Synchronization
Race Example (review)

- What's the expected output on line 11?
  - 2
- Possible output...
  - 1

```c
int numbers[2] = { 1, 1 }; // line 1
int sum = 0;

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, run, numbers[1]); // line 6
    for (int i = 0; i < 1; i++) { // line 7
        sum += numbers[i];
    }
    pthread_join(tid, NULL);
    printf("sum is %d\n", sum); // line 11
}

void* run(void* arg) { // line 14
    int* numbers = (int*) arg;
    for (int i = 0; i < 1; i++) { // line 16
        sum += numbers[i];
    }
    return NULL;
}
```
Race Example (review)

- Why can the outcome be 1? This line is the culprit.

```csharp
    sum += numbers[i];
```

- What does this look like in assembly?

```
1   movq ...(,%rsi,4), %rcx
2   movq ...(%rsi), %rdx
3   addq %rcx, %rdx
4   movq %rdx, ...(%rsi)
```

- Two threads T, T' have combinatorial number of interleavings
  - OK: T1, T2, T3, T4, T'1, T'2, T'3, T'4
  - BAD: T1, T'1, T2, T'2, T3, T'3, T4, T'4
  - Global sum is written as 1 by both threads at T4 & T'4
The Source of the Problem?

- What was the source of the race condition. What was being accessed by multiple threads, leading to incorrect results?
  - \texttt{sum}, a global variable shared between threads

- Ok, the solution is not to share variables across threads, right?
  - ‘Global variables are bad’™ anyway.

- Not so fast…
  - Sharing variables is actually useful thing when programming threads
  - Global variables are not the only variable type that can be shared.
Question: Which variables in a threaded C program are shared?
- The answer is not as simple as “global variables are shared” and “stack variables are private”

Definition of a shared variable
- A variable $x$ is shared if multiple threads reference some instance of $x$.

Requires answers to the following questions:
- What is the *memory model* for threads?
- Where are instances of variables stored in memory?
- How many threads might reference each of these instances?
Threads Memory Model

- **Conceptual model:**
  - Multiple threads run within the context of a single process
  - Each thread has its own separate *thread context*
    - Thread ID, stack, stack pointer, PC, condition codes, and GP registers
  - All threads share the remaining *process context*
    - Code, data, heap & shared library segments of the process address space

- **Operationally, this model is not strictly enforced:**
  - Register values are truly separate and protected, but…
  - *Any thread can read and write the stack of any other thread*

- **The mismatch between the conceptual and operational model is a source of confusion and errors**
Example of Sharing

Peer threads reference main thread's stack indirectly through global ptr variable

```c
char **ptr; /* global var */
int main()
{
pthread_t tid;
char* msgs[2] = {
    "Hello from foo",
    "Hello from bar"
};
ptr = msgs; // hmmm.

long i;
for (i = 0; i < 2; i++)
    pthread_create(&tid,
        NULL,
        thread,
        (void*) i);

pthread_exit(NULL);
}
```
Mapping Variable Instances to Memory

- **Global variables**
  - *Definition*: Variable declared outside of a function
  - Virtual memory contains exactly one instance of any global variable

- **Local variables**
  - *Definition*: Variable inside function without static attribute
  - Each thread stack contains one instance of each local variable

- **Local static variables**
  - *Definition*: Variable inside function with static attribute
  - Virtual memory contains exactly one instance of any local static variable.
Mapping Variable Instances to Memory

**Global var:** 1 instance (ptr . bss section)

**Local vars:** 1 instance (main.i, main.msgs)

**Local var:** 2 instances (p0.i [peer thread 0’s stack], p1.i [peer thread 1’s stack])

```c
int main()
{
    pthread_t tid;
    char* msgs[2] = {
        "Hello from foo",
        "Hello from bar",
    };
    ptr = msgs; // hmmm.
    long i;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
                       NULL,
                       thread,
                       (void*) i);
    pthread_exit(NULL);
}
```

**Local static var:** 1 instance (cnt . data section)
Shared Variable Analysis

- **Which variables are shared?**

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>main.i</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>main.msgs</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>p0.i</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>p1.i</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

- **Answer:** A variable x is shared if multiple threads reference at least one instance of x. Thus:
  - *ptr, cnt, and msgs* are shared
  - *.*i are not shared
Synchronizing Threads

- Shared variables are sometimes useful but they introduce the possibility of synchronization errors.
  - Like the one we saw last time

- How do we prevent such things?
  - We need to make sure that only one thread is mutating shared variables at a time.
  - This is known as mutual exclusion

- Moreover, we must protect critical sections.
Critical Sections

- A critical section is a part of a multi-threaded program that must not be concurrently executed by more than one of the program's threads.

- Critical sections access shared variables that are not safe for concurrent accesses.

- Critical sections must be protected
  - Must make sure accesses within a CS are not interleaved.
  - Or equivalently, CS must have atomicity
  - Atomicity is achieved via mutual exclusion.
Mutual Exclusion

- A *mutex*...
  - is a synchronization variable that is used to protect the access to shared variables.
  - surrounds critical sections so that one thread is allowed inside at a time.

- In practice, you (mentally) associates a mutex with a set of shared variables
Pthread Lock Functions

- There are three basic operations defined on a mutex.
  - `pthread_mutex_init(pthread_mutex_t *mutex, ..)`
    - Initializes the specified mutex to its default values
    - The second argument will always be NULL for us
  - `pthread_mutex_lock(pthread_mutex_t *mutex)`
    - Acquires a lock on the specified mutex variable.
    - If the mutex is already held by another thread, this call will block the calling thread until the mutex is unlocked.
  - `pthread_mutex_unlock(pthread_mutex_t *mutex)`
    - Unlocks a mutex variable.
    - An error is returned if mutex is already unlocked.
Example Critical Section

- If no synchronization, what happens when there are two concurrent calls with the same argument values?
  - T1: read account x = 100
  - T2: read account x = 100
  - T1: write account x = 90
  - T2: write account x = 90
  - T1: read account y = 100
  - T1: increment account y = 110
  - T2: read account y = 110
  - T2: increment account y = 120

```c
typedef struct {
    int id;
    int balance;
} account;

account* accounts[100];

void transfer(int x, int y, int amt) {
    accounts[x]->balance -= amt;
    accounts[y]->balance += amt;
}
```
The Easy Solution

- Put a mutex around the critical section in the transfer function
- The lock is associated with the array accounts.
  - In the programmer’s head, at least.
- Are there any drawbacks to this approach?

```c
typedef struct {
    int id;
    int balance;
    pthread_mutex_t m;
} account;

account* accounts[100];
 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void transfer(int x, int y, int amt) {
    pthread_mutex_lock(&m);
    accounts[x]->balance -= amount;
    accounts[y]->balance += amount;
    pthread_mutex_unlock(&m);
}
```
Another Mutex Example

- See lecture26/mutex.c
Problem with the Easy Solution

- There is a problem here...
  - "coarse-grained locking"
  - no concurrency
  - only one transfer happening at a time.

```c
typedef struct {
    int id;
    int balance;
    pthread_mutex_t m;
} account;

account* accounts[100];
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void transfer(int x, int y, int amt) {
    pthread_mutex_lock(&m);
    accounts[x]->balance -= amount;
    accounts[y]->balance += amount;
    pthread_mutex_unlock(&m);
}```
typedef struct {
    int id;
    int balance;
    pthread_mutex_t m;
} account;

account* accounts[100];

void transfer(int x, int y, int amt)
{
    pthread_mutex_lock(&accounts[x]->m);
    pthread_mutex_lock(&accounts[y]->m);
    accounts[x]->balance -= amount;
    accounts[y]->balance += amount;
    pthread_mutex_unlock(&accounts[x]->m);
    pthread_mutex_unlock(&accounts[y]->m);
}
Looks good! Right?

Then why did my entire banking system just stop functioning?

Hmmm.. looking at the system logs I see this...

- T1: transfer(X, Y, 10)
- T2: transfer(Y, X, 20)
Deadlock

- The following series of instructions happened…
  - T1: acquired X's lock
  - T2: acquired Y's lock
  - T1: blocked waiting for Y's lock to be released
  - T2: blocked waiting for X's lock to be released

- Neither can make progress! This is known as *deadlock*

- *A deadlock is any situation in which two or more competing actions are each waiting for the other to finish, and thus none ever do.*
**Deadlock con’t**

- Both processes need resources to continue execution.
- P1 requires additional resource R1 and is in possession of resource R2
- P2 requires additional resource R2 and is in possession of R1; neither process can continue.
Solution

- Acquire locks in the order based on account number

```c
void transfer(int x, int y, int amt)
{
    if (x < y) {
        pthread_mutex_lock(&accounts[x]->m);
        pthread_mutex_lock(&accounts[y]->m);
    } else {
        pthread_mutex_lock(&accounts[y]->m);
        pthread_mutex_lock(&accounts[x]->m);
    }
    accounts[x]->val -= amount;
    accounts[y]->val += amount;
    pthread_mutex_unlock(&accounts[x]->m);
    pthread_mutex_unlock(&accounts[y]->m);
}
```

- This approach works in general.
Condition Variables

- Locking is a simple kind of resource scheduling -- one thread at a time may enter a critical section.

- What about more complicated scheduling policy?
  - Supposed we need a mechanism to block thread(s) until some condition is true?

- Condition variables are synchronization variables that are used for signaling that some condition is met and that any waiting threads can proceed.
Waiting on a Condition

For example, suppose we want one function on one thread to produce a value and another function on another thread to consume that value?

```c
typedef struct {
    int* val;
} channel;

static channel c;

void send(int* v) {
    if (c->val == NULL) {
        c->val = v;
    } else {
        // wait until null
    }
}
```

```c
int* receive() {
    if (c->val != NULL) {
        int *v = c->val;
        c->val = NULL;
        return v;
    } else {
        // wait until non-null
    }
}
```
Pthread Condition Variable Functions

- Pthreads defines three basic operations on condition variables.
  - `int pthread_cond_init(cond, ...)`
    - Takes two arguments, the first of which is the condition variable itself. The second we don’t care about.
  - `int pthread_cond_wait(cond, mutex)`
    - The calling thread will wait until the condition represented by the `cond` variable is met.
  - `int pthread_cond_signal(cond)`
    - Sends a signal that wakes up exactly one thread that is waiting due to a call to `pthread_cond_wait`.
typedef struct {
    int* val;
} channel;

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

static channel c;

void send(int* v) {
    pthread_mutex_lock(&m);
    while (c.val != NULL) {
        pthread_cond_wait(&cv, &m);
    }
    c.val = v;
    pthread_mutex_unlock(&m);
}

int* receive() {
    pthread_mutex_lock(&m);
    if (c.val) {
        int* v = c->val;
        c.val = NULL;
        pthread_cond_signal(&cv);
        pthread_mutex_unlock(&m);
        return v;
    } else {
        pthread_mutex_unlock(&m);
        return NULL;
    }
}
Conditional Variable Usage

- **The general pattern is...**
  - **T1:**
    ```c
    lock(&m);
    while (condition != true)
      cond_wait(&cv, &m)
    ... do stuff...
    unlock(&m)
    ```
  - **T2:**
    ```c
    lock(&m)
    condition = true
    cond_signal(&cv)
    unlock(&m)
    ```
Another Conditional Variable Example

- See lecture26/cond_var.c
Summary

- Programmers need a clear model of how variables are shared by threads.

- Variables shared by multiple threads must be protected to ensure mutually exclusive access.

- Mutexes are a fundamental mechanism for enforcing mutual exclusion.

- Conditional variables can be used to signal between threads that some condition has been met.