

# Posix Thread Synchronization

*CS 241: Systems Programming  
Discussion, Week 4*

slides prepared by Sameer Sundresh

# **MP3**

**Motivation**

**Semaphores**

**Deadlocks**

**Producer-Consumer**

**MP3**

MP3

**Motivation**

Semaphores

Deadlocks

Producer-Consumer

# Motivation

Multiple threads execute *independently*, but have access to the *same* memory and resources.

This can be **unsafe** without coordination:

```
int x = 0;
void *up(void *arg) // in thread A
{ int i; for (i = 0; i < N; i++) x++; }

void *down(void *arg) // in thread B
{ int i; for (i = 0; i < N; i++) x--; }
```

# Coordination concept

C doesn't have native atomic constructs,  
but *what if it did?*

```
int x = 0;
void *up(void *arg)    // in thread A
{
    int i;
    for (i = 0; i < N; i++)
        atomic { // how to implement atomic?
            x++;
        }
}
```

# Synchronization constructs

Any of these may be used for synchronization:

- **Mutexes** (*simple but low-level*)
  - *Mutual exclusion locks*
- **Condition variables** (*kind of complicated*)
  - *Wait until the value of a special variable changes*
- **Semaphores** (*we'll discuss these today*)
  - *Abstraction of a resource counter*
  - *Can simulate mutexes & condition variables*

MP3

Motivation

**Semaphores**

Deadlocks

Producer-Consumer

# Semaphores

*Abstraction of a resource counter.*

Operations:

**sem\_wait(sem)**

**wait** until counter  $> 0$ ,  
then **decrement** counter

**sem\_post(sem)**

**increment** counter

These operations must be **atomic**.

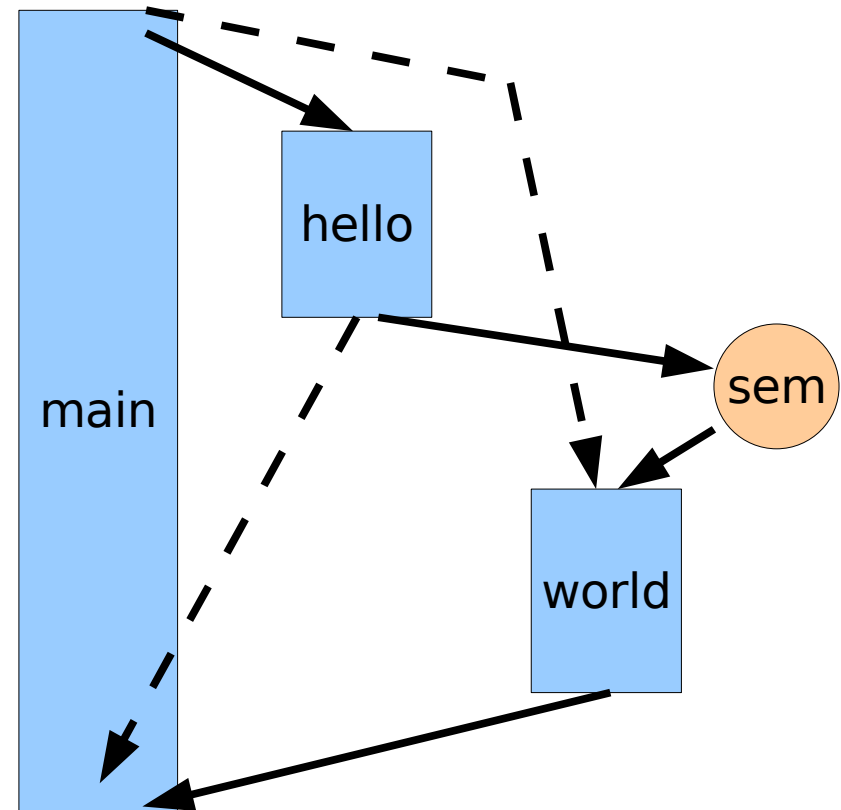
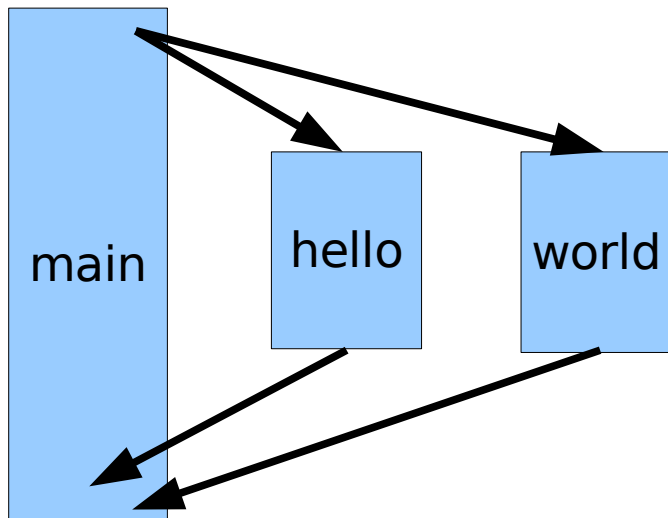
# Managing semaphores

Operations to create and destroy semaphores:

```
sem_t sem;
// ...
int main(int argc, char **argv)
{
    sem_init(&sem, 0, init_value);
    // ...use sem in here...
    sem_destroy(&sem);
}
```

# Example 1: Hello World!

Let's make Hello World! safe (see `disc4-ex1.c`):



# Example 2: up/down

Let's make up/down safe (see `disc4-ex2*.c`):

```
int x = 0; sem_t sem;
void *up(void *arg)    // in thread A
{
    int i;
    for (i = 0; i < N; i++) {
        sem_wait(&sem);
        x++;
        sem_post(&sem);
    }
}
```

MP3

Motivation

Semaphores

**Deadlocks**

Producer-Consumer

# Deadlocks (disc4-ex3.c)

All is not calm in semaphore land...

```
sem_init(&sem1, 0, 1);  
sem_init(&sem2, 0, 1);
```

```
// Thread A
```

```
sem_wait(&sem1);  
sem_wait(&sem2);
```

```
// potential deadlock!
```

```
sem_post(&sem2);  
sem_post(&sem1);
```

```
// Thread B
```

```
sem_wait(&sem2);  
sem_wait(&sem1);
```

```
sem_post(&sem1);  
sem_post(&sem2);
```

# Deadlock? (disc4-ex3.c)

```
sem_init(&sem1, 0, 1);  
sem_init(&sem2, 0, 1);
```

S1	S2	// Thread A	// Thread B
1	1	sem_wait(&sem1);	
0	1	sem_wait(&sem2);	
0	0	sem_post(&sem2);	
0	1	sem_post(&sem1);	
1	1		sem_wait(&sem2);
1	0		sem_wait(&sem1);
0	0		sem_post(&sem1);
1	0		sem_post(&sem2);
1	1		

OK...

# Deadlock! (disc4-ex3.c)

```
sem_init(&sem1, 0, 1);  
sem_init(&sem2, 0, 1);
```

S1	S2	// Thread A	// Thread B
1	1	sem_wait(&sem1);	
0	1		sem_wait(&sem2);
0	0	sem_wait(&sem2);	
0	0		sem_wait(&sem2);

**DEADLOCK**

# Deadlockn't (disc4-ex3b.c)

```
sem_init(&sem1, 0, 2); ← 2 copies of  
sem_init(&sem2, 0, 2); ← resources
```

S1	S2	// Thread A	// Thread B
2	2	sem_wait(&sem1);	
1	2	sem_wait(&sem2);	
1	1		sem_wait(&sem2);
1	0		sem_wait(&sem1);
0	0	sem_post(&sem2);	
0	1	sem_post(&sem1);	
1	1		sem_post(&sem1);
2	1		sem_post(&sem2);
2	2		

OK

MP3

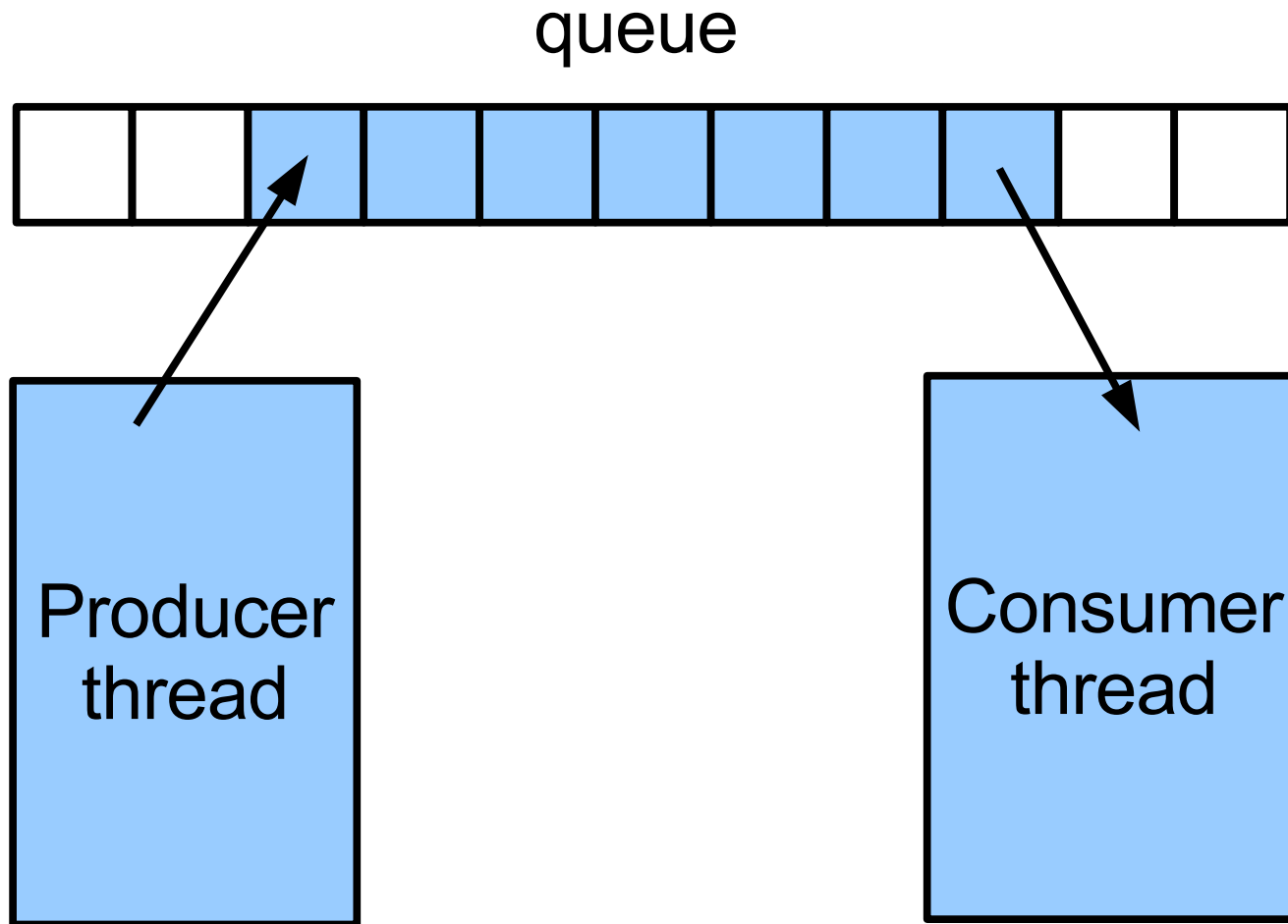
Motivation

Semaphores

Deadlocks

**Producer-Consumer**

# Producer-Consumer model



# Producer-Consumer model

Necessary atomic operations:

```
int enqueue(queue_t *q, void *datum);  
    // Return 1 on success, 0 on failure.
```

```
void *dequeue(queue_t *q);  
    // Wait if no data is available.
```

How do we implement these with semaphores?

See `disc4-ex4*.c`

# Unthreaded queue

First draft—get the queue operations working before we worry about threading

`disc4-ex4.c`

While it may *appear* to function correctly even with multiple threads under simple test cases, it's not *guaranteed* to be thread-safe.

# Queue with a lock

Now use a semaphore to mediate queue access:

*One resource, a single lock*

*Must hold the lock to enqueue or dequeue*

Try using it with multiple threads

`disc4-ex4b.c`

Try adding delays into the enqueue thread!

# Queue with lock & counter

Add a counter semaphore to keep track of how many elements are waiting in the queue:

*Use `sem_post()` to indicate an insertion*

*Use `sem_wait()` to claim a removal—and block the caller if the queue is empty.*

`disc4-ex4c.c`

Again try delays in the enqueue thread.