Lecture 14: Locks and Condition Variables

Operating Systems

Content taken from: https://pages.cs.wisc.edu/~remzi/OSTEP/

https://www.cse.iitb.ac.in/~mythili/os/

Last Class

- Process vs threads
    - Parent process spawns a child process using fork().
        - No sharing of memory between the parent and child
    - Parent process spawns a new thread using pthread_create()
        - Parent and new thread share the same address space (except the stack)
    - Threads are like separate processes except they share the same address space
- Why threads? Parallelism and effective usage of CPU
- OS schedules threads similar to how it schedules processes

Last Class

- Threads of the same process share data
- **Race Conditions:**
  - Two threads trying to access a **critical section** (updating a shared variable)
  - Leads to indeterminate/unexpected results
- **Mutual Exclusion:**
  - Ensure that only one thread at a time executes the critical section
  - Requires that the critical section executes **atomically** (cannot be interrupted by any interrupts etc.)
  - How? Using Locks

Locks

- For each critical section, use a lock
- Each thread need to get the lock before executing the critical section
- At a time, only one thread can have the lock
- If the lock is held by another thread, then all other threads wait until the lock is released
- Modern architectures provide hardware atomic instructions for implementing locks

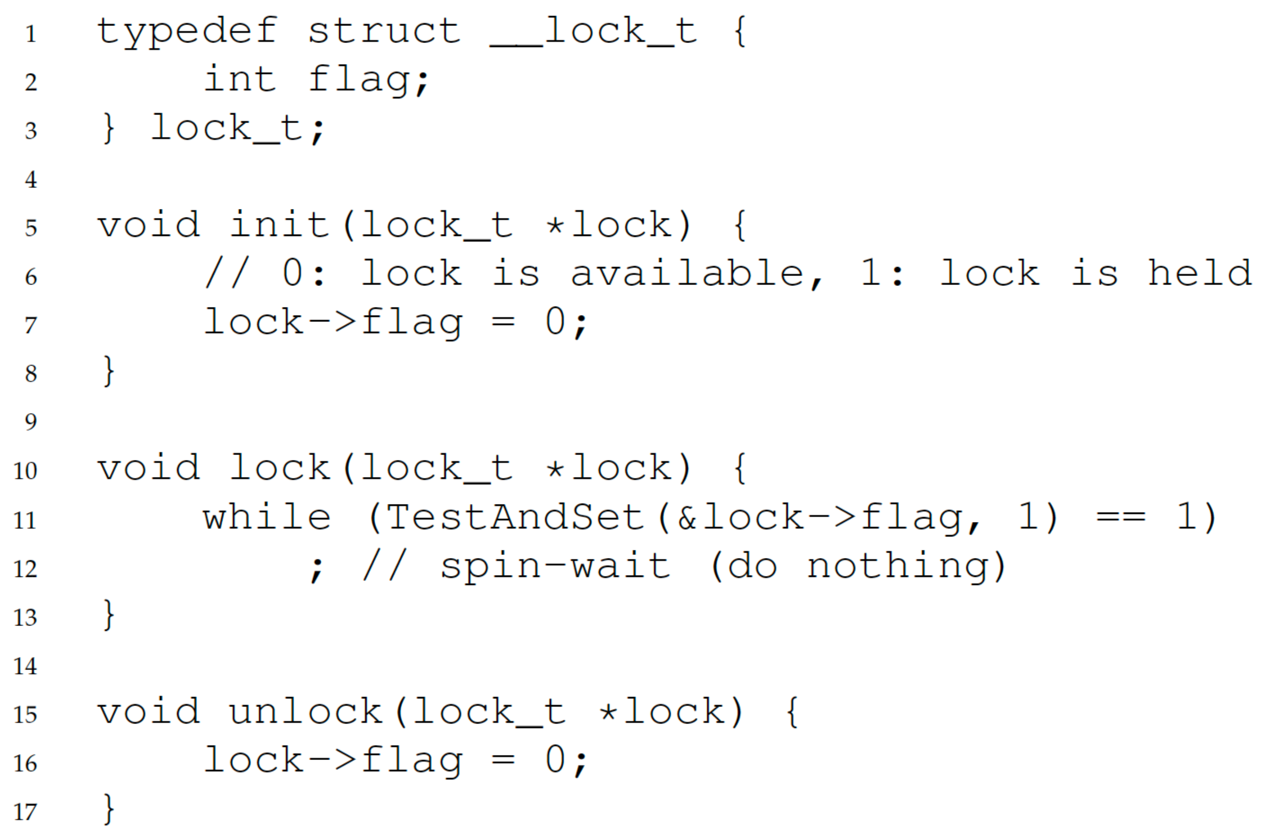
Test-and-Set Instruction

- Update a variable and return old value, all in one hardware instruction

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

Spin lock using test-and-set

- If TestAndSet(flag,1) returns 1, it means the lock is held by someone else, so spin until the flag changes to 0



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

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

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

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

Evaluating Spin Locks

- Correctness?
- Fairness?
- Performance?
- Think about the above properties when there are N threads being scheduled in a round-robin fashion?

How to avoid wasteful spinning?

- Can we avoid wasteful spinning while the thread is waiting for the lock to be released?
- Use **yield()**
- Yield() moves the waiting thread from running to ready state

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Yield() is helpful but…

- **Performance:** While better than spinning, yield() does have overhead due to the associated cost of context switch
- **Starvation/Fairness:** A thread may get caught in an endless yield loop while other threads repeatedly enter and exit the critical section
- **What is the problem in the approaches we discussed?**
    - Scheduler may make a bad choice: it may schedule a thread which yields the CPU immediately or spins for the entire time slice.
    - There is potential for waste and no prevention for starvation

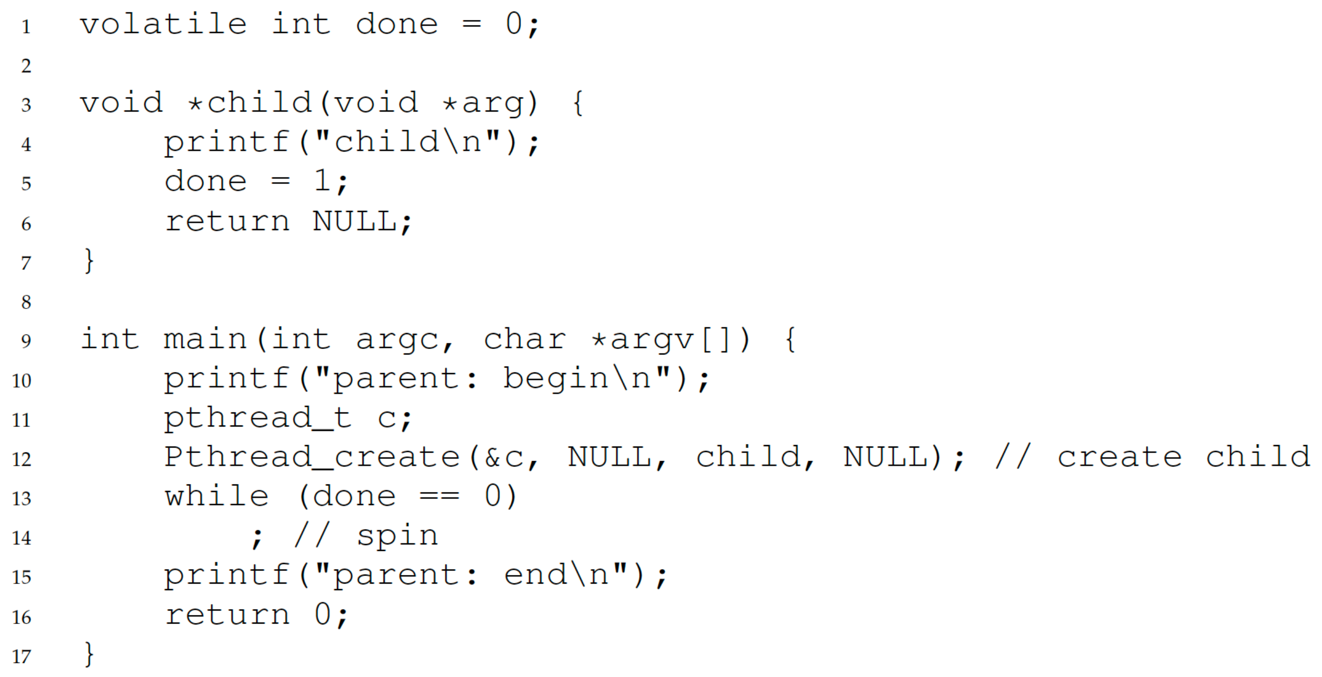
Using Sleep and Queues

- Use sleep instead of spinning or yielding
    - sleep moves the process from running to blocked
    - yield moves the process from running to ready
- We will have a queue to keep track of which threads are waiting to acquire the lock
- How to build a lock using sleep?
    - If the lock is held by another thread, add the calling thread to the queue and put it to sleep
    - If the lock is available, wake up the first sleeping thread from the queue

```c
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock
                                    // (for next thread!)
    m->guard = 0;
}
```
Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

Condition Variables: Waiting and Signaling

- Locks allow one type of synchronization between threads – mutual exclusion
- Another common requirement in multi-threaded applications – waiting and signaling
    - E.g., Thread T1 wants to continue only after T2 has finished some task
- Can accomplish this by busy-waiting on some variable, but inefficient
- Need a new synchronization primitive: condition variables



```c
volatile int done = 0;

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

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

Figure 30.2: **Parent Waiting For Child: Spin-based Approach**

Condition Variables

- A condition variable (CV) is a queue that a thread can put itself into when waiting on some condition
- Another thread that makes the condition true can signal the CV to wake up a waiting thread

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

Example

```c
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

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

Figure 30.3: **Parent Waiting For Child: Use A Condition Variable**

Why do we need the state variable done?

- Race condition: missed wakeup
  - Child runs immediately, signals, but no one sleeping yet
  - Parent decides to sleep and goes to sleep forever as child has already executed.

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Why do we need to lock before wait?

- Race condition: missed wakeup
  - Parent checks done to be 0, decides to sleep, interrupted
  - Child runs, sets done to 1, signals, but no one sleeping yet
  - Parent now resumes and goes to sleep forever
- Lock must be held when calling wait and signal with CV
- The wait function releases the lock before putting thread to sleep, so lock is available for signaling thread

```
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}

void thr_join() {
    if (done == 0)
        Pthread_cond_wait(&c);
}
```