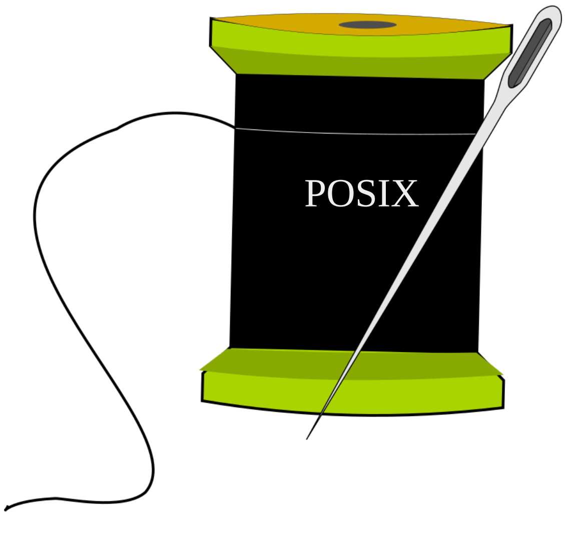
CS 470
Spring 2017

Mike Lam, Professor



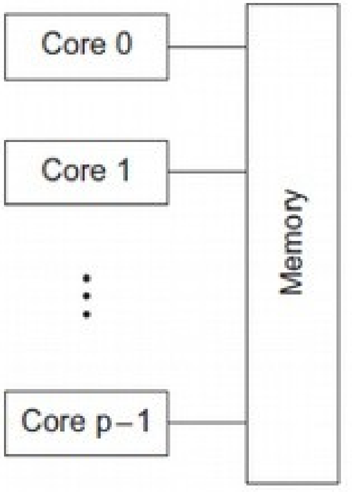
Multithreading & Pthreads

Case study

- Two people wish to compose a document together
  - Both have significant contributions
  - How might this work if collaborating via email?
  - How would it be different if working in the same room?
  - How would it be different with more than two people?

Multithreading

- A process is an instance of a running program
  - Private address space, shared files/sockets
- A thread is a single unit of execution
  - Private stack/registers, shared address space
- Multithreading libraries provide thread management
  - Spawn/kill capabilities
  - Synchronization mechanisms
  - POSIX threads: Pthreads

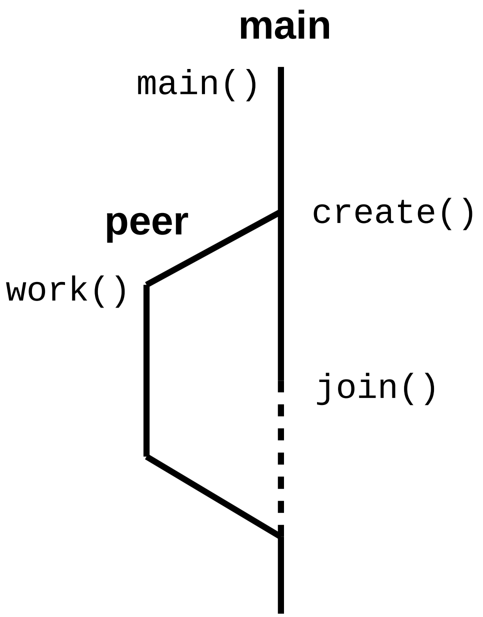


POSIX threads

- Pthreads – POSIX standard interface for threads in C
  - `pthread_create`: spawn a new 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 new thread!\n");
    return NULL;
}

int main ()
{
    printf("Spawning new thread ...\n");

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

    printf("Done!\n");

    return 0;
}
```



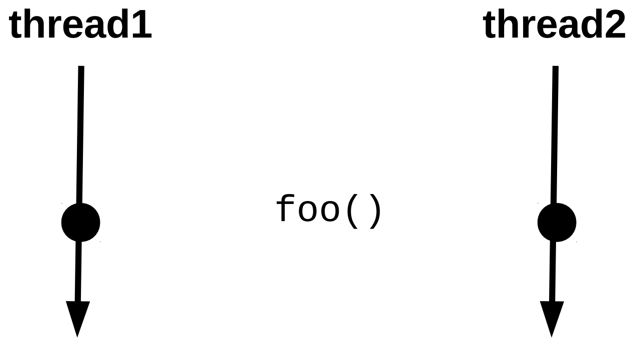
Shared memory

- Some data is shared 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

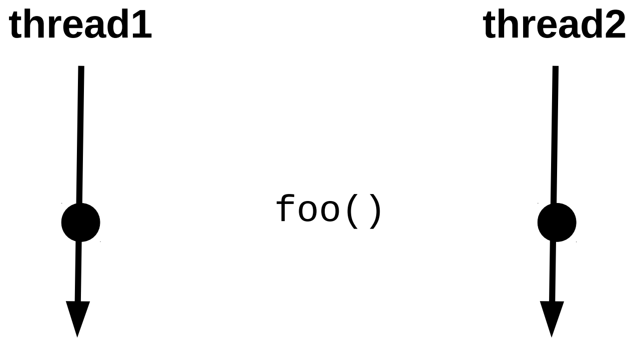
```
foo:
    irmovq x, %rcx
    irmovq 7, %rax
    mrmovq (%rcx), %rdx
    addq %rax, %rdx
    rmmovq %rdx, (%rcx)
    ret

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
```



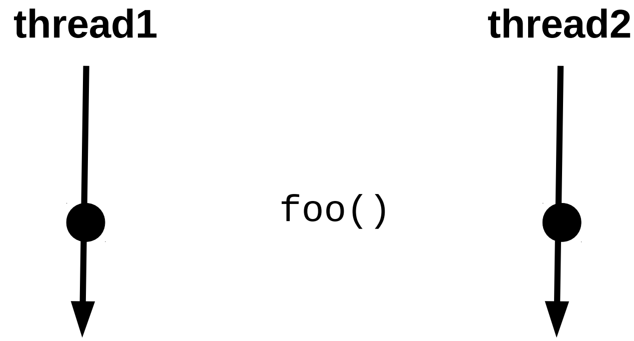

```
irmovq x, %rcx
irmovq 7, %rax
                                irmovq x, %rcx
                                irmovq 7, %rax
mrmovq (%rcx), %rdx
addq %rax, %rdx
rmmovq %rdx, (%rcx)
ret
                                mrmovq (%rcx), %rdx
                                addq %rax, %rdx
            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

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
```




```
irmovq x, %rcx
irmovq 7, %rax
mrmovq (%rcx), %rdx
```

```
irmovq x, %rcx
irmovq 7, %rax
mrmovq (%rcx), %rdx
```

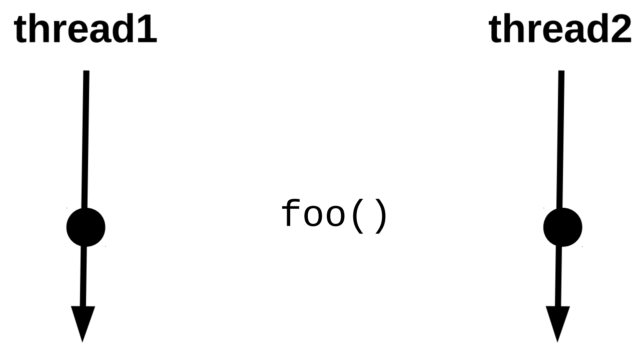
```
addq %rax, %rdx
rmmovq %rdx, (%rcx)
ret
```

```
addq %rax, %rdx
rmmovq %rdx, (%rcx)
ret
```

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
```



thread1:

```
irmovq x, %rcx
irmovq 7, %rax
mrmovq (%rcx), %rdx
```

thread2:

```
irmovq x, %rcx
irmovq 7, %rax
mrmovq (%rcx), %rdx
```

thread1:

```
addq %rax, %rdx
rmmovq %rdx, (%rcx)
ret
```

thread2:

```
addq %rax, %rdx
rmmovq %rdx, (%rcx)
ret
```

PROBLEM!

Issues with shared memory

- Nondeterminism
  - **Incorrect code can produce “correct” results**
  - Test suites cannot guarantee correctness!
- Data races
- Deadlock
- Starvation

Synchronization mechanisms

- Busy-waiting (wasteful!)
- Atomic instructions (e.g., `LOCK` prefix in x86)
- Pthreads
  - Mutex: simple mutual exclusion (“lock”)
  - Condition variable: lock + wait set (wait/signal/broadcast)
  - Semaphore: access to limited resources
    - Not technically part of Pthreads library (just the POSIX standard)
  - Barrier: ensure all threads are at the same point
    - Not present in all implementations (requires `--std=gnu99` on cluster)
- Java threads
  - Synchronized keyword: implicit mutex
  - Monitor: lock on object (wait/notify/notifyAll)

Common synchronization patterns

- Naturally (“embarrassingly”) parallel
  - No synchronization!
- Mutual exclusion
  - Use a lock
- Producer/consumer
  - Protect common buffer w/ lock
- Readers/writers
  - Multiple lock types
- Dining philosophers
  - Atomic acquisition of multiple locks

Synchronization granularity

- Granularity: level at which a structure is locked
  - Whole structure vs. individual pieces
  - If individual pieces, which pieces?
  - Simple locks vs. read/write locks
  - Tradeoff: coarse (lower granularity) vs. fine-grained (higher granularity) locks

Caching effects

- Caching
  - Keep frequently-used stuff in faster memory
- Cache line
  - Single unit of cached data
- Cache hits/misses
  - Was data in cache? (if so, hit; if not, miss)
- Cache invalidation
  - Writes to one cache can render another cache out-of-date
- False sharing
  - Unnecessary cache invalidation

Multithreading summary

- Shared memory parallelism has a lot of benefits
  - Low overhead for thread creation/switching
  - Uniform memory access times (symmetric multiprocessing)
- It also has significant issues
  - Limited scaling (# of cores)
  - Requires explicit thread management
  - Requires explicit synchronization (**HARD**!)
  - Caching problems can be difficult to diagnose
- Core design tradeoff: synchronization granularity
  - Higher granularity: simpler but slower
  - Lower granularity: more complex but faster