Synchronization

+ Concurrent Programming is Hard!

- **The ease with which threads share data and resources also makes them vulnerable to subtle and baffling errors.**
- **Classical problem classes of concurrent programs:**
  - *Races*: outcome depends on arbitrary scheduling decisions elsewhere in the system
  - *Deadlock*: improper resource allocation prevents forward progress
  - *Livelock / Starvation / Fairness*: external events and/or system scheduling decisions can prevent sub-task progress

Race Example

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

```
1  int numbers[2] = { 1, 1 };
2  int sum = 0;
3
4  int main() {
5    pthread_t tid;
6    pthread_create(&tid,NULL,run,numbers[1]);
7    for (int i = 0; i < 1; i++) {
8        sum += numbers[i];
9    }
10   pthread_join(tid, NULL);
11   printf("sum is %d\n", sum);
12 }
13
14 void* run(void* arg) {
15   int* numbers = (int*) arg;
16   for (int i = 0; i < 1; i++) {
17       sum += numbers[i];
18   }
19   return NULL;
20 }
```

+ Race Example

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


```c
1  int numbers[2] = { 1, 1 };
2  int sum = 0;
3
4  int main() {
5    pthread_t tid;
6    pthread_create(&tid,NULL,run,numbers[1]);
7    for (int i = 0; i < 1; i++) {
8        sum += numbers[i];
9    }
10   pthread_join(tid, NULL);
11   printf("sum is %d\n", sum);
12 }
13
14 void* run(void* arg) {
15   int* numbers = (int*) arg;
16   for (int i = 0; i < 1; i++) {
17       sum += numbers[i];
18   }
19   return NULL;
20 }
```

Race Example *con't*

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

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

- **What does this look like in assembly?**

```
1   movq numbers(,%rdi,4), %rcx    // %rcx = numbers[i]
2   movq (%rsi), %rdx              // %rdx = sum
3   addq %rcx, %rdx                // %rdx = %rcx + %rdx
4   movq %rdx, (%rsi)              // sum = %rdx
```

- **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`
    - `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?**
  - `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.

+ Shared Variables in Threaded C Programs

- **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

```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);
}
```

```c
void* thread(void *vargp)
{
    long i = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           i, ptr[i], ++cnt);
    return NULL;
}
```

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

+ 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
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);
}
```

```c
void* thread(void *vargp)
{
    long i = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           i, ptr[i], ++cnt);
    return NULL;
}
```

Local static var: 1 instance (`cnt` .data section)

+ Shared Variable Analysis

- **Which variables are shared?**

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`main.i`	yes	no	no
`main.msgs`	yes	yes	yes
`p0.i`	no	yes	no
`p1.i`	no	no	yes

- **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 (mutex)*
- **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 synchronization variable that is used to protect the access to shared variables.
  - surrounds critical sections so that one threads 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?**

```
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);
}
```

+ Fine-grained Locking

```c
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);
}
```

+ Fine-grained Locking *con't*

- **Looks good! Right?**
- **Then why did my entire banking system just stop functioning?**
- **Hmmm.. looking at the system logs I see this…**
  - `T1:transfer(1,2, 10)`
  - `T2:transfer(2,1, 20)`

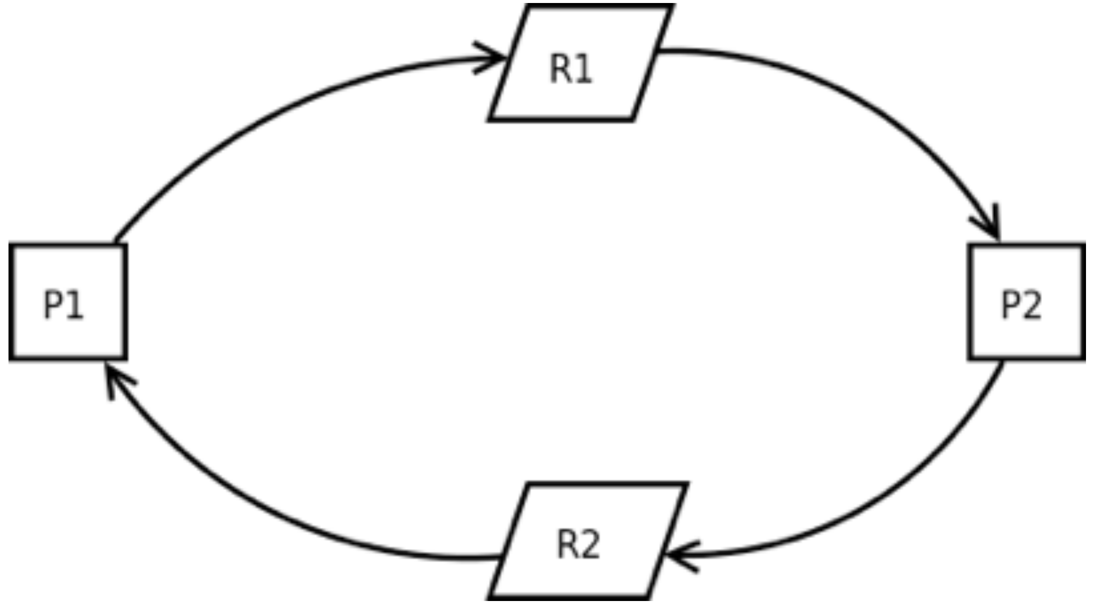
```
typedef struct {
  int id;
  int balance;
  pthread_mutex_t m;
} account;

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);
}
```

+ Deadlock

- **The following series of instructions happened…**
  - T1: acquired 2's lock
  - T2: acquired 1'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**

```
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;
  acounts[y]->val += amount;
  pthread_mutex_unlock(&accounts[x]->m);
  pthread_mutex_unlock(&accounts[y]->m);
}
```

- **This approach works in general.**