Data Races and Deadlocks
(or The Dangers of Threading)

CS449 Fall 2017
**Data Race**

- **Data Race**: a situation where two threads ‘race’ to access a memory location where at least one access is a write
  - Writes to `A[tail]` and writes to `tail` race with each other
  - Thread 1 interrupts Thread 0 when it does not want to be
    - When the queue in Thread 0 is still in an inconsistent state
    - Thread 1 will start operating on an inconsistent queue
  - Two reads are okay (reads do not change state so no inconsistency)

**Shared Data:**

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>8</th>
<th>5</th>
<th>6</th>
<th>20</th>
<th>?</th>
</tr>
</thead>
</table>

• `tail`
```

```
<table>
<thead>
<tr>
<th>var</th>
<th>6</th>
</tr>
</thead>
</table>
```

**Enqueue():**

- **Thread 0**
  ```
  A[tail] = 20;
  tail++;
  ```
- **Thread 1**
  ```
  A[tail] = 9;
  tail++;
  ```
Critical Sections

- **Critical Section**: a section of code where a thread does not want to be interrupted, while performing a critical set of actions
  - Doesn’t want another thread to modify state that is being accessed
  - Doesn’t want another thread to read state in middle of modification
  - In other words, doesn’t want a data race

- Ideally, this should happen:

![Diagram showing the sequence of events involving two threads entering and exiting critical sections](chart)
Synchronization

• Critical sections can always be interrupted
  – Two threads can be running in parallel on two CPUs
  – Even on a single CPU, context switches can happen

• Need a way to make critical sections “atomic”
  – Atom: smallest unit of matter that cannot be divided further using chemical means
  – Atomic: property of code where the code is always executed as a unit with no interruption

• Need cooperation from the Operating System (or the user-level thread scheduler)
Synchronization

• Already learned one synchronization primitive: pthread_join()

• But pthread_join() can only wait for a thread to terminate, not exit a critical section

• We need a different synchronization primitive
Mutex

• Analogy: if you don’t want to be interrupted what do you do?
  1. Enter a room, lock the door behind you before starting
  2. Exit the room, unlock the door so others can enter room

• Mutex: A resource that only one thread can own at a time, in mutual exclusion. Two operations are allowed:
  – lock(&mutex): Attempt to gain ownership of mutex
    (if mutex already owned by another thread, block current thread)
  – unlock(&mutex): Relinquishes ownership of mutex
    (if there are threads blocked on this mutex, unblock one of them)

• To protect a critical section:
  1. Before entering, lock mutex: now all other threads are blocked
  2. Before exiting, unlock mutex: now another thread can enter
Critical Sections

Shared Data:
- tail: 6
- mutex

Enqueue():
- lock(&mutex);
- A[tail] = 20;
- tail++;
- unlock(&mutex);

Blocked!
- Thread 0
- lock(&mutex);
- A[tail] = 9;
- tail++;
- unlock(&mutex);
- Thread 1

Thread 0

Thread 1
Pthread Mutex API

- **Mutex variable:** A resource that only one thread can own at a time

- `pthread_mutex_t`
  - The type of the mutex variable

- `PTHREAD_MUTEX_INITIALIZER`
  - A macro that returns the value of a mutex initialized to unlocked

- `pthread_mutex_lock (&mutex)`
  - Lock the `mutex` (if already locked, block thread)

- `pthread_mutex_unlock (&mutex)`
  - Unlock the `mutex` (if blocked threads, unblock one of them)

- Pointer to mutex passed to be able to modify it
Pthread Mutex API

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

int tail = 0;
int A[20];

// Initialize mutex to the unlocked state
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

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

- **Deadlock**: situation where two (or more) threads cannot make progress because they are waiting for each other to unlock a resource

- Caused when:
  1. There is **mutual exclusion** on resources
  2. A thread **holds** a resource and **waits** for another
  3. A **circular wait** scenario can occur
  4. **No preemption** of resources can occur
    - Preemption: forced reclamation of a resource
Deadlock Example: Dining Philosophers

- Philosophers eat/think
- Eating needs two chopsticks
- Pick one chopstick at a time
- What happens if each philosopher grabs chopstick on the right?
- Satisfies all conditions for deadlock
  1. Mutual exclusion
  2. Hold and wait
  3. Circular wait
  4. No preemption
- Philosophers will starve
Mutex Deadlock Example

Shared Data:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>mutex_a</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>mutex_b</td>
<td></td>
</tr>
</tbody>
</table>

Thread 0

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

Thread 1

```c
lock(&mutex_b);
lock(&mutex_a);
b++; a++; unlock(&mutex_a);
unlock(&mutex_b);
```
Solving Mutex Deadlocks

- Deadlocks are caused when:
  1. Mutual exclusion on resource
  2. Hold resources and waits for another
  3. Circular wait on resources
  4. No preemption of resource

- 1. and 4. cannot be avoided:
  - 1. Mutexes are by definition mutually exclusive
  - 4. You cannot force an unlock of a mutex (or atomicity is lost)

- Try to avoid 2. and 3.:
  - 2. Do not hold a mutex while waiting for another mutex
  - 3. Remove possibility of circular wait
Deadlock Solution 1: Remove Hold and Wait

**Original:**
lock(&mutex_a);
lock(&mutex_b);
a++; b++;
unlock(&mutex_b);
unlock(&mutex_a);

Thread 0
lock(&mutex_a);
a++;
unlock(&mutex_a);
lock(&mutex_b);
b++; a++;
unlock(&mutex_b);

Thread 1
lock(&mutex_b);
b++;
unlock(&mutex_b);
lock(&mutex_a);

**Fixed:**
lock(&mutex_a);
a++;
unlock(&mutex_a);
lock(&mutex_b);
b++; a++;
unlock(&mutex_b);

- Assuming a++, b++ can be separated without hurting atomicity
Deadlock Solution 2: Remove Circular Wait

**Original:**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(&amp;mutex_a);</td>
<td>lock(&amp;mutex_b);</td>
</tr>
<tr>
<td>lock(&amp;mutex_b);</td>
<td>lock(&amp;mutex_a);</td>
</tr>
<tr>
<td>a++; b++;</td>
<td>b++; a++;</td>
</tr>
<tr>
<td>unlock(&amp;mutex_b);</td>
<td>unlock(&amp;mutex_a);</td>
</tr>
<tr>
<td>unlock(&amp;mutex_a);</td>
<td>unlock(&amp;mutex_b);</td>
</tr>
</tbody>
</table>

**Fixed:**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(&amp;mutex_a);</td>
<td>lock(&amp;mutex_a);</td>
</tr>
<tr>
<td>lock(&amp;mutex_b);</td>
<td>lock(&amp;mutex_b);</td>
</tr>
<tr>
<td>a++; b++;</td>
<td>b++; a++;</td>
</tr>
<tr>
<td>unlock(&amp;mutex_b);</td>
<td>unlock(&amp;mutex_a);</td>
</tr>
<tr>
<td>unlock(&amp;mutex_a);</td>
<td>unlock(&amp;mutex_b);</td>
</tr>
</tbody>
</table>

- Always lock mutexes in the same order: number all mutexes
  - No thread will hold higher number mutex while waiting for lower number
  - All dependencies go from lower ➔ higher: no cycle can be formed
Is That Enough?

• Synchronization APIs seen so far
  – pthread_join: Waits for another thread to terminate and (optionally) produce a value
  – pthread_mutex_lock / _unlock: Waits for another thread to exit a critical section
  – Essentially waits for a certain condition to happen

• We need a synchronization primitive for all conditions in general
Pthread Condition API

- **Condition variable**: A condition under which a thread continues or is blocked

- `pthread_cond_t`
  - The type of the condition variable

- `pthread_cond_wait (&condition, &mutex)`
  - Blocks current thread and waits for `condition` to be signaled
  - Also unlocks `mutex` atomically before the blocking
  - Relocks `mutex` after waking up

- `pthread_cond_signal (&condition)`
  - Unblocks one thread waiting for `condition` (if none ignore)

- `pthread_cond_broadcast (&condition)`
  - Unblocks all threads waiting for `condition` (if none ignore)
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);
}
Producer/Consumer with only Mutexes

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *producer(void *junk) {
  while(1) {
    pthread_mutex_lock(&mutex);
    if( counter == N ) {
      pthread_mutex_unlock(&mutex);
      sleep(10); // How much to sleep?
      continue;
    }
    buffer[in] = ...;
    in = (in + 1) % N;
    counter++;
    // No wakeup
    pthread_mutex_unlock(&mutex);
  }
}

void *consumer(void *junk) {
  while(1) {
    pthread_mutex_lock(&mutex);
    if( counter == 0 )
      pthread_mutex_unlock(&mutex);
    sleep(10); // How much to sleep?
    continue;
  }
  ... = buffer[out];
  out = (out + 1) % N;
  counter--;
  // No wakeup
  pthread_mutex_unlock(&mutex);
}
```

- Correct code with no data races
- But no wakeup, so must guess how much to sleep: less efficient
Producer/Consumer with Condition Variables

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

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

void *producer(void *junk) {
    while(1) {
        pthread_mutex_lock(&mutex);
        if( counter == N )
            pthread_cond_wait(&prod_cond, &mutex);
        buffer[in] = ...;
        in = (in + 1) % N;
        counter++;
        if( counter == 1 )
            pthread_cond_signal(&cons_cond);
        pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *junk) {
    while(1) {
        pthread_mutex_lock(&mutex);
        if( counter == 0 )
            pthread_cond_wait(&cons_cond, &mutex);
        ... = buffer[out];
        out = (out + 1) % N;
        counter--;
        if( counter == (N-1) )
            pthread_cond_signal(&prod_cond);
        pthread_mutex_unlock(&mutex);
    }
}
```
Mutex in pthread_cond_wait()

- Note the mutex passed to pthread_cond_wait
  ```c
  pthread_mutex_lock(&mutex);
  if( counter == N )
      pthread_cond_wait(&prod_cond, &mutex);
  - Unlocks mutex and waits for condition in a single atomic operation
  ```
- Could we not have done something like this instead?
  ```c
  pthread_mutex_lock(&mutex);
  if( counter == N ) {
      pthread_mutex_unlock(&mutex);
      pthread_cond_wait(&prod_cond);
  }
  ```
- No. What if consumer thread intervenes between the unlock and wait, updates counter, and signals producer?
  - Signal would be lost, and the producer would be waiting forever
- Mutex makes check of condition and wait for condition atomic
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)
Pthread Semaphore API

• Semaphore variable: N resources that can be owned at any one time

• `sem_t`
  – The type of the semaphore variable

• `sem_init (&semaphore, pshared, value)`
  – Initializes `semaphore` to have value number of resources
  – pshared: If 0, used between threads. If 1, between processes

• `sem_wait (&semaphore)`
  – Decrements resources in `semaphore` (block thread if 0)

• `sem_post (&semaphore)`
  – Increments resources in `semaphore`
  – If previously 0 available resources, unblocks a waiting thread
#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 with Only Semaphores

```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()

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()

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);
    }
}

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);
    }
}
```
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()

void *producer(void *junk) {
    while(1) {
        1 sem_wait(&semmutex);
        2 sem_wait(&semempty); // BLOCKED!
        buffer[in] = total++;
        printf("Produced: %d\n", buffer[in]);
        in = (in + 1) % N;
        sem_post(&semfull);
        sem_post(&semmutex);
    }
}

void *consumer(void *junk) {
    while(1) {
        3 sem_wait(&semmutex); // BLOCKED!
        sem_wait(&semfull);
        printf("Consumed: %d\n", buffer[out]);
        out = (out + 1) % N;
        sem_post(&semempty);
        sem_post(&semmutex);
    }
}
```

- Producer holds `semmutex` while waiting for `semempty`

---

### Notes

- In the producer function, the producer holds `semmutex` while waiting for `semempty` for the first time. This prevents the consumer from dequeuing items.
- Similarly, in the consumer function, the consumer holds `semmutex` while waiting for `semfull` for the first time. This prevents the producer from enqueuing items.
- This creates a deadlock because the producer and consumer are waiting for each other's resources to be released.
Helgrind

• 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
Pitfall – Data Race on Stack

```c
void *thread_func(void *p) {
    printf("thread %d\n", *(int*)p);
    ...
}
int main() {
    pthread_t thread;
    int i;
    for(i = 0; i < 100; i++) {
        pthread_create(&thread, NULL, thread_func, (void *)&i);
    }
    ...
}

• i stack variable shared by all threads!
  – Main thread will overwrite i at every iteration
  – Will race with the read of i by all children threads
• Data races are not exclusive to heap and globals
```
void *thread_func(void *p) {
    printf("thread %d\n", *(int*)p);
    ...
    free(p); // don’t forget to free!
}

int main() {
    pthread_t thread;
    int i, *arg;
    for(i = 0; i < 100; i++) {
        arg = malloc(sizeof(int));
        *arg = i;
        pthread_create(&thread, NULL, thread_func, (void *)arg);
    }
    ...
}

• Now each thread has its own copy of i on the heap
Pitfall – Too Small Critical Section

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

- Technically removes data race (valgrind will not detect it)
  - All shared memory access are protected by a mutex
- Still doesn’t prevent illegal interleavings between threads
Pitfall – Too Small Critical Section

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

• Now only consistent state is exposed to other threads
```
Pitfall – Too Big Critical Section

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void* thread_func(void*)
{
    int val;
    pthread_mutex_lock(&mutex);
    val = compute(); // local computation of val
    enqueue(val);   // add val to a shared queue
    pthread_mutex_unlock(&mutex);
}
```

- Contains no data race
- But too big critical section allows no parallelism
Pitfall – Too Big Critical Section

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void* thread_func(void*)
{
    int val;
    val = compute(); // local computation of val
    pthread_mutex_lock(&mutex);
    enqueue(val);    // add val to a shared queue
    pthread_mutex_unlock(&mutex);
}

• Now `compute()` in each thread can run in parallel
```
void* thread_func1(void*) {
    pthread_mutex_lock(&mutex);
    for(i = 0; i < 1000; i++) sum += A[i];
    pthread_mutex_unlock(&mutex);
}

void* thread_func2(void*) {
    pthread_mutex_lock(&mutex);
    for(i = 0; i < 1000; i++) product *= A[i];
    pthread_mutex_unlock(&mutex);
}

• No data race so locking not needed
  – $A[i]$ is only read in both threads
  – Neither thread disturbs each other
Pitfall – Jerry-Rigged Synchronization

```c
int done = 0, value = 0;

void* producer_thread(void*) {
    value = ...;  // produce some value
    done = 1;  // signal that computation is done
}

void* consumer_thread(void*) {
    while(done != 1);  // wait until done
    ... = value;  // consume some value
}
```

- Looks correct as is but...
- Compiler optimization or high-performance CPU may flip the writes to ‘done’ and ‘value’
Pitfall – Jerry-Rigged Synchronization

```c
int done = 0, value = 0;
void* producer_thread(void*) {
    done = 1;  // signal that computation is done
    value = ...;  // produce some value
}
void* consumer_thread(void*) {
    while(done != 1);  // wait until done
    ... = value;  // consume some value
}
```

- Often mistakenly done by programmers to avoid the ‘overhead’ of POSIX calls
- Always use POSIX synchronization unless you know what you are doing (not many people in the world do)
Thank you!

Good luck on your finals