

# Semaphores

CSC 343, Operating Systems

# Topics covered in this lecture

- Condition variables
- Semaphores
- Signaling through condition variables and semaphores
- Concurrency bugs

This slide deck covers chapters 30, 31, 32 in OSTEP.

# Condition variables (CV)

In concurrent programming, a common scenario is one thread waiting for another thread to complete an action.

```
1  bool done = false;
2
3  /* called in the child to signal termination */
4  void thr_exit() {
5      done = true;
6  }
7  /* called in the parent to wait for a child thread */
8  void thr_join() {
9      while (!done);
10 }
```

# Condition variables (CV)

- Locks enable mutual exclusion of a shared region.
  - Unfortunately they are oblivious to ordering
- Waiting and signaling (i.e., T2 waits until T1 completes a given task) could be implemented by spinning until the value changes
- But spinning is incredibly *inefficient*
- New synchronization primitive: *condition variables*

# Condition variables (CV)

- Locks enable mutual exclusion of a shared region.
  - Unfortunately they are oblivious to ordering
- Waiting and signaling (i.e., T2 waits until T1 completes a given task) could be implemented by spinning until the value changes
- But spinning is incredibly *inefficient*
- New synchronization primitive: *condition variables*

# Condition variables (CV)

- Locks enable mutual exclusion of a shared region.
  - Unfortunately they are oblivious to ordering
- Waiting and signaling (i.e., T2 waits until T1 completes a given task) could be implemented by spinning until the value changes
- But spinning is incredibly *inefficient*
- New synchronization primitive: ***condition variables***

# Condition variables (CV)

- A CV allows:
  - A thread to wait for a condition
  - Another thread signals the waiting thread
- Implement CV using queues
- API: wait, signal or broadcast
  - wait: wait until a condition is satisfied
  - signal: wake up one waiting thread
  - broadcast: wake up all waiting threads
- On Linux, pthreads provides CV implementation

# Condition variables (CV)

- A CV allows:
  - A thread to wait for a condition
  - Another thread signals the waiting thread
- Implement CV using queues
- API: wait, signal or broadcast
  - wait: wait until a condition is satisfied
  - signal: wake up one waiting thread
  - broadcast: wake up all waiting threads
- On Linux, pthreads provides CV implementation



# Signal parent that child has exited

```
1  bool done = false;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4  /* called in the child to signal termination */
5  void thr_exit() {
6      pthread_mutex_lock(&m);
7      done = true;
8      pthread_cond_signal(&c);
9      pthread_mutex_unlock(&m);
10 }
11 /* called in the parent to wait for a child thread */
12 void thr_join() {
13     pthread_mutex_lock(&m);
14     while (!done)
15         pthread_cond_wait(&c, &m);
16     pthread_mutex_unlock(&m);
17 }
```

# Signal parent that child has exited (2)

- `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`
  - Assume mutex `m` is held; *atomically* unlock mutex when waiting, retake it when waking up
- Question: Why do we need to check a condition before sleeping?
- Thread may have already exited, i.e., no need to wait
  - Principle: Check the condition before sleeping
- Question: Why can't we use `if` when waiting?
- Multiple threads could be woken up, racing for done flag
  - Principle: `while` instead of `if` when waiting

# Signal parent that child has exited (2)

- `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`
  - Assume mutex `m` is held; *atomically* unlock mutex when waiting, retake it when waking up
- Question: Why do we need to check a condition before sleeping?
- Thread may have already exited, i.e., no need to wait
  - Principle: Check the condition before sleeping
- Question: Why can't we use `if` when waiting?
- Multiple threads could be woken up, racing for done flag
  - Principle: `while` instead of `if` when waiting

# Signal parent that child has exited (2)

- `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`
  - Assume mutex `m` is held; *atomically* unlock mutex when waiting, retake it when waking up
- Question: Why do we need to check a condition before sleeping?
- Thread may have already exited, i.e., no need to wait
  - Principle: Check the condition before sleeping
- Question: Why can't we use `if` when waiting?
- Multiple threads could be woken up, racing for done flag
  - Principle: `while` instead of `if` when waiting

# Signal parent that child has exited (2)

- `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`
  - Assume mutex `m` is held; *atomically* unlock mutex when waiting, retake it when waking up
- Question: Why do we need to check a condition before sleeping?
- Thread may have already exited, i.e., no need to wait
  - Principle: Check the condition before sleeping
- Question: Why can't we use `if` when waiting?
- Multiple threads could be woken up, racing for done flag
  - Principle: `while` instead of `if` when waiting

# Signal parent that child has exited (3)

- Question: Why do we need to protect `done` with mutex `m`?
- Mutex `m` allows one thread to access `done` for protecting against missed updates
  - Parent reads `done == false` but is interrupted
  - Child sets `done = true` and signals but no one is waiting
  - Parent continues and goes to sleep (forever)
- Lock is therefore required for wait/signal synchronization

# Signal parent that child has exited (3)

- Question: Why do we need to protect `done` with mutex `m`?
- Mutex `m` allows one thread to access `done` for protecting against missed updates
  - Parent reads `done == false` but is interrupted
  - Child sets `done = true` and signals but no one is waiting
  - Parent continues and goes to sleep (forever)
- Lock is therefore required for wait/signal synchronization

# Producer/consumer synchronization

- Producer/consumer is a common programming pattern
- For example: map (producers) / reduce (consumer)
- For example: a concurrent database (consumers) handling parallel requests from clients (producers)
  - Clients produce new requests (encoded in a queue)
  - Handlers consume these requests (popping from the queue)
- Strategy: use CV to synchronize
  - Make producers wait if buffer is full
  - Make consumers wait if buffer is empty (nothing to consume)



# Producer/consumer synchronization

- Producer/consumer is a common programming pattern
- For example: map (producers) / reduce (consumer)
- For example: a concurrent database (consumers) handling parallel requests from clients (producers)
  - Clients produce new requests (encoded in a queue)
  - Handlers consume these requests (popping from the queue)
- Strategy: use CV to synchronize
  - Make producers wait if buffer is full
  - Make consumers wait if buffer is empty (nothing to consume)

# Condition variables

- Programmer must keep state, orthogonal to locks
- CV enables access to critical section with a thread wait queue
- Always wait/signal while holding lock
- Whenever thread wakes, recheck state

# Semaphore

- A semaphore extends a CV with an integer as internal state
- `int sem_init(sem_t *sem, unsigned int value):`  
creates a new semaphore with `value` slots
- `int sem_wait(sem_t *sem):` waits until the semaphore has at least one slot, decrements the number of slots
- `int sem_post(sem_t *sem):` increments the semaphore (and wakes one waiting thread)
- `int sem_destroy(sem_t *sem):` destroys the semaphore and releases any waiting threads

# Concurrent programming: producer consumer

- One or more producers create items, store them in buffer
- One or more consumers process items from buffer
- Need synchronization for buffer
  - Want concurrent production and consumption
  - Use as many cores as available
  - Minimize access time to shared data structure

# Concurrent programming: producer consumer

- One or more producers create items, store them in buffer
- One or more consumers process items from buffer
- Need synchronization for buffer
  - Want concurrent production and consumption
  - Use as many cores as available
  - Minimize access time to shared data structure

# Concurrent programming

```
1 void *producer(void *arg) {
2     unsigned int max = (unsigned int)arg;
3     for (unsigned int i = 0; i < max; i++) {
4         put(i); // store in shared buffer
5     }
6     return NULL;
7 }
8 void *consumer(void *arg) {
9     unsigned int max = (unsigned int)arg;
10    for (unsigned int i = 0; i < max; i++) {
11        printf("%d\n", get(i)); // recv from buffer
12    }
13    return NULL;
14 }
pthread_t p, c;
pthread_create(&p, NULL, &producer, (void*)NUMITEMS);
pthread_create(&c, NULL, &consumer, (void*)NUMITEMS);
```

# Concurrent programming:

```
1 unsigned int buffer[BUFSIZE] = { 0 };
2 unsigned int cpos = 0, ppos = 0;
3
4 void put(unsigned int val) {
5     buffer[ppos] = val;
6     ppos = (ppos + 1) % BUFSIZE;
7 }
8
9 unsigned int get() {
10    unsigned long val = buffer[cpos];
11    cpos = (cpos + 1) % BUFSIZE;
12    return val;
13 }
```

What are the issues in this code?

- Producers may overwrite unconsumed entries
- Consumers may consume uninitialized or stale entries

# Concurrent programming:

```
1 unsigned int buffer[BUFSIZE] = { 0 };
2 unsigned int cpos = 0, ppos = 0;
3
4 void put(unsigned int val) {
5     buffer[ppos] = val;
6     ppos = (ppos + 1) % BUFSIZE;
7 }
8
9 unsigned int get() {
10    unsigned long val = buffer[cpos];
11    cpos = (cpos + 1) % BUFSIZE;
12    return val;
13 }
```

What are the issues in this code?

- Producers may overwrite unconsumed entries
- Consumers may consume uninitialized or stale entries



# Producer/consumer: use semaphores!

```
sem_t csem, psem;
```

```
/* BUFSIZE items are available for producer to create */  
sem_init(&psem, 0, BUFSIZE);
```

```
/* 0 items are available for consumer */  
sem_init(&csem, 0, 0);
```

# Producer: semaphores

```
1 void put(unsigned int val) {
2     /* we wait until there is buffer space available */
3     sem_wait(&psem);
4
5     /* store element in buffer */
6     buffer[ppos] = val;
7     ppos = (ppos + 1) % BUFSIZE;
8
9     /* notify consumer that data is available */
10    sem_post(&csem);
11 }
```

# Consumer: semaphores

```
1 unsigned int get() {
2     /* wait until data is produced */
3     sem_wait(&csem);
4
5     /* consumer entry */
6     unsigned long val = buffer[cpos];
7     cpos = (cpos + 1) % BUFSIZE;
8
9     /* notify producer that a space has freed up */
10    sem_post(&psem);
11    return val;
12 }
```

# Producer/consumer: remaining issues?

- We now synchronize between consumers and producers
  - Producer waits until buffer space is available
  - Consumer waits until data is ready
- How would you handle multiple producers/consumers?
  - Currently no synchronization between producers (or consumers)

# Producer/consumer: remaining issues?

- We now synchronize between consumers and producers
  - Producer waits until buffer space is available
  - Consumer waits until data is ready
- How would you handle multiple producers/consumers?
  - Currently no synchronization between producers (or consumers)

# Multiple producers: use locking!

```
/* mutex handling mutual exclusive access to ppos */
1 pthread_mutex_t pmutex = PTHREAD_MUTEX_INITIALIZER;
2
3 void put(unsigned int val) {
4     unsigned int mypos;
5     /* we wait until there is buffer space available */
6     sem_wait(&psem);
7     /* ppos is shared between all producers */
8     pthread_mutex_lock(&pmutex);
9     mypos = ppos;
10    ppos = (ppos + 1) % BUFSIZE;
11    /* store information in buffer */
12    buffer[mypos] = val;
13    pthread_mutex_unlock(&pmutex);
14    sem_post(&csem);
15 }
```

# Semaphores/spin locks/CVs are interchangeable

- Each is implementable through a combination of the others
- Depending on the use-case one is faster than the other
  - How often is the critical section executed?
  - How many threads compete for a critical section?
  - How long is the lock taken?

# Implementing a mutex with a semaphore

```
1 sem_t sem;  
2 sem_init(&sem, 1);  
3  
4 sem_wait(&sem);  
5 ... // critical section  
6 sem_post(&sem);
```



# Implementing a semaphore with CV/locks

```
1  typedef struct {
2      int value;           // sem value
3      pthread_mutex_t lock; // access to sem
4      pthread_cond_t cond; // wait queue
5  } sem_t;
6
7  void sem_init(sem_t *s, int val) {
8      s->value = val;
9      pthread_mutex_init(&(s->lock), NULL);
10     pthread_cond_init(&(s->cond), NULL);
11 }
```

# Implementing a semaphore with CV/locks

```
1 void sem_wait(sem_t *s) {
2   pthread_mutex_lock(&(s->lock));
3   while (s->value <= 0)
4     pthread_cond_wait(&(s->cond), &(s->lock));
5   s->value--;
6   pthread_mutex_unlock(&(s->lock));
7 }
8
9 void sem_post(sem_t *s) {
10  pthread_mutex_lock(&(s->lock));
11  s->value++;
12  pthread_cond_signal(&(s->cond));
13  pthread_mutex_unlock(&(s->lock));
14 }
```

# Reader/writer locks

- A single (exclusive) writer, multiple (N) concurrent readers
- Implement using two semaphores: `lock` for the data structure, `wlock` for the writer
  - Both semaphores initialized with (1)
  - Writer only waits/posts on `wlock` when acquiring/releasing
  - Reader waits on `lock`, increments/decrements reader count
  - If number of readers==0, must wait/post on `wlock`

# Reader/writer locks

```
1 void rwlock_acquire_readlock(rwlock_t *rw) {
2     sem_wait(&rw->lock);
3     rw->readers++;
4     if (rw->readers == 1)
5         sem_wait(&rw->wlock); // first r, also grab wlock
6     sem_post(&rw->lock);
7 }
8
9 void rwlock_release_readlock(rwlock_t *rw) {
10    sem_wait(&rw->lock);
11    rw->readers--;
12
13    if (rw->readers == 0)
14        sem_post(&rw->wlock); // last r, also release wlock
15    sem_post(&rw->lock);
16 }
```

# Bugs in concurrent programs

- Writing concurrent programs is hard!
- **Atomicity bug:** concurrent, unsynchronized modification (lock!)
- **Order-violating bug:** data is accessed in wrong order (use CV!)
- **Deadlock:** program no longer makes progress (locking order)

# Atomicity bugs

One thread checks value and prints it while another thread concurrently modifies it.

```
1  int shared = 24;
2
3  void T1() {
4    if (shared > 23) {
5        printf("Shared is >23: %d\n", shared);
6    }
7 }
8  void T2() {
9    shared = 12;
10 }
```

- T2 may modify shared between if check and printf in T1.
- Fix: use a common mutex between both threads when accessing the shared resource.

# Atomicity bugs

One thread checks value and prints it while another thread concurrently modifies it.

```
1  int shared = 24;
2
3  void T1() {
4    if (shared > 23) {
5        printf("Shared is >23: %d\n", shared);
6    }
7 }
8 void T2() {
9     shared = 12;
10 }
```

- T2 may modify shared between if check and printf in T1.
- Fix: use a common mutex between both threads when accessing the shared resource.

# Order-violating bug

One thread assumes the other has already updated a value.

Thread 1::

```
void init() {  
    mThread = PR_CreateThread(mMain, ...);  
    mThread->State = ...;  
}
```

Thread 2::

```
void mMain(...) {  
    mState = mThread->State;  
}
```

- Thread 2 may run before `mThread` is assigned in T1.
- Fix: use a CV to signal that `mThread` has been initialized.



# Order-violating bug

One thread assumes the other has already updated a value.

Thread 1::

```
void init() {  
    mThread = PR_CreateThread(mMain, ...);  
    mThread->State = ...;  
}
```

Thread 2::

```
void mMain(...) {  
    mState = mThread->State;  
}
```

- Thread 2 may run before `mThread` is assigned in T1.
- Fix: use a CV to signal that `mThread` has been initialized.

# Deadlock

Locks are taken in conflicting order.

```
void T1() {  
    lock(L1);  
    lock(L2);  
}
```

```
void T2() {  
    lock(L2);  
    lock(L1);  
}
```

- Threads 1/2 may be stuck after taking the first lock, program makes no more progress
- Fix: acquire locks in increasing (global) order.

# Deadlock

Locks are taken in conflicting order.

```
void T1() {  
    lock(L1);  
    lock(L2);  
}
```

```
void T2() {  
    lock(L2);  
    lock(L1);  
}
```

- Threads 1/2 may be stuck after taking the first lock, program makes no more progress
- Fix: acquire locks in increasing (global) order.

# Summary

- Spin lock, CV, and semaphore synchronize multiple threads
  - Spin lock: atomic access, no ordering, spinning
  - Condition variable: atomic access, queue, OS primitive
  - Semaphore: shared access to critical section with (int) state
- All three primitives are equally powerful
  - Each primitive can be used to implement both other primitives
  - Performance may differ!
- Synchronization is challenging and may introduce different types of bugs such as atomicity violation, order violation, or deadlocks.