Synchronization and Deadlocks

(or The Dangers of Threading)

CS449 Spring 2016

Race Condition

Shared Data:

tail
4 / 5

A[]							
1	8	5	6	20 / 9	?		

Enqueue():

Thread 0

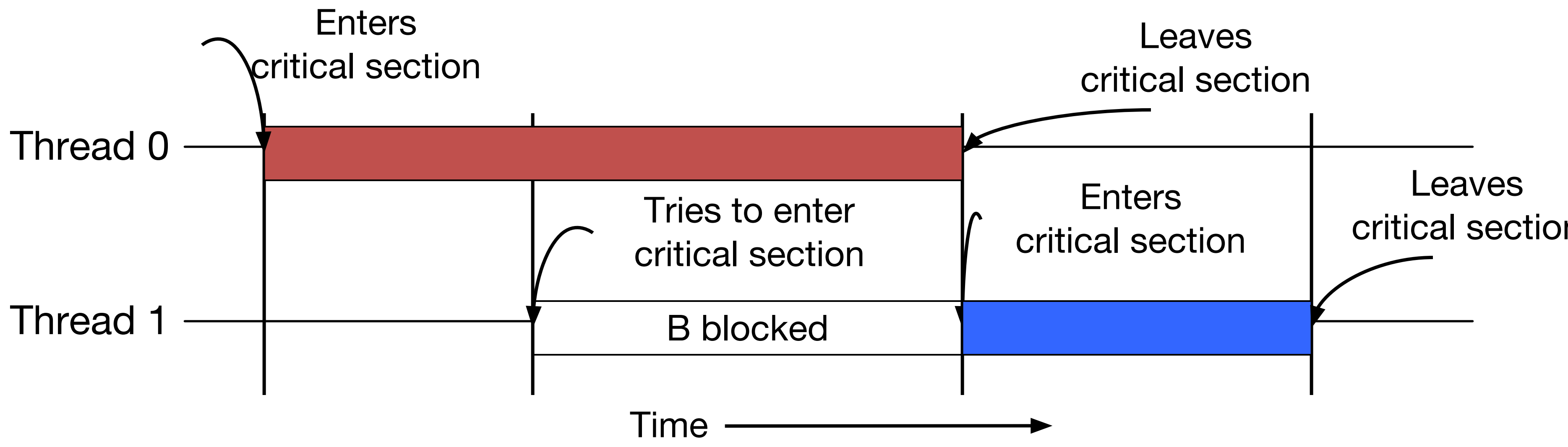
```
A[tail] = 20;
tail++;
```

Thread 1

```
A[tail] = 9;
tail++;
```

thread switch

Critical Sections



Synchronization

- Scheduling can be random and preemption can happen at any time
- Two threads can be running in parallel
- Need a way to make critical sections “atomic”
  - Atom: smallest unit that cannot be divided further using chemical means
  - Atomic: section of code appears to execute as a unit with no interleaving
- Need help from the Operating System (or the user thread scheduler)

Mutex

- MUTual EXclusion
- A mutex is a lock that only one thread can acquire
- All other threads attempting to enter the critical section will be blocked
- Thread holding lock releases lock when exiting critical section

Critical Sections

Shared Data:

tail 6

mutex 

1	8	5	6	20	9		

A[]

Enqueue():

thread switch

```
lock(&mutex);
A[tail] = 20;
tail++;
unlock(&mutex);
```

Thread 0

Blocked!

```
lock(&mutex);
A[tail] = 9;
tail++;
unlock(&mutex);
```

Thread 1

pthread_mutex_t

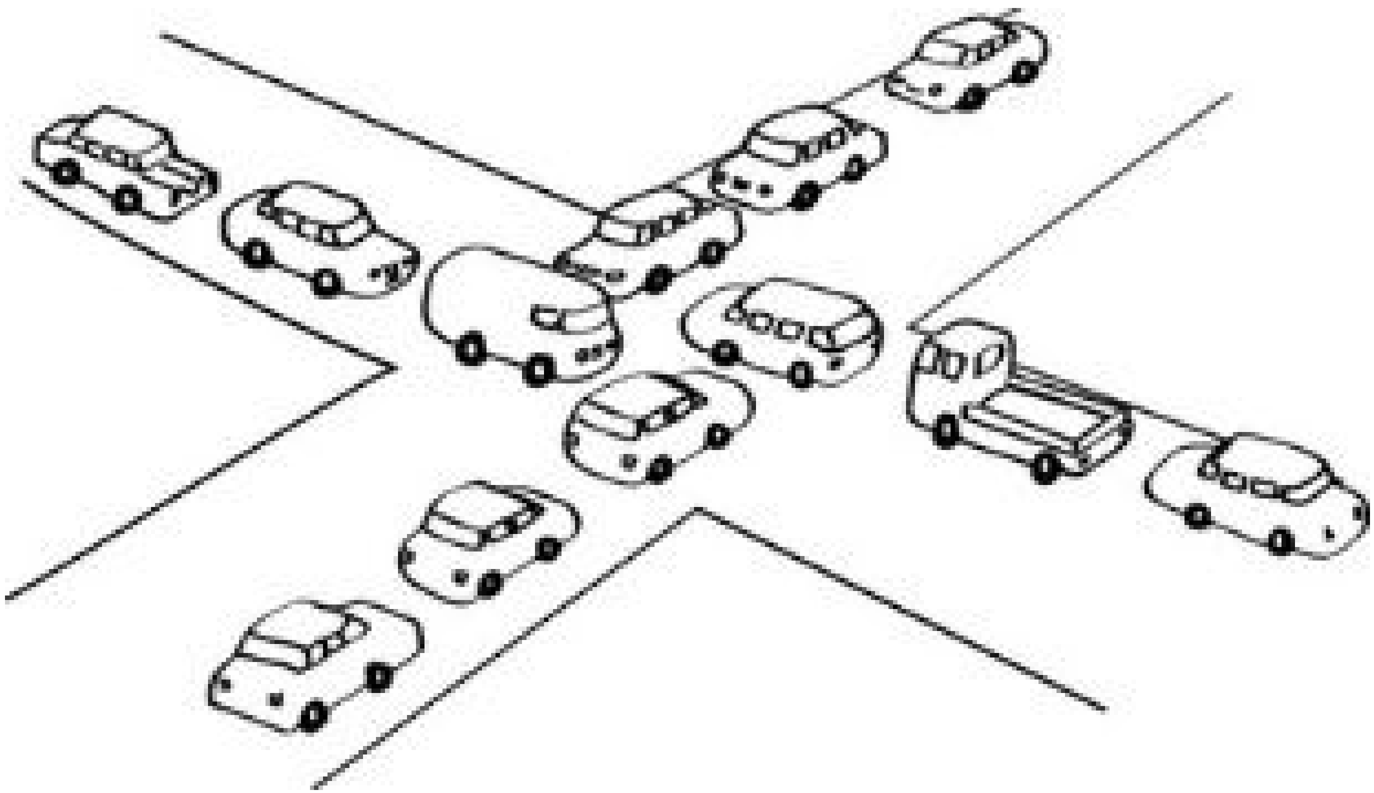
```
#include <stdio.h>
#include <pthread.h>

int tail = 0;
int A[20];

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void enqueue(int value)
{
        pthread_mutex_lock(&mutex);
        A[tail] = value;
        tail++;
        pthread_mutex_unlock(&mutex);
}
```

Sharing can lead to **deadlocks**!



Deadlocks

- “A set of processes is **deadlocked** if each process in the set is waiting for an event that only another process in the set can cause.”

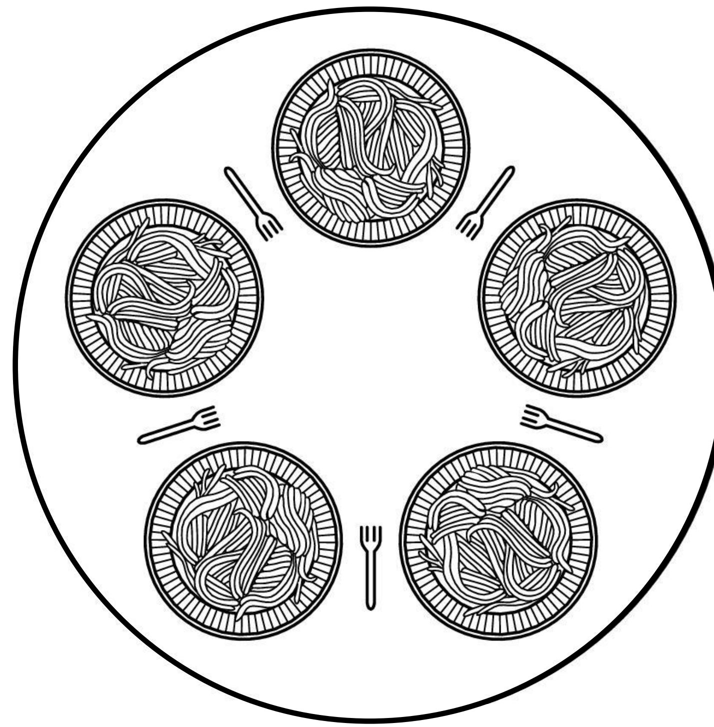
Deadlock Requirements

- Mutual Exclusion
  - Resource can only be held by one process at a time
- Hold and Wait
  - Process does not release its resources when waiting for other recourses
- No Preemption
  - Resources cannot be forcibly taken away
- Circular Wait
  - Processes are waiting for each other in a closed cycle

Deadlock Example: Dining Philosophers

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- What happens if each philosopher grabs the fork on the right?

1. Mutual exclusion
2. Hold and wait
3. No preemption of resource
4. *Circular wait*



Deadlock Example

Shared Data:

a | 0 | b | 0

mutex_a    mutex_b 

Blocked!

thread switch

```
lock(&mutex_a);
lock(&mutex_b);
a++; b++;
unlock(&mutex_b);
unlock(&mutex_a);
```

Thread 0

Blocked!

```
lock(&mutex_b);
lock(&mutex_a);
b++; a++;
unlock(&mutex_a);
unlock(&mutex_b);
```

Thread 1

thread switch

Handling Deadlocks

Detect

Prevent

Avoid

Ignore

Deadlock Solution 1: Remove Circular Wait

Shared Data:

a 0

b 0

mutex_a 

mutex_b 

```
lock(&mutex_a);
lock(&mutex_b);
a++; b++;
unlock(&mutex_b);
unlock(&mutex_a);
```

Thread 0

```
lock(&mutex_a);
lock(&mutex_b);
b++; a++;
unlock(&mutex_b);
unlock(&mutex_a);
```

Thread 1

- Rule: Acquire locks in the same order

Deadlock Solution 2: Remove Hold and Wait

Shared Data:

a | 0

b | 0

mutex_a 

mutex_b 

```
lock(&mutex_a);
a++;
unlock(&mutex_a);
lock(&mutex_b);
b++;
unlock(&mutex_b);
```

Thread 0

```
lock(&mutex_b);
b++;
unlock(&mutex_b);
lock(&mutex_a);
a++;
unlock(&mutex_a);
```

Thread 1

Producer/Consumer Problem

Shared variables

```
#define N 10;

int buffer[N];
int in = 0, out = 0, counter = 0;
```

Producer

```
while (1) {
  if (counter == N)
    sleep();

  buffer[in] = ... ;
  in = (in+1) % N;

  counter++;

  if (counter==1)
    wakeup(consumer);
}
```

Consumer

```
while (1) {
  if (counter == 0)
    sleep();

  ... = buffer[out];
  out = (out+1) % N;

  counter--;

  if (counter == N-1)
     wakeup(producer);
}
```

Condition Variables

- A condition under which a thread executes or is blocked
- pthread_cond_t
  - The type of the condition variable
- pthread_cond_wait (condition, mutex)
  - Blocks current thread until condition signaled
- pthread_cond_signal (condition)
  - Unblocks one thread waiting for condition

Producer/Consumer

```c
#define N 10
int buffer[N];
int counter = 0, in = 0, out = 0, total = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t prod_cond = PTHREAD_COND_INITIALIZER;
pthread_cond_t cons_cond = PTHREAD_COND_INITIALIZER;
```

```c
void *producer(void *junk) {
   while(1) {
      pthread_mutex_lock(&mutex);
      if( counter == N )
         pthread_cond_wait(&prod_cond,
                           &mutex);

      buffer[in] = total++;
      printf("Produced: %d\n",
                           buffer[in]);
      in = (in + 1) % N;
      counter++;

      if( counter == 1 )
         pthread_cond_signal(&cons_cond);

      pthread_mutex_unlock(&mutex);
   }
}
```

```c
void *consumer(void *junk) {
   while(1) {
      pthread_mutex_lock(&mutex);
      if( counter == 0 )
         pthread_cond_wait(&cons_cond,
                           &mutex);


      printf("Consumed: %d\n",
                           buffer[out]);
      out = (out + 1) % N;
      counter--;

      if( counter == (N-1) )
         pthread_cond_signal(&prod_cond);

      pthread_mutex_unlock(&mutex);
   }
}
```

Semaphores

- Mutex: controls access to one resource
  - Critical section protects access to a shared data structure
  - Analogy: Room with one desk where only one student can enter
- Semaphore: controls access to multiple resources
  - Analogy: Room with 5 desks where up to 5 students can enter
  - Internal counter keeps track of number of resources available
- Mutexes can be thought of as binary semaphores (semaphores than can only count up to 1 resource)

Producer/Consumer

```
#include <semaphore.h>

#define N 10
int buffer[N];
int in = 0, out = 0, total = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t semfull;  // sem_init(&semfull, 0, 0); in main()
sem_t semempty; // sem_init(&semempty, 0, N); in main()
```

```
void *producer(void *junk) {
    while(1) {
        sem_wait(&semempty);
        pthread_mutex_lock(&mutex);

        buffer[in] = total++;
        printf("Produced: %d\n",
                             buffer[in]);

        in = (in + 1) % N;

        pthread_mutex_unlock(&mutex);
        sem_post(&semfull);
    }
}
```

```
void *consumer(void *junk) {
    while(1) {
        sem_wait(&semfull);
        pthread_mutex_lock(&mutex);


        printf("Consumed: %d\n",
                             buffer[out]);

        out = (out + 1) % N;

        pthread_mutex_unlock(&mutex);
        sem_post(&semempty);
    }
}
```

Producer/Consumer

```
#include <semaphore.h>

#define N 10
int buffer[N];
int in = 0, out = 0, total = 0;

sem_t semmutex; // sem_init(&semmutex, 0, 1); in main()
sem_t semfull;  // sem_init(&semfull, 0, 0); in main()
sem_t semempty; // sem_init(&semempty, 0, N); in main()
```

```
void *producer(void *junk) {
    while(1) {
        sem_wait(&semempty);
        sem_wait(&semmutex);

        buffer[in] = total++;
        printf("Produced: %d\n",
                                    buffer
[in]);
        in = (in + 1) % N;

        sem_post(&semmutex);
        sem_post(&semfull);
    }
}
```

```
void *consumer(void *junk) {
    while(1) {
        sem_wait(&semfull);
        sem_wait(&semmutex);


        printf("Consumed: %d\n",

buffer[out]);
        out = (out + 1) % N;

        sem_post(&semmutex);
        sem_post(&semempty);
    }
}
```

Deadlock!

```c
#include <semaphore.h>

#define N 10
int buffer[N];
int in = 0, out = 0, total = 0;

sem_t semmutex; // sem_init(&semmutex, 0, 1); in main()
sem_t semfull;  // sem_init(&semfull, 0, 0); in main()
sem_t semempty; // sem_init(&semempty, 0, N); in main()
```

```c
void *producer(void *junk) {
    while(1) {
        sem_wait(&semmutex);
        sem_wait(&semempty);

        buffer[in] = total++;
        printf("Produced: %d\n",
                                    buffer
[in]);
        in = (in + 1) % N;

        sem_post(&semfull);
        sem_post(&semmutex);
    }
}
```

```c
void *consumer(void *junk) {
    while(1) {
        sem_wait(&semmutex);
        sem_wait(&semfull);


        printf("Consumed: %d\n",

buffer[out]);
        out = (out + 1) % N;

        sem_post(&semempty);
        sem_post(&semmutex);
    }
}
```

Valgrind

- Same tool we used for memory errors
- Helgrind: component of valgrind that does potential data race / deadlock detection
- Command: valgrind --tool=helgrind <program>
- Not perfect.
  - Can miss errors (sometimes)
  - Can report errors when there are none (sometimes)
- Not a replacement for sound programming