Lecture 4:

Synchronization problems
Condition variables, Monitors
Semaphores reminder

- How does it work?

  These people represent **waiting threads**. They aren't running on any CPU core.

  The bouncer represents a **semaphore**. He won't allow threads to proceed until instructed to do so.

- What if `signal()`/`post()` an empty queue:

  Let next 3 arrivals through.
More synchronization

• If wait and signal are both called some number of times, outcome is the same:

Source: preshing.com
Semaphore Limitations

• Can be hard to reason about synchronization

• Reason for waiting is embedded in the semaphore’s P() / wait() operation
  • E.g. “if count == 0, then sleep”
  • Sometimes you want a more complex wait condition
    • E.g. “if x == 0 and (y > 0 or z > 0), then sleep”
    • That condition must be checked outside the P()
  • But checking requires mutual exclusion
    • Easy to get stuck this way
Abstract Example

```
//shared state variables
int x, y, z; // some initial values
//mutual exclusion for shared vars
Semaphore mutex = 1;
//semaphore to wait if necessary
Semaphore no_go = 0;

compute_a_thing {
    P(mutex); //lock out others
    x = f1(x); //compute new x
    y = f2(y); //compute new y
    z = f3(z); //compute new z
    if (x != 0 || (y <= 0 && z <= 0))
        V(no_go);
    V(mutex); //up for grabs
}

use_a_thing {
    P(mutex); //lock out others
    if (x == 0 && (y > 0 || z > 0))
        P(no_go);
    // Now either x is non-zero
    // or y and z are non-positive
    // In this state, it is safe
    // to run “process” on x,y,z,
    // which may also change them.
    process(x, y, z);
    V(mutex);
}
```

• What can go wrong? How do we fix it?
What can go wrong?

• Deadlock.
  • If a use_a_thing thread has to wait on no_go, then no compute_a_thing thread can get mutex => no signal

• Changing state can “invalidate” previously sent signal (but signal can’t be revoked), for example:
  • T1: compute_a_thing, new x=1, y=1, z=1, good => V(no_go)
  • T2: compute_a_thing, new x=0, y=1, z=0, bad => just skip “if”
  • T3: use_a_thing, sees “bad” values for x,y,z, goes to P(no_go)
    • P() returns immediately (count was 1)
    • Even though x,y,z are not in usable state.

Must always re-check state of x,y,z before using them. Even after P() succeeds.
What can go wrong?

- Changing state can “invalidate” previously sent signal (but signal can’t be revoked) - other scenario:
  - T1: `compute_a_thing`, new $x=1, y=1, z=1$, $V(no\_go)$
  - T2: `compute_a_thing`, new $x=2, y=2, z=1$, $V(no\_go)$
  - T3: `use_a_thing`, sees $x \neq 0 =>$ good $=>$ proceed to run $process(x,y,z)$
    - Results in $x=0, y=1, z=0$ (“bad” values)
  - T4: `use_a_thing`, calls $P()$ and returns immediately
    - Even though $x,y,z$ are not in usable state.

Again, must always re-check state of $x,y,z$ before using them. Even after $P()$ succeeds.
Fixed Example?

//shared state variables
int x,y,z; // some initial values
//mutual exclusion for shared vars
Semaphore mutex = 1;
//semaphore to wait if necessary
Semaphore no_go = 0;

compute_a_thing {
    P(mutex); //lock out others
    x = f1(x);
    y = f2(y);
    z = f3(z);
    if (x != 0 || (y <= 0 && z <= 0))
        V(no_go);
    V(mutex); //up for grabs
}

use_a_thing {
    P(mutex); //lock out others
    while(x==0 && (y>0 || z>0)) {
        V(mutex); // no deadlock
        P(no_go);
        P(mutex);
    }
    // Now either x is non-zero
    // or y and z are non-positive
    process(x,y,z);
    V(mutex);
}

NOTE:
• Not very pretty, hard to read
• Semaphore count is not relevant
• May want to wake ALL waiters

• This is a very common pattern .. “Check condition, block and release mutex”
Condition Variables

- Abstract data type that encapsulates pattern of “release mutex, sleep, re-acquire mutex”
- Internal data is just a queue of waiting threads
  - Recall we didn’t need the semaphore count
- Operations are (each of these is atomic) – in pseudocode:
  - `cv_wait(struct cv *cv, struct mtx *mutex)`
    - Releases lock, waits, re-acquires mutex before return
  - `cv_signal(struct cv *cv)`
    - Wake one enqueued thread
  - `cv_broadcast(struct cv *cv)`
    - Wakes all enqueued threads

**CAUTION:** if no one is waiting, signal or broadcast has no effect. Not recorded for later use, as with semaphore V().
Using Condition Variables

- **Always** used together with locks
- The lock protects the shared data that is modified and tested when deciding whether to wait or signal/broadcast
- General Usage:

  ```
  lock_acquire(lock);
  while(condition not true) {
    cv_wait(cond, lock);
  }
  ... // do stuff
  cv_signal(cond); //or cv_broadcast(cond)
  lock_release(lock);
  ```
Pthreads Condition Variables API

```c
pthread_mutex_t mutex;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
(or pthread_cond_init() alternative)

pthread_cond_wait(&cv, &mutex);
pthread_cond_signal(&cv);
pthread_cond_broadcast(&cv);
```
Producer-Consumer revisited

Why do we need synchronization?

How do we solve the problem?

- We could use semaphores
- Or condition variables