CONCURRENCY: CONDITION VARIABLES, SEMAPHORES

Shivaram Venkataraman
CS 537, Spring 2019
- Project 3 is due next Monday 3/11

- Midterm is next Wednesday 3/13 at 5.15pm, details on Piazza
  - Includes all material covered till 3/12
  - Work out practice midterms before discussion!
Concurrency abstractions

How to implement producer-consumer pattern with CV/locks?

How can semaphores help this implementation?
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
ORDERING EXAMPLE: JOIN

```
pthread_t p1, p2;
Pthread_create(&p1, NULL, mythread, "A");
Pthread_create(&p2, NULL, mythread, "B");
// join waits for the threads to finish
Pthread_join(p1, NULL);
Pthread_join(p2, NULL);
printf("main: done\n [balance: %d]\n [should: %d]\n",
        balance, max*2);
return 0;
```

how to implement join()?
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
JOIN IMPLEMENTATION: CORRECT

Parent:

```c
void thread_join() {
    Mutex_lock(&m);  // w
    if (done == 0)   // x
        Cond_wait(&c, &m); // y
    Mutex_unlock(&m); // z
}
```

Child:

```c
void thread_exit() {
    Mutex_lock(&m);   // a
    done = 1;         // b
    Cond_signal(&c);  // c
    Mutex_unlock(&m); // d
}
```

Use mutex to ensure no race between interacting with state and wait/signal
RULES OF THUMB

Keep state in addition to CV’s!
CV’s are used to signal threads when state changes
If state is already as needed, thread doesn’t wait for a signal!

Hold mutex lock while calling wait/signal
Ensures no race between interacting with state and wait/signal
PRODUCER/CONSUMER PROBLEM
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
EXAMPLE: UNIX PIPES

start

Buf:

end
Implementation:
- reads/writes to buffer require locking
- when buffers are full, writers must wait
- when buffers are empty, readers must wait
PRODUCER/CONSUMER PROBLEM

Producers generate data (like pipe writers)

Consumers grab data and process it (like pipe readers)

Producer/consumer problems are frequent in systems (e.g. web servers)

General strategy use condition variables to:
  make producers wait when buffers are full
  make consumers wait when there is nothing to consume
Produce/Consumer Example

Start with easy case:
  - 1 producer thread
  - 1 consumer thread
  - 1 shared buffer to fill/consume (max = 1)

Numfull = number of buffers currently filled
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        if (numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}

Thread 1 state:

void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if (numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}

Thread 2 state:
WHAT ABOUT 2 CONSUMERS?

Can you find a problematic timeline with 2 consumers (still 1 producer)?
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m); // p1
        if (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
        if (numfull == 0) // c2
            Cond_wait(&cond, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&cond); // c5
        Mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}
HOW TO WAKE THE RIGHT THREAD?

Wake all the threads!?  
Better solution (usually): use two condition variables
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i); // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); // p6
    }
}

void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if (numfull == 0) // p2
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
PRODUCER/CONSUMER: TWO CVS

void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i); // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); // p6
    }
}

void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m); // c1
        if (numfull == 0) // c2
            Cond_wait(&fill, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&empty); // c5
        Mutex_unlock(&m); // c6
    }
}
PRODUCER/CONSUMER: TWO CVS AND WHILE

```c
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        while (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i); // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); // p6
    }
}

void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        while (numfull == 0) // p2
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
```
GOOD RULE OF THUMB 3

Whenever a lock is acquired, **recheck assumptions** about state!
Another thread could grab lock in between signal and wakeup from wait

Note that some libraries also have “spurious wakeups”
(may wake multiple waiting threads at signal or at any time)
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
**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
INTRODUCING SEMAPHORES

Condition variables have no state (other than waiting queue)
  – Programmer must track additional state

Semaphores have state: track integer value
  – State cannot be directly accessed by user program, but state determines behavior of semaphore operations
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*)`
Decrements sem value, Waits until value of sem is >= 0

**Signal or Post** *(sometime V() for Dutch)*

`sem_post(sem_t*)`
Increment sem value, then wake a single waiter
typedef struct __lock_t {
    sem_t sem;
} lock_t;

void init(lock_t *lock) {
}

void acquire(lock_t *lock) {
}

void release(lock_t *lock) {
}

sem_init(sem_t*, int initial)
sem_wait(sem_t*): Decrement, wait until value >= 0
sem_post(sem_t*): Increment value
    then wake a single waiter

https://tinyurl.com/cs537-sp19-bunny7
JOIN WITH CV VS SEMAPHORES

void thread_join() {
    Mutex_lock(&m);  // w
    if (done == 0)   // x
        Cond_wait(&c, &m);  // y
    Mutex_unlock(&m);  // z
}

void thread_exit() {
    Mutex_lock(&m);  // a
    done = 1;       // b
    Cond_signal(&c);  // c
    Mutex_unlock(&m);  // d
}

sem_t s;
sem_init(&s, ___-);

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

void thread_exit() {
    sem_post(&s)
}
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);
}
**PRODUCER/CONSUMER: SEMAPHORE #3**

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???
Consider three possible locations for mutual exclusion
Which work?? Which is best??

Producer 

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

Consumer 

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

Producer #2

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

Consumer #2

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

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.
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;
  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);
}

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

rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T1: release_readlock()
T4: acquire_readlock()
T5: acquire_readlock()
T3: release_writelock()
// what happens next?
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(): Waits until value > 0, then decrement (atomic)

Sem\_post(): Increment value, then wake a single waiter (atomic)

Can use semaphores in producer/consumer and for reader/writer locks
NEXT STEPS

Project 3: Out now!
Midterm details posted

Next class: How to build a semaphore, deadlocks