CS 261
Fall 2016

Mike Lam, Professor

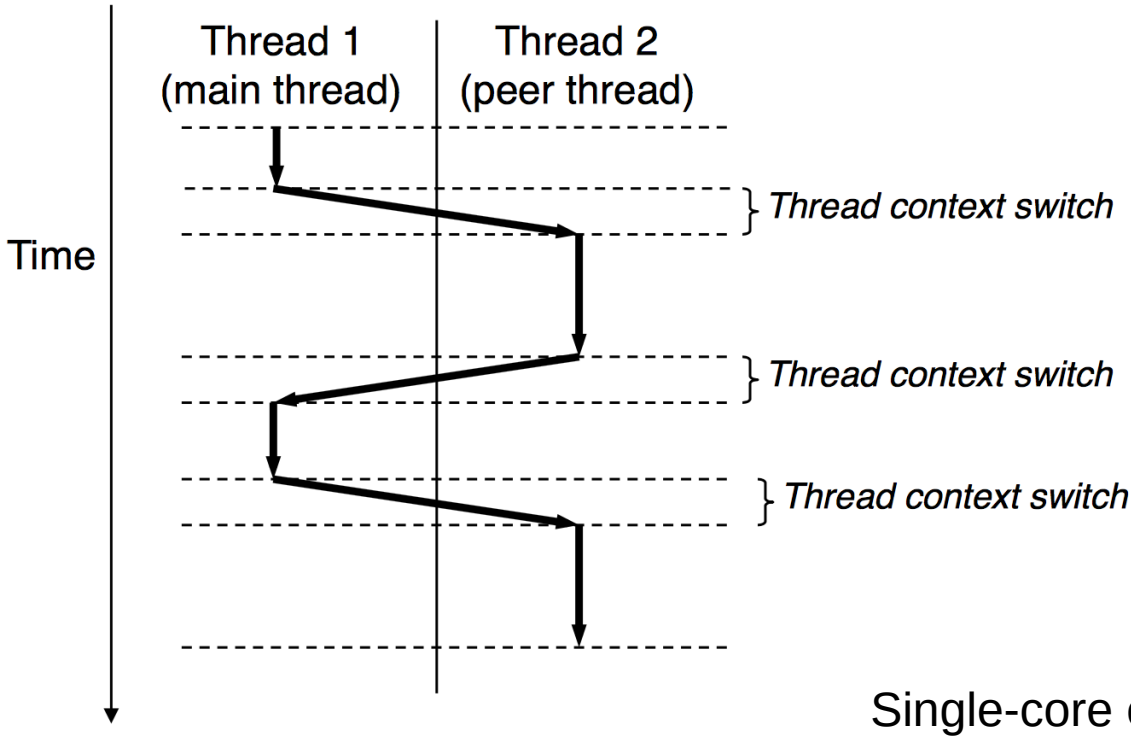
Threads

Processes vs. threads

- Process: currently-executing program
  - Code and state (PC, stack, data, heap)
  - Private address space
- Thread: unit of execution or logical flow
  - Exists within the context of a single process
  - Shares code/data/heap/files w/ other threads
  - Keeps private PC, stack, and registers
    - Stacks are technically shared, but harder to access

Processes vs. threads

- One main thread for each process
  - Can create multiple peer threads



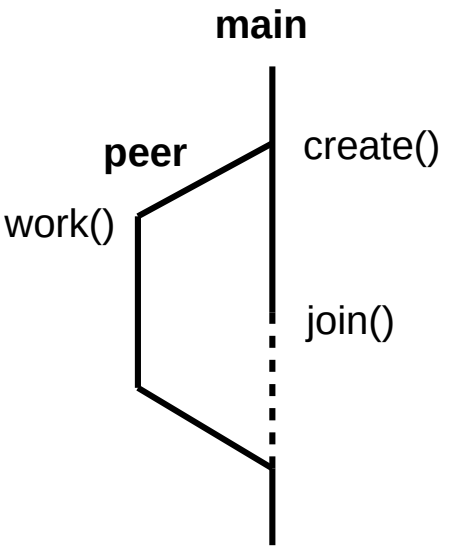
Single-core example

POSIX threads

- Pthreads – POSIX standard interface for threads in C
  - `pthread_create`: spawn a new child thread
    - `pthread_t` struct for storing thread info
    - attributes (or NULL)
    - thread work routine (function pointer)
    - thread routine parameter (void*)
  - `pthread_self`: get current thread ID
  - `pthread_exit`: terminate current thread
    - can also terminate implicitly by returning from the thread routine
  - `pthread_join`: wait for another thread to terminate

Thread creation example

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

void* work (void* arg)
{
    printf("Hello from work routine!\n");
    return NULL;
}

int main ()
{
    printf("Spawning single child ...\n");

    pthread_t child;
    pthread_create(&child, NULL, work, NULL);
    pthread_join(child, NULL);

    printf("Done!\n");

    return 0;
}
```



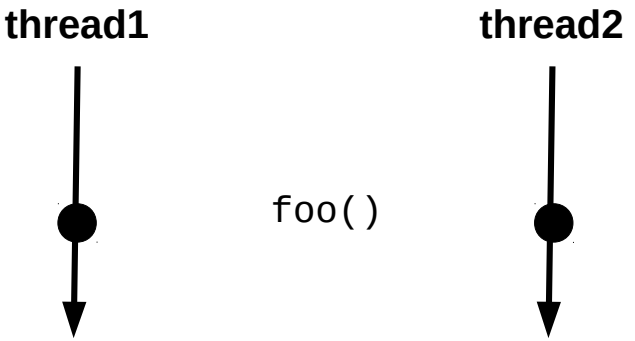
Shared memory

- Variables in threaded programs
  - Global variables (shared, single static copy)
  - Local variables (multiple copies, one on each stack)
    - Technically still shared if in memory, but harder to access
    - Not shared if cached in register
    - Safer to assume they're private
  - Local static variables (shared, single static copy)

Issues with shared memory

- Nondeterminism
- Data races and deadlock

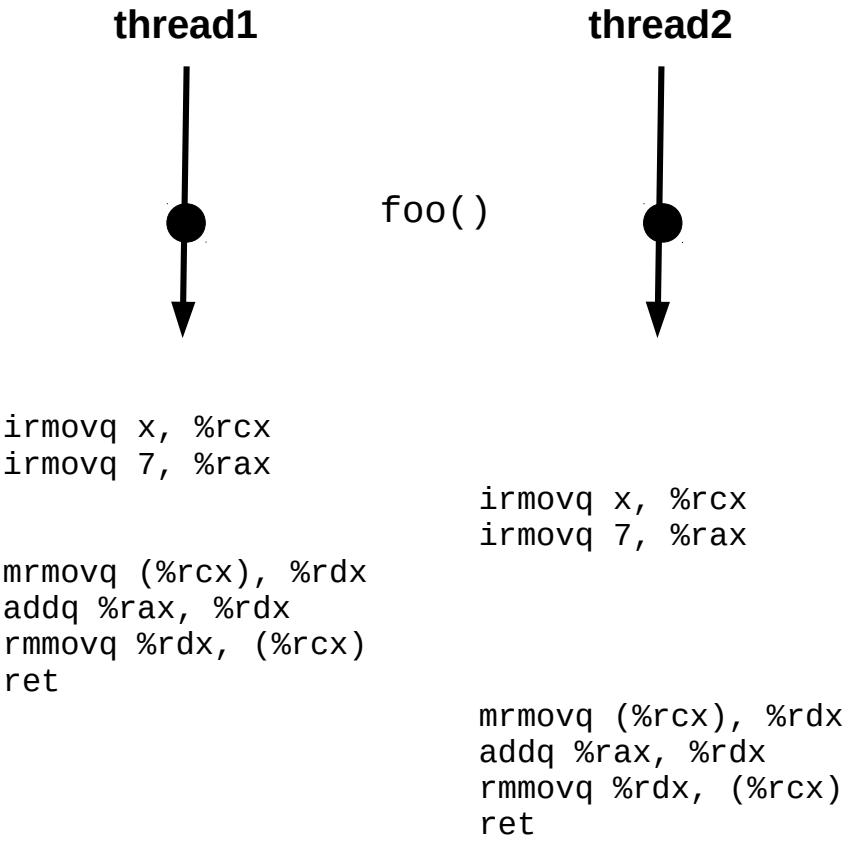
```
foo:
    irmovq x, %rcx
    irmovq 7, %rax
    mrmovq (%rcx), %rdx
    addq %rax, %rdx
    rmmovq %rdx, (%rcx)
    ret

x:
    .quad 0
```



Issues with shared memory

- Nondeterminism
- Data races and deadlock

```
foo:
    irmovq x, %rcx
    irmovq 7, %rax
    mrmovq (%rcx), %rdx
    addq %rax, %rdx
    rmmovq %rdx, (%rcx)
    ret

x:
    .quad 0
```

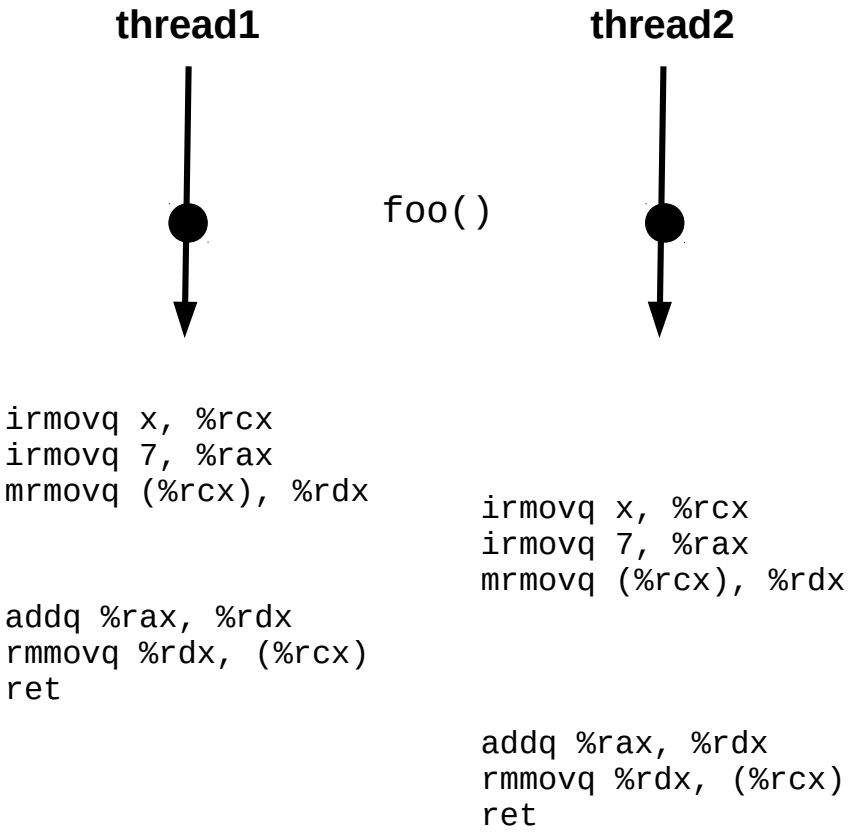


This interleaving is ok.

Issues with shared memory

- Nondeterminism
- Data races and deadlock

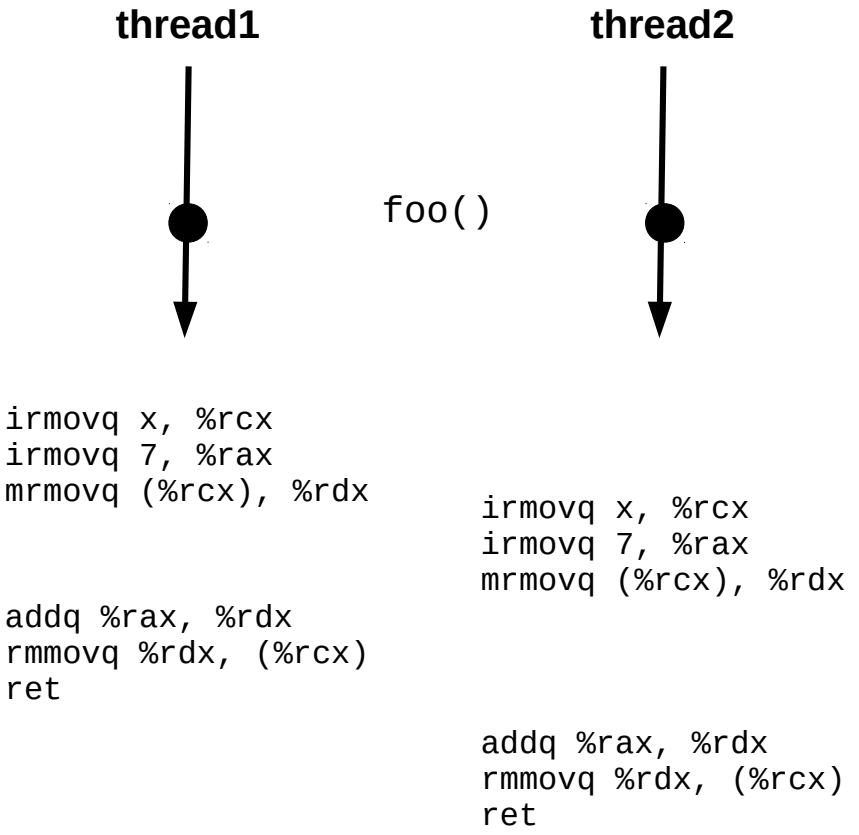
```
foo:
    irmovq x, %rcx
    irmovq 7, %rax
    mrmovq (%rcx), %rdx
    addq %rax, %rdx
    rmmovq %rdx, (%rcx)
    ret

x:
    .quad 0
```



Issues with shared memory

- Nondeterminism
- Data races and deadlock

```
foo:
    irmovq x, %rcx
    irmovq 7, %rax
    mrmovq (%rcx), %rdx
    addq %rax, %rdx
    rmmovq %rdx, (%rcx)
    ret

x:
    .quad 0
```

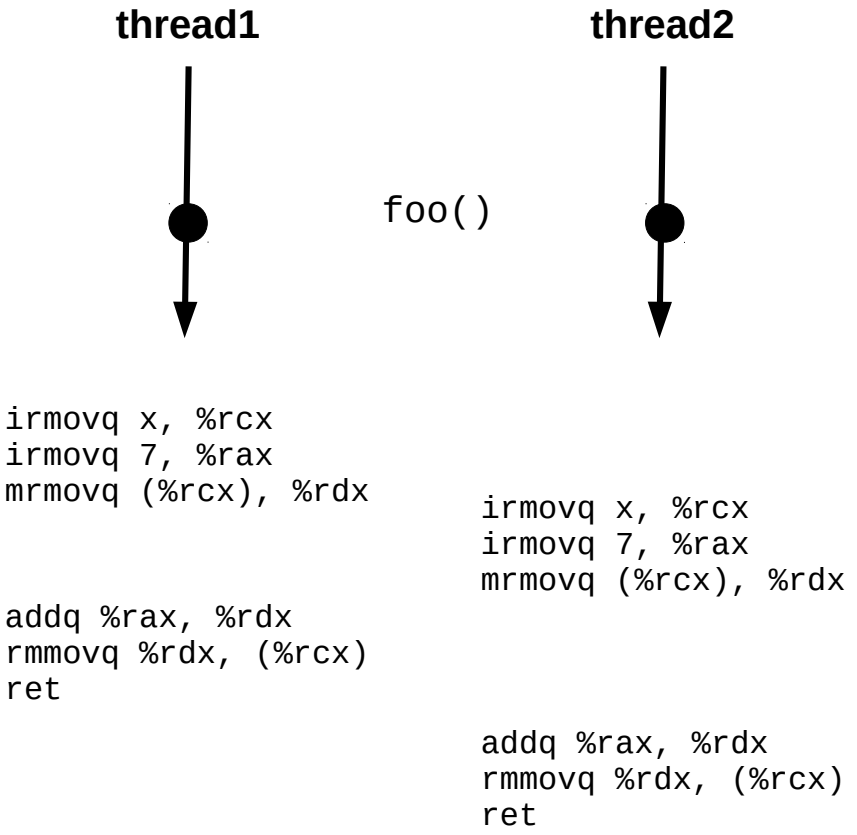


PROBLEM!

Issues with shared memory

- Nondeterminism
- Data races and deadlock

```
foo:
    irmovq x, %rcx
    irmovq 7, %rax
    mrmovq (%rcx), %rdx
    addq %rax, %rdx
    rmmovq %rdx, (%rcx)
    ret

x:
    .quad 0
```



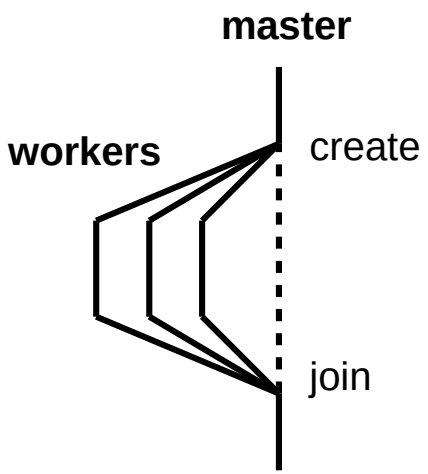
This will be a major topic in CS 361.

Processes vs. threads

- Process: currently-executing program
  - Code and state (PC, stack, data, heap)
  - **Private address space** not shared w/ other processes
  - Advantages: isolation, safety, and mutual exclusion
- Thread: unit of execution or logical flow
  - Exists within a process context
  - **Shared address space** w/ other threads
  - Private PC, registers, condition codes, and stack
  - Advantages: faster context switching, more shared resources

Processes vs. threads

- Common pattern: master/worker threads
  - One original (main thread) creates multiple “child” threads
  - Each worker thread does a chunk of the work
    - Coordinate via shared global data structure w/ locking
  - Main thread waits for workers, then aggregates results



Parallelism & operating systems

- Uniprogramming / batch (1950s) - **CS 261**
  - One process at a time w/ complete control of CPU
  - Minimal OS (mostly for launching programs)
- Multiprogramming / multitasking / time sharing (1960s) - CS 261, **CS 450**
  - Multiple processes taking turns on a single CPU
  - Increased utilization, lower response time
  - OS handles context switching
- (Symmetric) multiprocessing (1970s) - **CS 361**, CS 450, CS 470
  - Multiple processes share multiple CPUs or cores
  - Increased throughput, increased parallelism
  - OS handles scheduling and communication
- Distributed processing (1980s) - CS 361, **CS 470**
  - Multiple processes share multiple computers
  - Massive scaling; OS no longer sufficient (other middleware required)

Automatic parallelism

- Wouldn't it be great if the compiler could automatically parallelize our programs?
  - This is a HARD problem
  - In some cases, it is (kind of) possible
  - Approach #1: code annotations in existing language
    - Example: OpenMP (CS 450, **CS 470**)
  - Approach #2: new language designed for parallelism
    - Example: Chapel (**CS 470**)

```
int a[100];

#pragma omp parallel for
int i;
for (i=0; i < 100; i++)
    a[i] = i*i;
```

OpenMP example

```
var a: [100] int;

forall i in 0..100 do
    a[i] = i*i;
```

Chapel example