ILLINOIS TECH

College of Computing

CS 450 Operating Systems
Semaphore

Yue Duan

Concurrency Goals

- Mutual Exclusion
  - Keep two threads from executing in a critical section concurrently
  - We solved this with **locks**
- Dependent Events
  - We want a thread to wait until some particular event has occurred
  - Or some condition has been met
  - Solved with **condition variables and semaphores**

Condition Variables

- CV:
  - queue of waiting threads
- **B** waits for a signal on CV before running
  - wait(CV, ...);
- **A** sends signal() on CV when time for **B** to run
  - signal(CV, ...);

API

- **cond_wait**(cond_t * cv, mutex_t * lock)
  - assumes lock is held when wait() is called
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning
- **cond_signal**(cond_t * cv)
  - wake a single waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, NOP

CV Rules of Thumb

- Keep state in addition to CVs
  - numfull in producer/consumer problem
- Always **cond_wait**() or **cond_signal**() with lock held
- Use different CVs for different conditions
- Recheck state assumptions when waking up from waiting
  - use while instead of if

Semaphore

- CVs only have a queue.
  - **State is managed by the programmer!**
- Semaphores include some state (namely, a counter), which is managed by the **implementation**.
  - less error-prone!
- Not easy to use as a general condition variable
- Pthreads just have locks and condition variables, but no semaphores

Semaphores (API)

- **sem_init**(sem_t * s, int init_count);
- **sem_wait**(sem_t * s);
  - decrements count, goes to sleep if == -1
  - sometimes also called p() or down()
- **sem_post**(sem_t * s);
  - increments count, wakes any waiters (sleepers)
  - sometimes also called v() or up()

thread_join()

with locks and CVs

```
void thread_join () {
        mutex_lock(&m);
        if (done == 0)
                cond_wait(&c, &m);
        mutex_unlock(&m);
}
void thread_exit () {
        mutex_lock(&m);
        done = 1;
        cond_signal(&c);
        mutex_unlock(&m);
}
```

with semaphores

```
sem_t sem;
sem_init(&sem, ???);

void thread_join () {
        sem_wait(&sem);
}
void thread_exit () {
        sem_post(&sem);
}
```

Claim: Semaphores are equally powerful as lock+CVs

Types

- Binary semaphore
  - represents single access to a resource
  - guarantees mutual exclusion to a critical section
  - equals to a **lock**

```
sem_t m;
sem_init(&m, 0, X); // initialize to X; what should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

Types

- General semaphore
  - multiple threads pass the semaphore determined by count
    - mutex has count = 1, counting has count = N
  - represents a resource with many units available
  - or a resource allowing some unsynchronized concurrent access (e.g., reading)

Producer/consumer with semaphores

- Simple case: one consumer/one producer
- Single shared buffer between them
  - max = 1
- **Constraints**:
  - Producer must wait for buffer to be non-full before producing
  - Consumer must wait for buffer to be non-empty before consuming
- Use **2 semaphores** to get it right

Producer/consumer with semaphores

Producer

```
while (1) {
        sem_wait(&emptyBuffer);
        put(&buffer);

        sem_post(&fullBuffer);
}
```

Consumer

```
while (1) {
        sem_wait(&fullBuffer);
        get(&buffer);

        sem_post(&emptyBuffer);
}
```

- **What should the initial counts be?**
  - **emptyBuffer: Initialize to** **1**
  - **fullBuffer: Initialize to** **0**

Producer/consumer with semaphores

- Simple case: one consumer/one producer
- Single shared (**circular**) buffer (**with N slots**) between them
- **Constraints**:
  - Producer must wait for buffer to be non-full before producing
  - Consumer must wait for buffer to be non-empty before consuming
- Use **2 semaphores** to get it right

Producer/consumer with semaphores

Producer

```
i = 0;
while (1) {
      sem_wait(&emptyBuffer);
      put(&buffer[i]);
      i = (i + 1) % N;
      sem_post(&fullBuffer);
}
```

Consumer

```
j = 0;
while (1) {
      sem_wait(&fullBuffer);
      get(&buffer[j]);
      j = (j + 1) % N;
      sem_post(&emptyBuffer);
}
```

- **What should the initial counts be?**
  - **emptyBuffer: Initialize to N**
  - **fullBuffer: Initialize to 0**

Producer/consumer with semaphores

- General case: **multiple producers/multiple consumers**
- Single shared (**circular**) buffer (**with N slots**) between them
- **Constraints**:
    - Producer must wait for buffer to be non-full before producing
    - Consumer must wait for buffer to be non-empty before consuming
- Use **2 semaphores** to get it right

Producer/consumer with semaphores

Producer

```
i = 0;
while (1) {
        sem_wait(&emptyBuffer);
        put(&buffer[i]);
        i = (i + 1) % N;
        sem_post(&fullBuffer);
}
```

Consumer

```
j = 0;
while (1) {
        sem_wait(&fullBuffer);
        get(&buffer[j]);
        j = (j + 1) % N;
        sem_post(&emptyBuffer);
}
```

- **Will this work?**
    - **no, why not?**
    - **that’s right, mutual exclusion!**

Adding Mutual Exclusion

Producer

```
i = 0;
while (1) {
    sem_wait(&mutex);
    sem_wait(&emptyBuffer);
    put(&buffer[i]);
    i = (i + 1) % N;
    sem_post(&fullBuffer);
    sem_post(&mutex);
}
```

Consumer

```
j = 0;
while (1) {
    sem_wait(&mutex);
    sem_wait(&fullBuffer);
    get(&buffer[j]);
    j = (j + 1) % N;
    sem_post(&emptyBuffer);
    sem_post(&mutex);
}
```

- Does it work?
- What's the problem?
    - deadlock

Adding Mutual Exclusion

Producer

```
i = 0;
while (1) {
    sem_wait(&emptyBuffer);
    sem_wait(&mutex);
    put(&buffer[i]);
    i = (i + 1) % N;
    sem_post(&mutex);
    sem_post(&fullBuffer);
}
```

Consumer

```
j = 0;
while (1) {
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    get(&buffer[j]);
    j = (j + 1) % N;
    sem_post(&mutex);
    sem_post(&emptyBuffer);
}
```

- Correct version!
- Is there a even better version?

Reader-Writer Locks

- Different data structure accesses might require different kinds of locking
  - inserts change the state of a list
  - lookups simply read the data structure
  - as long as no insert is on-going, many lookups can proceed concurrently
- Let multiple reader threads grab lock (shared)
- Only one writer thread can grab lock (exclusive)
  - No reader threads
  - No other writer threads

Reader-Writer Locks

- General design
  - use a **writelock** semaphore to ensure that only a single writer can
    - acquire the lock
    - enter the critical section to update the data structure
  - when acquiring a read lock
    - the reader first acquires **lock**
    - increments the **readers** variable
    - the reader also acquires the **write** lock
      - by calling **sem_wait**() on the **writelock** semaphore

Reader-Writer Locks

```
void rwlock_init(rwlock_t *l) {
	l->readers = 0;
	sem_init(&l->lock, 1);
	sem_init(&l->writelock, 1);
}
```

```
void rw_readlock (rwlock_t *l) {
	sem_wait(&l->lock); // grab read lock
	l->readers++;      // this is the critical section
	if (readers == 1) // since there are readers, writer must wait
		sem_wait(&l->writelock);
	sem_post(&l->lock); // other readers can continue
}
```

Reader-Writer Locks

```
void rw_readunlock (rwlock_t *l) {
      sem_wait(&l->lock); // grab read lock
      l->readers--;       // this is the critical section
      if (readers == 0) // no more readers, writers can cont.
            sem_post(&l->writelock);
      sem_post(&l->lock); // other readers can continue
}

   void rw_writelock (rwlock_t *l) {
         sem_wait(&l->writelock); // grab write lock
         // only continues if there are no readers!
   }

   void rw_writeunlock (rwlock_t *l) {
         sem_post(&l->writelock); // release write lock
   }
```

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T1: release_readlock()
T4: acquire_readlock()
T5: acquire_readlock()
T3: release_writelock()
// what happens next?

THANK YOU!