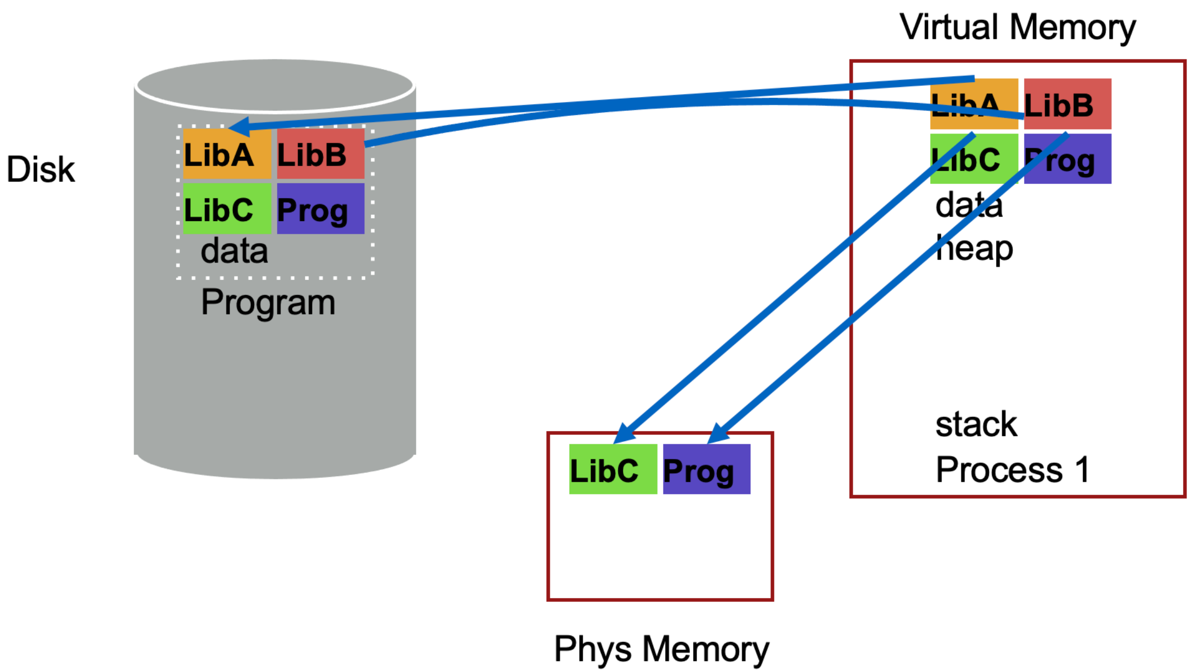
Virtual Memory



Page Selection
Page Replacement

1,2,3,1,2,4,1,4,2,3,2

Miss: 1,2,3			
	1	2	3
Hit: 1	1	2	3
Hit: 2	1	2	3
Miss:4, Replace:3	1	2	4
Hit: 1	1	2	4
Hit: 4	1	2	4
Hit: 2	1	2	4
Miss:3, Replace:1	2	4	3
Hit: 2	2	4	3

$$AMAT = (T_m) + (Miss\% * T_d)$$

OPT
FIFO
LRU

Page Replacement Comparison

Add more physical memory, what happens to performance?

- LRU, OPT: Add more memory, 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 a subset of bigger cache

- FIFO: Add more memory, usually have fewer page faults
  - Belady’s anomaly: but there are cases where we have more page faults!

Consider access stream: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Consider physical memory size: 3 pages vs. 4 pages

How many misses with FIFO?

3 pages: 9 misses
4 pages: 10 misses

Problems with LRU-based Replacement

LRU does not consider frequency of accesses

- Is a page accessed **once** in the past equal to one accessed **N** times?
- Common workload problem:
  - Scan (sequential read, never used again) one large data region flushes memory

Solution: Track frequency of accesses to page

Pure LFU (Least-frequently-used) replacement

- Problem: LFU can never forget pages from the far past

Implementing LRU

Perfect LRU on Software

- 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

Perfect LRU on Hardware

- Associate timestamp with each page (e.g., PTE)
- When page is referenced: Associate current system timestamp with page
- When need victim: Scan through PTEs 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 more
- Goal: Find an old page, but not necessarily the oldest

Clock Algorithm

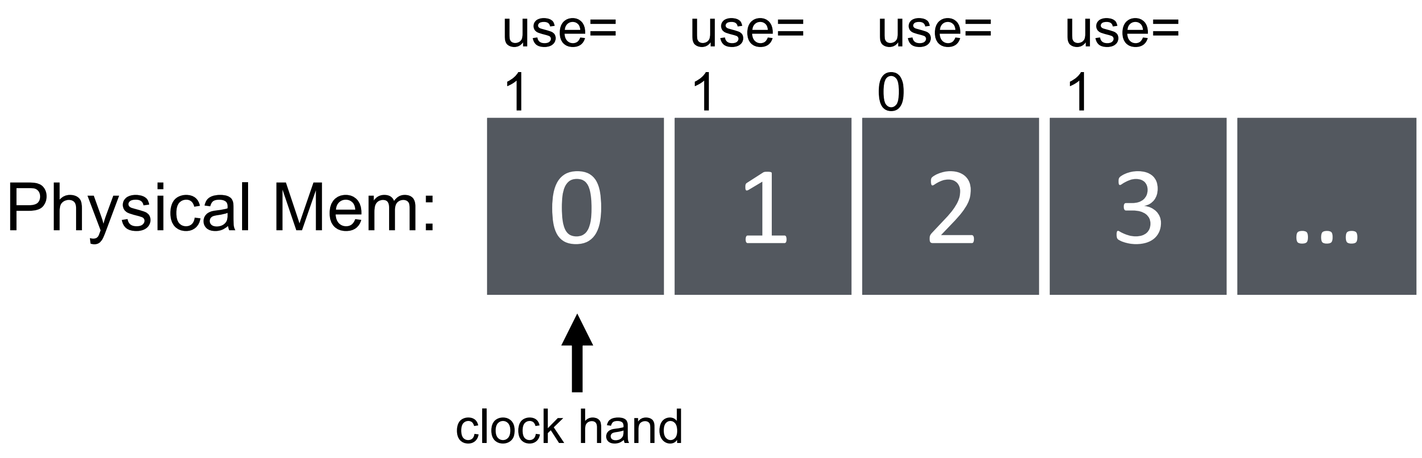
Hardware

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

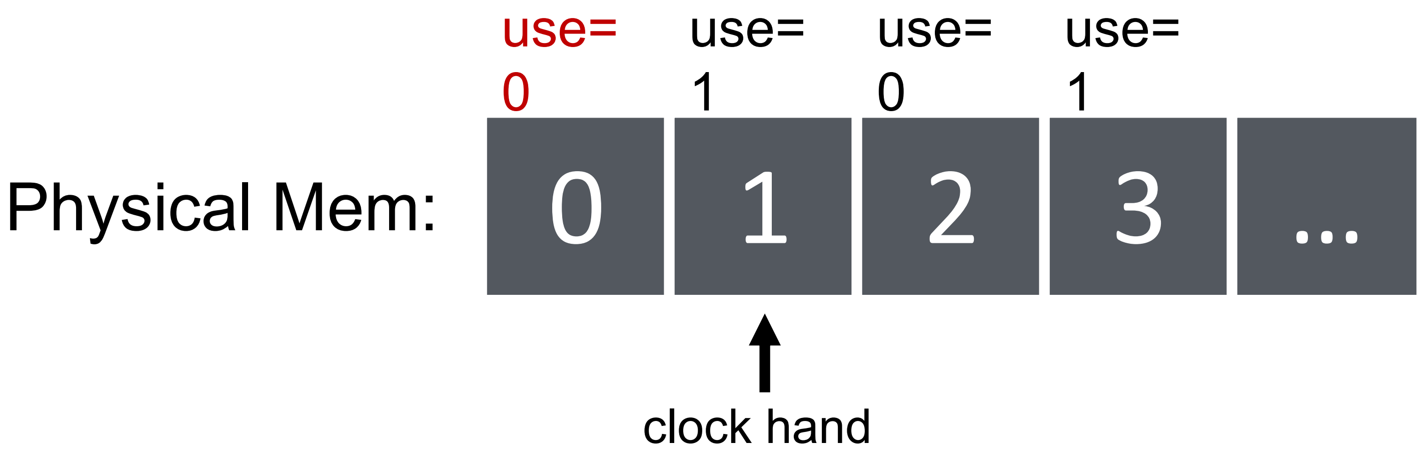
Operating System

- Page replacement: Look for page with `use` bit cleared (has not been referenced for a while)
- Implementation:
  - Keep pointer to last examined page frame (“clock hand”)
  - Traverse pages in circular fashion (like a clock)
  - Clear `use` bits as you search
  - Stop when find page with already cleared `use` bit, replace this page

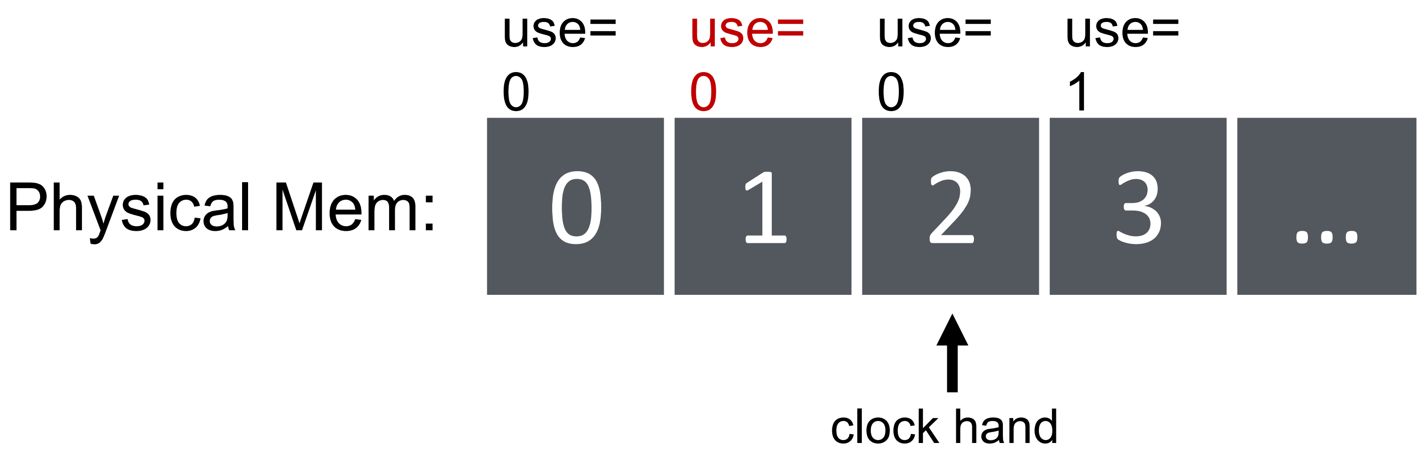
Clock:
Look For a Page



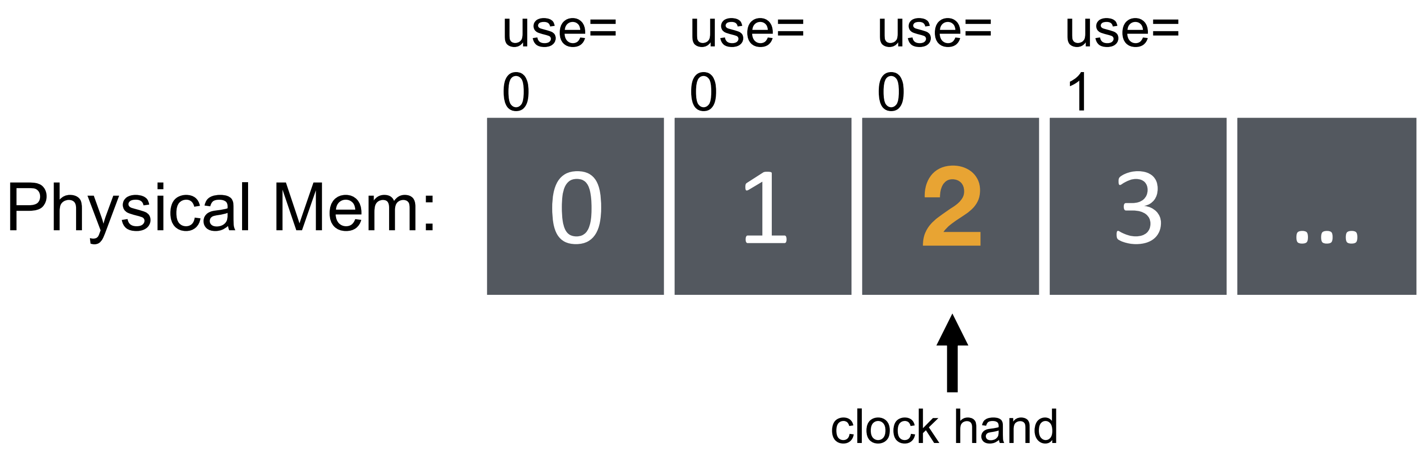
Clock:
Look For a Page



Clock:
Look For a Page

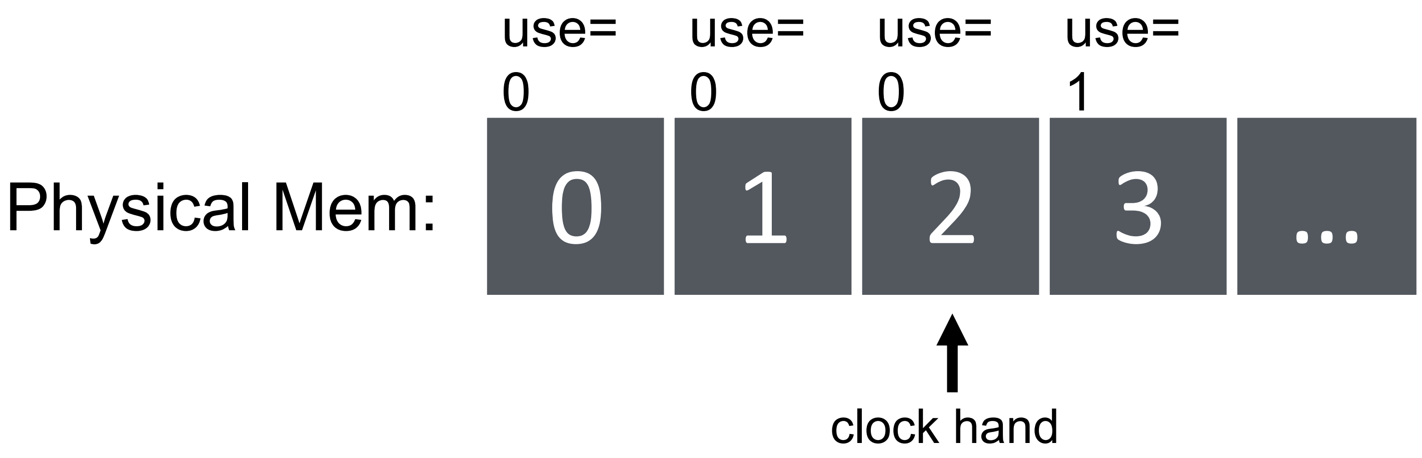


Clock:
Look For a Page



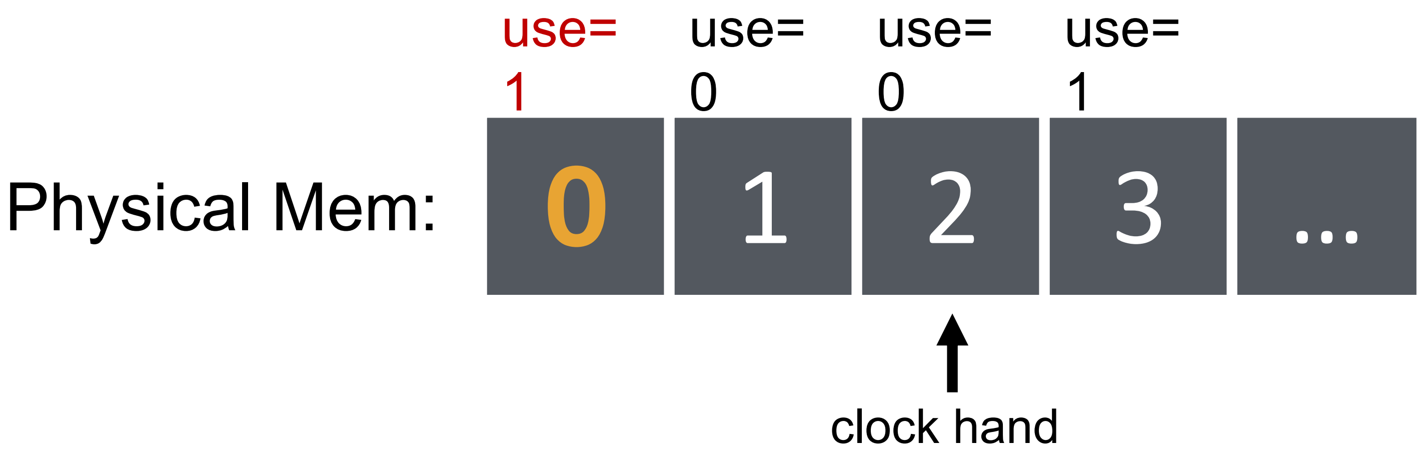
evict **page 2** because it has not been recently used

Clock:
Look For a Page

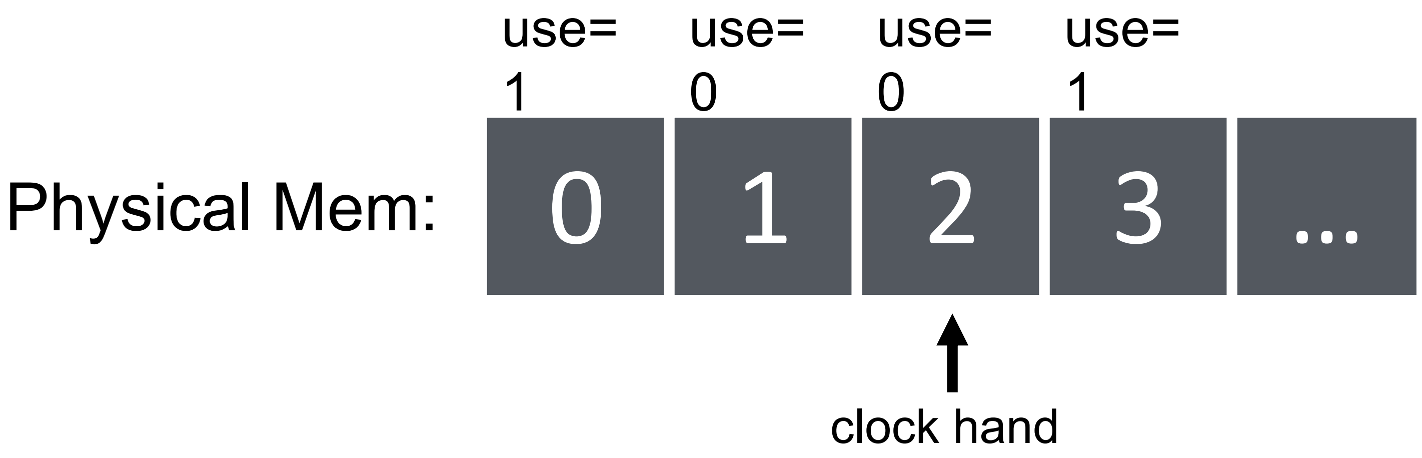


page 0 is accessed…

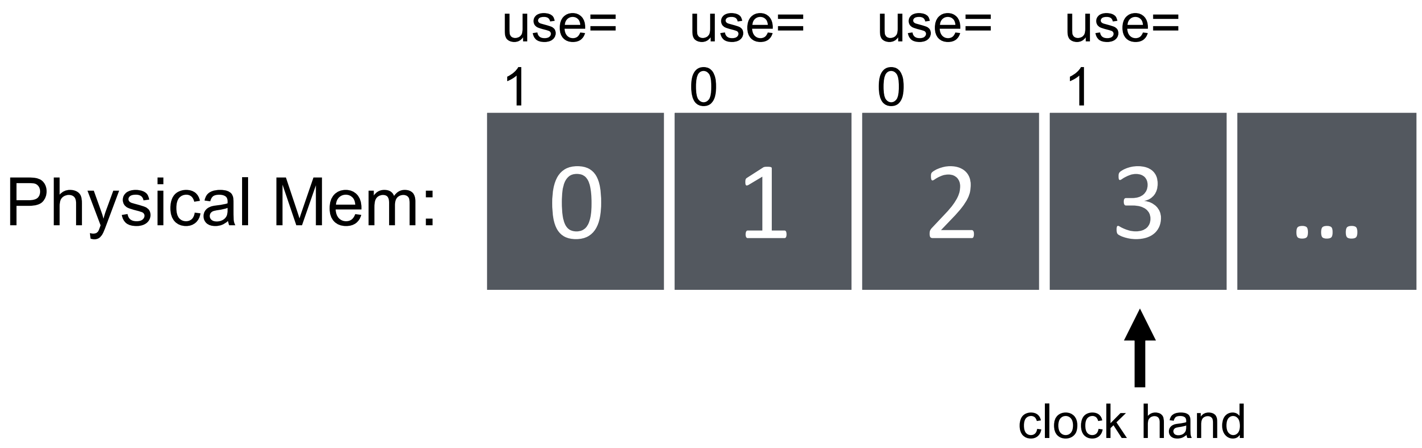
Clock:
Look For a Page



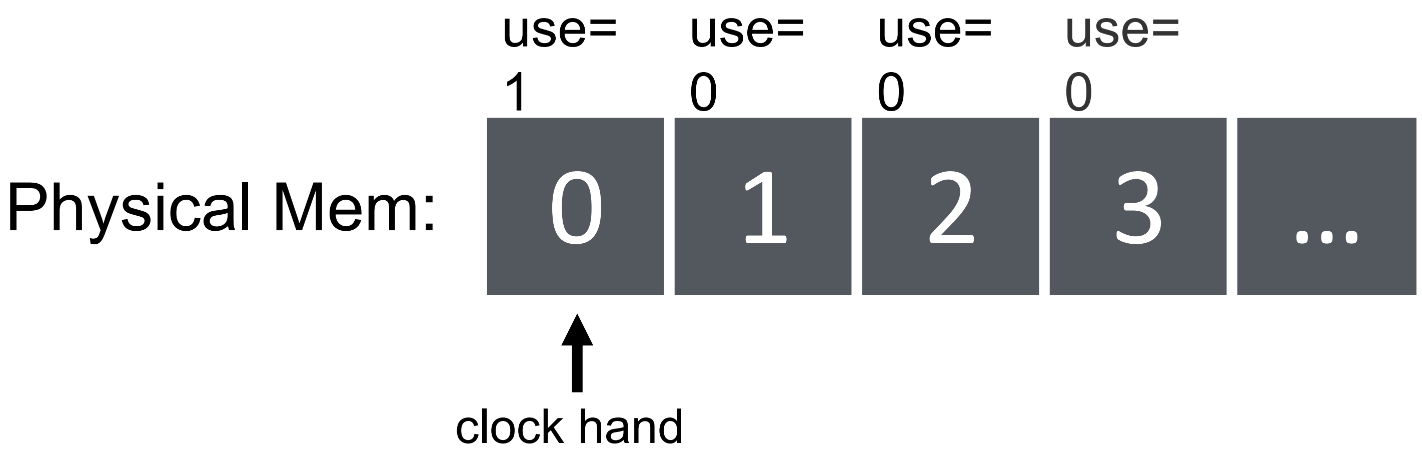
Clock:
Look For a Page



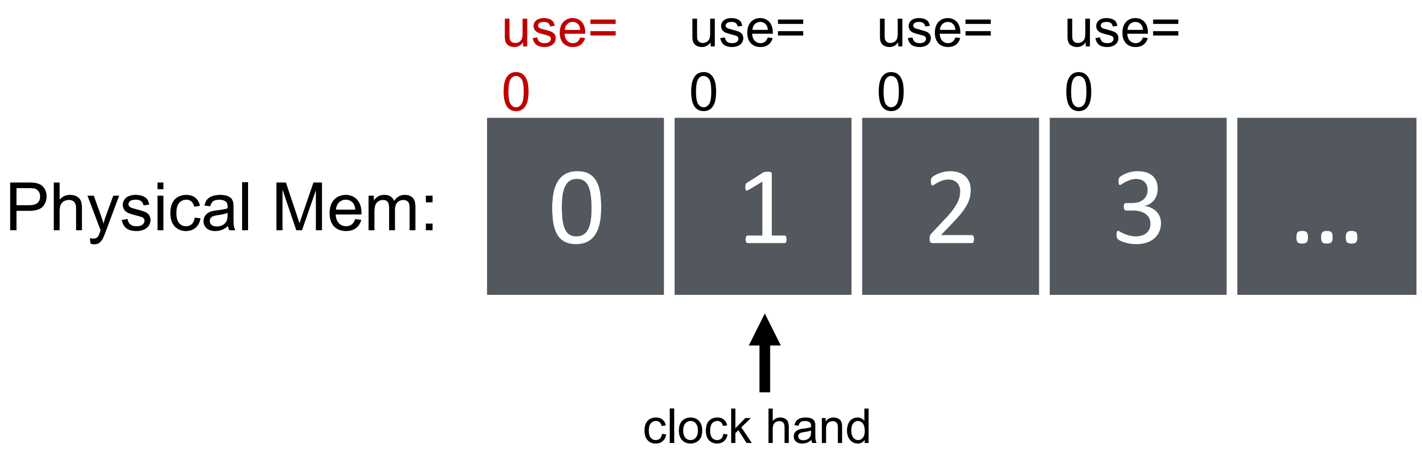
Clock:
Look For a Page



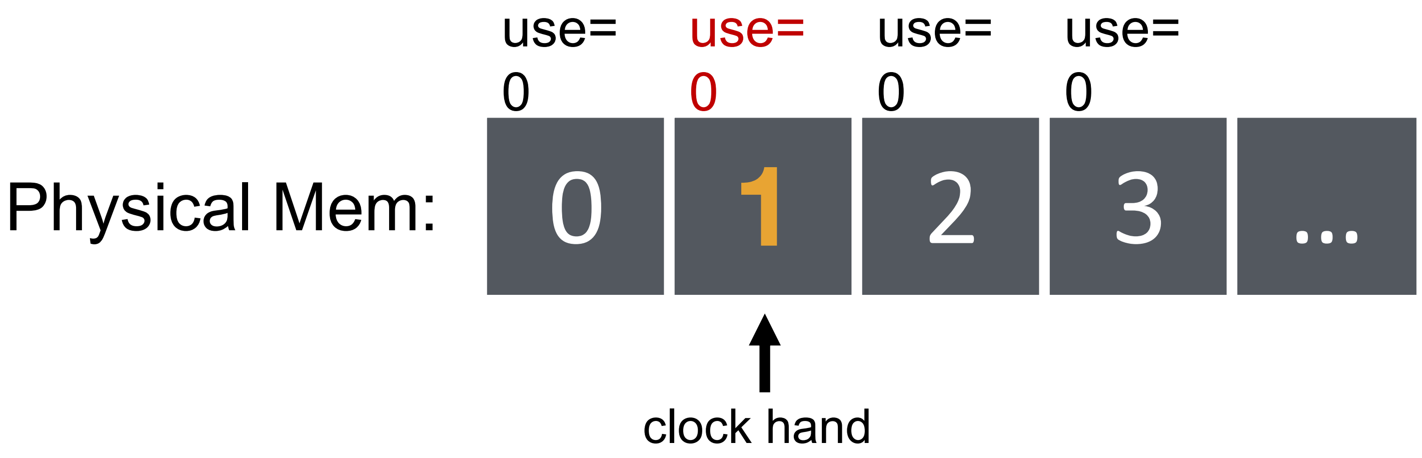
Clock:
Look For a Page



Clock:
Look For a Page



Clock:
Look For a Page



evict **page 1** because it has not been recently used

Clock Extensions

Use modified ("dirty") bit to prefer to retain modified pages in memory

- Intuition: More expensive to replace dirty pages
  - Modified pages must be written to disk, clean pages do not have to be
- First replace pages that have `use` bit and `modified` bit cleared

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

Add software counter ("chance") to track use frequency

- Intuition: Want to differentiate pages by how much they are accessed
- Increment software counter if `use` bit is 0
- Replace when chance exceeds some specified limit

What if no hardware support?

What can the OS do if hardware does not have `use` bit (or `dirty` bit)?

- Can the OS “emulate” these bits?

Think about this question:

- Can the OS get control (i.e., generate a trap) every time `use` bit should be set? (i.e., when the page is accessed?)

Conclusion

Illusion of virtual memory: Processes can run when the sum of virtual address spaces is larger than physical memory

Mechanism:

- Extend page table entry with “present” bit
- OS handles page faults (or page misses) by reading in the desired page from disk

Policy:

- Page selection – demand paging, prefetching, hints
- Page replacement – OPT, FIFO, LRU, others

Implementations (clock) approximate LRU

Concurrency

Concurrency

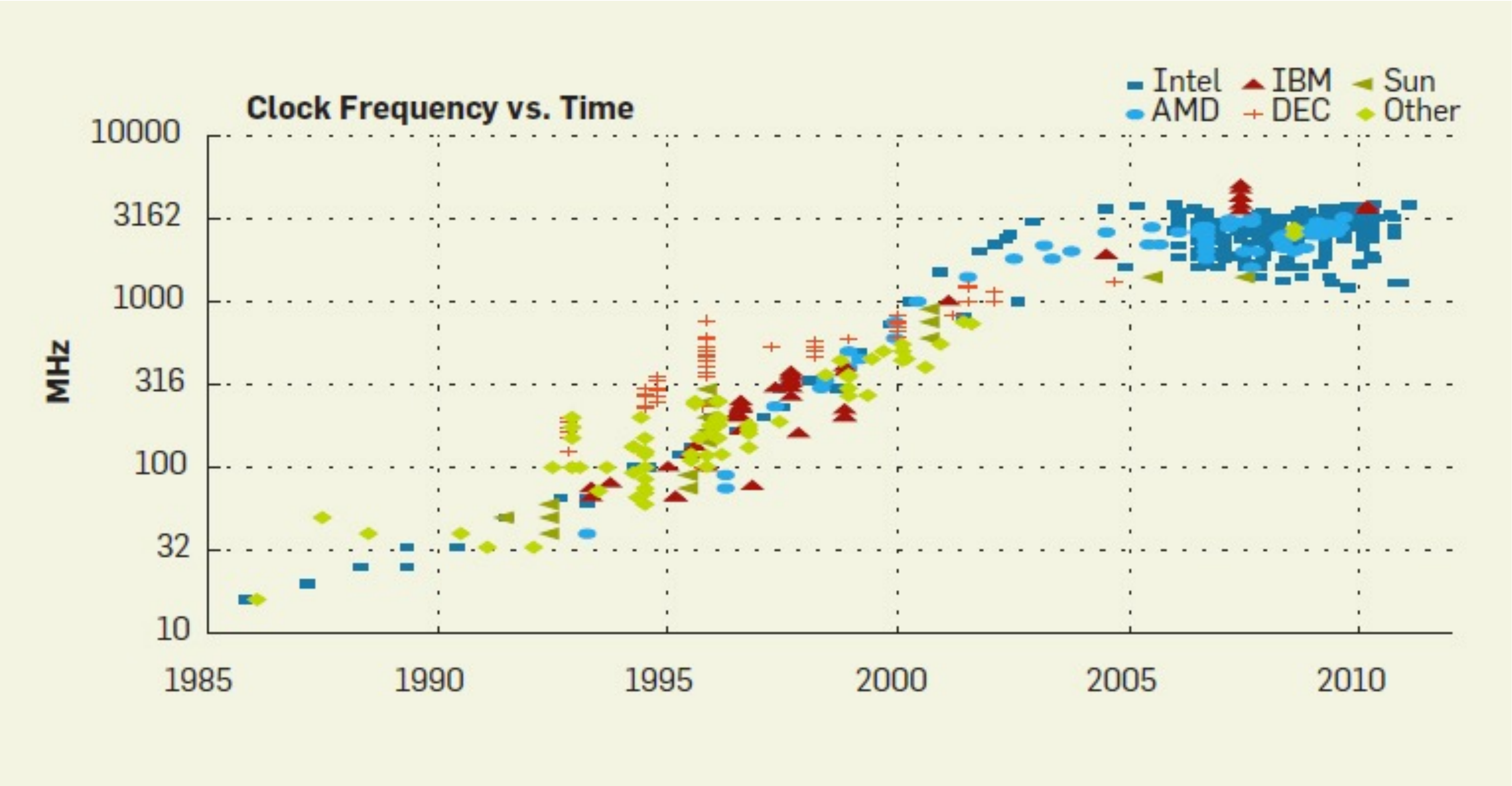
- **Questions answered:**
- Why is concurrency useful?
- What is a thread and how does it differ from processes?
- What can go wrong if scheduling of critical sections is not atomic?

Motivation for concurrency: Blocking

- Operations proceeding at the same time: blocking for I/O, while doing other useful work

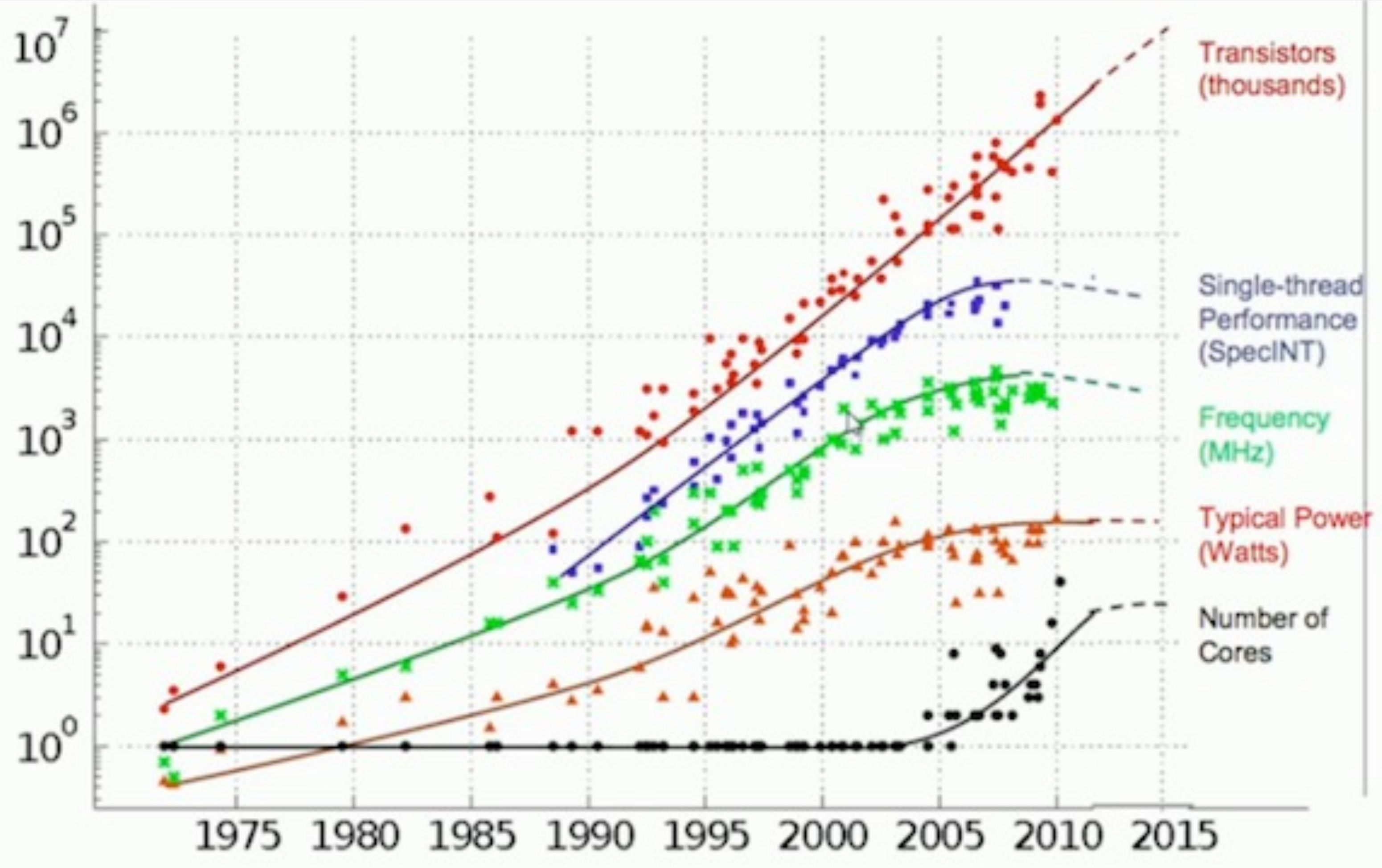
- Example: web server
  - Serve the first request by reading a file from disk
  - Serve a second request by running computation

Motivation for Concurrency: Parallelism



http://cacm.acm.org/magazines/2012/4/147359-cpu-db-recording-microprocessor-history/fulltext

Motivation for Concurrency: Parallelism



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

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

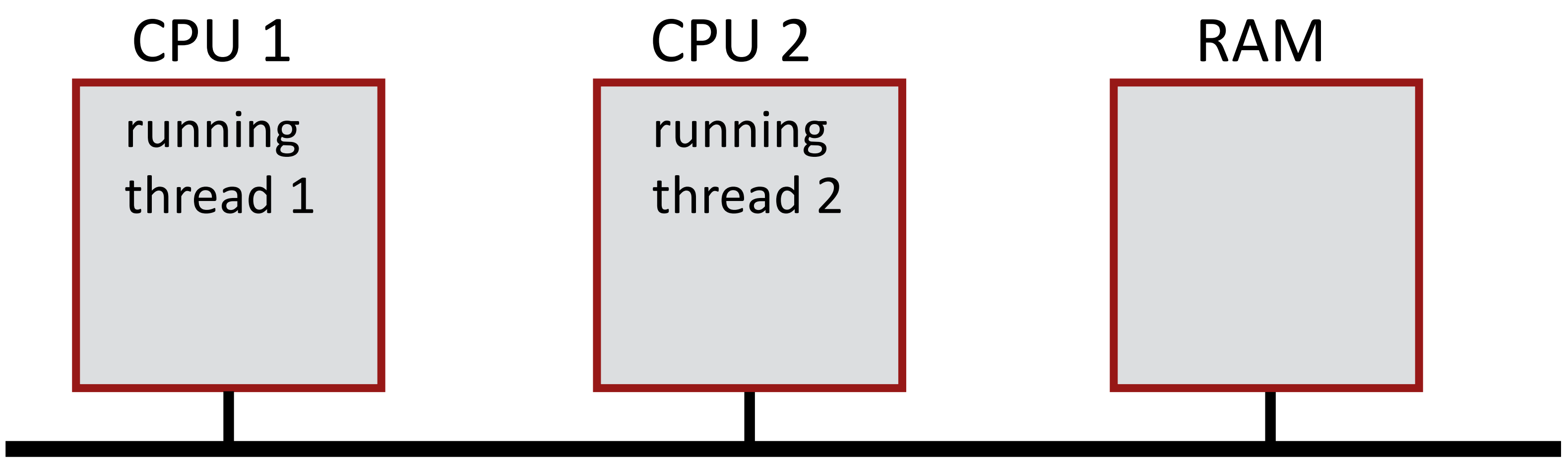
Divide large task across several cooperative threads

Communicate through shared address space

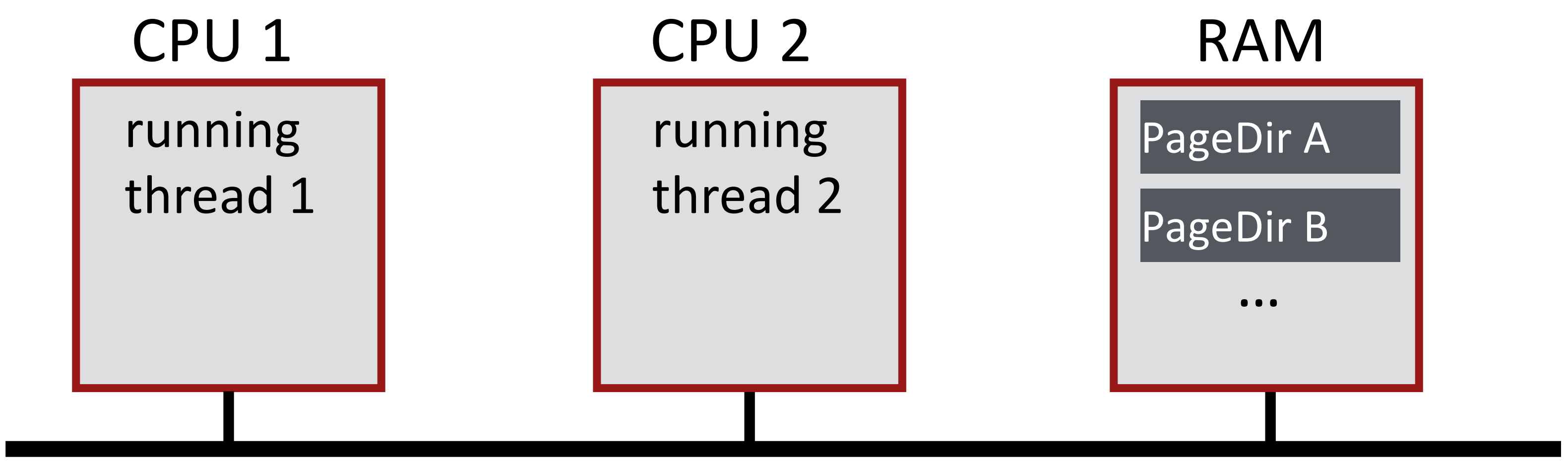
Common Programming Models

Multi-threaded programs tend to be structured as:

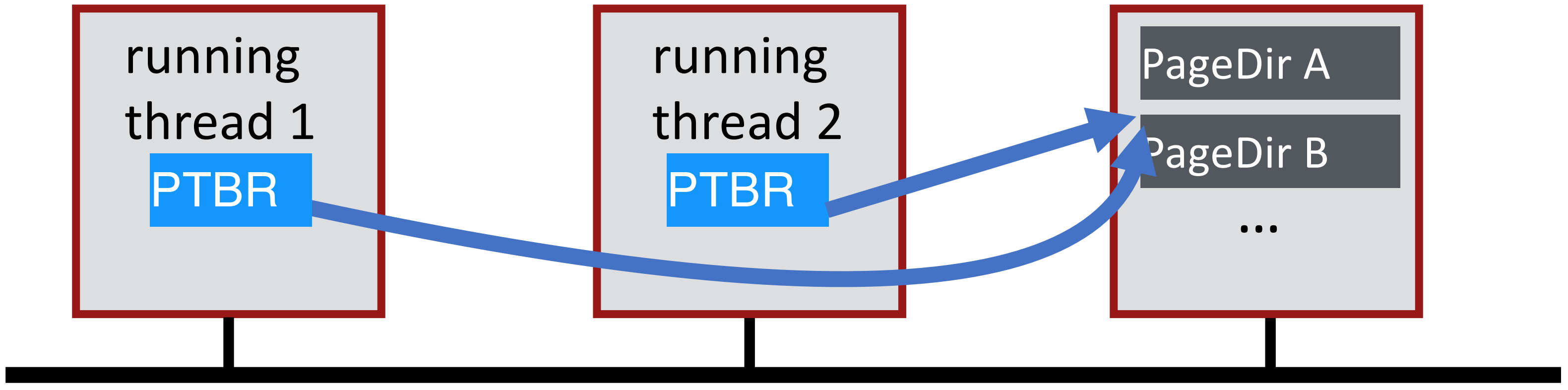
- **Producer/consumer**
  Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
- **Pipeline**
  Task is divided into series of subtasks, each of which is handled in series by a different thread
- **Defer work with background thread**
  One thread performs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

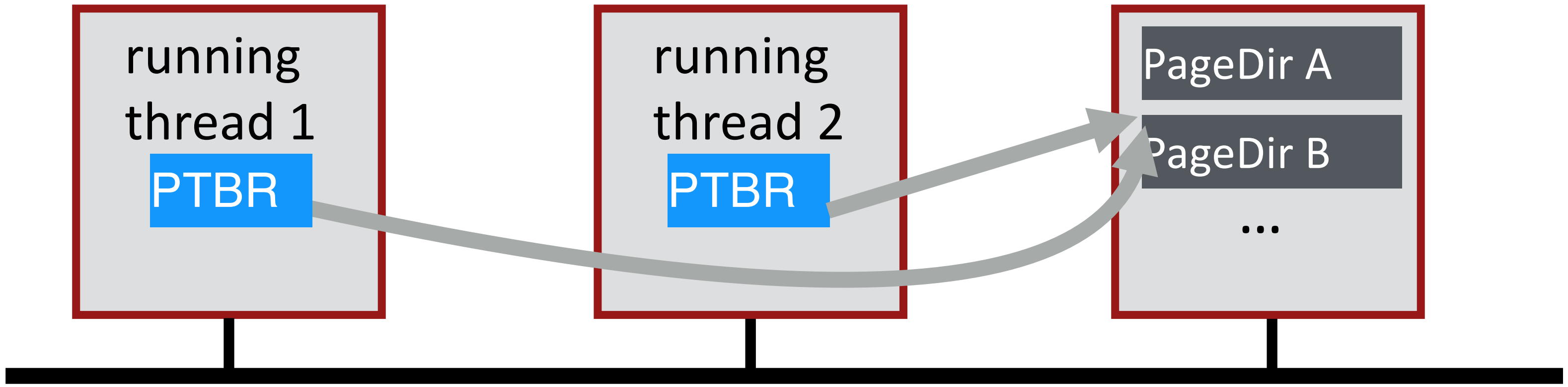


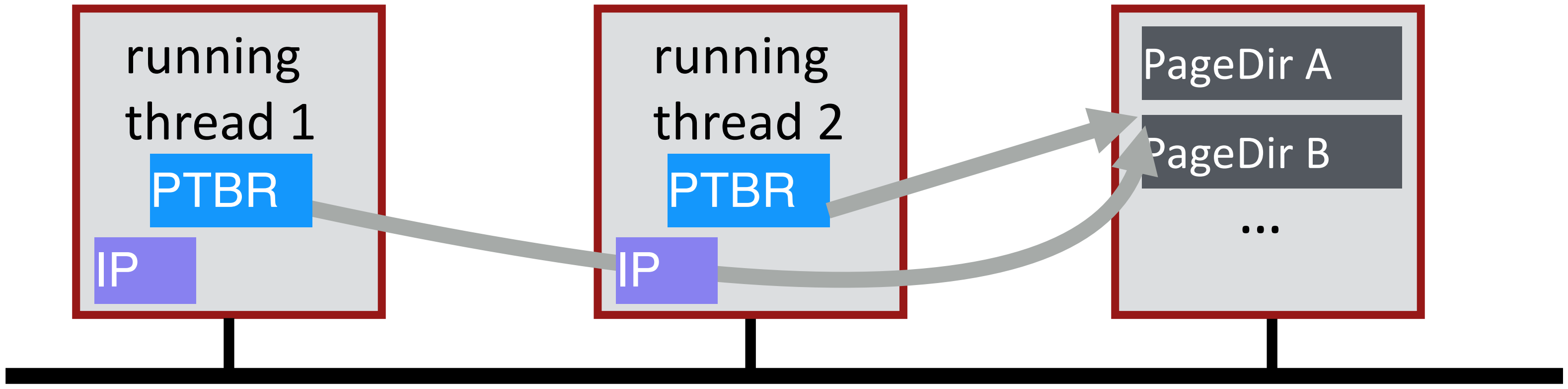
What state do threads share?



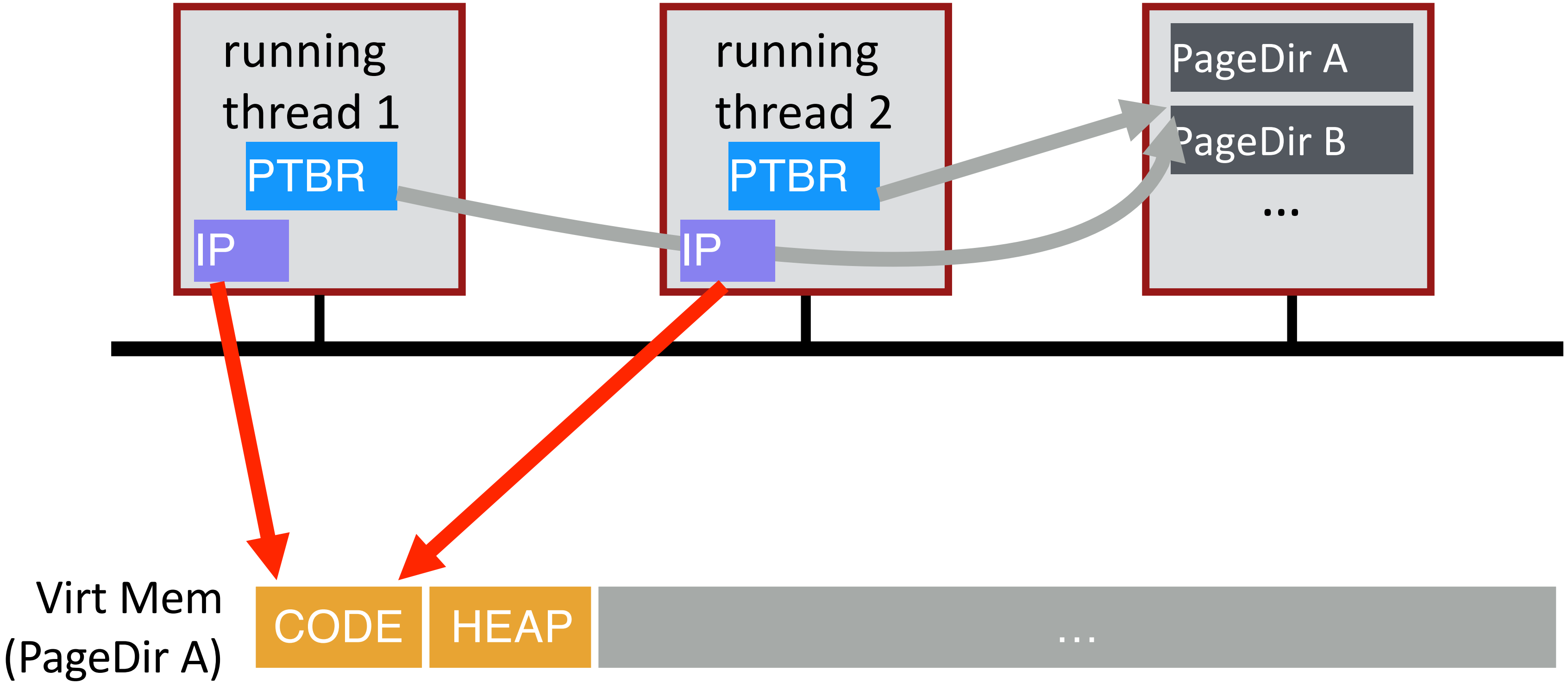
What threads share page directories?

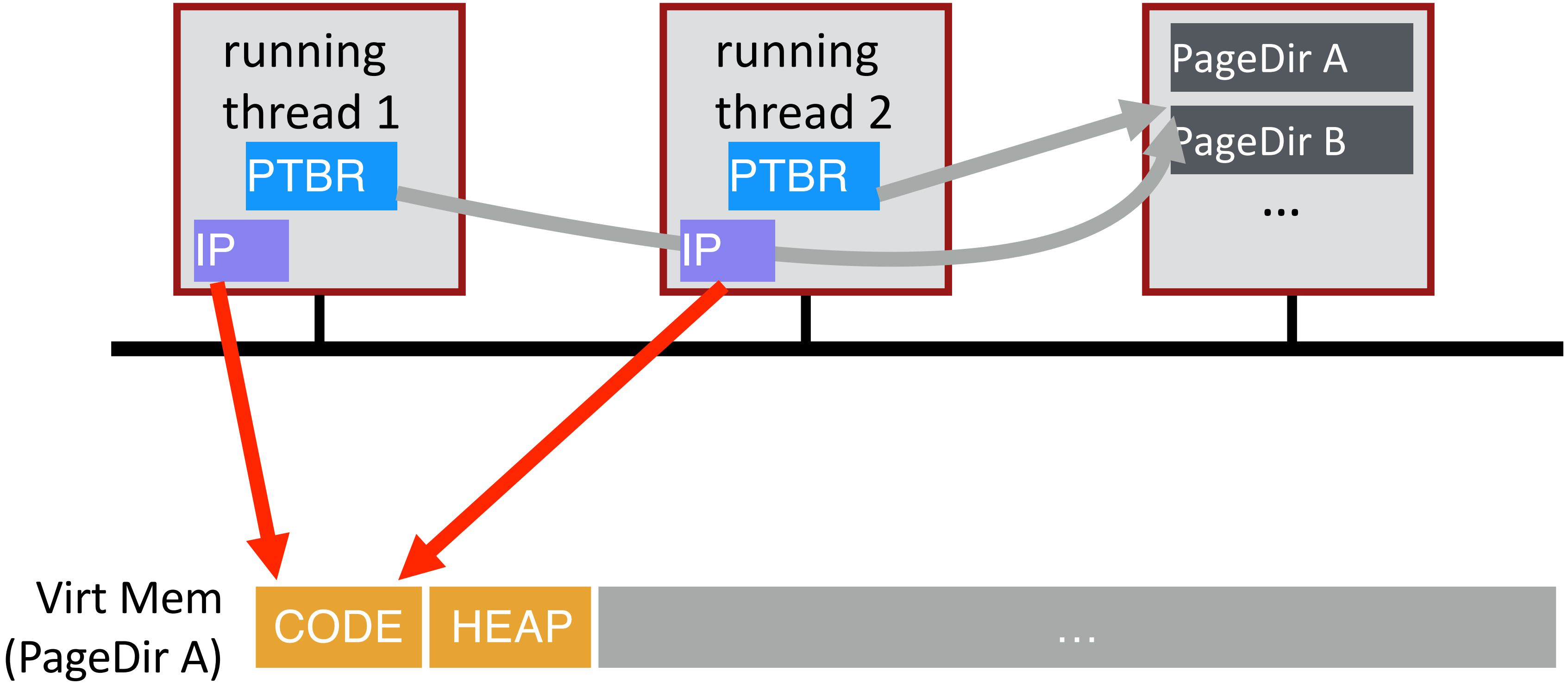

running thread 1
PTBR
running thread 2
PTBR
PageDir A
PageDir B
...


running thread 1
PTBR
running thread 2
PTBR
PageDir A
PageDir B
...



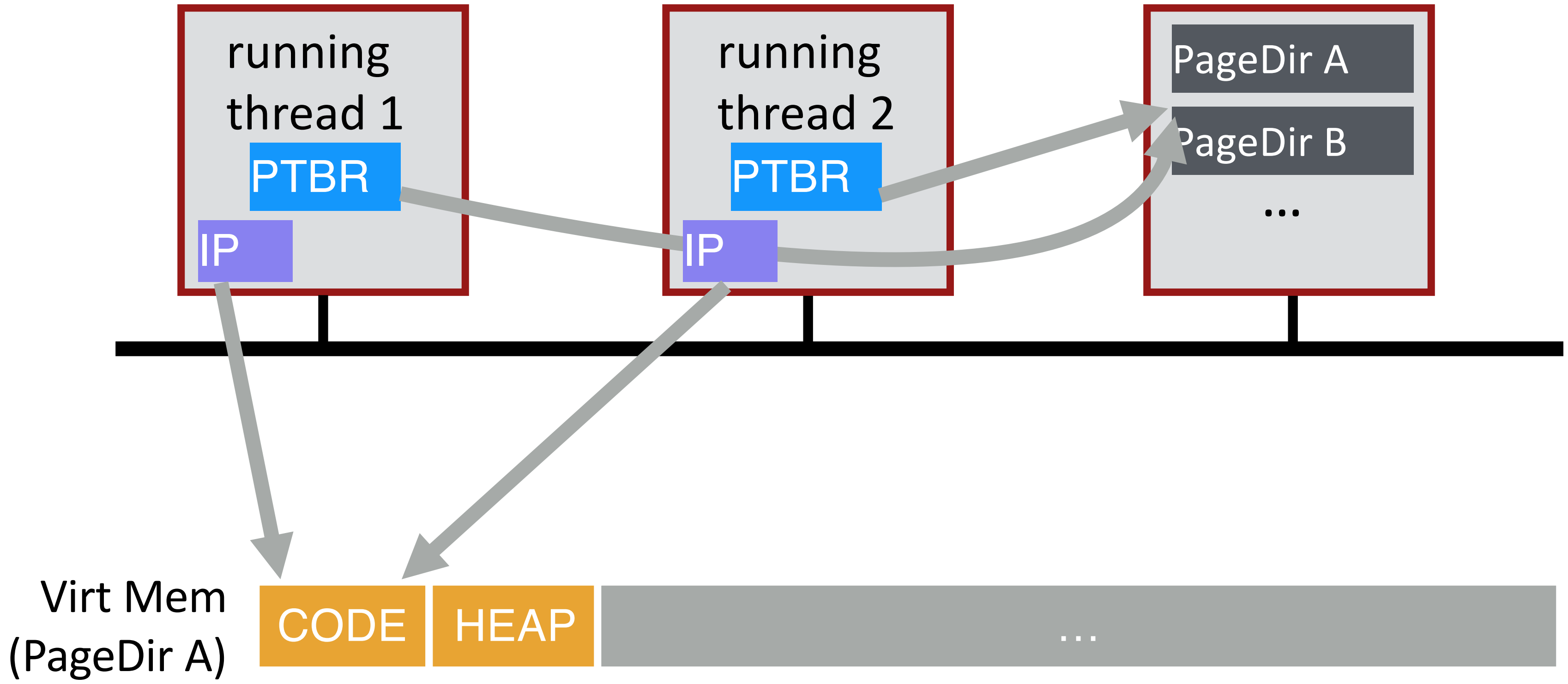
Do threads share Instruction Pointer?

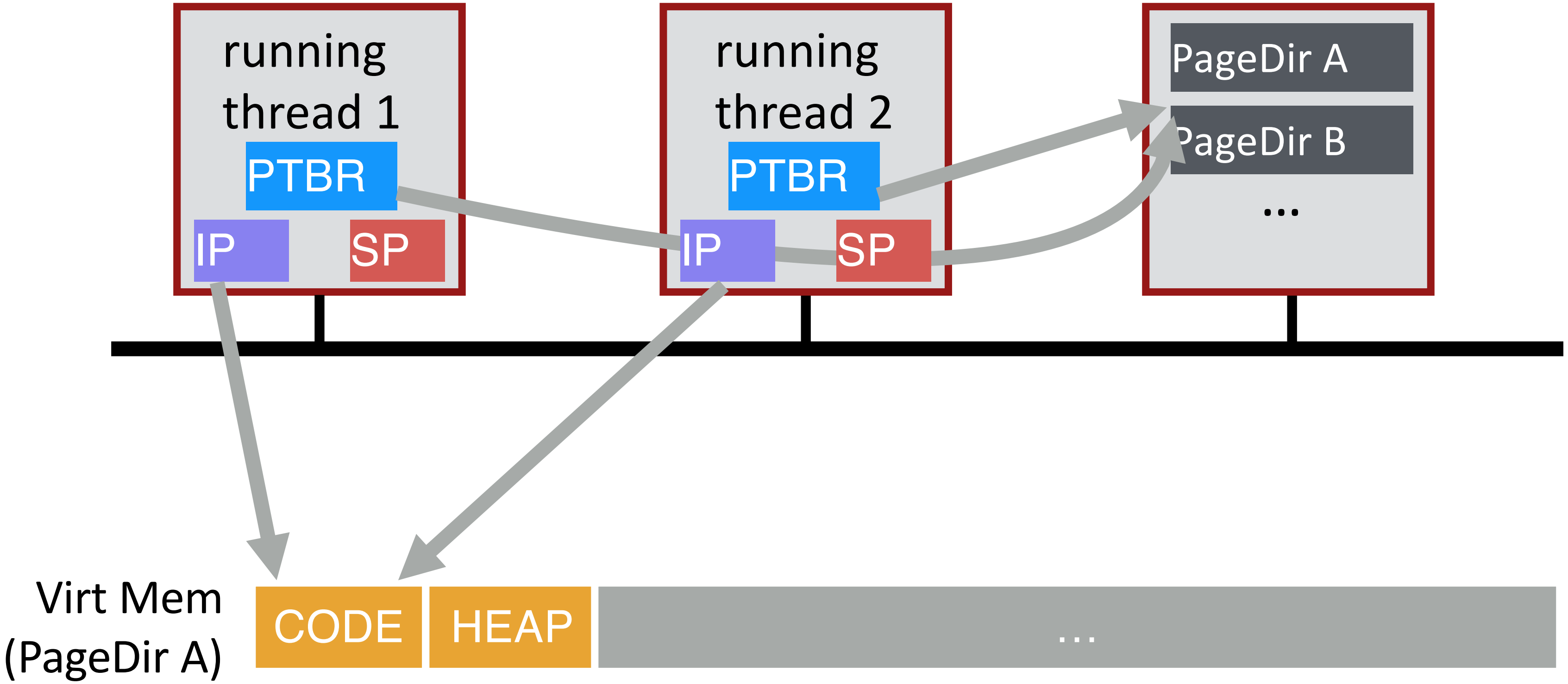

running
thread 1
PTBR
IP
running
thread 2
PTBR
IP
PageDir A
PageDir B
...
Virt Mem
(PageDir A)
CODE
HEAP
...



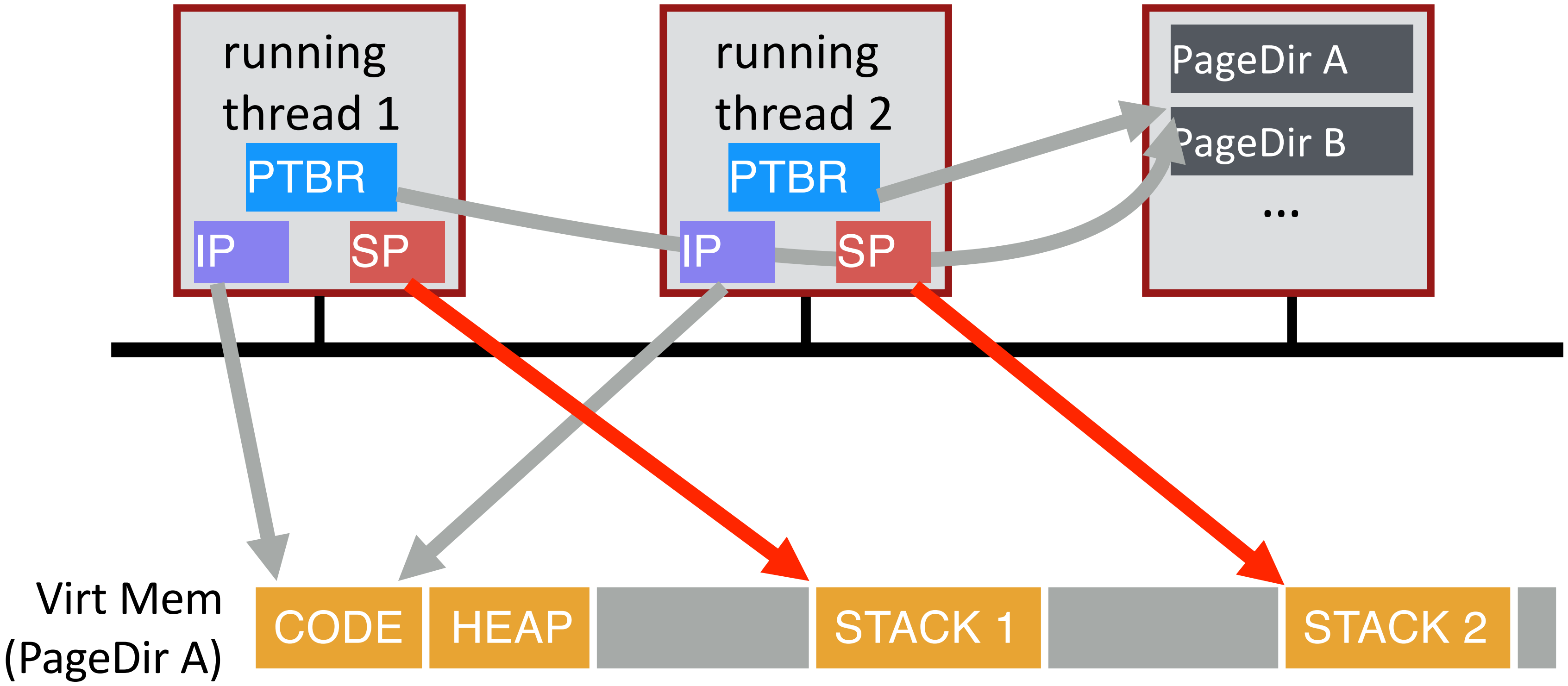
Share code, but each thread may be executing different code at the same time

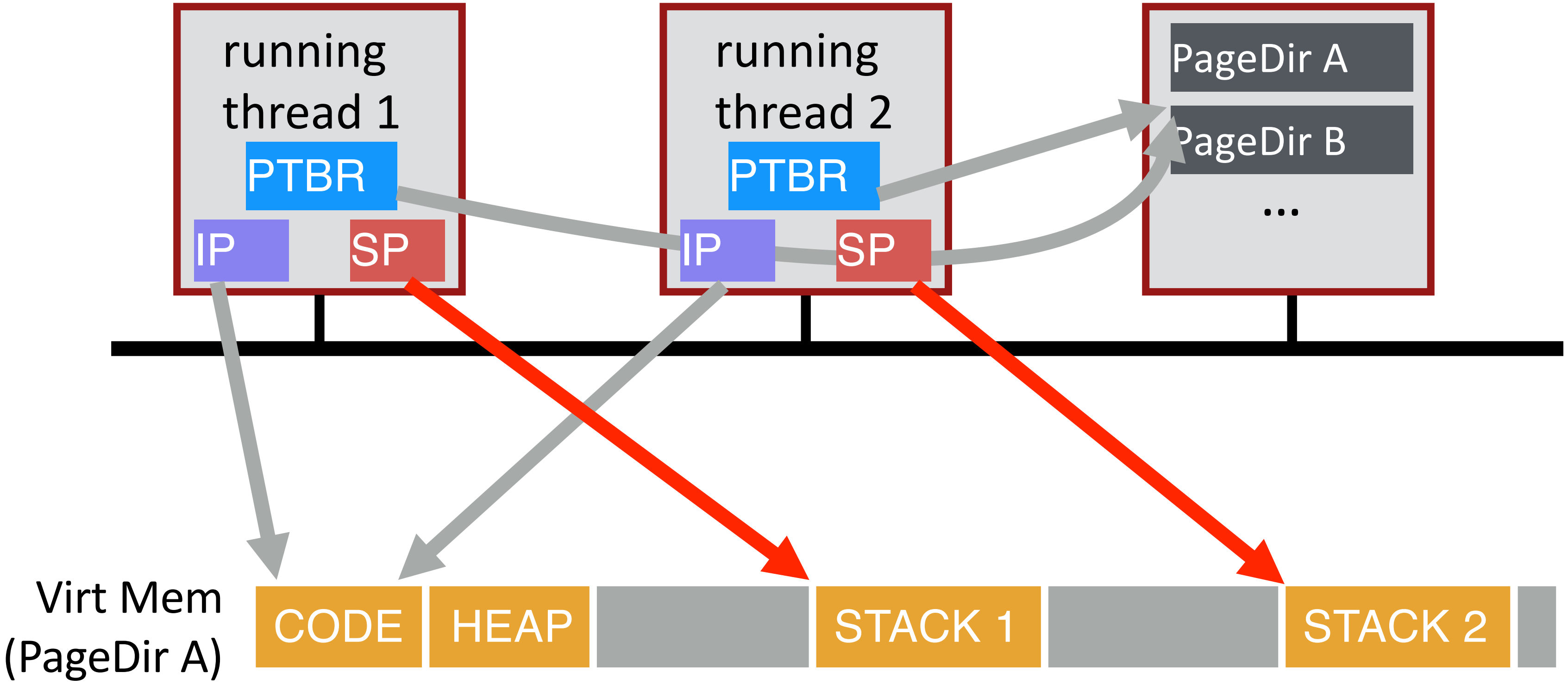
→ Different Instruction Pointers


running thread 1
PTBR
IP
running thread 2
PTBR
IP
PageDir A
PageDir B
...
Virt Mem
(PageDir A)
CODE
HEAP
...



Do threads share stack pointer?


running thread 1
PTBR
IP
SP
running thread 2
PTBR
IP
SP
PageDir A
PageDir B
...
Virt Mem
(PageDir A)
CODE
HEAP
STACK 1
STACK 2



threads executing different functions need different stacks

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)

THREAD API

Variety of thread systems exist

- POSIX Pthreads

Common thread operations

- Create
- Exit
- Join (instead of wait() for processes)

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

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
- OS must scale well with increasing number of threads

Thread Schedule #1

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

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

process control blocks:

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

T1 →

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

Thread Schedule #1

State:
0x9cd4: 100
%eax: 100
%rip = 0x19a

process control blocks:

%eax: ?	%eax: ?
%rip: 0x195	%rip: 0x195

T1

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

Thread Schedule #1

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

process control blocks:

%eax: ?	%eax: ?
%rip: 0x195	%rip: 0x195

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

T1

Thread Schedule #1

State:

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

process control blocks:

%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

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

T1

Thread Schedule #1

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

process control blocks:

%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

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

T1

Thread Context Switch

Thread Schedule #1

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

process control blocks:

%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

T2 →

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

Thread Schedule #1

State:
0x9cd4: 101
%eax: 101
%rip = 0x19a

process control blocks:

%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

T2

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

Thread Schedule #1

State:
0x9cd4: 101
%eax: 102
%rip = 0x19d

process control blocks:

%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

T2

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

Thread Schedule #1

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

process control blocks:

%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

T2

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

Thread Schedule #1

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

process control blocks:

%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

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

T2

Desired Result!

Let’s consider another schedule…

Thread Schedule #2

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

process control blocks:

%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

T1 →

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

Thread Schedule #2

State:
0x9cd4: 100
%eax: 100
%rip = 0x19a

process control blocks:

%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

T1

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

Thread Schedule #2

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

process control blocks:

%eax: ?
%rip: 0x195

%eax: ?
%rip: 0x195

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

T1

Thread Context Switch

Thread Schedule #2

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

process control blocks:

%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

T2 →

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

Thread Schedule #2

State:
0x9cd4: 100
%eax: 100
%rip = 0x19a

process control blocks:

%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

T2

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

Thread Schedule #2

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

process control blocks:

%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

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

T2

Thread Schedule #2

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

process control blocks:

%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

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

T2

Thread Schedule #2

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

process control blocks:

%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

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

T2

Thread Context Switch

Thread Schedule #2

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

process control blocks:

%eax: 101 %rip: 0x19d	%eax: 101 %rip: 0x1a2

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- T1 → 0x19d mov %eax, 0x9cd4

Thread Schedule #2

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

process control blocks:

%eax: 101 %rip: 0x1a2	%eax: 101 %rip: 0x1a2

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

T1

Thread Schedule #2

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

process control blocks:

%eax: 101	%eax: 101
%rip: 0x1a2	%rip: 0x1a2

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

T1

WRONG Result! Final value of balance is 101

Timeline View

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

How much is added to shared variable?

3: correct!

Timeline View

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

How much is added?

2: incorrect!

Timeline View

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

How much is added?

1: incorrect!

Timeline View

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

How much is added? 3: correct!

Timeline View

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

How much is added? 2: incorrect!

Non-Determinism

Concurrency leads to non-deterministic results

- Not deterministic result: different results even with same inputs
- race conditions: results depend on execution timing

Whether bug manifests depends on CPU schedule!

Passing tests means little

How to program: imagine scheduler is malicious
Assume scheduler will pick bad ordering at some point…

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
```

critical section

More general:

Need mutual exclusion for critical sections

- if process A is in critical section C, process B can't
- (okay if other processes do unrelated work)

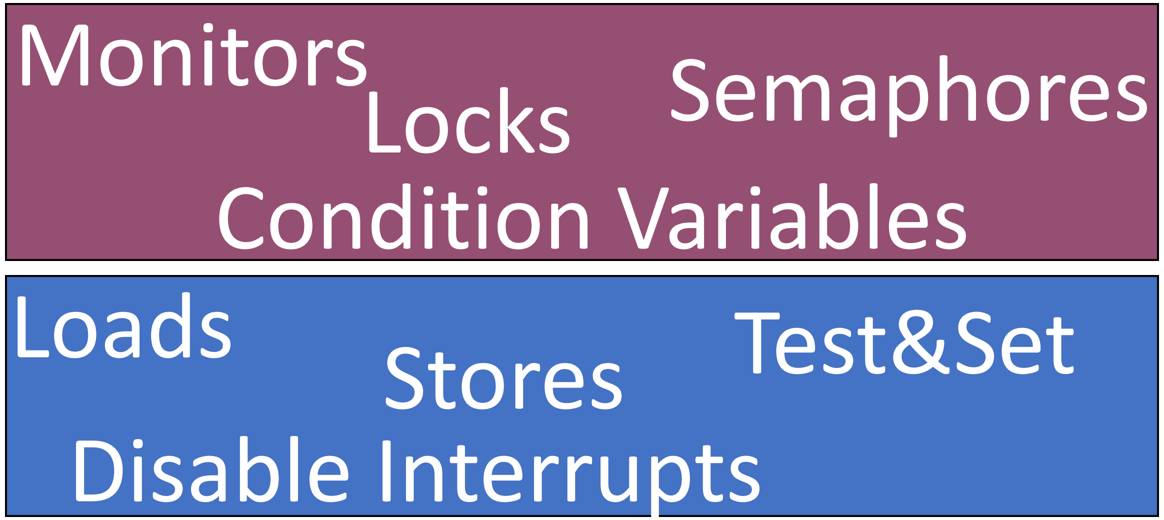
Synchronization

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Why is this an OS (rather than app) concern?

Motivation: Build them once and get them right



Locks

Goal: Provide mutual exclusion (mutex)

Three common operations:

- 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);`

Other Examples

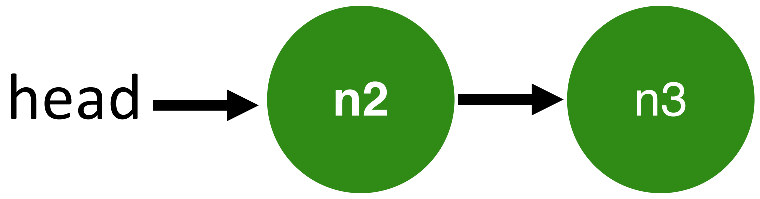
Consider multi-threaded applications that do more than increment shared balance

Multi-threaded application with shared linked-list

- All concurrent:
  - Thread A inserting element a
  - Thread B inserting element b
  - Thread C looking up element c

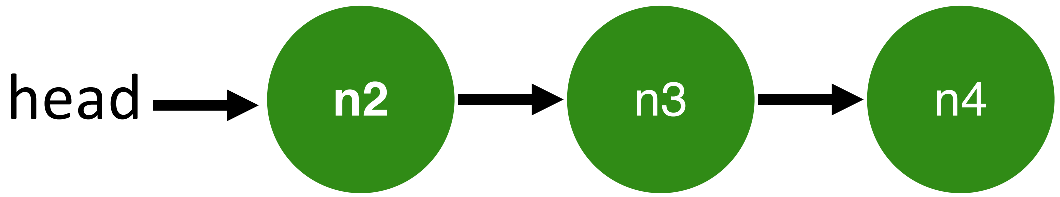
Shared Linked List

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
}
```



Shared Linked List

```
Void List_Insert(list_t *L, int key) {
		node_t *new =
malloc(sizeof(node_t));
		assert(new);
		new->key = key;
		new->next = L->head;
		L->head = new;
}
int List_Lookup(list_t *L,  int key) {
		node_t *tmp = L->head;
		while (tmp) {
			if (tmp->key == key)
					return
1;
			tmp = tmp->next;
		}
return 0;
}
```

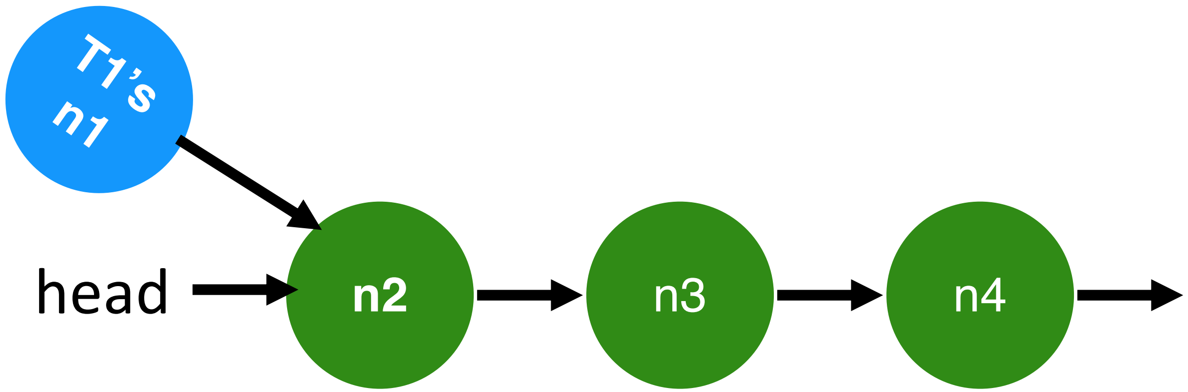


What can go wrong?
Find a schedule that leads to problem?

Linked-List Race

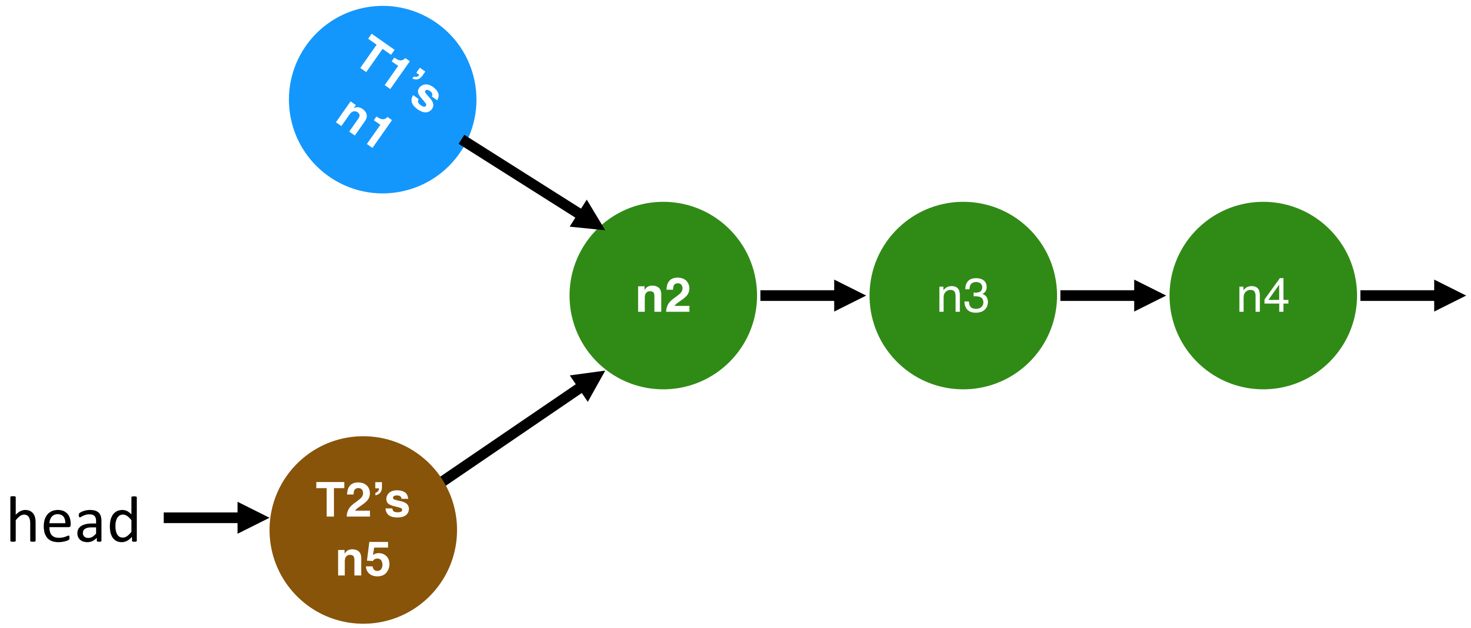
Thread 1	Thread 2
new->key = key	
new->next = L->head	

Linked-List Race



Linked-List Race

Thread 1	Thread 2
new->key = key	
new->next = L->head	
Cntxt_Switch()	new->key = key
	new->next = L->head
	L->head = new


T1's
n1
n2
n3
n4
head
T2's
n5

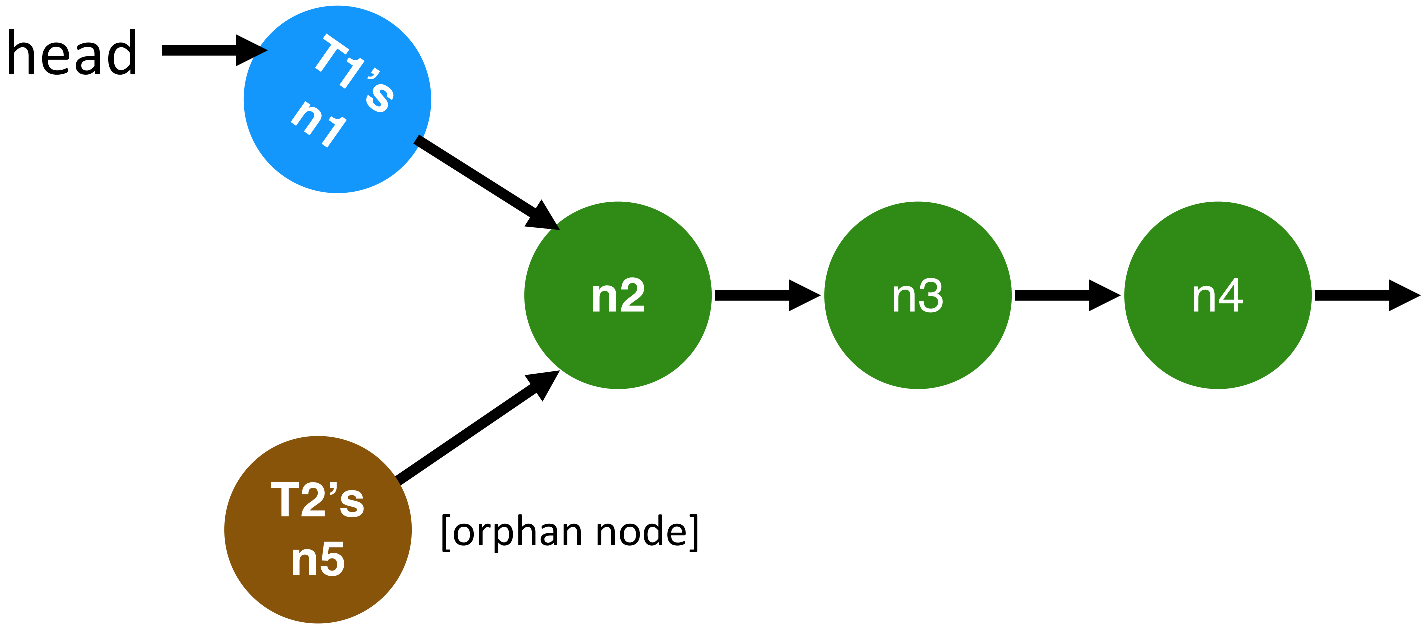
Linked-List Race

Thread 1	Thread 2
new->key = key	
new->next = L->head	
Cntxt_Switch()	new->key = key
	new->next = L->head
	L->head = new
	Cntxt_Switch()
L->head = new	

Both entries point to old head

Only one entry (which one?) can be the new head.

Resulting Linked List



Locking Linked Lists

```
Void List_Insert(list_t *L, int key) {
        node_t *new =
                malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}

int List_Lookup(list_t *L, int key) {
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)
                                return
1;
                tmp = tmp->next;
        }
    return 0;
}
```

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
}
```

How to add locks?

Locking Linked Lists

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
}
```

How to add locks?

pthread_mutex_t lock;

One lock per list

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
        pthread_mutex_t lock;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}
```

Locking Linked Lists : Approach #1

Consider everything critical section

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

```
Void List_Insert(list_t *L,  int key) {

  node_t *new = malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}
int List_Lookup(list_t *L, int key) {
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)

                return 1;
                tmp = tmp->next;
        }
    return 0;
}
```

Locking Linked Lists : Approach #2

Can critical section be smaller?

```
Void List_Insert(list_t *L,  int key) {
    Pthread_mutex_lock(&L->lock);
    node_t *new = malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
    Pthread_mutex_unlock(&L->lock);
}

int List_Lookup(list_t *L, int key) {
    Pthread_mutex_lock(&L->lock);
        node_t *tmp = L->head;
        while (tmp) {
            if (tmp->key == key)

                return 1;
            tmp = tmp->next;
        }
    Pthread_mutex_unlock(&L->lock);
    return 0;
}
```

Locking Linked Lists : Approach #2

Can critical section be smaller?

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

```
Void List_Insert(list_t *L,  int key) {

  node_t *new = malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}
int List_Lookup(list_t *L, int key) {
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)

                return 1;
                tmp = tmp->next;
        }
    return 0;
}
```

Locking Linked Lists : Approach #3

Can critical section be smaller?

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

~~Pthread_mutex_lock(&L->lock);~~

If no List_Delete(), locks not needed

~~Pthread_mutex_unlock(&L->lock);~~

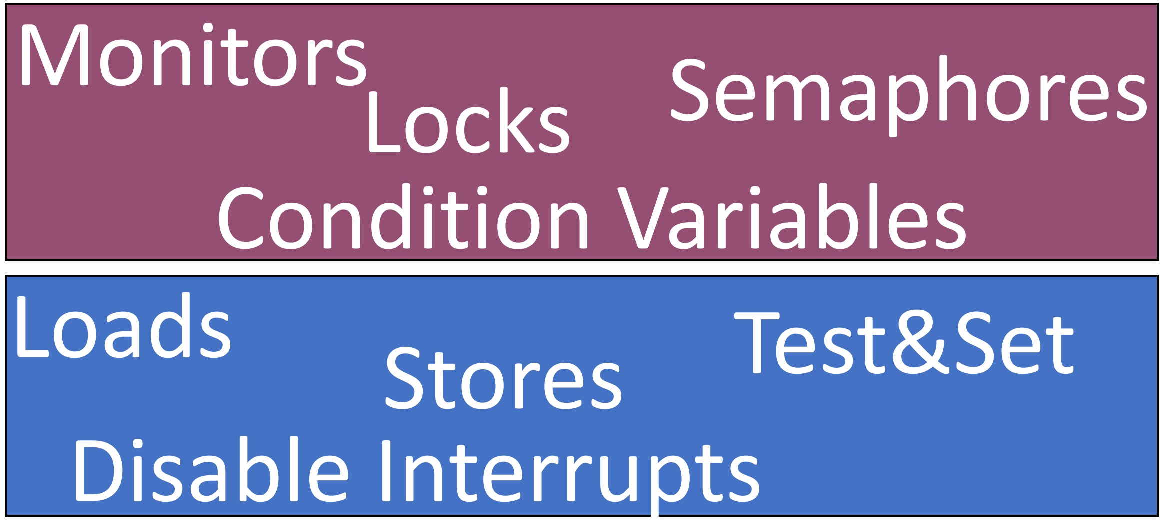
```
Void List_Insert(list_t *L, int key) {

  node_t *new = malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}
int List_Lookup(list_t *L, int key) {
        node_t *tmp = L->head;
        while (tmp) {
              if (tmp->key == key)

              return 1;
              tmp = tmp->next;
        }
    return 0;
}
```

Implementing Synchronization

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right



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 (e.g., spinning)

Implementing Synchronization

To implement, need atomic operations

Atomic operation: No other instructions can be interleaved

Examples of atomic operations

- Code between interrupts on uniprocessors
  - Disable timer interrupts, don't do any I/O
- Loads and stores of words
  - Load r1, B
  - Store r1, A
- **Special hw instructions**
  - **Test&Set**
  - **Compare&Swap**

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??

Implementing LOCKS: w/ Load+Store

Code uses a single **shared** lock variable

```
Boolean lock = false; // shared variable
Void acquire(Boolean *lock) {
     while (*lock) /* wait */ ;
     *lock = true;
}
Void release(Boolean *lock) {
     *lock = false;
}
```

Why doesn't this work? Example schedule that fails with 2 threads?

*lock == 0 initially

Thread 1	Thread 2
`while(*lock == 1);`	
	`while(*lock == 1);`
	`*lock = 1`
`*lock = 1`	

```
// xchg(int *addr, int newval)
// return what was pointed to by addr
// at the same time, store newval into addr

int xchg(int *addr, int newval) {
  int old = *addr;
  *addr = newval;
  return old;
}
```

```
xchg(                    *addr,               newval)
{
          result;
    asm volatile("lock; xchgl %0, %1" :
                 "+m" (*addr), "=a" (result) :
                 "1" (newval) : "cc");
    return result;
}
```

LOCK Implementation with XCHG

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    lock->flag = ??;
}

void acquire(lock_t *lock) {
    ????;
    // spin-wait (do nothing)
}

void release(lock_t *lock) {
    lock->flag = ??;
}
```

```
int xchg(int *addr, int newval)
```

XCHG Implementation

```
typedef struct __lock_t {
     int flag;
} lock_t;

void init(lock_t *lock) {
     lock->flag = 0;
}

void acquire(lock_t *lock) {
     while(xchg(&lock->flag, 1) == 1) ;
     // spin-wait (do nothing)
}

void release(lock_t *lock) {
     lock->flag = 0;
}
```

Other Atomic HW Instructions

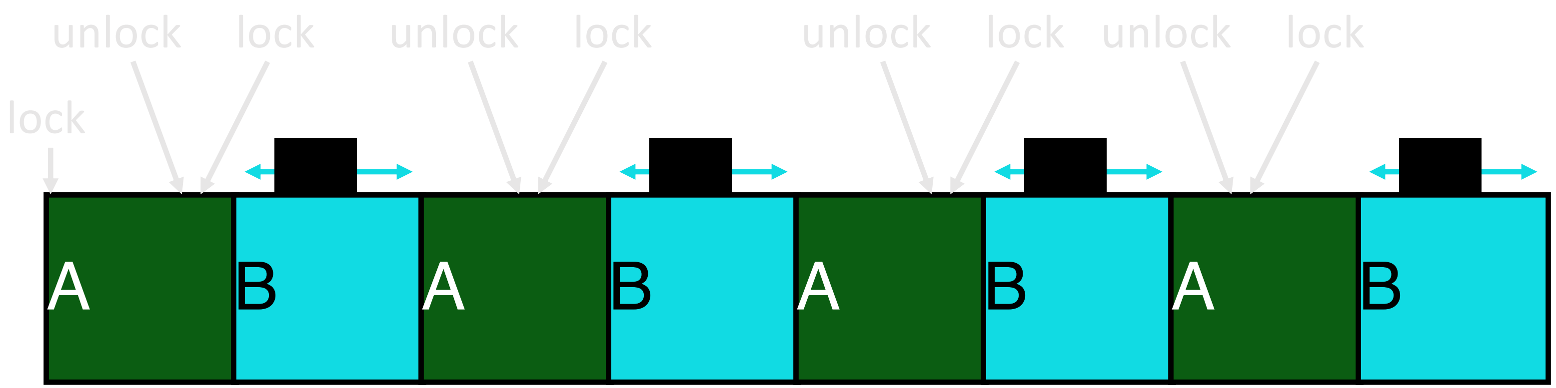
```
int CompareAndSwap(int *addr, int expected, int new) {
  int actual = *addr;
  if (actual == expected)
    *addr = new;
  return actual;
}
```

```
void acquire(lock_t *lock) {
        while(CompareAndSwap(&lock->flag, ?, ?)
                                        == ?) ;
        // spin-wait (do nothing)
}
```

Other Atomic HW Instructions

```
int CompareAndSwap(int *ptr, int expected, int new) {
  int actual = *addr;
  if (actual == expected)
    *addr = new;
  return actual;
}
```

```
void acquire(lock_t *lock) {
        while(CompareAndSwap(&lock->flag, 0, 1)
                                        == 1) ;
        // spin-wait (do nothing)
}
```



unlock
lock
unlock
lock
unlock
lock
unlock
lock
lock
A
B
A
B
A
B
A
B

Idea: reserve each thread’s turn to use a lock.

Each thread spins until their turn.

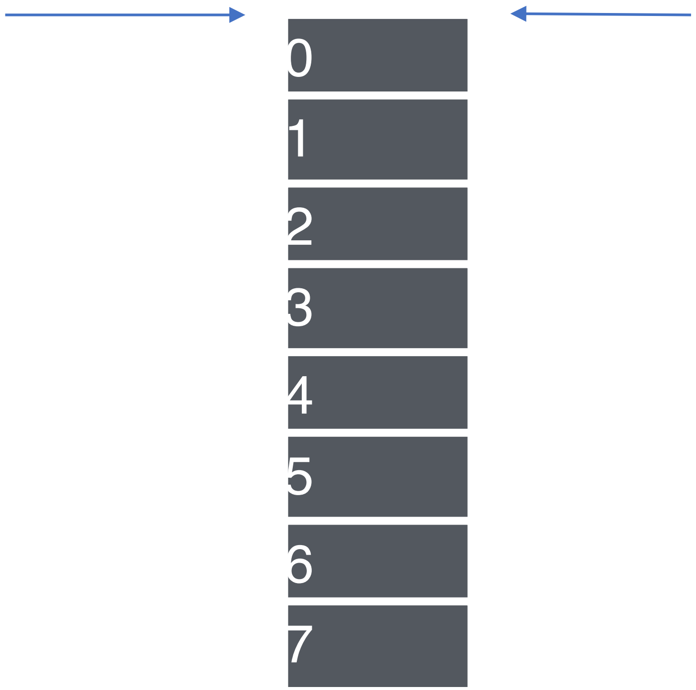
Use new atomic primitive, fetch-and-add:

```
int FetchAndAdd(int *ptr) {
  int old = *ptr;
  *ptr = old + 1;
  return old;
}
```

```
Acquire: Grab ticket;
Spin while not thread’s ticket != turn
```

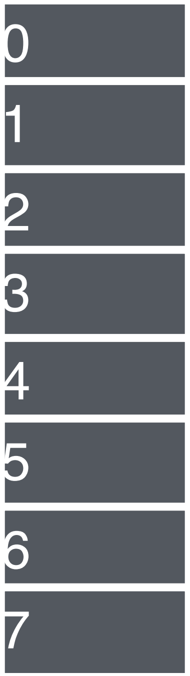
```
Release: Advance to next turn
```

Ticket Lock Example



Ticket Lock ExampLE

A lock(): gets ticket 0, spins until turn = 0 →runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++ (turn = 1)
B runs
A lock(): gets ticket 3, spins until turn=3
B unlock(): turn++ (turn = 2)
C runs
C unlock(): turn++ (turn = 3)
A runs
A unlock(): turn++ (turn = 4)
C lock(): gets ticket 4, runs

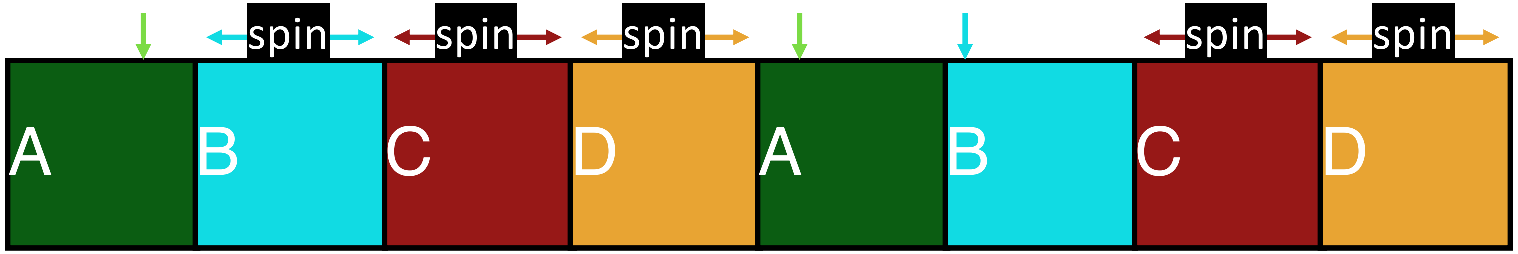


```
typedef struct __lock_t {
        int ticket;
        int turn;
}

void lock_init(lock_t *lock) {
        lock->ticket = 0;
        lock->turn = 0;
}
```

```
void acquire(lock_t *lock) {
        int myturn = FAA(&lock->ticket);
        while (lock->turn != myturn); // spin
}

void release (lock_t *lock) {
        FAA(&lock->turn);
}
```

Fast when…
- many CPUs
- locks held a short time
- advantage: avoid context switch

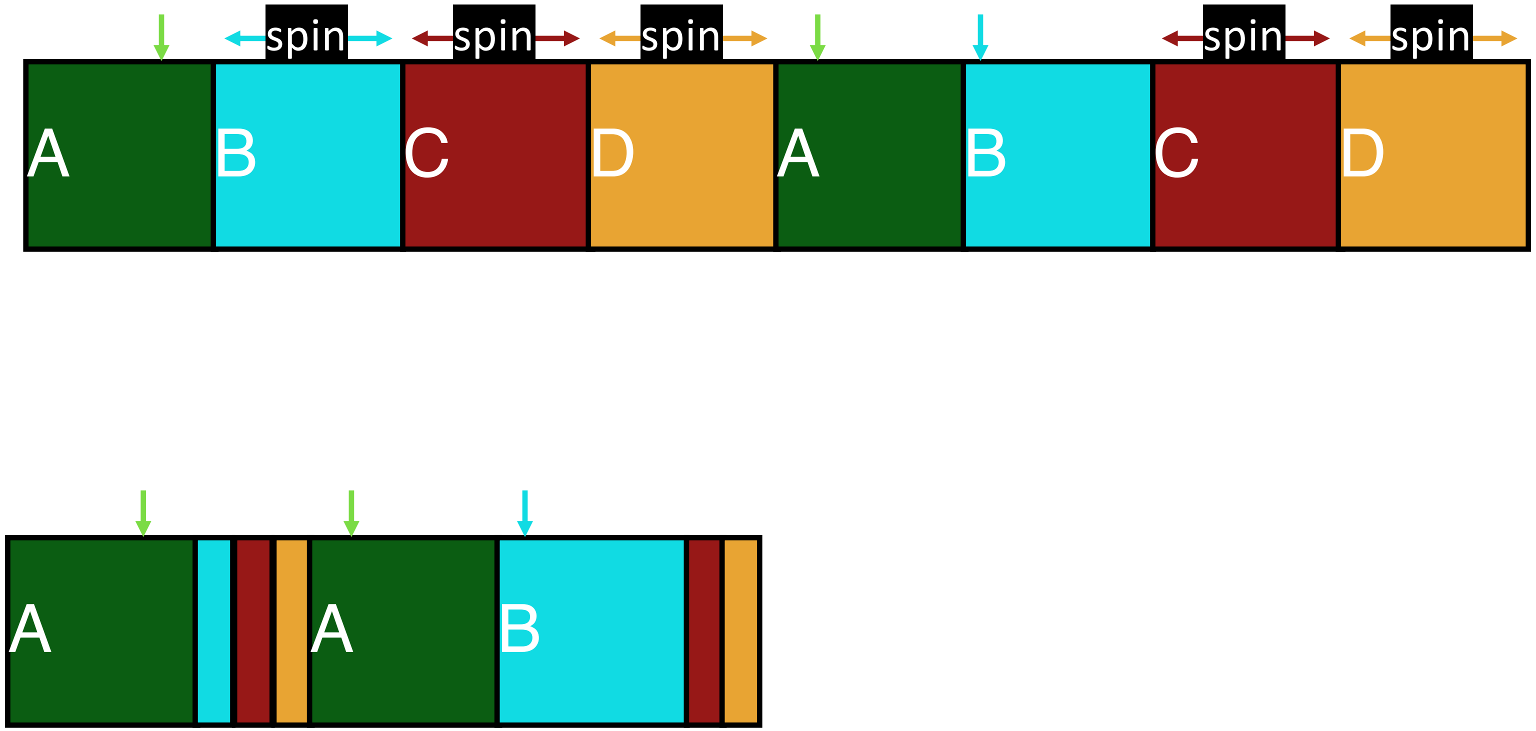
Slow when…
- one CPU
- locks held a long time
- disadvantage: spinning is wasteful


spin
spin
spin
spin
spin
A
B
C
D
A
B
C
D

```
typedef struct __lock_t {
        int ticket;
        int turn;
}

void lock_init(lock_t *lock) {
        lock->ticket = 0;
        lock->turn = 0;
}
```

```
void acquire(lock_t *lock) {
        int myturn = FAA(&lock->ticket);
        while(lock->turn != myturn)
}

void release (lock_t *lock) {
        FAA(&lock->turn);
}
```

Yield Instead of Spin



Waste…

- Without yield: O(threads * **time_slice**)
- With yield: O(threads * **context_switch**)

So even with yield, spinning is slow with high thread contention

Next improvement: Block and put thread on waiting queue instead of spinning

Mutual exclusion (e.g., A and B don’t run at same time)

- solved with *locks*

Ordering (e.g., B runs after A does something)

- solved with *condition variables* and *semaphores*

wait(cond_t *cv, mutex_t *lock)

- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

signal(cond_t *cv)

- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return, doing nothing

Parent:

Child:

void thread_exit() {

w

Mutex_lock(&m);

// a

x

done = 1;

y

// b

z

Cond_signal(&c);

// c

Note: Cond_wait also releases mutex before waiting provided condition is not met yet

Mutex_unlock(&m);

// d

z

}

Parent: w x y

Child: a b c d

Producers generate data (like pipe writers)

Consumers grab data and process it (like pipe readers)

Use condition variables to:

make producers wait when buffers are full

make consumers wait when there is nothing to consume

Broken Implementation of Producer Consumer

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m); // p1
                while(numfull == max)
//p2
        Cond_wait(&cond, &m); //p3
                do_fill(i); // p4
                Cond_signal(&cond); //p5
                Mutex_unlock(&m); //p6
        }
}
```

m

Cond_wait(&cond

m

cond

m

wait()

wait()

Producer: p4 p5 p6 p1 p2 p3 p1 p2

Consumer1: c1 c2 c3

Consumer2: c1 c2 c3 c2 c4 c5

producer consumer2

Producer/Consumer: Two CVs

Is this correct? Can you find a bad schedule?

1. consumer1 waits because numfull == 0
2. producer increments numfull, wakes consumer1
3. before consumer1 runs, consumer2 runs, grabs entry, sets numfull=0.
4. consumer1 then reads bad data.

Producer: p1 p2 p4 p5 p6

Consumer1: c1 c2 c3 c4! ERROR

Consumer2: c1 c2 c4 c5 c6

Whenever a lock is acquired, recheck assumptions about state!

Use “while” intead of “if”

Possible for another thread to grab lock between signal and wakeup from wait

- Difference between Mesa (practical implementation) and Hoare (theoretical) semantics
- Signal() simply makes a thread runnable, does not guarantee thread run next

Note that some libraries also have “spurious wakeups”

- May wake multiple waiting threads at signal or at any time

Producer/Consumer: Two CVs and WHILE

Is this correct? Can you find a bad schedule?

Correct!

- no concurrent access to shared state
- every time lock is acquired, assumptions are reevaluated
- a consumer will get to run after every `do_fill()`
- a producer will get to run after every `do_get()`

Keep state in addition to CV's

Always do wait/signal with lock held

Whenever thread wakes from waiting, recheck state

Conclusions

Concurrency is needed to obtain high performance by utilizing 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 (race conditions)

Use locks to provide mutual exclusion

Improving performance requires reducing critical section cost