CS 423
Operating System Design: Swapping and Intro to Concurrency
02/26

Ram Alagappan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

AGENDA / LEARNING OUTCOMES

Finish discussion on swapping

Summary: virtual memory

Concurrency (second piece):

- What is the motivation for concurrent execution
- What are some 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** to determine which pages live in memory and which on disk

Virtual Address Space Mechanisms

Basic mechanism: use present bit to say whether or not in memory

If not present, PTE stores location on disk

If not present, HW raises an interrupt:

- Called a page fault
- OS must handle the page fault
- Needs to identify a victim page to swap and bring in the required page
- Modifies the PTEs to reflect these changes

Dirty bit indicates whether or not the page was written since the last time it was brought from disk

If dirty, cannot just drop for replacement, but needs to write back

Soft vs. Hard Page Faults

Hard:

- requires reading the page from disk
- expensive

Soft:

- Page already in memory, but OS need to do some work, cheaper
- E.g., COW – forked child modifies a page

Swapping to Remote Memory

Via RDMA networks which are very fast

Doesn’t require interrupting CPU on the receiver end

END RECAP

SWAPPING POLICIES

SWAPPING Policies

Goal: Minimize number of page faults

- Page faults require milliseconds to handle (reading from disk)
- Implication: Plenty of time for OS to make good decision

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 **thrown out** to disk?

Page Selection

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)
- Two costs of bad prefetching?

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, ...

Hints with madvise()

int madvise(void *addr, size_t length, int advice) – allows user program to give hints to OS about how page can be prefetched

MADV_SEQUENTIAL

MADV_WILLNEED

MADV_RANDOM

Many more on linux…

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

Page Replacement

Which page in main memory should selected as victim?

- Write out victim page to disk if modified (dirty bit set)
- If victim page is not modified (clean), just discard

OPT: Replace page not used for longest time in future

- Advantages: Guaranteed to minimize number of page faults
- Disadvantages: Requires that OS predict the future; Not practical, but good for comparison

Page Replacement

FIFO: Replace page that has been in memory the longest

- Intuition: First referenced long time ago, done with it now
- Advantages: Fair: All pages receive equal residency; Easy to implement
- Disadvantage: Some pages may always be needed

LRU: Least-recently-used: Replace page not used for longest time in past

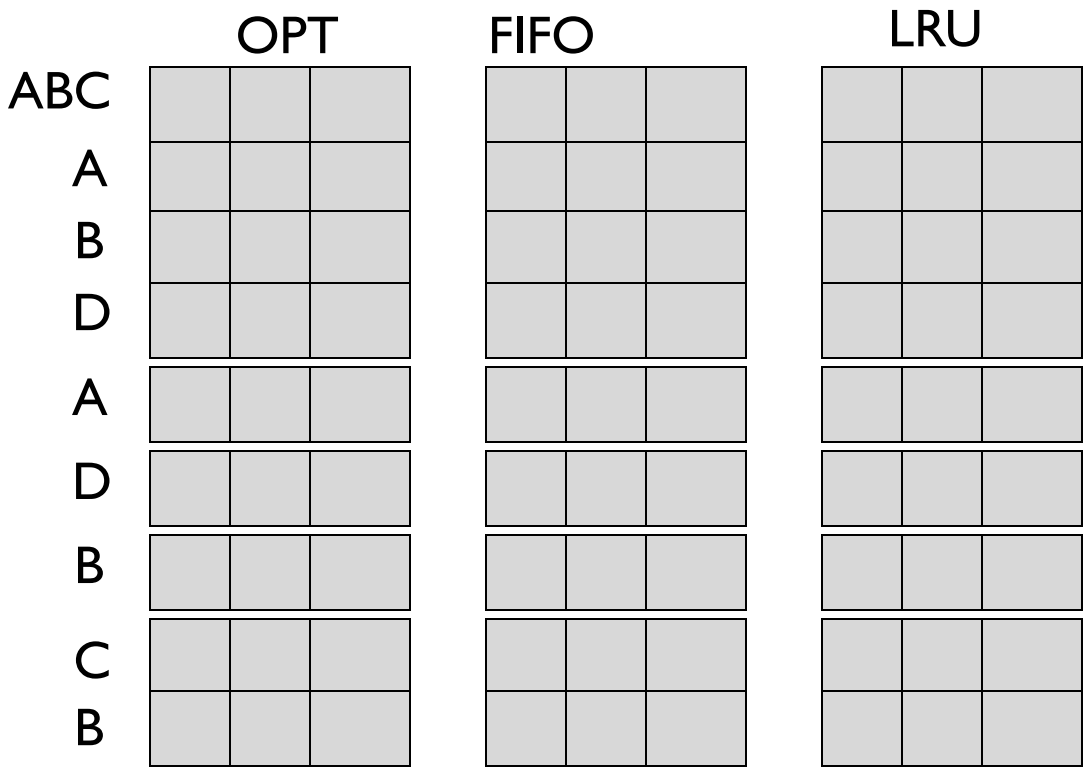
- Intuition: Use past to predict the future
- Advantages: With locality, LRU approximates OPT
- Disadvantages:
  - Harder to implement, must track which pages have been accessed

Page Replacement - chat for 2 mins

Page reference string: ABCABDADBCB

Metric:
Miss count

Three pages of physical memory



Page Replacement

Page reference string: ABCABDADBCB

Metric:
Miss count

Three pages of physical memory

	OPT			FIFO			LRU		
ABC	A	B	C	A	B	C	A	B	C
A	A	B	C	A	B	C	A	B	C
B	A	B	C	A	B	C	A	B	C
D	A	B	D	D	B	C	A	B	D
A	A	B	D	D	A	C	A	B	D
D	A	B	D	D	A	C	A	B	D
B	A	B	D	D	A	B	A	B	D
C	C	B	D	C	A	B	C	B	D
B	C	B	D	C	A	B	C	B	D

LRU vs. OPT

Think of a workload/setting where LRU will be way worse than OPT…

Page Replacement Comparison

Add more physical memory, what happens to performance?

LRU, OPT:

- Guaranteed to have fewer (or same number of) page faults
- Smaller memory sizes are guaranteed to contain a subset of larger memory sizes
- Stack property: smaller cache always subset of bigger

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 Perfect LRU (conceptually)

Software Perfect LRU

- OS maintains ordered list of physical pages by reference time
- When page is referenced: Move page to front of list
- When need victim: Pick page at back of list
- 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)

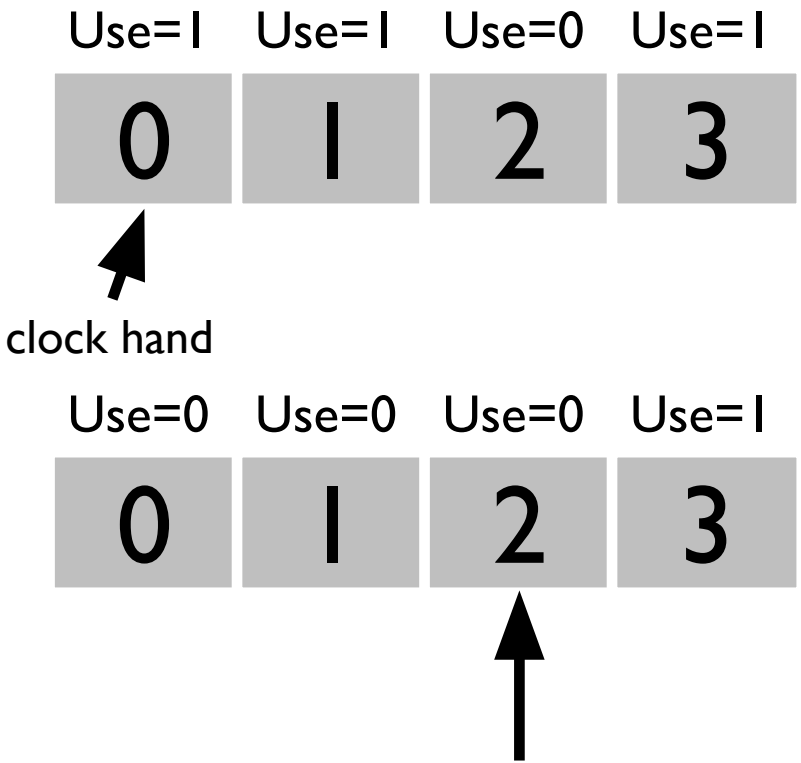
In practice, do not implement Perfect LRU

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

Clock Algorithm

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

CLOCK: LOOK FOR A PAGE



Evict contents of frame 2, load in new page

Clock Algorithm - Who does what - HW or OS

- Keep use (or reference) bit for each page frame
- When page is referenced: set use bit
- Page replacement: Look for page with use bit cleared (has not been referenced for awhile)
  - 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

Clock Extensions

Replace multiple pages at once

- Intuition: Expensive to run replacement algorithm and to write single block to disk
- Find multiple victims each time and track free list

Use dirty bit to give preference to 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 ("change")

Intuition: Better ability to differentiate across pages (how much they are being accessed)
Increment software counter if use bit is 0
Replace when chance exceeds some specified limit

Linux 2Q

Active list and inactive list

On first reference, put in inactive list, if accessed again, move to active list

Otherwise, it will be swapped to disk (uses FIFO to make the decision)

Active list is very similar to a clock

Move pages that not referenced to the inactive list until active list is 2/3rd of physical memory size

SUMMARY: VIRTUAL MEMORY

Abstraction: Virtual address space with code, heap, stack

Address translation

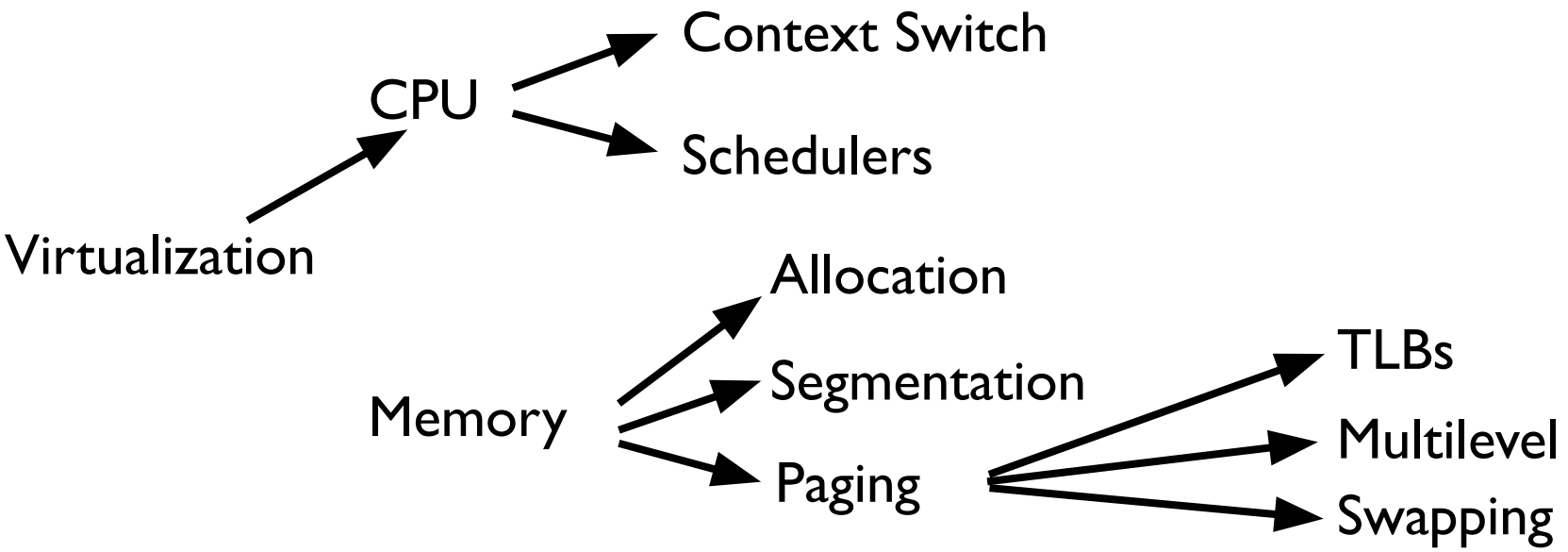
- Contiguous memory: base, bounds, segmentation
- Using fixed sizes pages with page tables

Challenges with paging

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

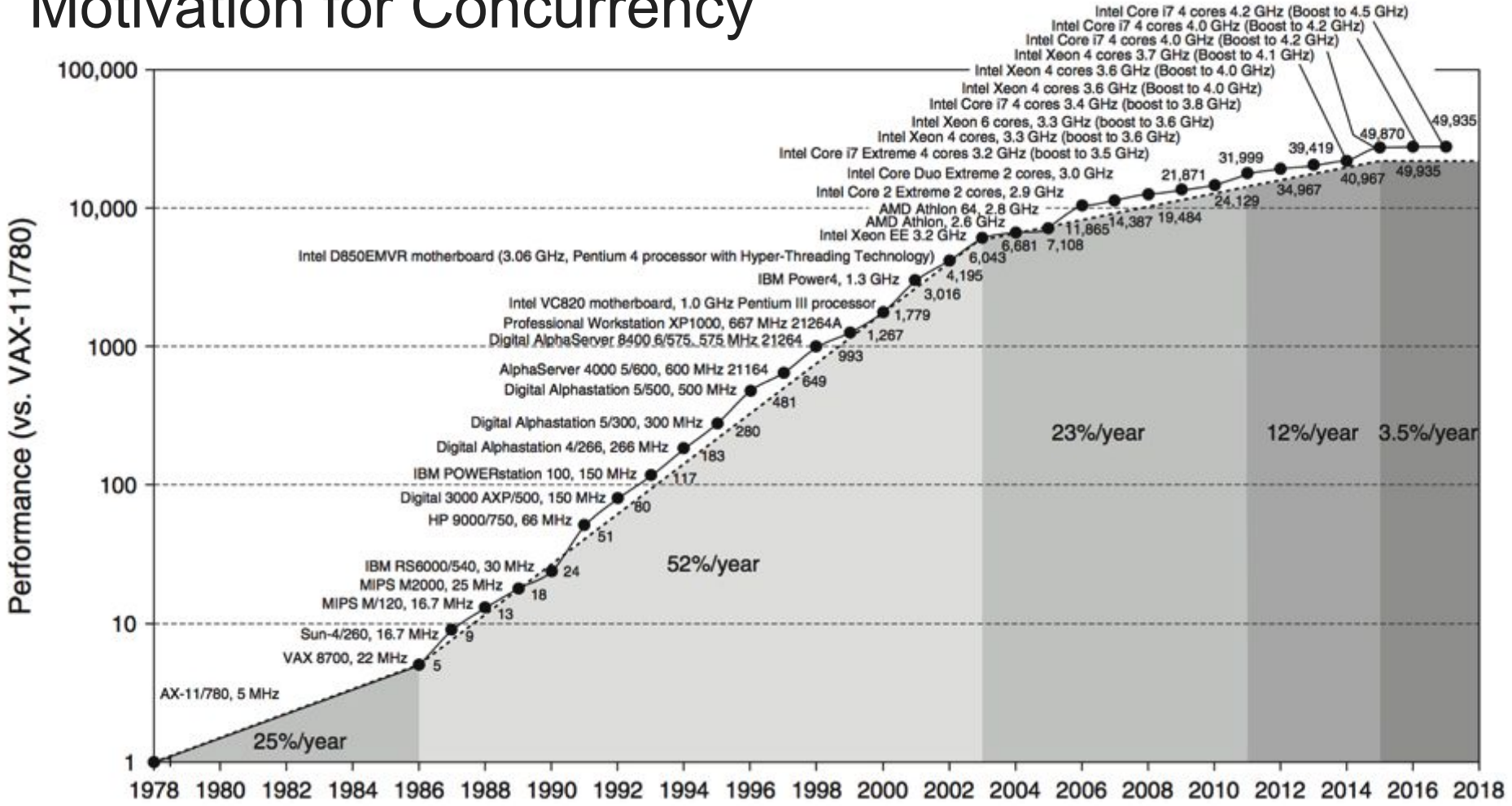
Larger address spaces: Swapping mechanisms, policies (LRU, Clock)

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 (why expensive?)

CONCURRENCY: OPTION 2

New abstraction: thread

Threads are like processes, except:

multiple threads of same process share an address space

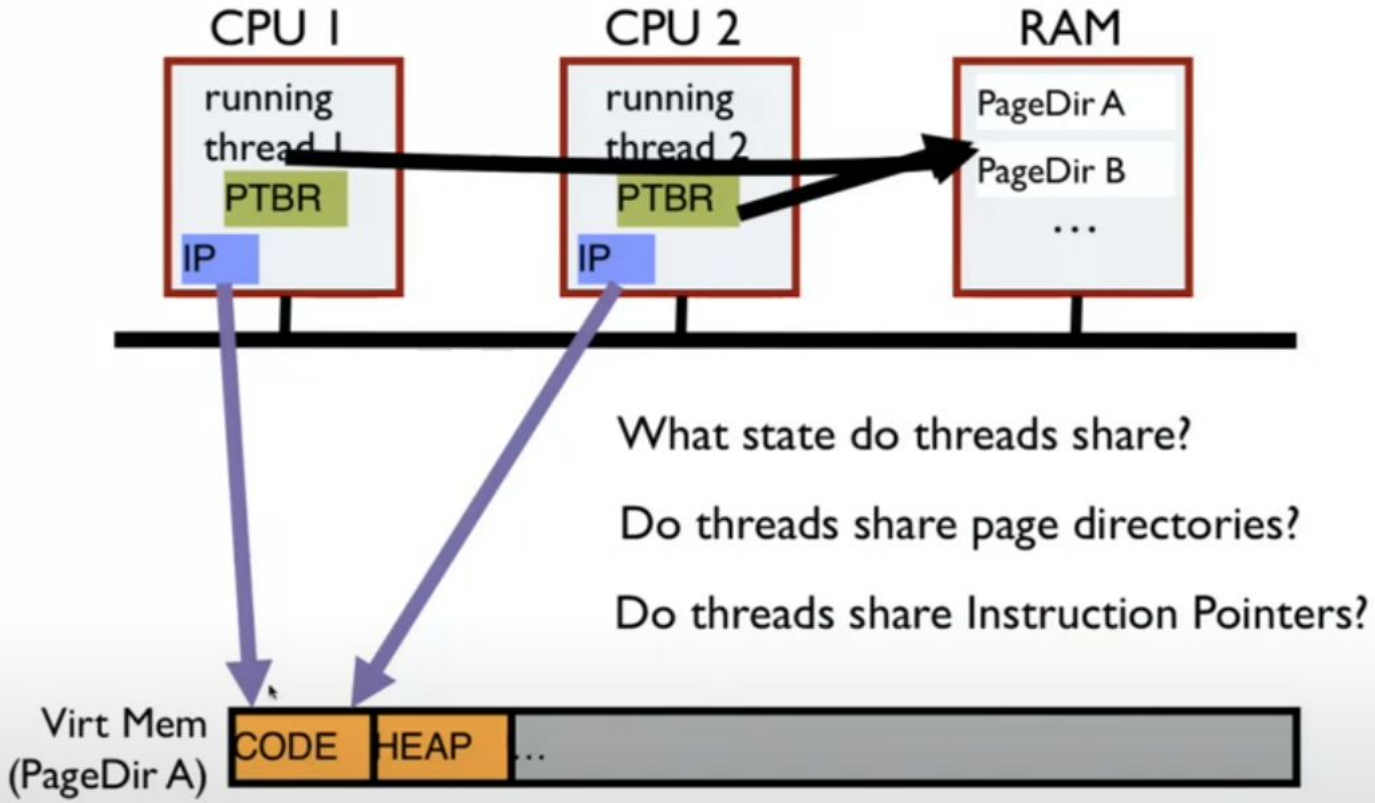
Divide large task across several cooperative threads

Communicate through shared address space

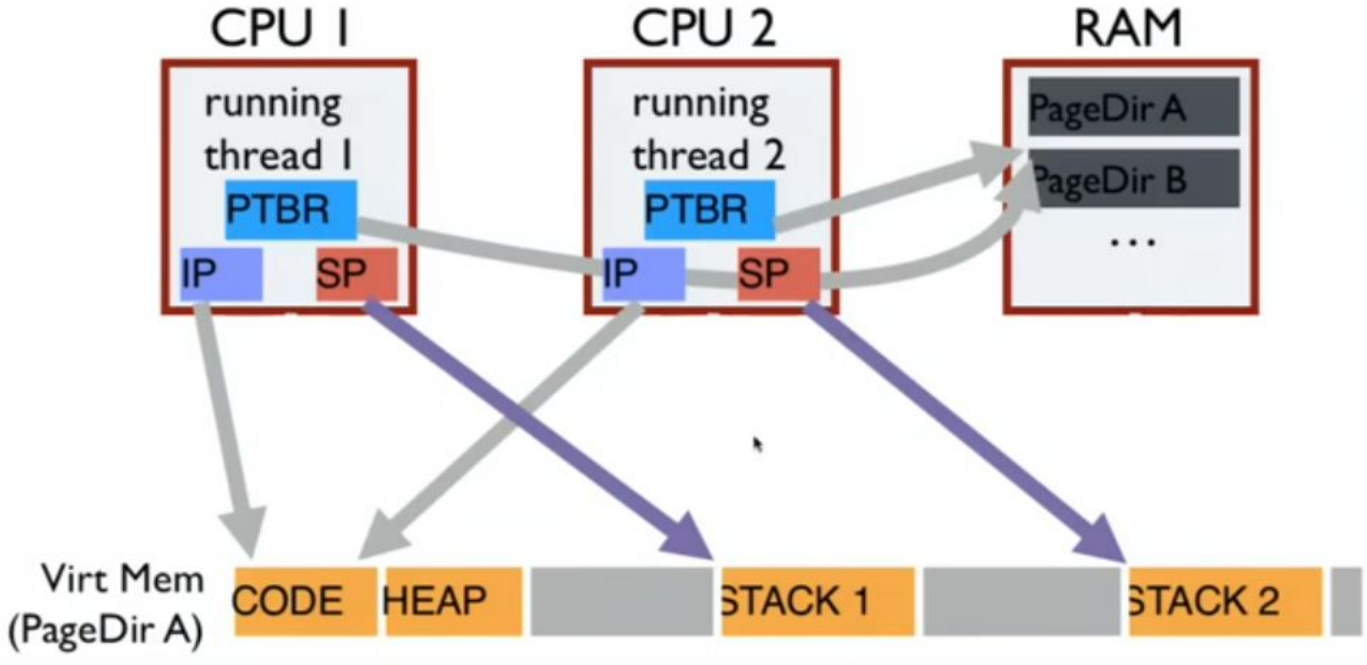
Common Programming Models

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:

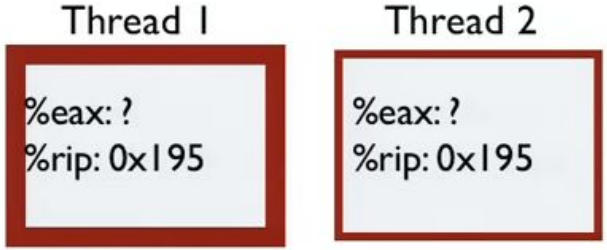
Thread 1	Thread 2
%eax: ? %rip: 0x195	%eax: ? %rip: 0x195


```
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:

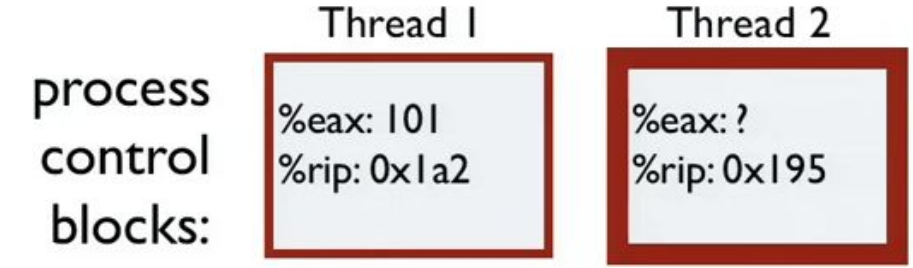



```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

T1

THREAD SCHEDULE #1

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



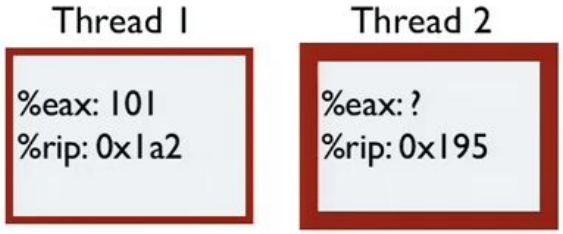
T2

```
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

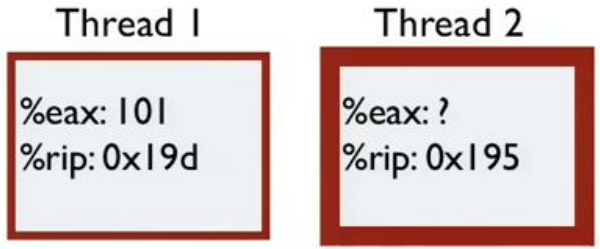

```
        0x195   mov 0x9cd4, %eax
        0x19a   add $0x1, %eax
T1      0x19d   mov %eax, 0x9cd4A
```

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

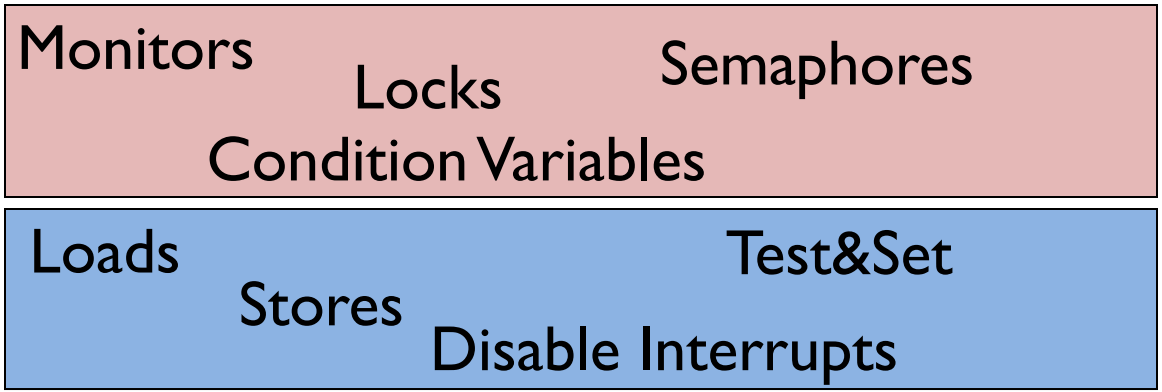
if thread A is in critical section C, thread B isn't

(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!