Concurrency: Semaphores

Questions Answered in this Lecture:
• What is a semaphore? Why is it useful? How different from CV?
• How does one implement a lock with a semaphore?
• How to implement producer/consumer with semaphores?
• How to implement reader/writer locks with semaphores?
Recall: CV API

- **wait**(cond_t * cv, mutex_t * lock)
  - assumes lock is held when `wait()` is called (*why?*)
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning

- **signal**(cond_t * cv)
  - wake a *single* waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, NOP

- **broadcast**(cond_t * cv)
  - wake *all* waiters
  - if no waiting threads, NOP
CV Rules of thumb

• Keep state in addition to CVs
• Always wait() or signal() with lock held
• Recheck state assumptions when waking up from waiting
void *producer (void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m); // p1
        while (numfull == max) // p2
            cond_wait(&cond, &m); // p3
        do_fill(i); // p4
        cond_signal(&cond); // p5
        mutex_unlock(&m); // p6
    }
}

void *consumer (void *arg) {
    while (1) {
        mutex_lock(&m); // c1
        while (numfull == 0) // c2
            cond_wait(&cond, &m);
        int tmp = do_get(); // c4
        cond_signal(&cond); // c5
        mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}
How do we fix this?

• Use 2 CVs! (*one for each logical condition* that we’re signaling)
• Consumers only signal producers
• producers only signal consumers!
• Other option (*covering conditions*) broadcast to everyone!
```c
void *producer (void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m); // p1
        while (numfull == max) // p2
            cond_wait(&empty_cond, &m); // p3
        do_fill(i); // p4
        cond_signal(&full_cond); // p5
        mutex_unlock(&m); // p6
    }
}

void *consumer (void *arg) {
    while (1) {
        mutex_lock(&m); // c1
        while (numfull == 0) // c2
            cond_wait(&full_cond, &m);
        int tmp = do_get(); // c4
        cond_signal(&empty_cond); // c5
        mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}
```
Semaphores
What are they?

• Another (earlier) solution to the bounded buffer problem (Dijkstra in the 60s)

• Ensure **mutual** exclusion of a critical section

• Ensure **ordering** among threads in the execution of a concurrent program
Difference between CVs and Semaphores?

• CVs only have a queue. *State is managed by the programmer!*

• Semaphores include some state (namely, *a counter*), which is managed by the implementation
Semaphores (API)

• **sem_init**({\texttt{sem_t * s, int init\_count}});
• **sem_wait**({\texttt{sem_t * s}}); // decrements count, goes to // sleep if == -1
  • sometimes also called p() or down()
• **sem_post**({\texttt{sem_t * s}}); // increments count, wakes any //waiters (sleepers)
  • sometimes also called v() or up()
thread_join()

with locks and CVs

```c
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

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

void thread_join () {
    sem_wait(&sem);
}

void thread_exit () {
    sem_post(&sem);
}
```
Equivalence

• **Claim**: Semaphores are equally powerful as lock+CVs

• One might be *more convenient* for a particular use case, but that’s not the point

• This means *we can build each out of the other*
Proof outline

- Locks
  - Semaphores
- CV’s
  - Semaphores
- Semaphores
  - Locks
  - CV’s
Proof outline
typedef struct {
    } lock_t;

void init(lock_t *lock) {
}

void acquire(lock_t *lock) {
}

void release(lock_t *lock) {
}
typedef struct {
    sem_t sem;
} lock_t;

void init(lock_t *lock) {
    sem_init(&lock->sem, ??);
}

void acquire(lock_t *lock) {
    sem_wait(&lock->sem);
}

void release(lock_t *lock) {
    sem_post(&lock->sem);
}
typedef struct {
    sem_t sem;
} lock_t;

void init(lock_t *lock) {
    sem_init(&lock->sem, 1);
}

void acquire(lock_t *lock) {
    sem_wait(&lock->sem);
}

void release(lock_t *lock) {
    sem_post(&lock->sem);
}
Proof outline

- Locks
- Semaphores

- CV’s
- Semaphores

- Semaphores
- Locks
- CV’s
This one is surprisingly subtle

- **Challenge**: go do it on your own
- Maybe see Andrew Birrell’s experience report from Microsoft...
Proof outline

- Locks
  - Semaphores
- CV’s
  - Semaphores

Semaphores

- Locks
- CV’s
typedef struct {
    // ???
} sem_t;

void sem_init(sem_t *s, int init_count) {
    // ???
}
typedef struct {
  int count;
  cond_t cond;
  lock_t lock;
} sem_t;

void sem_init(sem_t *s, int init_count) {
  s->count = init_count;
  cond_init(&s->cond);
  lock_init(&s->lock);
}
void sem_wait (sem_t *s) {
    // ???
}

void sem_post (sem_t *s) {
    // ???
}
```c
void sem_wait (sem_t *s) {
    lock_acquire(&s->lock);
    // atomic stuff
    lock_release(&s->lock);
}

void sem_post (sem_t *s) {
    lock_acquire(&s->lock);
    // atomic stuff
    lock_release(&s->lock);
}
```
void sem_wait (sem_t *s) {
    lock_acquire(&s->lock);
    while (s->count < 0)
        cond_wait(&s->cond);
    s->value--;
    lock_release(&s->lock);
}

void sem_post (sem_t *s) {
    lock_acquire(&s->lock);
    // atomic stuff
    lock_release(&s->lock);
}
```c
void sem_wait (sem_t *s) {
    lock_acquire(&s->lock);
    while (s->count < 0)
        cond_wait(&s->cond);
    s->value--;
    lock_release(&s->lock);
}

void sem_post (sem_t *s) {
    lock_acquire(&s->lock);
    s->count++;
    cond_signal(&s->cond);
    lock_release(&s->lock);
}
```
Producer/consumer with semaphores

• Simplest case: one consumer/one producer
• Single shared buffer 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

while (1) {
    sem_wait(&emptyBuffer);
    put(&buffer);
    sem_post(&fullBuffer);
}

Consumer

while (1) {
    sem_wait(&fullBuffer);
    get(&buffer);
    sem_post(&emptyBuffer);
}

What should initial counts be?
Producer/consumer with semaphores

• Simplest 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

\[ i = 0; \]
\[
\text{while (1) { }
\]
\[
\text{sem\\_wait}(&\text{emptyBuffer});
\]
\[
\text{put}(&\text{buffer}[i]);
\]
\[
i = (i + 1) \% N;
\]
\[
\text{sem\\_post}(&\text{fullBuffer});
\]
\[
\text{}}\]

Consumer

\[ j = 0; \]
\[
\text{while (1) { }
\]
\[
\text{sem\\_wait}(&\text{fullBuffer});
\]
\[
\text{get}(&\text{buffer}[j]);
\]
\[
j = (j + 1) \% N;
\]
\[
\text{sem\\_post}(&\text{emptyBuffer});
\]
\[
\text{}}\]

What should initial counts be?
MPMC

• General case: multiple producers/multiple consumers
• Single shared (circular) buffer (with N slots) between them
• Share

• 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

\[ i = 0; \]
\[ \text{while } (1) \{ \]
\[ \quad \text{sem}_\text{wait}(&\text{emptyBuffer}); \]
\[ \quad \text{put}(&\text{buffer}[i]); \]
\[ \quad i = (i + 1) \% N; \]
\[ \quad \text{sem}_\text{post}(&\text{fullBuffer}); \]
\[ \} \]

Consumer

\[ j = 0; \]
\[ \text{while } (1) \{ \]
\[ \quad \text{sem}_\text{wait}(&\text{fullBuffer}); \]
\[ \quad \text{get}(&\text{buffer}[j]); \]
\[ \quad j = (j + 1) \% N; \]
\[ \quad \text{sem}_\text{post}(&\text{emptyBuffer}); \]
\[ \} \]

Will this work?
Producer

i = 0;
while (1) {
    sem_wait(&emptyBuffer);
    i = findempty(&buffer);
    put(&buffer[i]);
    sem_post(&fullBuffer);
}

Consumer

j = 0;
while (1) {
    sem_wait(&fullBuffer);
    j = findfull(&buffer);
    get(&buffer[j]);
    sem_post(&emptyBuffer);
}
Producer

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

Consumer

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

What’s the problem?
Producer

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

Consumer

j = 0;
while (1) {
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    j = findfull(&buffer);
    get(&buffer[j]);
    sem_post(&mutex);
    sem_post(&emptyBuffer);
}
Works, but limits concurrency

Producer

\[
i = 0;
\]

while (1) {
    sem_wait(&emptyBuffer);
    sem_wait(&mutex);
    i = findempty(&buffer);
    put(&buffer[i]);
    sem_post(&mutex);
    sem_post(&fullBuffer);
}

Consumer

\[
j = 0;
\]

while (1) {
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    j = findfull(&buffer);
    get(&buffer[j]);
    sem_post(&mutex);
    sem_post(&emptyBuffer);
}
Increases concurrency (in producing/consuming)

Producer

\[i = 0;\]
\[\text{while (1)} \{\]
  \[\text{sem\_wait(\&emptyBuffer);}\]
  \[\text{sem\_wait(\&mutex);}\]
  \[i = \text{findempty(\&buffer);}\]
  \[\text{sem\_post(\&mutex);}\]
  \[\text{put(\&buffer[i]);}\]
  \[\text{sem\_post(\&fullBuffer);}\]
\[\}\]

Consumer

\[j = 0;\]
\[\text{while (1)} \{\]
  \[\text{sem\_wait(\&fullBuffer);}\]
  \[\text{sem\_wait(\&mutex);}\]
  \[j = \text{findfull(\&buffer);}\]
  \[\text{sem\_post(\&mutex);}\]
  \[\text{get(\&buffer[j]);}\]
  \[\text{sem\_post(\&emptyBuffer);}\]
\[\}\]
Reader Writer Locks

• Operations on shared data are not symmetric
• If many threads attempt to read a lock:
  • Normal locks will prevent this
  • But, if there no changes to the state, why shouldn’t they be able to?
• So writers (changers of state) should be treated differently from readers
A Conceptual Solution

• As long as there are no writers, readers can proceed concurrently
• Writers must wait for readers to drain
• Readers must wait for writers to finish (writers have exclusive access)
typedef struct {
    sem_t  lock;
    sem_t  writelock;
    int    readers;
} rwlock_t;
Reader/writer locks

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

```c
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

```c
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
}
```
Reader/writer locks

```c
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
}
```
Stepping back

• We’ve considered mechanisms for *mutual exclusion* and *ordering* of events
• Mostly we’ve talked about them in the context of *threads executing concurrently*
• Is this more broadly applicable? **Hint**: is the universe concurrent?
• E.g.
  • What if two base stations are trying to send a firmware update to Mars rover at the same time?
    • How do we ensure atomicity?
    • What does spinning/waiting look like?
• Put another way, **how do we ensure mutual exclusion and ordering for a distributed system?** (answered in CS550)