CONCURRENCY: INTRODUCTION

Shivaram Venkataraman

CS 537, Spring 2019

ADMINISTRIVIA

- Project 2b is out. Due Feb 27th, 11:59
- Project 2a grading in progress

Discussion:

Makefile tutorial

How to return values from a syscall

AGENDA / LEARNING OUTCOMES

Virtual memory: Summary

Concurrency

- What is the motivation for concurrent execution?
- What are some of the challenges?

RECAP

SWAPPING INTUITION

Idea: OS keeps unreferenced pages on disk

- Slower, cheaper backing store than memory

Process can run when not all pages are loaded into main memory

OS and hardware cooperate to make large disk seem like memory

- Same behavior as if all of address space in main memory

Requirements:

- OS must have **mechanism** to identify location of each page in address space → in memory or on disk
- OS must have **policy** for determining which pages live in memory and which on disk

VIRTUAL MEMORY MECHANISMS

First, hardware checks TLB for virtual address

– if TLB hit, address translation is done; page in physical memory

Else ...

– Hardware or OS walk page tables → VPN → PPN

– If PTE designates page is present, then page in physical memory (i.e., present bit is cleared) → Page Table

Else

– Trap into OS (not handled by hardware)

– OS selects victim page in memory to replace → if not enough memory

  • Write victim page out to disk if modified (add dirty bit to PTE)

– OS reads referenced page from disk into memory

– Page table is updated, present bit is set ← invalidate TLB

– Process continues execution

PAGE SELECTION

When

Process
↳ binary
libraries

Demand paging: Load page only when page fault occurs

- Intuition: Wait until page must absolutely be in memory
- When process starts: No pages are loaded in memory
- Problems: Pay cost of page fault for every newly accessed page

Prepaging (anticipatory, prefetching): Load page before referenced

- OS predicts future accesses (oracle) and brings pages into memory early
- Works well for some access patterns (e.g., sequential)

Polluting | wasting memory | disk operations

Hints: Combine above with user-supplied hints about page references

- User specifies: may need page in future, don't need this page anymore, or sequential access pattern, ...
- Example: madvise() in Unix

MEM limit


https://tinyurl.com/cs537-sp19-bunny2

PAGE REPLACEMENT EXAMPLE

Page reference string: ABCABDADBCB

Metric:
Miss count

Three pages of physical memory

	OPT			FIFO				LRU			
	5 misses						3 misses				3 misses
ABC				A	B	C		A	B	C	
A				A	B	C		A	B	C	
B				A	B	C		A	B	C	
D				D	B	C	+4	A	B	D	4th
A				D	A	C	+5	A	B	D	
D				D	A	C		A	B	D	
B				D	A	B	+6	A	B	D	
C				C	A	B	+7	C	B	D	+5
B				C	A	B		C	B	D	

FIFO 7 misses

LRU 5 misses

IMPLEMENTING LRU

Software Perfect LRU

- OS maintains ordered list of physical pages by reference time
- When page is referenced: Move page to front of list → update
- When need victim: Pick page at back of list → replacement
- Trade-off: Slow on memory reference, fast on replacement

Hardware Perfect LRU

- Associate timestamp register with each page
- When page is referenced: Store system clock in register
- When need victim: Scan through registers to find oldest clock
- Trade-off: Fast on memory reference, slow on replacement (especially as size of memory grows)

traversing can be slow

In practice, do not implement Perfect LRU

- LRU is an approximation anyway, so approximate more
- Goal: Find an old page, but not necessarily the very oldest

CLOCK ALGORITHM

Hardware

- Keep use (or reference) bit for each page frame
- When page is referenced: set use bit

bits are cheap!

use = 1 page was used recently

Operating System

- Page replacement: Look for page with use bit cleared (has not been referenced for awhile)
- Implementation:
  - Keep pointer to last examined page frame
  - Traverse pages in circular buffer
  - Clear use bits as search
  - Stop when find page with already cleared use bit, replace this page

use = 0 cleared

CLOCK: LOOK FOR A PAGE

Use

	Use=1	Use=0	Use=1	Use=1	
Physical Mem:	0	1	2	3	...

↑ clock hand

We need to find page to evict

if use_bit == 1
    clear use_bit
    move clock
else select page

Page 0 is accessed
↳ set use_bit to 1

Approx LRU

SUMMARY: VIRTUAL MEMORY

external fragmentation

Abstraction: Virtual address space with code, heap, stack

Address translation

- Contiguous memory: base, bounds, segmentation
- Using fixed sizes pages with page tables

Challenges with paging

- Extra memory references: avoid with TLB
- Page table size: avoid with multi-level paging, inverted page tables etc.

cache

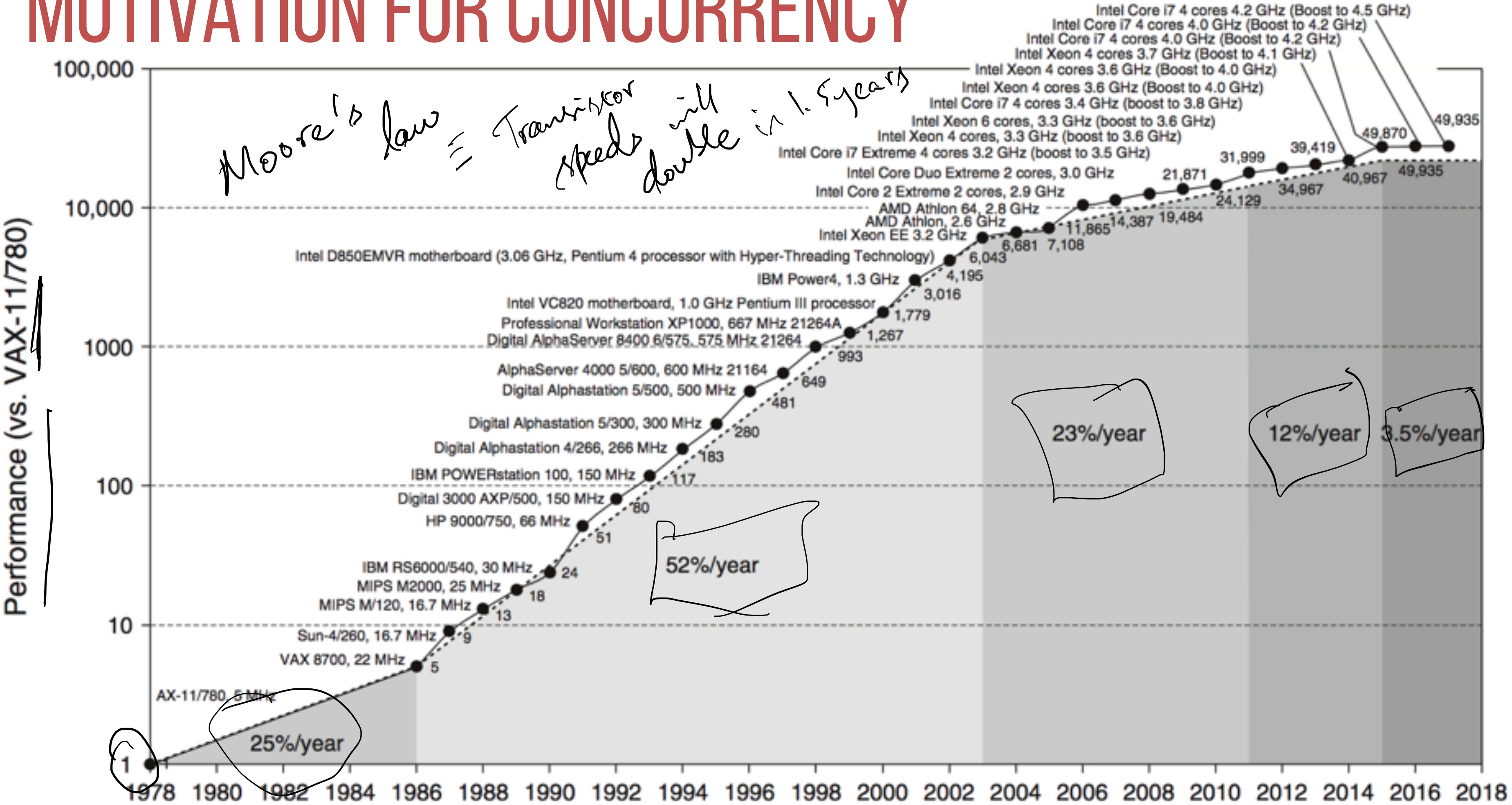
Larger address spaces: Swapping mechanisms, policies (LRU, Clock)

REVIEW: EASY PIECE 1

- Virtualization
  - CPU
    - Context Switch
    - Schedulers
  - Memory
    - Allocation
    - Segmentation
    - Paging
      - TLBs
      - Multilevel
      - Swapping

CONCURRENCY

MOTIVATION FOR CONCURRENCY



MOTIVATION

CPU Trend: Same speed, but multiple cores

Goal: Write applications that fully utilize many cores

Option 1: Build apps from many communicating **processes**

streams of execution

isolation, parallelism

- Example: Chrome (process per tab)
- Communicate via pipe() or similar

Pros?

- Don't need new abstractions; good for security

Cons?

- Cumbersome programming
- High communication overheads
- Expensive context switching (why expensive?)

CONCURRENCY: OPTION 2

threads vs process

New abstraction: thread → streams of execution

code, stack, heap??

Threads are like processes, except:

multiple threads of same process share an address space

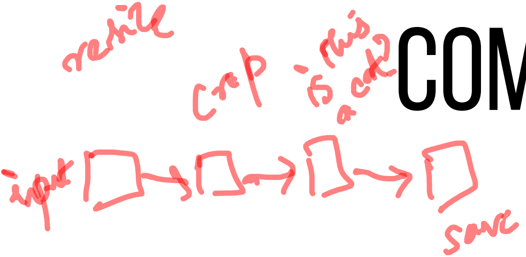
Divide large task across several cooperative threads

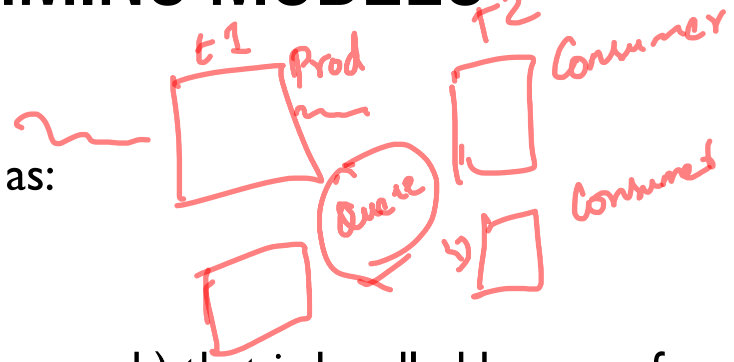
Communicate through shared address space

→ fine grained

high performance

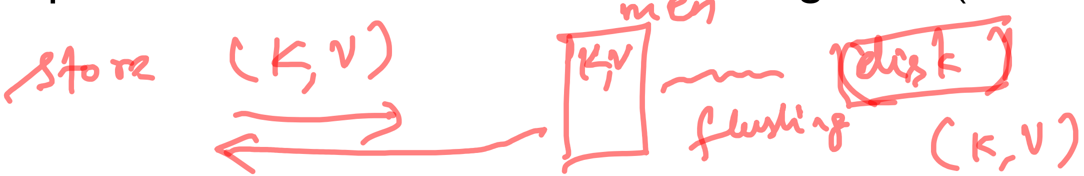
COMMON PROGRAMMING MODELS

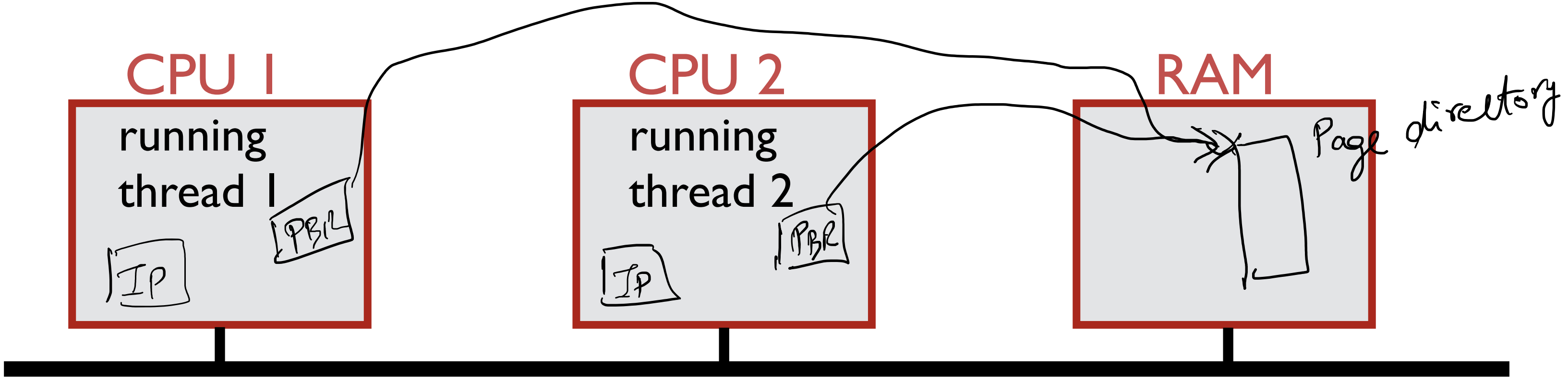




Multi-threaded programs tend to be structured as:

- **Producer/consumer**
  Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
- **Pipeline**
  Task is divided into series of subtasks, each of which is handled in series by a different thread
- **Defer work with background thread** → expensive
  One thread performs non-critical work in the background (when CPU idle)

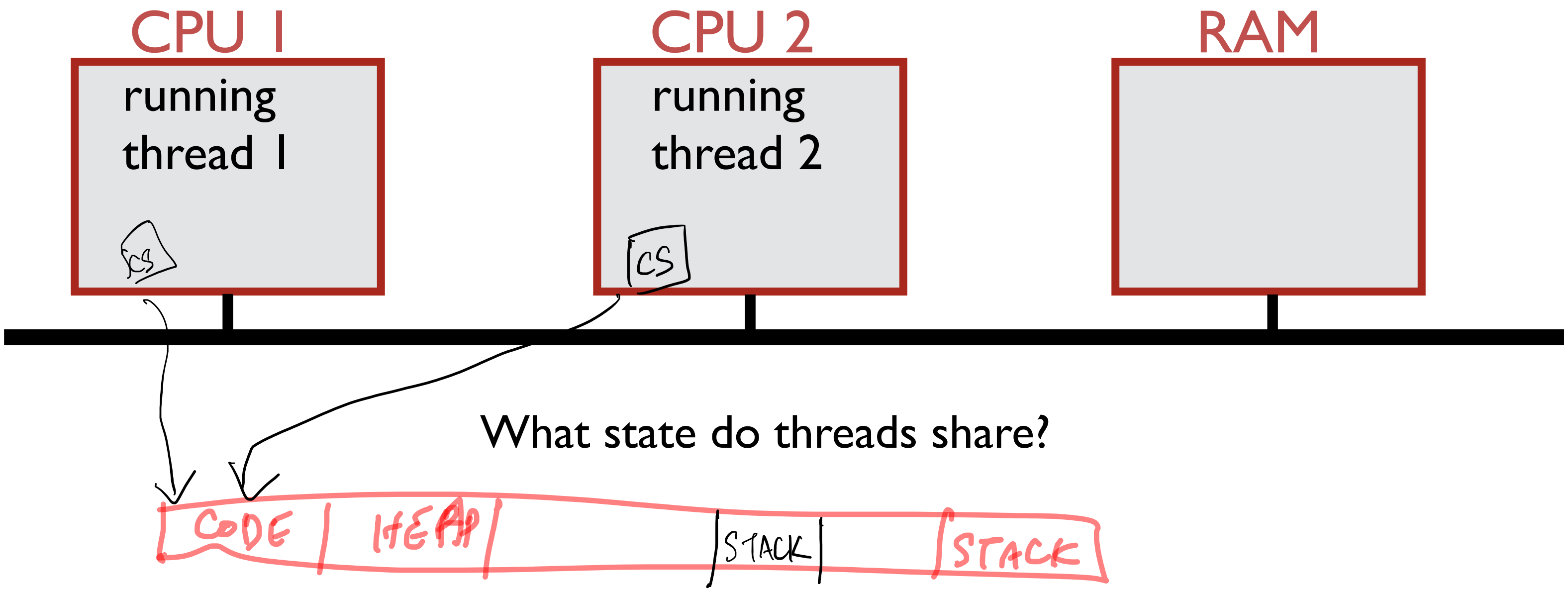




What state do threads share?

Page directories, Page tables ? Shared

instruction pointed ? Not shared



What state do threads share?

Code segment → Shared

Stack ? Not stack

THREAD VS. PROCESS

Multiple threads within a single process share:

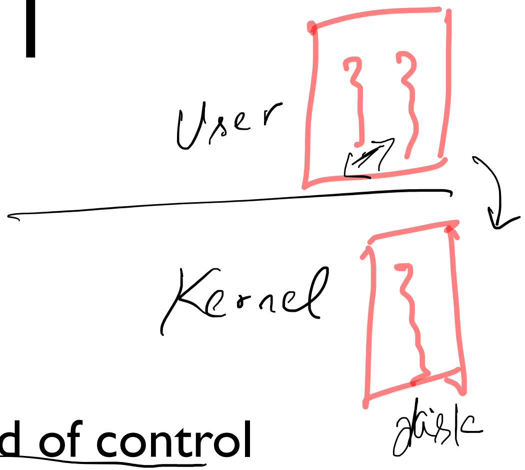
- Process ID (PID)
- Address space: Code (instructions), Most data (heap)
- Open file descriptors
- Current working directory
- User and group id

Each thread has its own

- Thread ID (TID)
- Set of registers, including Program counter and Stack pointer
- Stack for local variables and return addresses
  (in same address space)

register files

OS SUPPORT: APPROACH 1



User-level threads: Many-to-one thread mapping

- Implemented by user-level runtime libraries
  - Create, schedule, synchronize threads at user-level
- OS is not aware of user-level threads
  - OS thinks each process contains only a single thread of control

Advantages

- Does not require OS support; Portable
- Can tune scheduling policy to meet application demands
- Lower overhead thread operations since no system call

since no trap

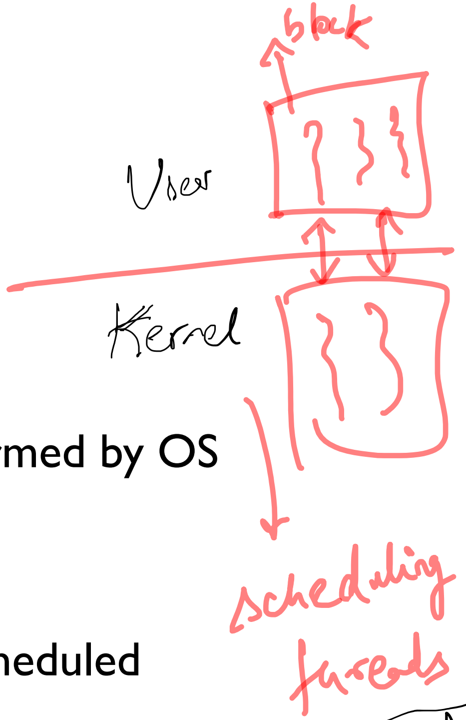
Disadvantages?

- Cannot leverage multiprocessors

Only Programming model

- Entire process blocks when one thread blocks

OS SUPPORT: APPROACH 2



Kernel-level threads: One-to-one thread mapping

- OS provides each user-level thread with a kernel thread
- Each kernel thread scheduled independently
- Thread operations (creation, scheduling, synchronization) performed by OS

Advantages

- Each kernel-level thread can run in parallel on a multiprocessor
- When one thread blocks, other threads from process can be scheduled

Disadvantages

- Higher overhead for thread operations
- OS must scale well with increasing number of threads

thread per request

THREADS DEMO

THREAD SCHEDULE #1

`balance = balance + 1;` balance at `0x9cd4`

State:

`0x9cd4:` ~~100~~ 101 ← 102

`%eax:`

`%rip = 0x195`

thread control blocks:

Thread 1	Thread 2
%eax: 101	%eax: 102
%rip: 0x19d	%rip: 0x195

T1

- `0x195   mov 0x9cd4, %eax`
- `0x19a   add $0x1, %eax` ←
- `0x19d   mov %eax, 0x9cd4`

0x195
0x19a

T2

THREAD SCHEDULE #2

`balance = balance + 1; balance at 0x9cd4`

State: 101
`0x9cd4: 100` 101
`%eax:`
`%rip = 0x195`

thread control blocks:

	Thread 1	Thread 2
%eax:	101	101
%rip:	0x195	0x195

T1

- `0x195   mov 0x9cd4, %eax`
- `0x19a   add $0x1, %eax`
- `0x19d   mov %eax, 0x9cd4`

Context switch T2
mov
add
mov

Final answer 101 ≡ Context switch

T1 resumes

TIMELINE VIEW

100

TIME

Thread 1	Thread 2
mov 0x123, %eax 100	
add %0x1, %eax	
mov %eax, 0x123 101	
	mov 0x123, %eax 101
	add %0x2, %eax
	mov %eax, 0x123 103

3 increment ✓ Correct

BUNNY

tinyurl.com/cs537-sp19-bunny3

NON-DETERMINISM

Concurrency leads to non-deterministic results

- Different results even with same inputs
- race conditions

Whether bug manifests depends on CPU schedule!

How to program: imagine scheduler is malicious?!

WHAT DO WE WANT?

Want 3 instructions to execute as an uninterruptable group

That is, we want them to be atomic

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

More general: Need mutual exclusion for critical sections

if thread A is in critical section C, thread B isn't

(okay if other threads do unrelated work)

SYNCHRONIZATION

Build higher-level synchronization primitives in OS

Operations that ensure correct ordering of instructions across threads

Use help from hardware

Motivation: Build them once and get them right

Monitors
Locks
Semaphores
Condition Variables

Loads
Stores
Test&Set
Disable Interrupts

CONCURRENCY SUMMARY

Concurrency is needed for high performance when using multiple cores

Threads are multiple execution streams within a single process or address space (share PID and address space, own registers and stack)

Context switches within a critical section can lead to non-deterministic bugs

LOCKS

Goal: Provide mutual exclusion (mutex)

Allocate and Initialize

- Pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;

Acquire

- Acquire exclusion access to lock;
- Wait if lock is not available (some other process in critical section)
- Spin or block (relinquish CPU) while waiting
- Pthread_mutex_lock(&mylock);

Release

- Release exclusive access to lock; let another process enter critical section
- Pthread_mutex_unlock(&mylock);

THREADS DEMO2

NEXT STEPS

Project 2b: Out now

Next class: How to implement locks?

Discussion:

- Makefile tutorial
- How to return values from a syscall