- Project 3 is due Monday 3/11

- Midterm is next Wednesday 3/13 at 5.15pm, details on Piazza

- Discussion: Midterm review, Q&A

- Fill out mid semester course evaluation https://aefis.wisc.edu/
Concurrency abstractions
   How to implement semaphores?
   What are common pitfalls with concurrent execution?
RECAP
CONCURRENCY OBJECTIVES

**Mutual exclusion** (e.g., A and B don’t run at same time)
solved with *locks*

**Ordering** (e.g., B runs after A does something)
solved with *condition variables* and *semaphores*
**SUMMARY: CONDITION VARIABLES**

**wait** (cond_t *cv, mutex_t *lock)
- assumes the lock is held when wait() is called
- 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, just return, doing nothing
SUMMARY: RULES OF THUMB FOR CVS

1. Keep state in addition to CV’s

2. Always do wait/signal with lock held

3. Whenever thread wakes from waiting, recheck state
Semaphore Operations

Allocate and Initialize

```c
sem_t sem;
sem_init(sem_t *s, int initval) {
    s->value = initval;
}
```

User cannot read or write value directly after initialization.

Wait or Test (sometime P() for Dutch) `sem_wait(sem_t*)`
Decrement sem value, waits until value of sem is \( \geq 0 \)

Signal or Post (sometime V() for Dutch) `sem_post(sem_t*)`
Increment sem value, then wake a single waiter
A pipe may have many writers and readers

Internally, there is a finite-sized buffer

Writers add data to the buffer
   - Writers have to wait if buffer is full

Readers remove data from the buffer
   - Readers have to wait if buffer is empty
PRODUCER/CONSUMER: SEMAPHORES #1

Single producer thread, single consumer thread
Single shared buffer between producer and consumer

Use 2 semaphores
- emptyBuffer: Initialize to ________
- fullBuffer: Initialize to __________

Producer
while (1) {
  sem_wait(&emptyBuffer);
  Fill(&buffer);
  sem_signal(&fullBuffer);
}

Consumer
while (1) {
  sem_wait(&fullBuffer);
  Use(&buffer);
  sem_signal(&emptyBuffer);
}
Single producer thread, single consumer thread
Shared buffer with \( N \) elements between producer and consumer
Use 2 semaphores
   - emptyBuffer: Initialize to ____________
   - fullBuffer: Initialize to ____________

Producer
\( i = 0; \)
while (1) {
    sem_wait(&emptyBuffer);
    Fill(&buffer[i]);
    \( i = (i+1)\%N; \)
    sem_signal(&fullBuffer);
}

Consumer
\( j = 0; \)
While (1) {
    sem_wait(&fullBuffer);
    Use(&buffer[j]);
    \( j = (j+1)\%N; \)
    sem_signal(&emptyBuffer);
}
Final case:
- Multiple producer threads, multiple consumer threads
- Shared buffer with N elements between producer and consumer

Requirements
- Each consumer must grab unique filled element
- Each producer must grab unique empty element
PRODUCER/CONSUMER: MULTIPLE THREADS

Producer

while (1) {
    sem_wait(&emptyBuffer);
    my_i = findempty(&buffer);
    Fill(&buffer[my_i]);
    sem_signal(&fullBuffer);
}

Consumer

while (1) {
    sem_wait(&fullBuffer);
    my_j = findfull(&buffer);
    Use(&buffer[my_j]);
    sem_signal(&emptyBuffer);
}

Are my_i and my_j private or shared? Where is mutual exclusion needed???
PRODUCER/CONSUMER: MULTIPLE THREADS

Does this work?

Producer #1

```
sem_wait(&mutex);
sem_wait(&emptyBuffer);
my_i = findempty(&buffer);
Fill(&buffer[my_i]);
sem_signal(&fullBuffer);
sem_signal(&mutex);
```

Consumer #1

```
sem_wait(&mutex);
sem_wait(&fullBuffer);
my_j = findfull(&buffer);
Use(&buffer[my_j]);
sem_signal(&emptyBuffer);
sem_signal(&mutex);
```
**PRODUCER/CONSUMER: MULTIPLE THREADS**

Works, but limits concurrency:
Only 1 thread at a time can be using or filling different buffers
PRODUCER/CONSUMER: MULTIPLE THREADS

Producer #3

```c
sem_wait(&emptyBuffer);
sem_wait(&mutex);
myi = findempty(&buffer);
sem_signal(&mutex);
Fill(&buffer[myi]);
sem_signal(&fullBuffer);
```

Consumer #3

```c
sem_wait(&fullBuffer);
sem_wait(&mutex);
myj = findfull(&buffer);
sem_signal(&mutex);
Use(&buffer[myj]);
sem_signal(&emptyBuffer);
```

Works and increases concurrency; only finding a buffer is protected by `mutex`;
Filling or Using different buffers can proceed concurrently.
READER/WRITER LOCKS

Let multiple reader threads grab lock (shared)
Only one writer thread can grab lock (exclusive)
  – No reader threads
  – No other writer threads

Let us see if we can understand code
typedef struct _rwlock_t {
    sem_t lock;  // "reader"
    sem_t writelock;
    int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 1);
    sem_init(&rw->writelock, 1);
}
void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers++;
    if (rw->readers == 1)
        sem_wait(&rw->writelock);
    sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers--;
    if (rw->readers == 0)
        sem_post(&rw->writelock);
    sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
    sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
    sem_post(&rw->writelock);
}
READER WRITER LOCKS

T1: acquire_readlock()  →  RUNNING
T2: acquire_readlock()  →  RUNNING
T3: acquire_writelock()  →  BLOCKED WRITE LOCK

What is the status of T2?

T6: acquire_writelock()
T4: acquire_readlock()  →  WAITING FOR READ LOCK
T5: acquire_readlock()  →  WAITING FOR READ LOCK

What is the status of T4?
Typedef struct {
    int value;
    cond_t cond;
    lock_t lock;
} sem_t;

void sem_init(sem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    lock_init(&s->lock);
}

sem_wait(): Decrement and waits until value >= 0
sem_post(): Increment value, then wake a single waiter
Build Semaphore from Lock and CV

```c
sem_wait(sem_t *s) {
    lock_acquire(&s->lock);
    s->value--;
    while (s->value < 0)
        cond_wait(&s->cond);
    lock_release(&s->lock);
}

sem_post(sem_t *s) {
    lock_acquire(&s->lock);
    s->value++;
    cond_signal(&s->cond);
    lock_release(&s->lock);
}
```

**Locks vs. Semaphores vs. CV's**

- **Locks**: Easiest to use but single-threaded
- **Semaphores**: Equally powerful
- **CV's**: Most powerful

**sem_wait():** Decrement and waits until value >= 0

**sem_post():** Increment value, then wake a single waiter
Semaphores are equivalent to locks + condition variables
  – Can be used for both mutual exclusion and ordering
Semaphores contain **state**
  – How they are initialized depends on how they will be used
  – Init to 0: Join (1 thread must arrive first, then other)
  – Init to N: Number of available resources

sem_wait(): Decrement and waits until value >= 0
sem_post(): Increment value, then wake a single waiter (atomic)
Can use semaphores in producer/consumer and for reader/writer locks
CONCURRENCY BUGS
"The accidents occurred when the high-power electron beam was activated instead of the intended low power beam, and without the beam spreader plate rotated into place. Previous models had hardware interlocks in place to prevent this, but Therac-25 had removed them, depending instead on software interlocks for safety. The software interlock could fail due to a race condition."

"...in three cases, the injured patients later died."

Lu et al. [ASPLOS 2008]:
For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.
Thread 1:

```c
if (thd->proc_info) {
    ...
    fputs(thd->proc_info, ...);
    ...
}
```

Thread 2:
```
thd->proc_info = NULL;
```

What's wrong?
Thread 1:

pthread_mutex_lock(&lock);
if (thd->proc_info) {
    ...
    fputs(thd->proc_info, ...);
    ...
}

pthread_mutex_unlock(&lock);

Thread 2:

pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
**CONCURRENCY STUDY**

Lu *etal.* [ASPLOS 2008]:
For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.
Ordering: Mozilla

Thread 1:

```c
void init() {
...
    mThread = PR_CreateThread(mMain, ...);
...
}
```

Thread 2:

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

What's wrong?
Thread 1:
void init() {
    ...
    mThread = PR_CreateThread(mMain, ...);

    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);
    ...
}

Thread 2:
void mMain(...) {
    ...

    mutex_lock(&mtLock);
    while (mtInit == 0) {
        Cond_wait(&mtCond, &mtLock);
        Mutex_unlock(&mtLock);
    }
    mState = mThread->State;
    ...
}
Lu et al. [ASPLOS 2008]:
For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.
DEADLOCK

No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does
Thread 1:

lock(&A);
lock(&B);

Thread 2:

lock(&B);
lock(&A);
Circular Dependency

Thread 1 holds Lock A, which is wanted by Lock B. Lock B holds Lock A, which is wanted by Thread 2.

This forms a circular dependency.
FIX DEADLOCKED CODE

Thread 1:
lock(&A);
lock(&B);

Thread 2:
lock(&B);
lock(&A);
**NON-CIRCULAR DEPENDENCY**

1. **Thread 1** holds **Lock A**
2. **Lock B** is wanted by **Thread 2**
3. **Lock A** is wanted by **Thread 2**
4. **Thread 1** holds **Lock B**

The diagram illustrates a non-circular dependency where **Thread 1** holds **Lock A**, which is wanted by **Thread 2**. **Lock B** is held by **Thread 1** and also wanted by **Thread 2**.
set_t *set_intersection (set_t *s1, set_t *s2) {
    set_t *rv = malloc(sizeof(*rv));
    mutex_lock(&s1->lock);
    mutex_lock(&s2->lock);
    for(int i=0; i<s1->len; i++) {
        if(set_contains(s2, s1->items[i])
            set_add(rv, s1->items[i]);
    }
    mutex_unlock(&s2->lock);
    mutex_unlock(&s1->lock);
}

Thread 1: rv = set_intersection(setA, setB);
Thread 2: rv = set_intersection(setB, setA);
ENCAPSULATION

Modularity can make it harder to see deadlocks

Solution?

if (m1 > m2) {
    // grab locks in high-to-low address order
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}
Deadlocks can only happen with these four conditions:

1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

Can eliminate deadlock by eliminating any one condition
1. MUTUAL EXCLUSION

Problem: Threads claim exclusive control of resources that they require.

Strategy: Eliminate locks!

Try to replace locks with atomic primitive:

```c
int CompAndSwap(int *addr, int expected, int new)
Returns 0 fail, 1 success
```
void add (int *val, int amt) {
    Mutex_lock(&m);
    *val += amt;
    Mutex_unlock(&m);
}

void add (int *val, int amt) {
    do {
        int old = *value;
    } while(!CompAndSwap(val, ___, old+amt);
}
void insert (int val) {
    node_t *n = Malloc(sizeof(*n));
    n->val = val;
    lock(&m);
    n->next = head;
    head = n;
    unlock(&m);
}

void insert (int val) {
    node_t *n = Malloc(sizeof(*n));
    n->val = val;
    do {
        n->next = head;
    } while (!CompAndSwap(&head, n->next, n));
}
2. HOLD-AND-WAIT

Problem: Threads hold resources allocated to them while waiting for additional resources

Strategy: Acquire all locks atomically **once**. Can release locks over time, but cannot acquire again until all have been released

How to do this? Use a meta lock:

Disadvantages?
3. NO PREEMPTION

Problem: Resources (e.g., locks) cannot be forcibly removed from threads that are
Strategy: if thread can’t get what it wants, release what it holds

```
top:
  lock(A);
  if (trylock(B) == -1) {
    unlock(A);
    goto top;
  }
  ...  
```

Disadvantages?
4. CIRCULAR WAIT

Circular chain of threads such that each thread holds a resource (e.g., lock) being requested by next thread in the chain.

Strategy:
- decide which locks should be acquired before others
- if A before B, never acquire A if B is already held!
- document this, and write code accordingly

Works well if system has distinct layers
NEXT STEPS

Project 3: Out now!
Midterm details posted

Next class: Midterm review