OSTEP
Concurrency
Semaphores

Questions answered in this lecture:

Review: How to implement join with condition variables?

Review: How to implement producer/consumer with condition variables?

What is the difference between **semaphores** and condition variables?

How to implement a **lock** with semaphores?

How to implement semaphores with locks and condition variables?

How to implement **join** and producer/consumer with semaphores?

How to implement **reader/writer locks** with semaphores?

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*

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

broadcast(cond_t *cv)
- wake all waiting threads (if >= 1 thread is waiting)
- if there are no waiting thread, just return, doing nothing

Join Implementation: Correct

Parent:

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

Child:

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

Parent:	w	x	y				z
Child:				a	b	c	

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

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.

Use condition variables to:
make producers wait when buffers are full
make consumers wait when there is nothing to consume

Broken Implementation of Producer Consumer

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m); // p1
                while(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
                while(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
        }
}
```

			wait()			wait()				signal()				wait()			signal()
Producer:							p1	p2	p4	p5	p6	p1	p2	p3			
Consumer1:	c1	c2	c3														
Consumer2:				c1	c2	c3									c2	c4	c5

does last signal wake producer or consumer2?

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
	}
}
```

Is this correct? Can you find a bad schedule?

1. consumer1 waits because numfull == 0
2. producer increments numfull, wakes consumer1
3. before consumer1 runs, consumer2 runs, grabs entry, sets numfull=0.
4. consumer2 then reads bad data.

Producer:				p1	p2	p4	p5	p6						
Consumer1:	c1	c2	c3											c4! ERROR
Consumer2:									c1	c2	c4	c5	c6	

CV Rule of Thumb 3

Whenever a lock is acquired, recheck assumptions about state!

Use “while” instead of “if”

Possible for another thread to grab lock between signal and wakeup from wait

- Difference between Mesa (practical implementation) and Hoare (theoretical) semantics
- Signal() simply makes a thread runnable, does not guarantee th read run next

Note that some libraries also have “spurious wakeups”

- May wake multiple waiting threads at signal or at any time

Producer/Consumer: Two CVs and WHILE

```
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)
                        Cond_wait(&fill, &m);
                int tmp = do_get();
                Cond_signal(&empty);
                Mutex_unlock(&m);
        }
}
```

Is this correct? Can you find a bad schedule?

Correct!

- no concurrent access to shared state
- every time lock is acquired, assumptions are reevaluated
- a consumer will get to run after every do_fill()
- a producer will get to run after every do_get()

Summary: rules of thumb for CVs

Keep state in addition to CV's

Always do wait/signal with lock held

Whenever thread wakes from waiting, recheck state

Semaphore: A definition

An object with an integer value

- We can manipulate with two routines; `sem_wait()` and `sem_post()`.

- Initialization

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // initialize s to the value 1
```

  - Declare a semaphore `s` and initialize it to the value 1
  - The second argument, 0, indicates that the semaphore is shared between *threads in the same process*.

Semaphore: Interact with semaphore

- `sem_wait()`

```
int sem_wait(sem_t *s) {
  decrement the value of semaphore s by one
  wait if value of semaphore s is negative
}
```

  - If the value of the semaphore was *one* or *higher* when called `sem_wait()`, **return right away**.
  - It will cause the caller to suspend execution waiting for a subsequent post.
  - When negative, the value of the semaphore is equal to the number of waiting threads.

Semaphore: Interact with semaphore (Cont.)

- `sem_post()`

```
int sem_post(sem_t *s) {
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```

- Simply **increments** the value of the semaphore.
- If there is a thread waiting to be woken, **wakes** one of them up.

Binary Semaphores (Locks)

- What should **X** be?
  - The initial value should be **1**.

```
sem_t m;
sem_init(&m, 0, X); // initialize semaphore to X; what should X be?

sem_wait(&m);
//critical section here
sem_post(&m);
```

Thread Trace: Single Thread Using A Semaphore

Value of Semaphore	Thread 0	Thread 1
1		
1	`call sema_wait()`	
0	`sem_wait() returns`	
0	`(crit sect)`	
0	`call sem_post()`	
1	`sem_post() returns`	

Thread Trace: Two Threads Using A Semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	`call sem_wait()`	Running		Ready
0	`sem_wait() retruns`	Running		Ready
0	`(crit set: begin)`	Running		Ready
0	*Interrupt; Switch → T1*	Ready		Running
0		Ready	`call sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem < 0)→sleep`	sleeping
-1		Running	*Switch → T0*	sleeping
-1	`(crit sect: end)`	Running		sleeping
-1	`call sem_post()`	Running		sleeping
0	`increment sem`	Running		sleeping
0	`wake(T1)`	Running		Ready
0	`sem_post() returns`	Running		Ready
0	*Interrupt; Switch → T1*	Ready		Running
0		Ready	`sem_wait() retruns`	Running
0		Ready	`(crit sect)`	Running
0		Ready	`call sem_post()`	Running
1		Ready	`sem_post() returns`	Running

Semaphores As Condition Variables

```
sem_t s;

void *
child(void *arg) {
   printf("child\n");
   sem_post(&s); // signal here: child is done
   return NULL;
}

 int
 main(int argc, char *argv[]) {
   sem_init(&s, 0, X); // what should X be?
   printf("parent: begin\n");
   pthread_t c;
   pthread_create(c, NULL, child, NULL);
   sem_wait(&s); // wait here for child
   printf("parent: end\n");
   return 0;
 }
```

A Parent Waiting For Its Child

```
parent: begin
child
parent: end
```

The execution result

- What should **x** be?
  - The value of semaphore should be set to is **0**.

Thread Trace: Parent Waiting For Child (Case 1)

- The parent call `sem_wait()` before the child has called `sem_post()`.

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*`(Child exists; is runnable)`*	Ready
0	`call sem_wait()`	Running		Ready
-1	`decrement sem`	Running		Ready
-1	`(sem < 0)→sleep`	sleeping		Ready
-1	*`Switch→Child`*	sleeping	`child runs`	Running
-1		sleeping	`call sem_post()`	Running
0		sleeping	`increment sem`	Running
0		Ready	`wake(Parent)`	Running
0		Ready	`sem_post() returns`	Running
0		Ready	*`Interrupt; Switch→Parent`*	Ready
0	`sem_wait() retruns`	Running		Ready

Thread Trace: Parent Waiting For Child (Case 2)

- The child runs to completion before the parent call `sem_wait()`.

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	*`Interrupt; switch→Child`*	Ready	`child runs`	Running
0		Ready	`call sem_post()`	Running
1		Ready	`increment sem`	Running
1		Ready	`wake(nobody)`	Running
1		Ready	`sem_post() returns`	Running
1	`parent runs`	Running	*`Interrupt; Switch→Parent`*	Ready
1	`call sem_wait()`	Running		Ready
0	`decrement sem`	Running		Ready
0	`(sem<0)→awake`	Running		Ready
0	`sem_wait() retruns`	Running		Ready

The Producer/Consumer (Bounded-Buffer) Problem

- **Producer**: `put()` interface
  - Wait for a buffer to become *empty* in order to put data into it.
- **Consumer**: `get()` interface
  - Wait for a buffer to become *filled* before using it.

```
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
  buffer[fill] = value;    // line f1
  fill = (fill + 1) % MAX;       // line f2
}

int get() {
  int tmp = buffer[use];  // line g1
  use = (use + 1) % MAX;  // line g2
  return tmp;
}
```

The Producer/Consumer (Bounded-Buffer) Problem

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
       sem_wait(&empty);        // line P1
       put(i);                  // line P2
       sem_post(&full);         // line P3
 }
}

void *consumer(void *arg) {
 int i, tmp = 0;
 while (tmp != -1) {
       sem_wait(&full);         // line C1
       tmp = get();             // line C2
       sem_post(&empty);        // line C3
       printf("%d\n", tmp);
 }
}
…
```

First Attempt: Adding the Full and Empty Conditions

A Solution: Adding Mutual Exclusion

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
        sem_wait(&mutex);        // line p0 (NEW LINE)
        sem_wait(&empty);        // line p1
        put(i);                  // line p2
        sem_post(&full);         // line p3
        sem_post(&mutex);        // line p4 (NEW LINE)
  }
}

(Cont.)
```

Adding Mutual Exclusion (Incorrectly)

A Solution: Adding Mutual Exclusion

```
(Cont.)
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
            sem_wait(&mutex);        // line c0 (NEW LINE)
            sem_wait(&full);         // line c1
            int tmp = get();         // line c2
            sem_post(&empty);        // line c3
            sem_post(&mutex);        // line c4 (NEW LINE)
            printf("%d\n", tmp);
    }
}
```

Adding Mutual Exclusion (Incorrectly)

A Solution: Adding Mutual Exclusion (Cont.)

- Imagine two thread: one producer and one consumer.
  - The consumer **acquire** the `mutex` (line c0).
  - The consumer **calls** `sem_wait()` on the full semaphore (line c1).
  - The consumer is **blocked** and **yield** the CPU.
    - The consumer <u>still holds the mutex</u>!
  - The producer **calls** `sem_wait()` on the binary `mutex` semaphore (line p0).
  - The producer is now **stuck** waiting too. a classic deadlock.

Finally, A Working Solution

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
        sem_wait(&empty);       // line p1
        sem_wait(&mutex);       // line p1.5 (MOVED MUTEX HERE…)
        put(i);                 // line p2
        sem_post(&mutex);       // line p2.5 (… AND HERE)
        sem_post(&full);        // line p3
  }
 }

(Cont.)
```

Adding Mutual Exclusion (Correctly)

Finally, A Working Solution

```
(Cont.)
  void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
          sem_wait(&full);          // line c1
          sem_wait(&mutex);         // line c1.5 (MOVED MUTEX HERE…)
          int tmp = get();          // line c2
          sem_post(&mutex);         // line c2.5 (… AND HERE)
          sem_post(&empty);         // line c3
          printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // …
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with …
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
    // …
}
```

Adding Mutual Exclusion (Correctly)

Reader-Writer Locks

- Imagine a number of concurrent list operations, including **inserts** and simple **lookups**.
  - **insert:**
    - Change the state of the list
    - A traditional critical section makes sense.
  - **lookup:**
    - Simply *read* the data structure.
    - As long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently.

This special type of lock is known as a reader-write lock.

Reader-Writer

- Only **a single writer** can acquire the lock.
- Once a reader has acquired a read lock,
  - **More readers** will be allowed to acquire the read lock too.
  - A writer will <u>have to wait</u> until all readers are finished.

```
typedef struct _rwlock_t {
   sem_t lock;        // binary semaphore (basic lock)
   sem_t writelock;   // used to allow ONE writer or MANY readers
   int readers;       // count of readers reading in critical section
} rwlock_t;

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

void rwlock_acquire_readlock(rwlock_t *rw) {
   sem_wait(&rw->lock);
   …
```

A Reader-Writer Locks (Cont.)

```
    rw->readers++;
    if (rw->readers == 1)
          sem_wait(&rw->writelock); // first reader acquires 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); // last reader releases 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);
}
```

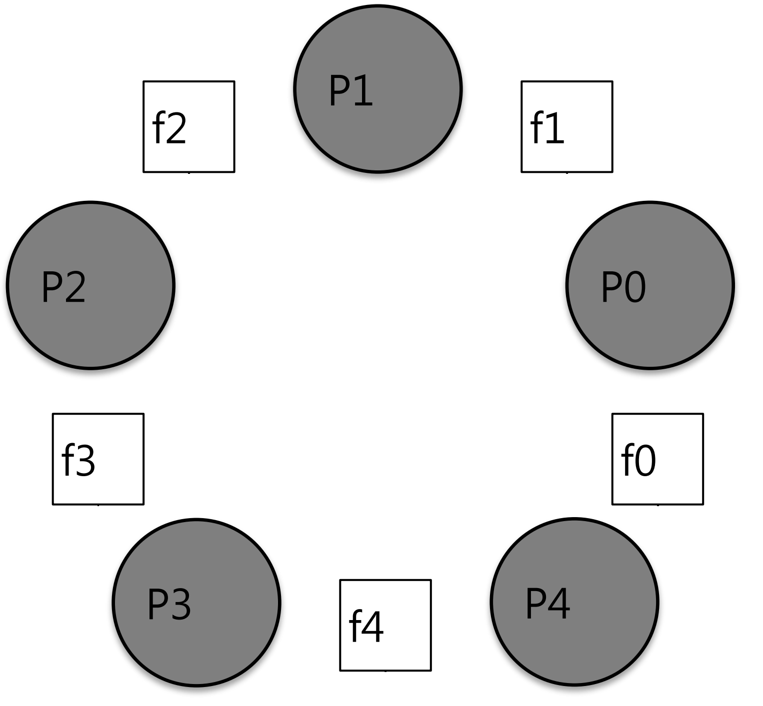
A Reader-Writer Locks (Cont.)

- The reader-writer locks have fairness problem.
  - It would be relatively easy for reader to **starve writer**.
  - How to prevent more readers from entering the lock once a writer is waiting?

  - 숙제 (due 10/26 hard copy로 수업시간에 낼것, 1장으로)

The Dining Philosophers

- Assume there are five "**philosophers**" sitting around a table.
  - Between each pair of philosophers is a single fork (five total).
  - The philosophers each have times where they **think**, and don't need any forks, and times where they **eat**.
  - In order to *eat*, a philosopher needs two forks, both the one on their *left* and the one on their *right*.
  - **The contention for these forks.**



The Dining Philosophers (Cont.)

- Key challenge
  - There is **no deadlock**.
  - **No** philosopher **starves** and never gets to eat.
  - **Concurrency** is high.

```
while (1) {
        think();
        getforks();
        eat();
        putforks();
}
```

Basic loop of each philosopher

```
// helper functions
int left(int p) { return p; }

int right(int p) {
        return (p + 1) % 5;
}
```

Helper functions (Downey's solutions)

- Philosopher `p` wishes to refer to the for on their left → call `left(p)`.
- Philosopher `p` wishes to refer to the for on their right → call `right(p)`.

The Dining Philosophers (Cont.)

- We need some **semaphore**, one for each fork: `sem_t forks[5]`.

```
void getforks() {
   sem_wait(forks[left(p)]);
   sem_wait(forks[right(p)]);
}

void putforks() {
   sem_post(forks[left(p)]);
   sem_post(forks[right(p)]);
}
```

The `getforks()` and `putforks()` Routines (Broken Solution)

- Deadlock occur!
  - If each philosopher happens to **grab the fork on their left** before any philosopher can grab the fork on their right.
  - Each will be stuck *holding one fork* and waiting for another, *forever*.

A Solution: Breaking The Dependency

- Change how forks are acquired.
  - Let's assume that philosopher 4 acquire the forks in a *different order*.

```
void getforks() {
    if (p == 4) {
            sem_wait(forks[right(p)]);
            sem_wait(forks[left(p)]);
    } else {
            sem_wait(forks[left(p)]);
            sem_wait(forks[right(p)]);
    }
}
```

- There is no situation where each philosopher grabs one fork and is stuck waiting for another. **The cycle of waiting is broken**.

How To Implement Semaphores

- Build our own version of semaphores called Zemaphores

```
typedef struct __Zem_t {
   int value;
   pthread_cond_t cond;
   pthread_mutex_t lock;
} Zem_t;

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
   s->value = value;
   Cond_init(&s->cond);
   Mutex_init(&s->lock);
}

void Zem_wait(Zem_t *s) {
   Mutex_lock(&s->lock);
   while (s->value <= 0)
   Cond_wait(&s->cond, &s->lock);
   s->value--;
   Mutex_unlock(&s->lock);
}
…
```

How To Implement Semaphores (Cont.)

```c
void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

- Zemaphore don't maintain the invariant that *the value of* the semaphore.
  - The value never be lower than zero.
  - This behavior is **easier** to implement and **matches** the current Linux implementation.

Semaphore (LINUX actual)

- **sem_wait**(sem)
  - decrements (locks) the semaphore pointed to by *sem*.
  - If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately.
  - If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call
- **sem_post**(sem)
  - increments (unlocks) the semaphore pointed to by *sem*.
  - If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a [sem_wait( )](sem_wait) call will be woken up and proceed to lock the semaphore.
- **sem_init(sem_t** ***sem*****, int** ***pshared*****, unsigned int** ***value*****)**
  - pshared indicates sharing between threads(0)/processes(nonzero)

Condition Variable

Queue:

Queue:

A

wait()

Queue:

A

Queue

A B

wait()

Queue

Queue:

B

signal()

Queue:

B

Queue:

signal()

Queue:

Queue:

signal()

Queue:

nothing to do!

signal()

Queue:

C

Queue:

C

wait()

Queue:

C

If we weren’t careful, C may sleep forever.

Thread Queue: Signal Queue:

Thread Queue: Signal Queue:

A

wait()

Thread Queue: Signal Queue:

A

Thread Queue: Signal Queue:

signal()

Thread Queue: Signal Queue:

Thread Queue: Signal Queue:

signal

signal()

Thread Queue: Signal Queue:

signal

Thread Queue:

A

Signal Queue:

signal

wait()

Thread Queue: Signal Queue:

wait()

Thread Queue: Signal Queue:

signal was not lost do to some race condition!

wait()

Thread Queue: Signal Queue:

Actual Implementation

Use counter instead of Signal Queue
- all signals are the same

If the counter is positive, don't bother to queue a thread upon wait().

CV's don't keep extra state, so CV users must.
Semaphores keep extra state, so users sometimes don't.

Actual Definition (see handout)

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

sem_wait(sem_t *s) {
    s->value -= 1
    wait if s->value < 0
}

sem_post(sem_t *s) {
    s->value += 1
    wake one waiting thread (if there
   are any)
}
```

wait and post are atomic

Actual Definition (see handout)

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

sem_wait(sem_t *s) {
    s->value -= 1
    wait if s->value < 0
}

sem_post(sem_t *s) {
    s->value += 1
    wake one waiting thread (if there
   are any)
}
```

value = 4: 4 waiting signals

value = -3: 3 waiting threads

Join example

Join is simpler with semaphores than CV’s.

Join w/ CV

```
int done = 0;
mutex_t m = MUTEX_INIT;
cond_t c = COND_INIT;


    Mutex_lock(&m);
    done = 1;  cond_s
    ignal(&c);  Mutex
    _unlock(&m);



    Pthread_create(c, NULL, child, NULL);
    Mutex_lock(&m);  while(
    done == 0)
        Cond_wait(&c, &m);
    Mutex_unlock(&m);
```

Join w/ Semaphore

```
sem_t s;


    sem_post(&s);



    sem_init(&s, ?);


    Pthread_create(c, NULL, child, NULL);
    sem_wait(&s);
```

Join w/ Semaphore

```
sem_t s;


   sem_post(&s);



   sem_init(&s, ?);


   Pthread_create(c, NULL, child, NULL);
   sem_wait(&s);
```

Join w/ Semaphore

```
sem_t s;


   sem_post(&s);



   sem_init(&s, ?);
```

What is this int?

```
   Pthread_create(c, NULL, child, NULL);
   sem_wait(&s);
```

Join w/ Semaphore

```
sem_t s;


   sem_post(&s);



   sem_init(&s, ?);


   Pthread_create(c, NULL, child, NULL);
   sem_wait(&s);
```

Join w/ Semaphore

```
sem_t s;


   sem_post(&s);



   sem_init(&s, 0);


   Pthread_create(c, NULL, child, NULL);
   sem_wait(&s);
```

Run it!
(sem-join.c)

Equivalence Claim

Semaphores are equally powerful to Locks+CVs.
- what does this mean?

Either may be more convenient, but that's not relevant.

Equivalence means we can build each over the other.

Condition Variables vs 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

```
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 word)

Waits until value of sem is > 0, then decrements sem value

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

Increment sem value, then wake a single waiter

Join with CV vs Semaphores

CVs:

```
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
}
```

Semaphores:

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

```
sem_t s;
sem_init(&s, ???);
```

Initialize to 0 (so sem_wait() must wait...)

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

```
void thread_exit() {
                sem_post(&s)
}
```

Equivalence Claim

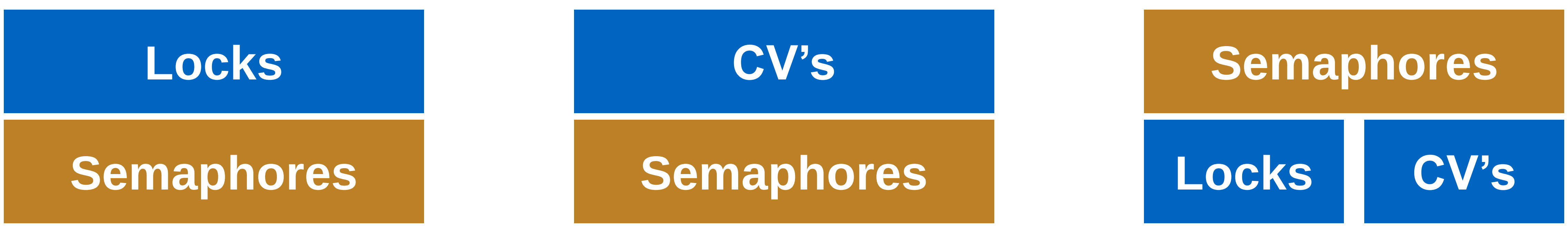
Semaphores are equally powerful to Locks+CVs
 - what does this mean?

One might be more convenient, but that's not relevant

Equivalence means each can be built from the other

Proof Steps

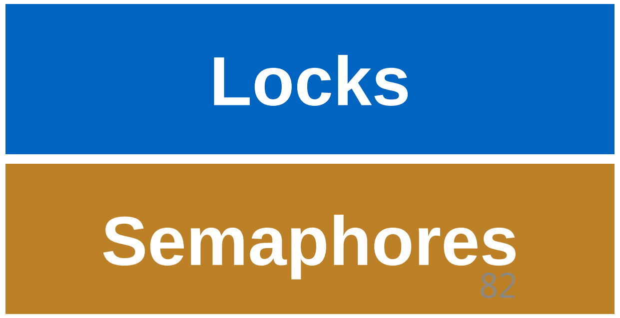
Want to show we can do these three things:



Build Lock from Semaphore

```
typedef struct __lock_t {
// whatever data structs you need go here
} lock_t;

void init(lock_t *lock) {
}

void acquire(lock_t *lock) {
}

void release(lock_t *lock) {
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter



Build Lock from Semaphore

```
typedef struct __lock_t {
        sem_t sem;
} lock_t;

void init(lock_t *lock) {
        sem_init(&lock->sem, ??);
}
void acquire(lock_t *lock) {
        sem_wait(&lock->sem);
}
void release(lock_t *lock) {
        sem_post(&lock->sem);
}
```

1 → 1 thread can grab lock

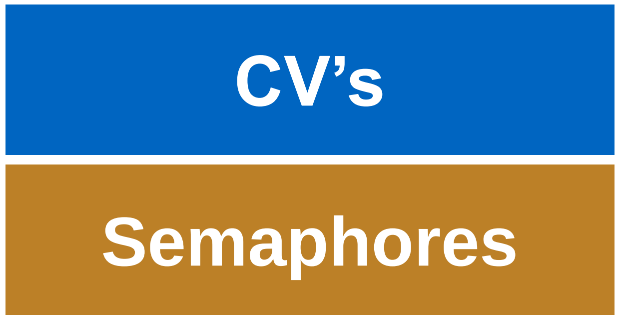
Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Locks

Semaphores

Building CV's over Semaphores

Possible, but really hard to do right



Read about Microsoft Research's attempts:
http://research.microsoft.com/pubs/64242/ImplementingCVs.pdf

Build Semaphore from Lock and CV

```
Typedef struct {
	// what goes here?


} sem_t;

Void sem_init(sem_t *s, int value) {
	// what goes here?


}
```

Semaphores

Locks | CV's

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Build Semaphore from Lock and CV

```
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);
}
```

Semaphores

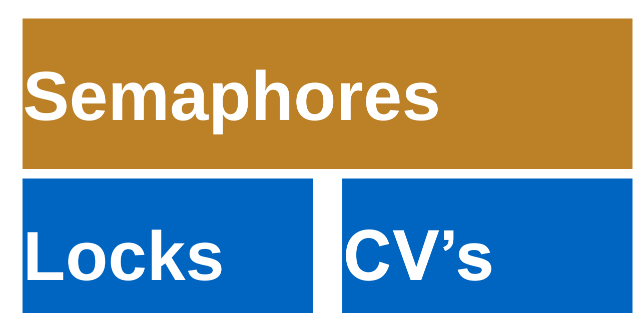
Locks   CV's

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Build Semaphore from Lock and CV

```
Sem_wait{sem_t *s) {
        // what goes here?




}
```

```
Sem_post{sem_t *s) {
        // what goes here?



}
```

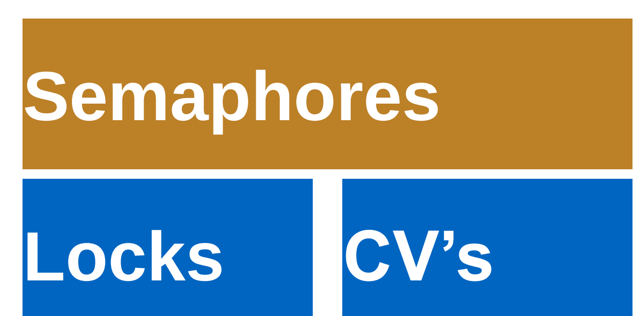


Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Build Semaphore from Lock and CV

```
Sem_wait{sem_t *s) {
      lock_acquire(&s->lock);
      // this stuff is atomic


      lock_release(&s->lock);
}
```

```
Sem_post{sem_t *s) {
      lock_acquire(&s->lock);
      // this stuff is atomic

      lock_release(&s->lock);
}
```



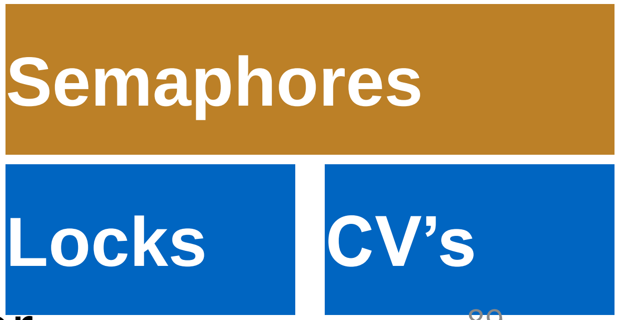
Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Build Semaphore from Lock and CV

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

```
Sem_post{sem_t *s) {
    lock_acquire(&s->lock);
    // this stuff is atomic

    lock_release(&s->lock);
}
```

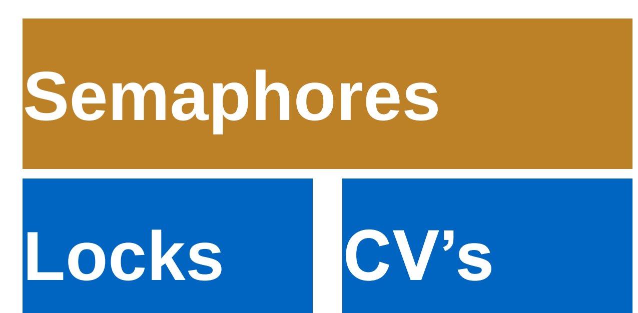
Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter



Build Semaphore from Lock and CV

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

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



Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Producer/Consumer: Semaphores #1

Simplest case:

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

Requirements

- Consumer must wait for producer to fill buffer
- Producer must wait for consumer to empty buffer (if filled)

Requires 2 semaphores

- emptyBuffer: Initialize to ??? 1 → 1 empty buffer; producer can run 1 time first
- fullBuffer: Initialize to ??? 0 → 0 full buffers; consumer can run 0 times first

Producer

```
While (1) {
        sem_wait(&emptyBuffer);
        Fill(&buffer);
        sem_signal(&fullBuffer);
}
```

Consumer

```
While (1) {
        sem_wait(&fullBuffer);
        Use(&buffer);
        sem_signal(&emptyBuffer);
}
```

Producer/Consumer: Semaphores #2

Next case: **Circular Buffer**

- Single producer thread, single consumer thread
- Shared buffer with **N** elements between producer and consumer

Requires 2 semaphores

- emptyBuffer: Initialize to ???   N → N empty buffers; producer can run N times first
- fullBuffer: Initialize to ???   0 → 0 full buffers; consumer can run 0 times first

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
- **Why will previous code (shown below) not work???**

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);
}
```

Are i and j private or shared? Need each producer to grab unique buffer

Producer/Consumer: Multiple Threads

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

```
While (1) {
        sem_wait(&emptyBuffer);
        myi = findempty(&buffer);
        Fill(&buffer[myi]);
        sem_signal(&fullBuffer);
}
```

Consumer

```
While (1) {
        sem_wait(&fullBuffer);
        myj = findfull(&buffer);
        Use(&buffer[myj]);
        sem_signal(&emptyBuffer);
}
```

Are myi and myj private or shared? Where is mutual exclusion needed???

Producer/Consumer: Multiple Threads

Consider three possible locations for mutual exclusion

Which work??? Which is best???

Producer #1

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

Consumer #1

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

Problem: Deadlock at mutex (e.g., consumer runs first; won't release mutex)

Producer/Consumer: Multiple Threads

Consider three possible locations for mutual exclusion

Which work??? Which is best???

Producer #2

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

Consumer #2

```
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

Consider three possible locations for mutual exclusion

Which work??? Which is best???

Producer #3

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

Consumer #3

```
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

Goal:

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…

Reader/Writer Locks

```
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);
}

```

Reader/Writer Locks

```
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???
```

Semaphores

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 1: Mutex
- 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 relationships and for re ader/writer locks

Summary

Locks+CVs are good primitives, but not always convenient.

Possible to build other abstractions such as semaphores.

Advice: if you always use an abstraction the same way,
build another abstraction over the first!