# CS323 Operating Systems

## Semaphores

Mathias Payer and Sanidhya Kashyap

EPFL, Fall 2021

# 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.

```
bool done = false;

/* called in the child to signal termination */
void thr_exit() {
  done = true;
}
/* called in the parent to wait for a child thread */
void thr_join() {
  while (!done);
}
```

# 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

# 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
  - Usually implemented as queues
  - Another thread signals the waiting thread

# Condition variables (CV)

- A CV allows a thread to wait for a condition
  - Usually implemented as queues
  - Another thread signals the waiting thread
- 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

```c
bool done = false;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
/* called in the child to signal termination */
void thr_exit() {
  pthread_mutex_lock(&m);
  done = true;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}
/* called in the parent to wait for a child thread */
void thr_join() {
  pthread_mutex_lock(&m);
  while (!done)
    pthread_cond_wait(&c, &m);
  pthread_mutex_unlock(&m);
}
```

# 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
- Principle: checking condition before sleeping
  - Thread may have already exited, i.e., no need to wait
- Principle: `while` instead of `if` when waiting
  - Multiple threads could be woken up, racing for `done` flag

# Signal parent that child has exited (3)

- Lock (mutex) for concurrent access to `done` protects against missed updates
  - Parent reads `done == false` but is interrupted
  - Child sets `done = true` and signals but noone is waiting
  - Parent continues and goes to sleep (forever)
- Lock is therefore required for wait/signal synchronization

Demo: `22-thread_exit.c`

# 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)

# 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

# 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

```c
void *producer(void *arg) {
    unsigned int max = (unsigned int)arg;
    for (unsigned int i = 0; i < max; i++) {
        put(i); // store in shared buffer
    }
    return NULL;
}
void *consumer(void *arg) {
    unsigned int max = (unsigned int)arg;
    for (unsigned int i = 0; i < max; i++) {
        printf("%d\n", get(i)); // recv from buffer
    }
    return NULL;
}
pthread_t p, c;
pthread_create(&p, NULL, &producer, (void*)NUMITEMS);
pthread_create(&c, NULL, &consumer, (void*)NUMITEMS);
```

```
unsigned int buffer[BUFSIZE] = { 0 };
unsigned int cpos = 0, ppos = 0;

void put(unsigned int val) {
    buffer[ppos] = val;
    ppos = (ppos + 1) % BUFSIZE;
}

unsigned int get() {
    unsigned long val = buffer[cpos];
    cpos = (cpos + 1) % BUFSIZE;
    return val;
}
```

What are the issues in this code?

# Concurrent programming: producer consumer

```c
unsigned int buffer[BUFSIZE] = { 0 };
unsigned int cpos = 0, ppos = 0;

void put(unsigned int val) {
    buffer[ppos] = val;
    ppos = (ppos + 1) % BUFSIZE;
}

unsigned int get() {
    unsigned long val = buffer[cpos];
    cpos = (cpos + 1) % BUFSIZE;
    return val;
}
```

What are the issues in this code?

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

# Producer/consumer: use semaphores!

```c
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

```
void put(unsigned int val) {
    /* we wait until there is buffer space available */
    sem_wait(&psem);

    /* store element in buffer */
    buffer[ppos] = val;
    ppos = (ppos + 1) % BUFSIZE;

    /* notify consumer that data is available */
    sem_post(&csem);
}
```

# Consumer: semaphores

```
unsigned int get() {
    /* wait until data is produced */
    sem_wait(&csem);

    /* consumer entry */
    unsigned long val = buffer[cpos];
    cpos = (cpos + 1) % BUFSIZE;

    /* notify producer that a space has freed up */
    sem_post(&psem);
    return val;
}
```

# Producer/consumer: remaining issues?

- We now synchronize between consumers and producers
  - Producer waits until buffer space is available
  - Consumer waits until data is ready

# 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)

Demo: `22-producer.c`

# Multiple producers: use locking!

```
/* mutex handling mutual exclusive access to ppos */
pthread_mutex_t pmutex = PTHREAD_MUTEX_INITIALIZER;

void put(unsigned int val) {
    unsigned int mypos;
    /* we wait until there is buffer space available */
    sem_wait(&psem);
    /* ppos is shared between all producers */
    pthread_mutex_lock(&pmutex);
    mypos = ppos;
    ppos = (ppos + 1) % BUFSIZE;
    /* store information in buffer */
    buffer[mypos] = val;
    pthread_mutex_unlock(&pmutex);
    sem_post(&csem);
}
```

# 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

```
sem_t sem;
sem_init(&sem, 1);

sem_wait(&sem);
... // critical section
sem_post(&sem);
```

# Implementing a semaphore with CV/locks

```
typedef struct {
    int value;              // sem value
    pthread_mutex_t lock; // access to sem
    pthread_cond_t cond;  // wait queue
} sem_t;

void sem_init(sem_t *s, int val) {
    s->value = val;
    pthread_mutex_init(&(s->lock), NULL);
    pthread_cond_init(&(s->cond), NULL);
}
```

# Implementing a semaphore with CV/locks

```
void sem_wait(sem_t *s) {
    pthread_mutex_lock(&(s->lock));
    while (s->value <= 0)
        pthread_cond_wait(&(s->cond), &(s->lock));
    s->value--;
    pthread_mutex_unlock(&(s->lock));
}

void sem_post(sem_t *s) {
    pthread_mutex_lock(&(s->lock));
    s->value++;
    pthread_cond_signal(&(s->cond));
    pthread_mutex_unlock(&(s->lock));
}
```

Demo: `22-semaphore.c`

# 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

```
void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1)
    sem_wait(&rw->wlock); // first r, also grab wlock
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0)
    sem_post(&rw->wlock); // last r, also release wlock
  sem_post(&rw->lock);
}
```

# 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.

```c
int shared = 24;

void T1() {
    if (shared > 23) {
        printf("Shared is >23: %d\n", shared);
    }
}
void T2() {
    shared = 12;
}
```

# Atomicity bugs

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

```c
int shared = 24;

void T1() {
    if (shared > 23) {
        printf("Shared is >23: %d\n", shared);
    }
}
void T2() {
    shared = 12;
}
```

- 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;
}
```

# 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);
}
```

# 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.

Don't forget to get your learning feedback through the Moodle quiz!