CS 61 Lecture 24 Nov 21, 2019

Concurrency Overview

1 execution unit → concurrency by MULTIPROGRAMMING

History

- 1 processor / 1 program
- single processor / many processes
- many processors / many processes

TODAY → many processors / one process

Threads

- user-level abstraction of multiple execution resources
- one process can have multiple threads of control simultaneously
  - need more: cache, registers, rip

process	thread
· individualized view of hardware resources	
· isolation: individual views of memory, separate addr spaces	· shared memory running on top of virtual memory
· file abstraction	

C++ std::thread

- maintain process isolation
- std::thread(threadfunc, &n)

method defining function to be threaded

argument to threadfunc

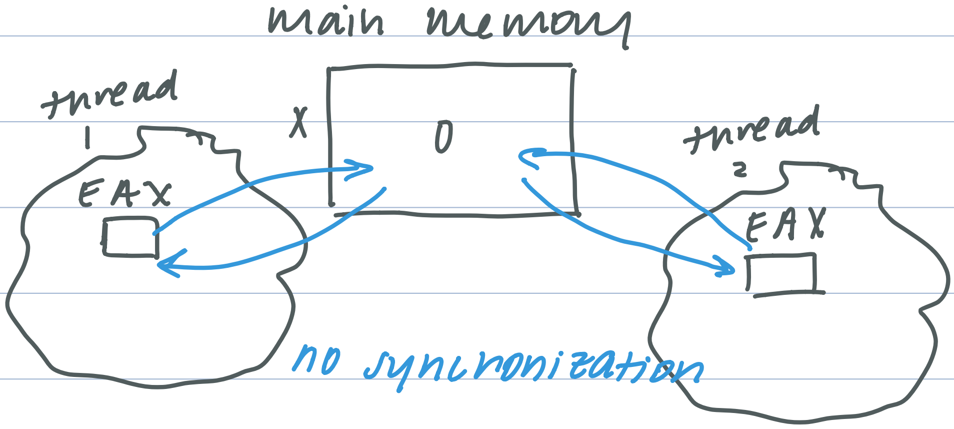
th[i].join() ← wait for thread to complete th[i].detach() ← don't wait

threadfunc (unsigned * x) ← increment int to 10,000,000

```
movl x, %eax
addl $1, %eax
movl %eax, x
```

equal to

```
addl $1, x
```



C++ std::atomic ← single thread accesses and modifies

- looking to build synchronization objects

```
mov   $0x..., %eax
nopl  (%rax)
lock  addl $0x1, (%rdi)
sub   $0x1, %eax
```

C++ std::mutex

- mutual exclusion, only one thread has access to a piece of code at a time
- mutex.lock();

  <<code>>

  mutex.unlock();

mutex objects

- 1 of 2 states < unlocked / locked
- initialization: starts unlocked
- mutex::lock() wait till state == unlocked

atomically set state = locked

- mutex::unlock()   assert locked

  set state = unlocked

* when 2 threads access same normal variable → undefined behavior

```
struct mutex
    std::atomic<int> state = unlocked;
     void lock() {
          while (state == locked) {}
                 load()
           state = locked;
           store()
     }

     void unlock() {
          state = unlocked;
          store()
     }
```

two threads could access this

STILL WRONG!