CS 423
Operating System Design: Swapping and Intro to Concurrency
Feb 24

Ram Kesavan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

ANNOUNCEMENTS

MP1 due tomorrow (2/25) 11:59 pm CT

Pop quizzes

4Cr: next paper on “Huge Pages”, due 3/3

Misc Resources section:

- VM/Paging/MMU YT video has a small mistake

AGENDA / LEARNING OUTCOMES

Finish the discussion on swapping

A pop-quiz

Start on concurrency (second piece):

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

RECAP

Swap Space

Designated & reserved fraction of persistent storage

- Sizing is configurable; usually multiple of the physical memory

Persistent storage is accessed in units of “blocks”

- Typically, block size is equal to or a multiple of page size.
- “Paging out”: write a page out to swap space
- “Paging in”: read a page in from swap space*

OS tracks free space in the swap space

- Simple block addressing: linearly from 0 to n

*not necessarily

Virtual Address Space Mechanisms

Basic mechanism: use present bit to say whether page is in physical memory

If not present, PTE typically points to location on swap device

- Unless its code/static data (backed by the binary)

On access, HW raises an exception called page fault

- OS handles the page fault
- Bring in the required page from swap space (or the binary)
- Modifies the PTEs to reflect these changes

Swap/page daemon creates free space based on two watermarks (low and hi)

- If victim-page has dirty bit set, it needs to be written out swap space
- Else it can be reclaimed immediately

Interaction With OS Scheduler

During TLB miss handling, process is in _____ state

(Hard) page faults are expensive: OS must read from disk

During page fault handling, process is in _____ state

Once page fault handling kicks off the read I/O, process moves to ____ state

The process moves to _____state once the read I/O completes and sets the PTE's present bit.

END RECAP

SWAP SPACE LOCATION

The fact that swap space is persistent is somewhat *incidental*

- It happens to be the next tier in the memory hierarchy!

Persistence allows features, such as laptop hibernation

- Smartphones typically use zRAM instead

Local: SSD or hard drive or USB device

Remote: storage on a different system

- RDMA-based swapping

SWAPPING POLICIES

SWAPPING Policies

Goal: Minimize number of page faults

- Page faults require millisecs to handle (reading from swap)
- Implication: Lots of time for OS to make good decisions

OS has two decisions

- Page selection
  When should a page (or pages) on disk be **brought into** memory?

- Page replacement
  Which resident page (or pages) in memory should be **kicked out?**

Page Selection

Demand paging: Load page only when page fault occurs

- Intuition: Wait until the last minute; when page is absolutely needed
- When a process starts up: No pages are loaded in memory
- Problem: Pay cost of page fault for every newly accessed page

Pre-paging (anticipatory, prefetching): Load page beforehand

- OS predicts future access and prefetches pages into memory
- Works well for some access patterns (e.g., sequential)
- Two costs of bad prefetching?

Some combination of the two, together with hints

Hints: User-supplied hints about page references

Hints with madvise()

int madvise(void *addr, size_t length, int advice) - allows user to give hints to OS

MADV_RANDOM // kernel may disable prefetching

MADV_SEQUENTIAL // kernel may aggressively prefetch…

// …and kick out after access

MADV_WILLNEED

MADV_DONTNEED

and more…

Generally, hard to tune and extract better performance than OS default

Page Replacement

One potential dimension for selecting victim pages kick out:

- Write out victim page to disk if modified (dirty bit set)
- If victim page is not modified (clean), just discard
- So when it makes sense, pick clean pages as victims

Now the policies

OPTIMAL: Replace the page to be accessed furthest in the future

- Pro: Guaranteed to minimize #page faults
- Con: Requires knowing the future! Impractical, but good for comparisons

Page Replacement

FIFO: Replace page that has been in memory the longest

- Intuition: Referenced long time ago, get rid of it
- Advantage: Fair as each page gets equal residency; Easy to implement
- Disadvantage: Some pages may remain popular

LRU: Least-recently-used

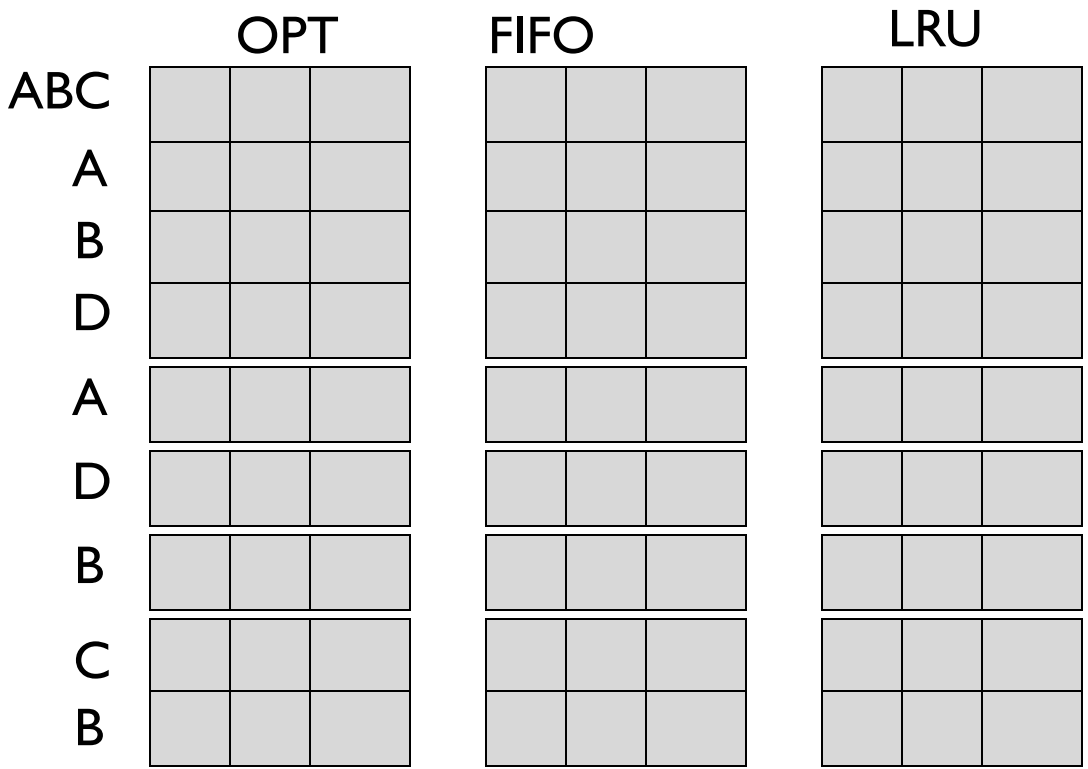
- Intuition: Recent past predicts the near future
- Advantages: With locality, LRU approximates OPTIMAL
- Disadvantages:
  - Harder to implement, must track which pages have been accessed

Page Replacement - chat for 2 mins

Page reference string: ABCABDADBCB

Metric:
Miss count

Three pages of physical memory



LRU vs. OPT

QUESTION: Think of a workload & config where LRU will be way worse than OPT. Random replacement might do better that LRU!

Page Replacement Comparison

What happens to performance with increase in physical memory size?

LRU, OPT:

- Guaranteed to have fewer (or same number of) page faults
- Stack property:
  - cache of size N+1 includes contents of cache of size N

FIFO:

- Usually have fewer page faults
- Belady's anomaly: May actually have more page faults!

FIFO PERFORMANCE MAY DECREASE!

Consider access stream: ABCDABEABCDE

Physical memory size: 3 pages vs. 4 pages

How many misses with FIFO?

9 misses with 3 pages and 10 misses with 4 pages

3 Pages
A, B, C, D, A, B, E miss
A hit, B hit, C, D miss
E hit

4 pages
A, B, C, D miss
A hit, B hit, E miss –
A, B, C, D, E miss

IMPLEMENTING LRU

Software Perfect LRU

- OS maintains ordering of physical pages by reference time
- When page is referenced: modify the ordering
- When need victim: Pick page with oldest time
- Trade-off: Slow on memory reference, fast on replacement

Hardware Perfect LRU

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

In practice, do not implement Perfect LRU

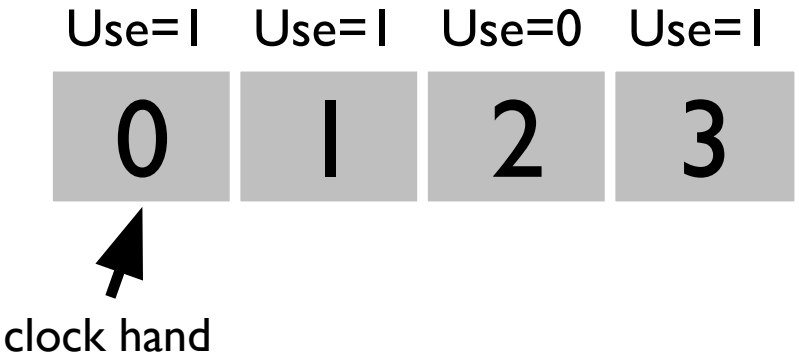
- LRU is an approximation anyway, so approximate some more :-)
- Goal: Find an old page, but not necessarily the very oldest

CLOCK ALGORITHM: APPROX LRU

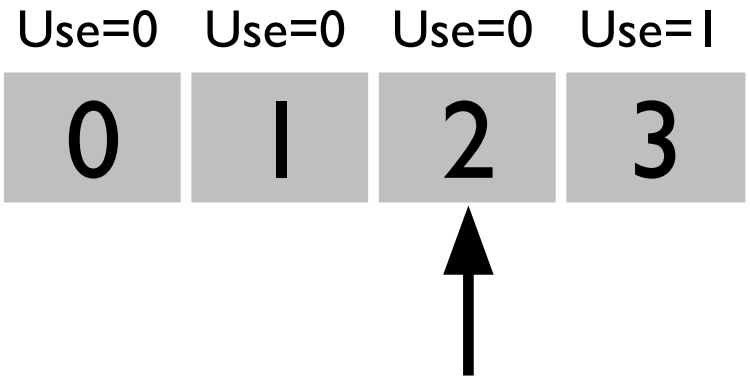
- PTE comprises a use/reference bit
- Is set whenever a page is referenced
- Page replacement: Look for page with use bit cleared
  (has not been referenced for awhile)
- Implementation:
  - Keep pointer to last examined page frame
  - Traverse pages in circular fashion
  - Clear use bits as search
  - Stop when find page with already cleared use bit, replace this page

CLOCK: LOOK FOR A PAGE

Physical Mem:



Evict contents of frame 2, load in new page



CLOCK EXTENSIONS

Replace multiple pages at once

- Intuition: Efficient to write pages to swap space in bulk
- Find multiple victims each time and track free list

Use dirty bit to bias against replacing dirty pages

- Intuition: More expensive to replace dirty pages
  Dirty pages must be written to disk, clean pages do not
- Replace pages that have use bit and dirty bit cleared

Add software counter (“unused”)

Intuition: Can tell how often pages are accessed
Increment software counter if use bit is 0
Replace when change exceeds some specified limit

LINUX 2Q

Linux has a unified page cache (will cover later in course)

Maintains active list vs inactive list

On first reference, put page in inactive list

- If accessed again, promoted to active list
- Pick victim pages from inactive list in FIFO order to swap out

In background: move pages from active to inactive list until active list is <= 2/3rd of physical memory size

- Protects against larger-than-cache-cyclic-access pattern (will cover later)

LINUX CODE

include/linux/swap.h

- Data structure used to track swap area
  - struct swap_info_struct
- Operates in units of 256 pages “clusters”
  - struct swap_cluster_info
- Parallelism: each CPU is given a cluster

OUT OF MEMORY

What happens when total memory allocated to all processes > RAM + swap space?

1. OS constantly pages in/out from swap space–aka *thrashing*

   ProcessA kicks out p2 to page in p1, ProcessB kicks out p1 to page in p2, and so on…

   System noticeably slows down

2. OOM Killer (kernel) gets invoked

   Kill processes that hog memory; typically younger processes

3. User processes: malloc() returns null

SUMMARY: VIRTUAL MEMORY

Abstraction: Virtual address space with code, heap, stack

Address translation

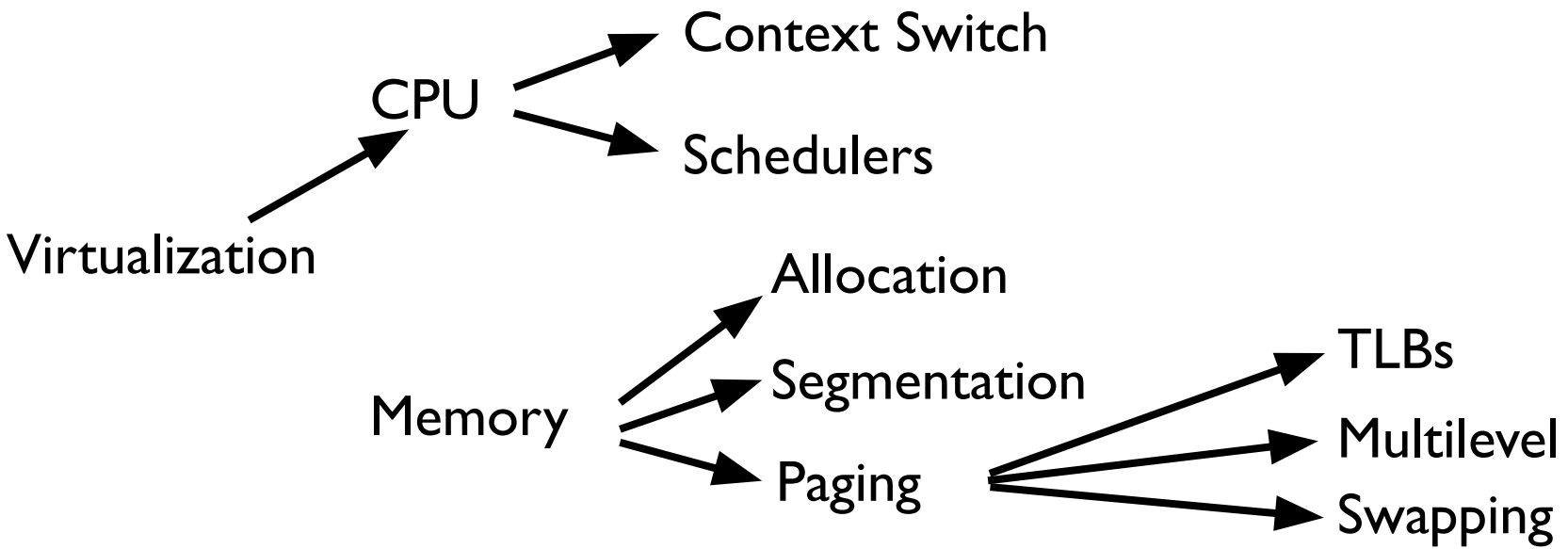
- Contiguous memory: base, bounds, segmentation
- Using fixed sizes pages with page tables

Challenges with paging

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

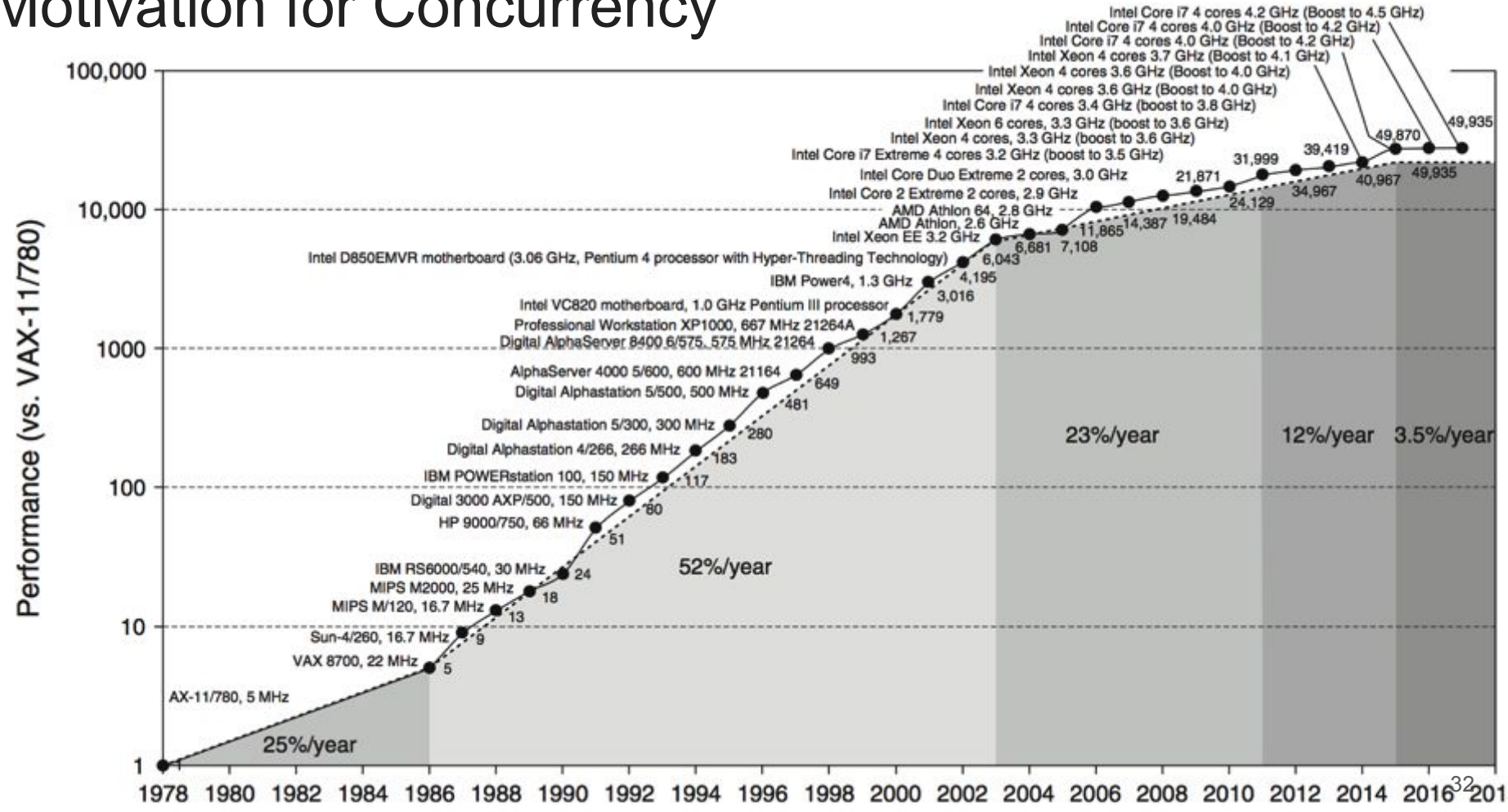
Larger address spaces: Swapping mechanisms, policies (LRU, Clock)

REVIEW: EASY PIECE 1



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**

- 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

CONCURRENCY: OPTION 2

New abstraction: thread

Threads are like processes, except:

multiple threads of same process share an address space

Divide large task across several cooperative threads

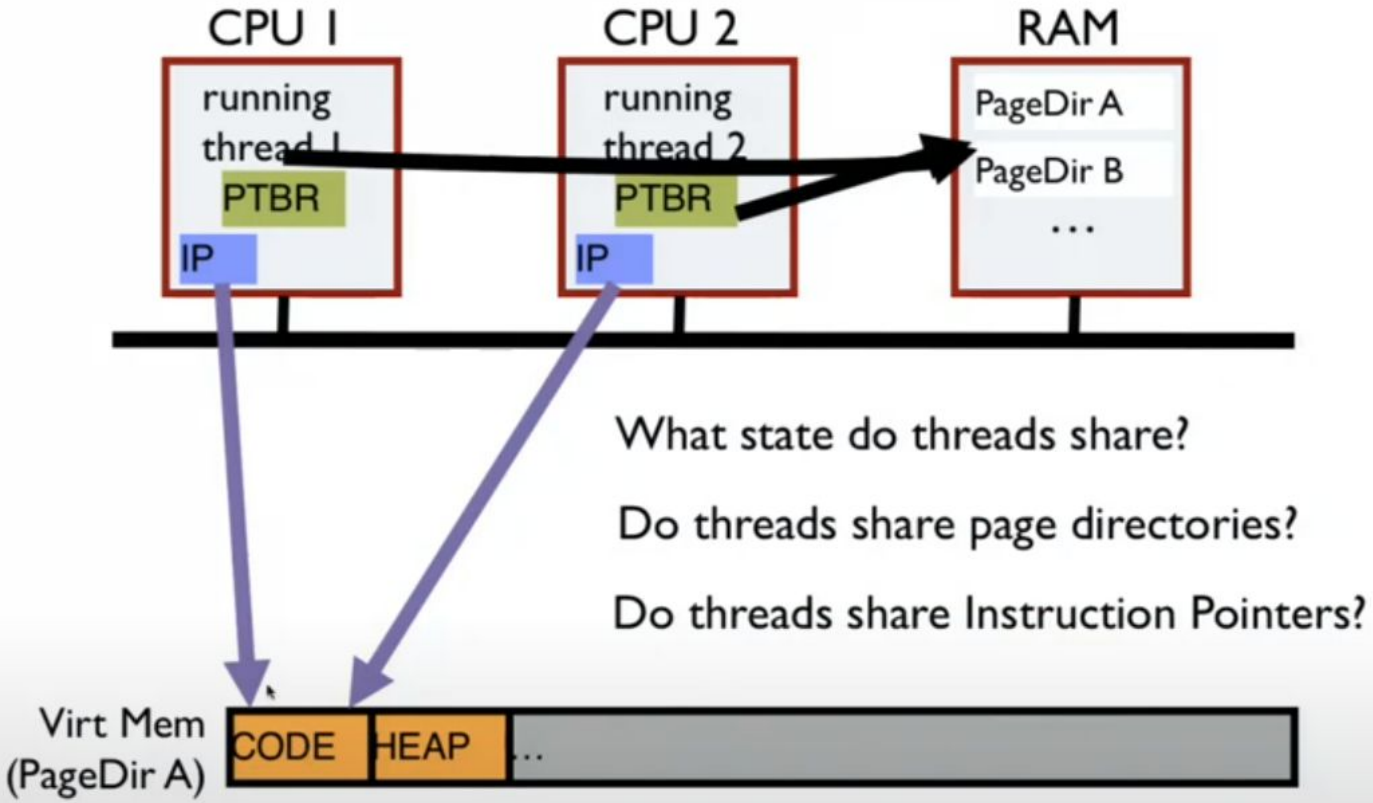
Communicate through shared address space

We discuss POSIX thread library, aka **pthreads**

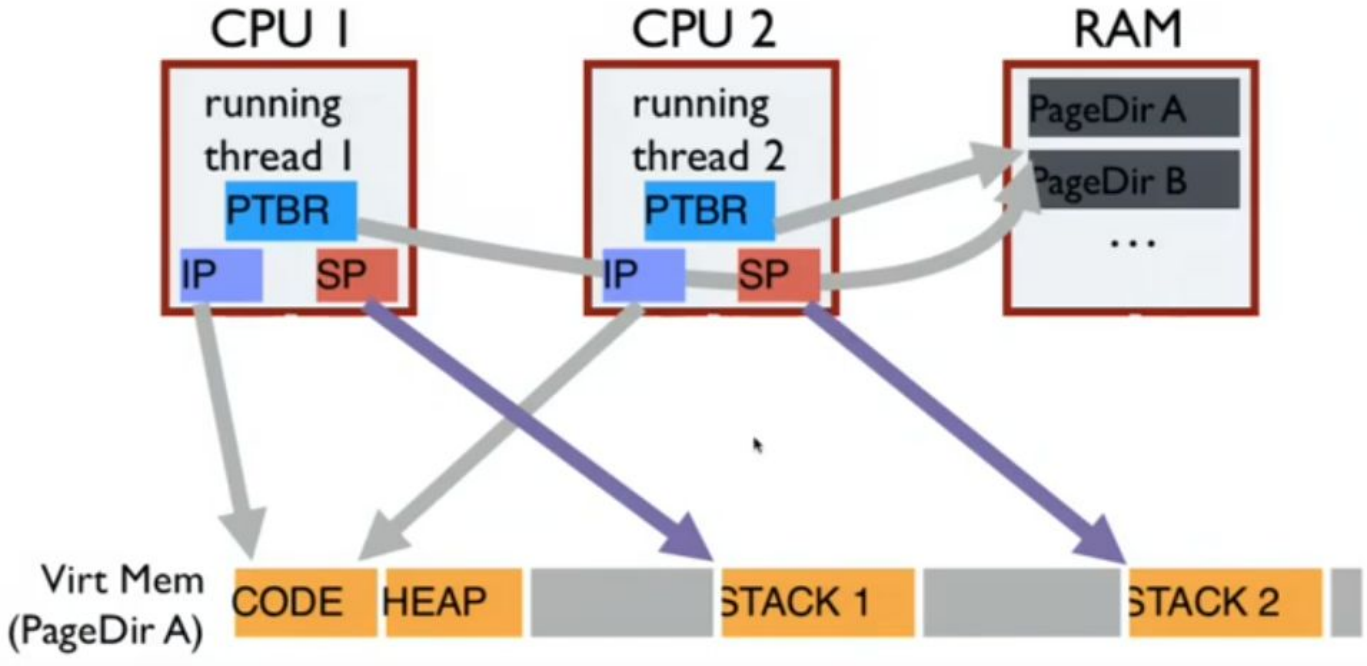
Common Programming Models

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**
  One thread performs non-critical work in the background (when CPU idle)



Share code, but each thread may be executing different code at the same time
→ Different Instruction Pointers



Do threads share stack pointer?

Threads executing different functions need different stacks
(But, stacks are in same address space, so trusted to be cooperative)

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)

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 policies to suit application
- Lower overhead thread operations since no system call

Disadvantages?

- Cannot leverage multiprocessors
- 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

THREAD SCHEDULE

```
volatile int balance = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
      balance++;
    }
    pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[]) {
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", balance);
    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", balance);
    return 0;
}
```

THREAD SCHEDULE #1

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

Registers are virtualized by OS;
Each thread thinks it has own

State:
0x9cd4: 100
%eax: ?
%rip = 0x195

process control blocks:

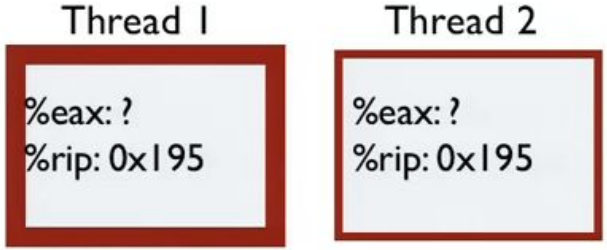
Thread 1	Thread 2
%eax: ? %rip: 0x195	%eax: ? %rip: 0x195


```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

THREAD SCHEDULE #1

State:
`0x9cd4: 101`
`%eax: 101`
`%rip = 0x1a2`

process control blocks:

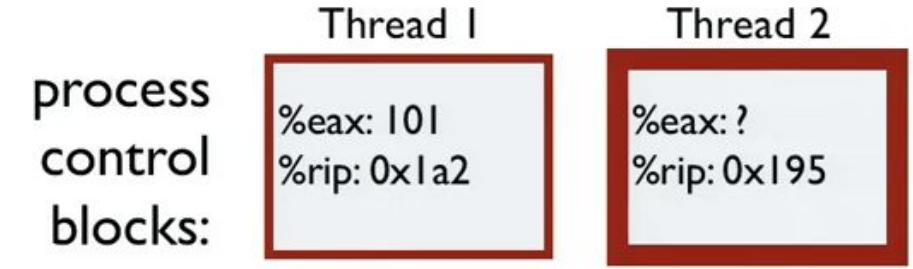



```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

T1

THREAD SCHEDULE #1

State:
0x9cd4: 101
%eax: ?
%rip = 0x195



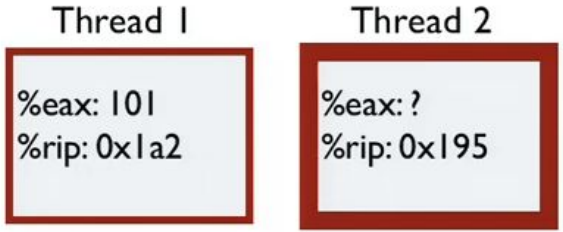
T2

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

THREAD SCHEDULE #1

State:
`0x9cd4: 102`
`%eax: 102`
`%rip = 0x1a2`

process control blocks:




```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

T2

Desired Result!

THREAD SCHEDULE #2

State:
0x9cd4: 100
%eax: 101
%rip = 0x19d

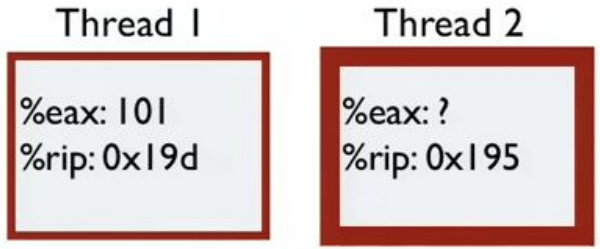

```
        0x195   mov 0x9cd4, %eax
        0x19a   add $0x1, %eax
T1      0x19d   mov %eax, 0x9cd4A
```

Thread Context Switch before T1 executes 0x19d

THREAD SCHEDULE #2

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

process control blocks:




```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

T2

THREAD SCHEDULE #2

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2


```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

T1

WRONG Result! Final value of balance is 101

TIMELINE VIEW

Thread 1	Thread 2	Balance	T1-eax	T2-eax
mov 0x123, %eax		0	0	
add %0x1, %eax			1	
mov %eax, 0x123		1		
	mov 0x123, %eax			1
	add %0x2, %eax			3
	mov %eax, 0x123	3		

How much is added to shared variable? 3: correct!

TIMELINE VIEW

Thread 1	Thread 2	Balance	T1-eax	T2-eax
		0	0	
mov 0x123, %eax			1	
add %0x1, %eax				
	mov 0x123, %eax	1		0
mov %eax, 0x123				
	add %0x2, %eax			2
	mov %eax, 0x123	2		

How much is added?

2: incorrect!

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

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

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

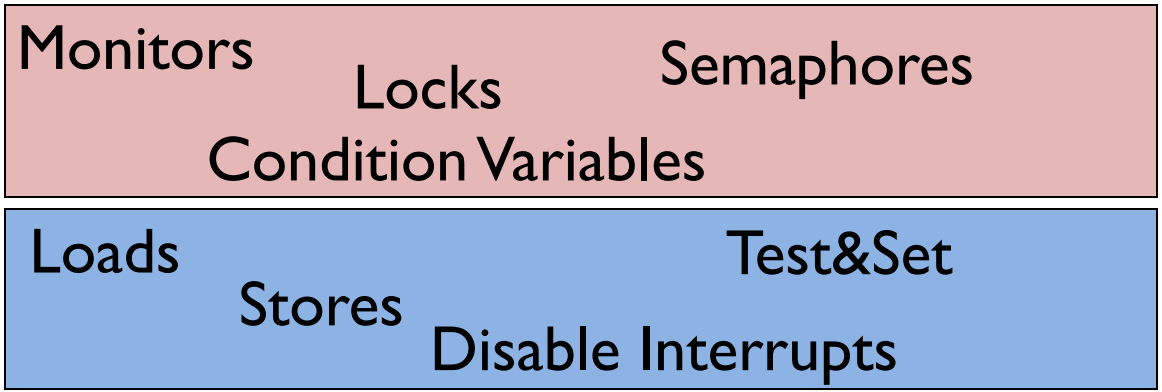
if thread A is in critical section C, thread B isn't

(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



LOCKS

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);

Thread-Safe Queue

```
tryget() {
    item = NULL;
    lock.acquire();
    if (front < tail) {
        item = buf[front % MAX];
        front++;
    }
    lock.release();
    return item;
}
```

```
tryput(item) {
    lock.acquire();
    if ((tail – front) < size) {
        buf[tail % MAX] = item;
        tail++;
    }
    lock.release();
}
```

Initially: front = tail = 0; lock = FREE; MAX is buffer capacity

If tryget return NULL, can we be sure that there are no elements in the queue at that point?

What if we do tryget in a loop?

LOCK IMPLEMENTATION GOALS

Correctness

- *Mutual exclusion*
  Only one thread in critical section at a time
- *Progress* (deadlock-free)
  If several simultaneous requests, must allow one to proceed
- *Bounded* (starvation-free)
  Must eventually allow each waiting thread to enter

Fairness: Each thread waits for same amount of time

Performance: CPU is not used unnecessarily

Implementing Synchronization

Approaches

- Disable interrupts
- Using atomic hardware instructions

Implementing Locks: W/ Interrupts

Turn off interrupts for critical sections
- Prevent dispatcher from running another thread
- Code between interrupts executes atomically

```
void acquire(lockT *l) {
    disableInterrupts();
}
```

```
void release(lockT *l) {
    enableInterrupts();
}
```

Disadvantages?

Only works on uniprocessors

Process can keep control of CPU for arbitrary length

Cannot perform other necessary work

Using Atomic Instructions

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0: lock is available, 1: lock is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Test – return old value

Set – set the passed in value

HW does them atomically!

NEXT STEPS

Next class: More about locks!