    - ?
  - more than one instance of a shared resource is available
    - ?
  - multiple threads can read concurrently, but only one may write at a time
    - ?
Which Lock Do I Use?

- Consider **what** is shared and what type of **access** is desired
  - only one thread at a time allowed
    - **mutex** example: global count variable
  - more than one instance of a shared resource is available
    - **semaphore** example: multiple free slots in a shared buffer
  - multiple threads can read concurrently, but only one may write at a time
    - **readers-writer lock** example: lookup from global list, web-proxy cache