Pthread Synchronization

Lecture 9
March 10, 2025

To Dos

Reading for next time

Midterm next class

Questions about program 4?

Pthread Locks

```
int pthread_mutex_init(pthread_mutex_t *mutex_p,
                       const pthread_mutexattr_t *attr_p)
```

- To create a lock, need to
  - Declare a `pthread_mutex_t` variable
    - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER`
  - Call `pthread_mutex_init()` on that variable
    - Second argument sets attributes different from default and can be NULL

Pthread mutexes: locking and unlocking

```
int pthread_mutex_lock(pthread_mutex_t *mutex_p)

int pthread_mutex_unlock(pthread_mutex_t *mutex_p)
```

- `pthread_mutex_lock()` blocks if someone already has lock
  - Returns 0 on success and thread has lock
- Pthread_mutex_unlock() returns 0 on success
  - Must have lock for this to be successful

Pthread Condition Variables

```
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr)
```

- To create a condition variable, need to
  - Declare a `pthread_cond_t` variable
    - `pthread_cond_t cond = PTHREAD_COND_INITIALIZER`
  - Call `pthread_cond_init()` on that variable
    - Second argument sets attributes different from default and can be NULL

Pthread CV: wait/signal/broadcast

```
int pthread_cond_wait(pthread_cond_t*p, pthread_mutex_t *mutex)
int pthread_cond_signal(pthread_cond_t *cond_var_p)
int pthread_cond_broadcast(pthread_cond_t *cond_var_p)
```

```
int num_threads;
int val;

void *Hello(void *rank)
{
  int tmp = val+1;
  if((long)rank % 2 ==  0)
    sleep(1);
  val = tmp;
  return NULL;
}
```

```
int main(int argc, char *argv[])
{
  long pthread;
  num_threads = 4;
  val = 0;
  pthread_t ids[num_threads];

  for(long i = 0; i < num_threads; i++){
    pthread_create(&ids[i], NULL, Hello, (void*)i);
  }
  for(int i = 0; i < num_threads; i++){
    pthread_join(ids[i], NULL);
  }
  printf("Value of val %d\n", val);
  return 0;
}
```

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int num_threads;
int val;

void *Hello(void *rank)
{
  pthread_mutex_lock(&mutex);
  int tmp = val+1;
  if((long)rank % 2 ==  0)
    sleep(1);
  val = tmp;
  pthread_mutex_unlock(&mutex);
  return NULL;
}
```

```
int main(int argc, char *argv[])
{
  long pthread;
  num_threads = 4;
  val = 0;
  pthread_t ids[num_threads];
  pthread_mutex_init(&mutex, NULL);

  for(long i = 0; i < num_threads; i++){
    pthread_create(&ids[i], NULL, Hello,
                   (void*)i);
  }
  for(int i = 0; i < num_threads; i++){
    pthread_join(ids[i], NULL);
  }
  printf("Value of val %d\n", val);
  return 0;
}
```

Semaphores

- Semaphore has a non-negative integer value
  - `P()` atomically waits for value to become > 0, then decrements
  - `V()` atomically increments value (waking up waiter if needed)
- Semaphores are like integers except:
  - Only operations are `P` and `V`
  - Operations are atomic
    - If value is 1, two P's will result in value 0 and one waiter
- Semaphores are useful for
  - mutual exclusion and general waiting for another thread to do something (e.g., fork/join)

How Can We Use a Semaphore to Create Mutex?

Example from Earlier: Bounded Buffer

```
tryget() {
    item = NULL;
    lock.acquire();
    if (front < tail) {
      item=buf[front % MAX];
      front++;
    }
    lock.release();
    return item;
}
```

```
tryput(item) {
    lock.acquire();
    if((tail - front)< MAX){
      buf[tail % MAX]=item;
      tail++;
    }
    lock.release();
}
```

Initially: `front = tail = 0; lock = FREE; MAX` is buffer capacity

Semaphore Bounded Buffer

```
get() {
  fullSlots.P();
  mutex.P();
  item = buf[front % MAX];
  front++;
  mutex.V();
  emptySlots.V();
  return item;
}
```

```
put(item) {
  emptySlots.P();
  mutex.P();
  buf[last % MAX] = item;
  last++;
  mutex.V();
  fullSlots.V();
}
```

Initially: `front = last = 0;` `MAX` is buffer capacity

`mutex = 1; emptySlots = MAX; fullSlots = 0;`

Using Semaphores w/ Pthreads

- Semaphores not part of pthreads
- `#include <semaphore.h>`
- Type is `sem_t`
- `int sem_init(sem_t* semaphore_p, int shared, unsigned val)`
  - `shared` : whether shared across processes. Should set to 0.
  - `val` : initial value
- `int sem_post(sem_t* semaphore_p) // Up/V`
- `int sem_wait(sem_t* semaphore_p) // Down/P`

```c
#include <semaphore.h>

int num_threads;
int val;
sem_t semaphore;

void *Hello(void *rank)
{
  sem_wait(&semaphore);
  val++;
  sem_post(&semaphore);
  return NULL;
}
```

```c
int main(int argc, char *argv[])
{
  long pthread;
  num_threads = 4;
  val = 0;
  pthread_t ids[num_threads];

  sem_init(&semaphore, 0, 1);

  for(long i = 0; i < num_threads; i++){
    pthread_create(&ids[i], NULL, Hello,
                   (void*)i);
  }

  for(int i = 0; i < num_threads; i++){
    pthread_join(ids[i], NULL);
  }

  printf("Value of val %d\n", val);
  return 0;
}
```

Barriers

- Point of synchronization that all threads must reach before any proceed
- No implementation in pthreads
- Incredibly useful for parallel codes where all threads work independently and then combine results for next stage
    - Functionally may do something akin to MPI's reduce, gather, or scatter functionality at the end of a barrier

Improving Lock Performance

- Locking granularity
  - Coarse grained: less concurrency
  - Fine grained: more concurrency, often more locks, often harder to keep track of everything
- Try locks
  - Before trying to get a lock, see if it's already in use
  - Trylocks attempt to get the lock, but return instantly if already in use
    - `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
- Sometimes you have data that will be read lots by many different threads but updated rarely
  - Don't want to serialize the threads when they're just reading the data
  - Still need to insure correct updates

Read-Write Locks

- Multiple readers grab read locks and can access shared data simultaneously
  - Threads that want to write must wait until all readers have released read locks
- Writer lock provides exclusive access to shared data
  - Only 1 writer at a time
  - No readers allowed when there is a writer

- How does write frequency impact performance?

Use of Read-Write Locks?

Read-Write Locks

- Type is `pthread_rwlock_t`
- Initialize variable to `PTHREAD_RWLOCK_INITIALIZER`
- `int pthread_rwlock_init(pthread_rwlock_t* rwlock)`

- `int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);`
- `int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);`

- `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);`
- There are try versions of both the read and write locks

Thread Safety

- Libraries frequently have static or global variables declared in them
- When you use threaded code, access to those variables has potential to cause race conditions
- Need to only use thread-safe versions of libraries when writing multithreaded programs

What do you remember about caches?