CS 423
Operating System Design: Midterm Review
03/26

Ram Alagappan

Logistics

Today: review

Friday: midterm exam (take home)

Will release the exam by 12 noon Friday and it is due 12 noon Saturday

Need only 2-3 hours if you are well prepared

Logistics

Exam: will post link to exam on Piazza (not anywhere else)

Keep looking for a post (will be available just before 12 noon)

You must type in answers (not hand-written)

At the top of the exam, there will be a link to a google form – this is where you will submit/upload your finished exam

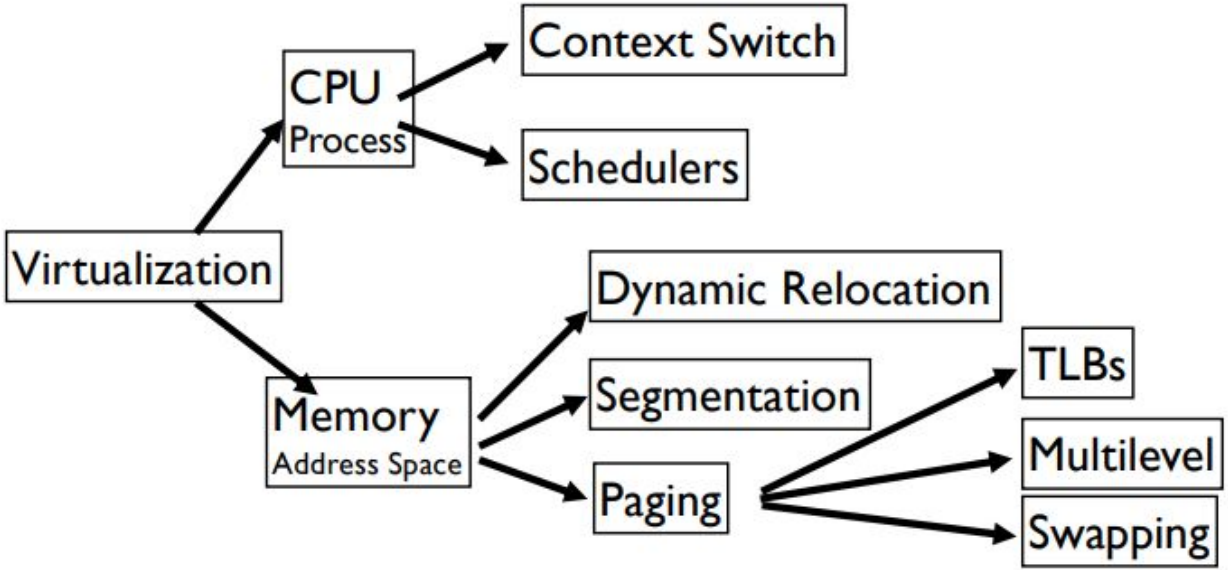
The form will close at 12 noon Saturday

More about the exam

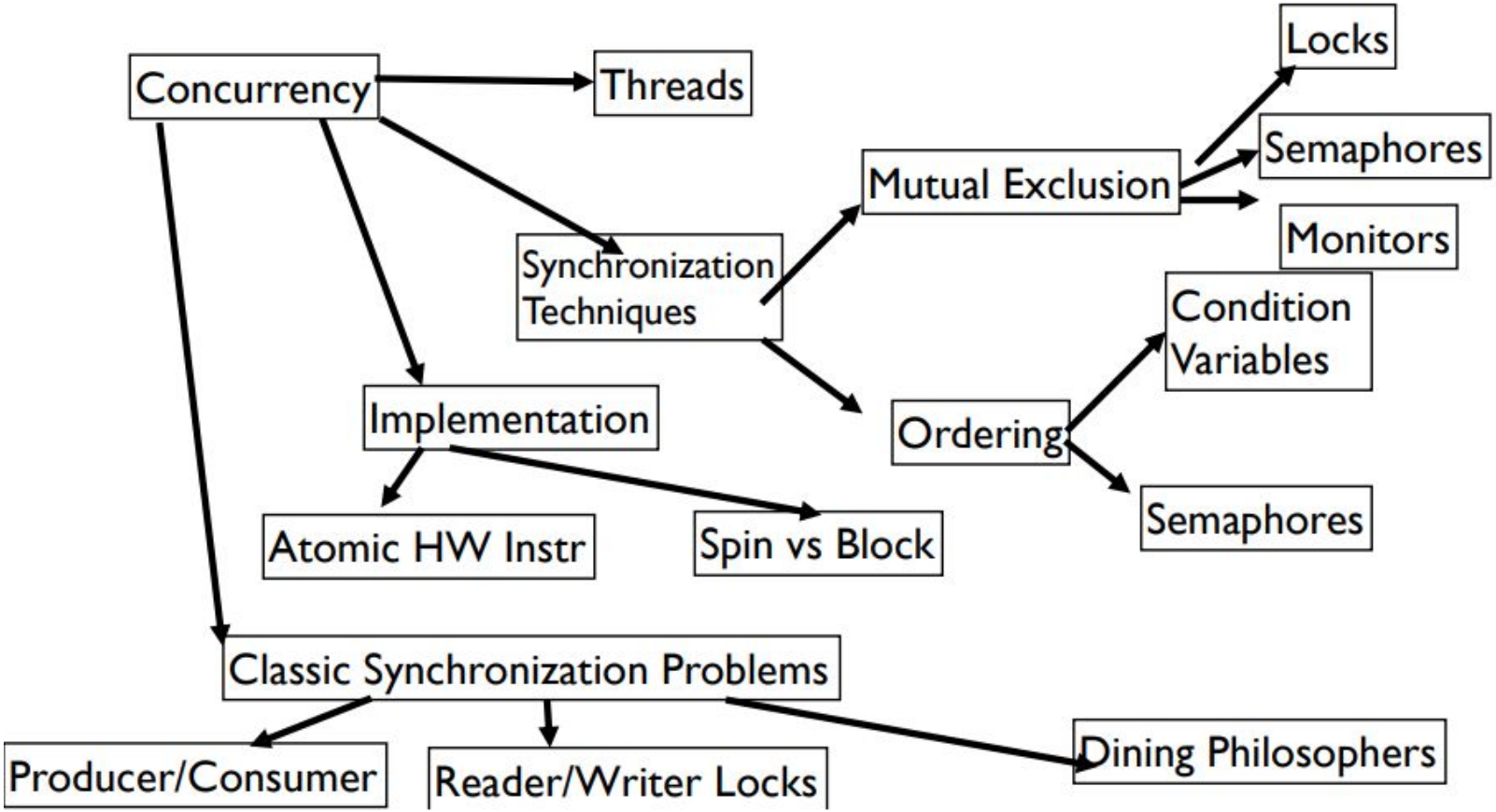
Mostly 4 or 5 questions

Questions based on lecture materials and MPs

REVIEW: EASY PIECE 1



REVIEW: EASY PIECE 2



REVIEW: PROCESSES VS THREADS

```
int a = 0;
int main() {
  fork();
  a++;
  fork();
  a++;
  if (fork() == 0) {
    printf("Hello!\n");
  } else {
    printf("Goodbye!\n");
  }
  a++;
  printf("a is %d\n", a);
}
```

How many times will "Hello!\n" be displayed?

What will be the **final** value of "a" as displayed by the final line of the program?

REVIEW: PROCESSES VS THREADS

```
volatile int balance = 0;
void *mythread(void *arg) {
    int result = 0;
    result = result + 200;
    balance = balance + 200;
    printf("Result is %d\n", result);
    printf("Balance is %d\n", balance);
    return NULL;
}
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final Balance is %d\n", balance);
}
```

How many total threads are part of this process?

When thread p1 prints "Result is %d\n", what value of `result` will be printed?

When thread p1 prints "Balance is %d\n", what value of `balance` will be printed?

Scheduling

Scheduling mechanism: context switch

HW saves some registers, OS does remaining

Scheduling policies: metrics like turnaround time, response time

Policies: FIFO, SJF, STCF (preemptive version of SJF)

All above must know job time

In reality don't know, RR is a good scheme when time unknown

MLFQ: GENERAL PURPOSE SCHEDULER

Must support two job types with distinct goals

- “interactive” programs care about response time
- “batch” programs care about turnaround time

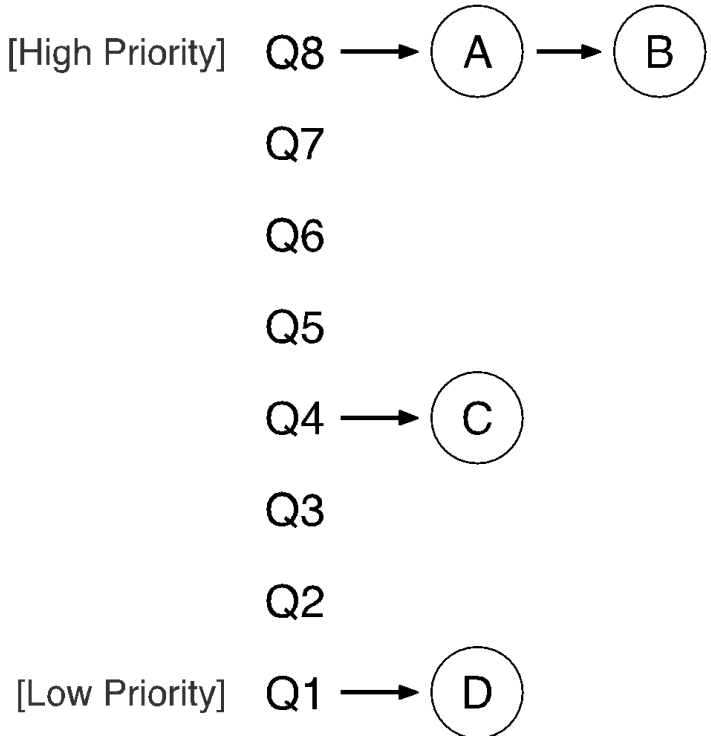
Approach:

Multiple levels of round-robin

Each level has higher priority than lower level

Can preempt them

MLFQ EXAMPLE



"Multi-level" – Each level is a queue

Rules for MLFQ

Rule 1: If priority(A) > Priority(B)
A runs

Rule 2: If priority(A) == Priority(B),
A & B run in RR

CHALLENGES

How to set priority?

Does a process stay in one queue (static) or move between queues (dynamic)?

Approach:

Use past behavior of process to predict future!

Common approach in systems when don't have perfect knowledge

Guess how CPU burst (job) will behave based on past CPU bursts

MORE MLFQ RULES

Rule 1: If priority(A) > Priority(B), A runs

Rule 2: If priority(A) == Priority(B), A & B run in RR

Rule 3: Processes start at top priority

Rule 4: If job uses whole slice, demote process

(longer time slices at lower priorities)

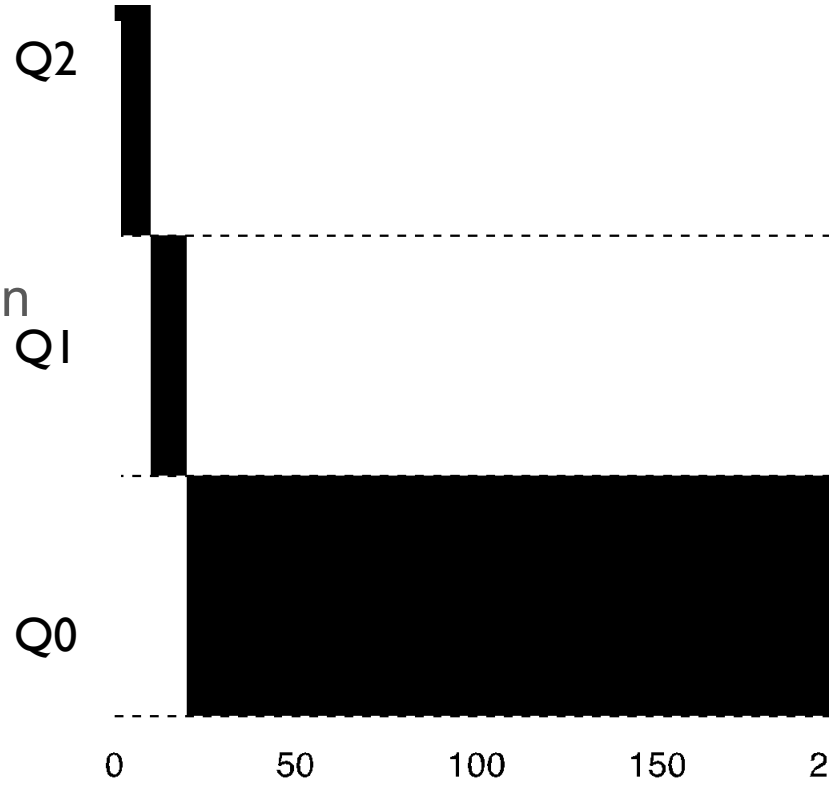
ONE LONG JOB

Starts at top

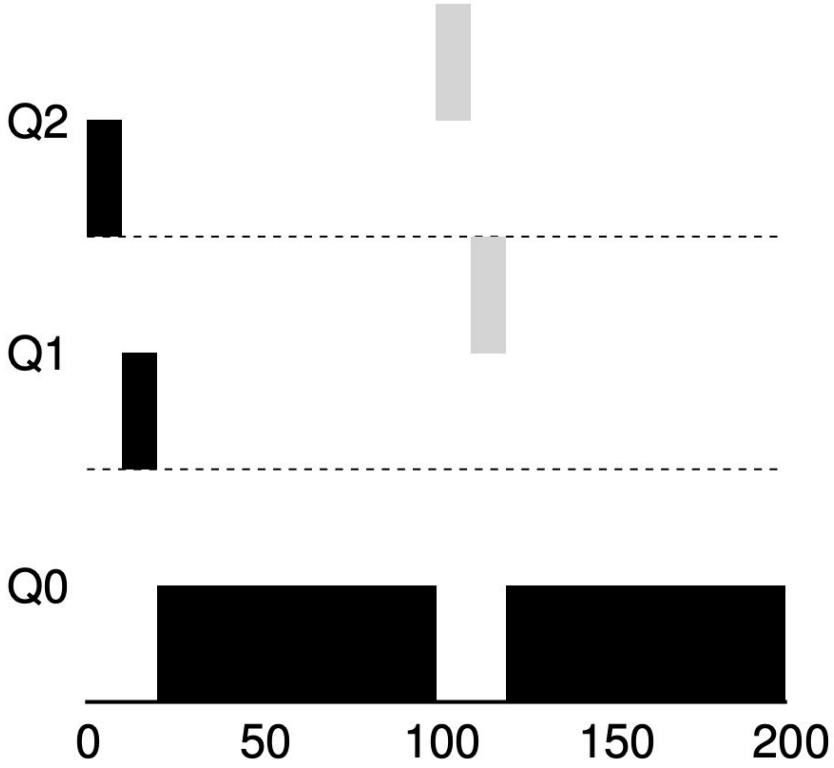
Uses whole slice

Moves down

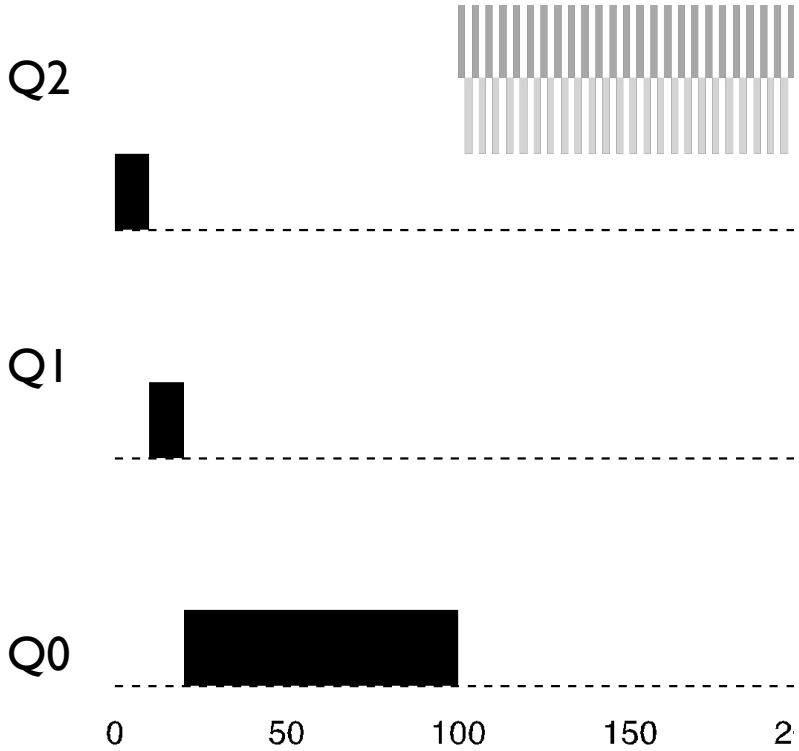
Moves down again



INTERACTIVE SHORT PROCESS JOINS

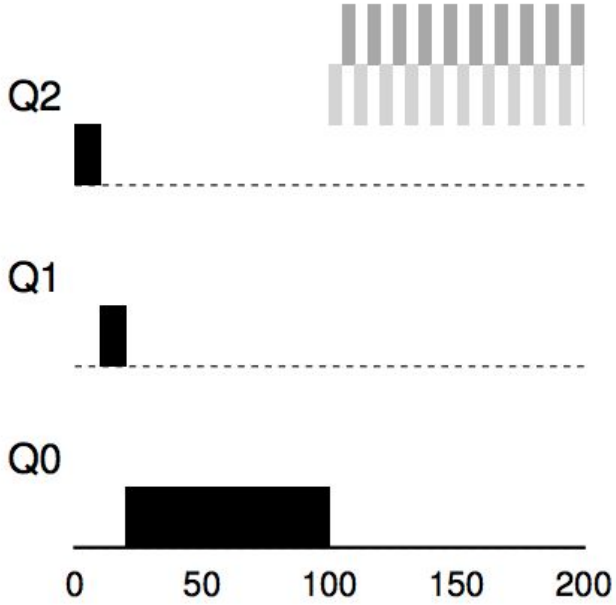


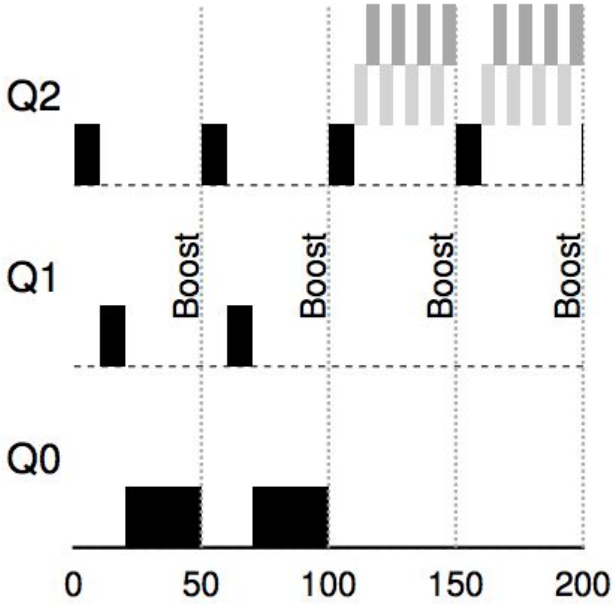
MLFQ PROBLEMS?



What is the problem with this schedule ?

AVOIDING STARVATION

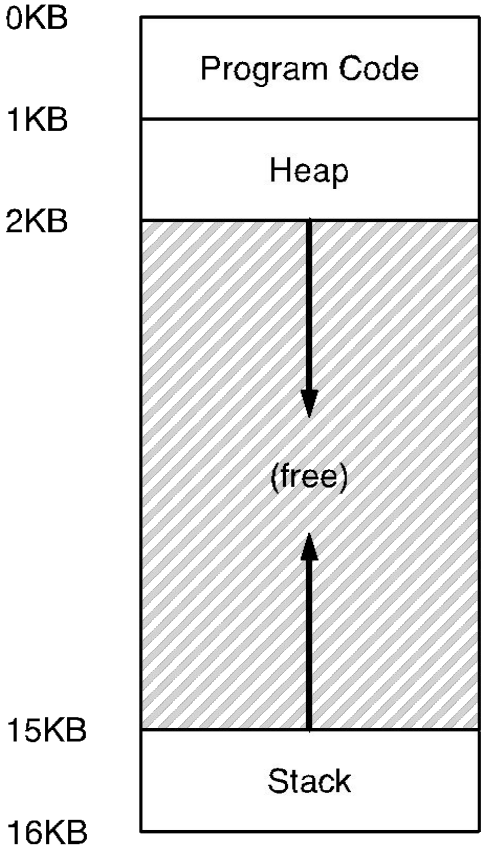




Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

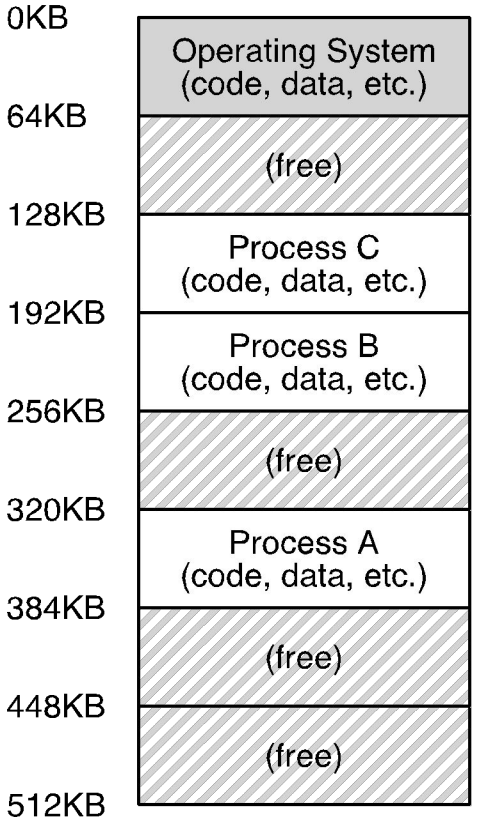
Q: After the second boost, why are *all* jobs not run in RR!?

ABSTRACTION: ADDRESS SPACE



Address Space:
Each process has its own set of addresses

OS aims to provide Illusion of private memory



HOW TO VIRTUALIZE MEMORY

Problem: How to run multiple processes simultaneously?

Addresses are “hardcoded” into process binaries

How to avoid collisions?

Possible Solutions for Mechanisms (covered today):

1. Time Sharing
2. Static Relocation
3. Base
4. Base+Bounds

Assumptions: each AS is the same size, AS is smaller than physical memory, AS can be placed contiguously in physical memory (not realistic…)

4) DYNAMIC WITH BASE+BOUNDS

Base register: smallest physical addr (or starting location)

Bounds register: size of this process's virtual address space

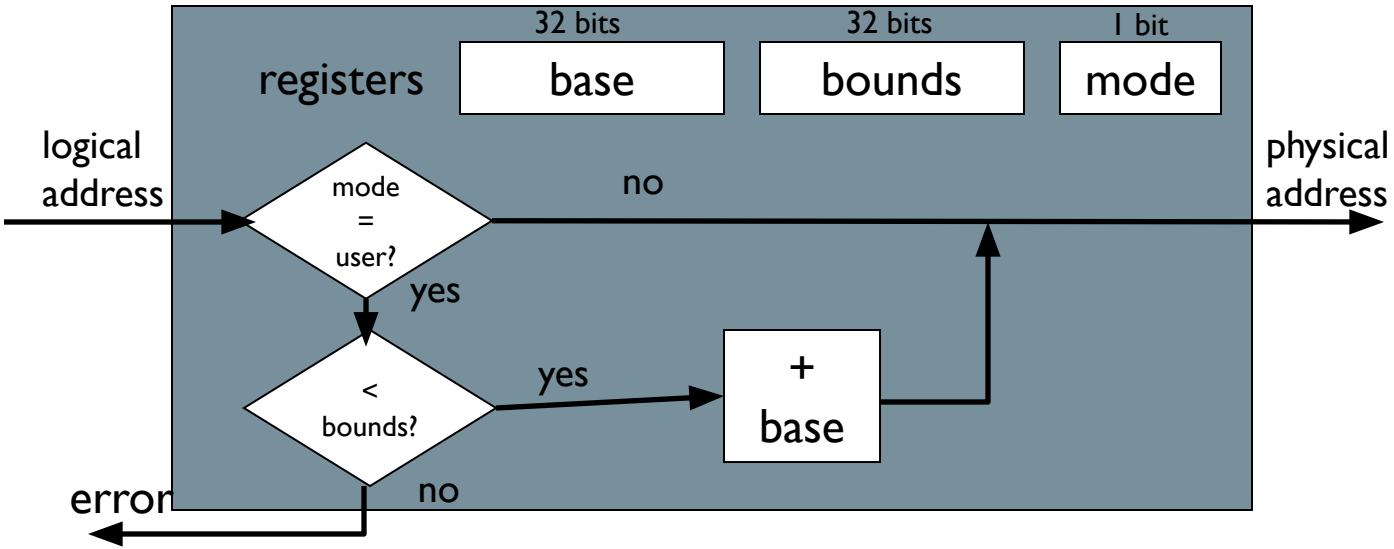
- Sometimes defined as largest physical address (base + size)

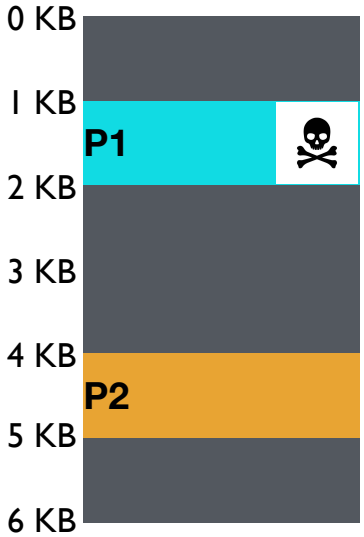
OS kills process if process loads/stores beyond bounds

Implementation of BASE+BOUNDS

Translation on every memory access of user process

- MMU compares logical address to bounds register
  if logical address is greater, then generate error
- MMU adds base register to logical address to form physical address





Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	Interrupt OS!

Can P1 hurt P2?

Managing Processes with Base and Bounds

Context-switch: Add base and bounds registers to PCB

Steps

- Change to privileged mode
- Save base and bounds registers of old process
- Load base and bounds registers of new process
- Change to user mode and jump to new process

Are there cases where you won't change Base and Bounds register during context switches?

Protection requirement

- User process cannot change base and bounds registers
- User process cannot change to privileged mode

Base and Bounds Advantages

Provides protection (both read and write) across address spaces
Supports dynamic relocation
    Can place process at different locations initially and also move address spaces

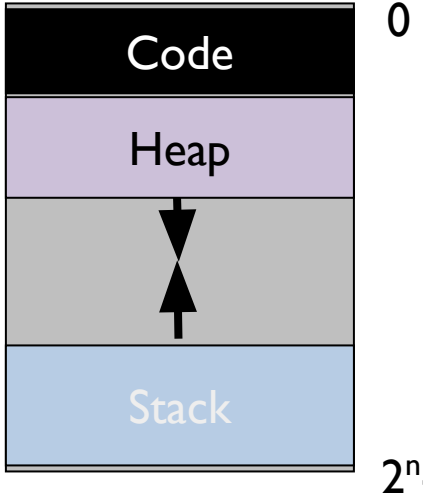
Simple, inexpensive implementation: Few registers, little logic in MMU
Fast: Add and compare in parallel

Base and Bounds DISADVANTAGES

Disadvantages

- Each process must be allocated contiguously in physical memory
  Must allocate memory that may not be used by process

- No partial sharing: Cannot share parts of address space



5) Segmentation

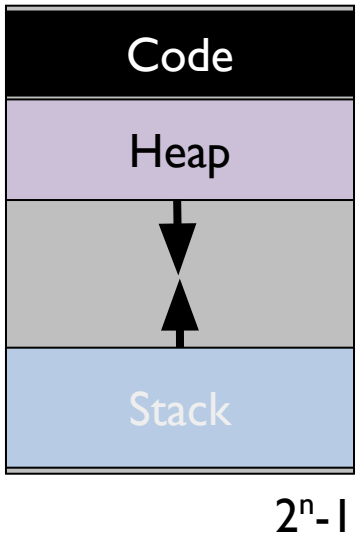
Divide address space into logical segments

- Each segment corresponds to logical entity in address space (code, stack, heap)

Each segment has separate base + bounds register

Each segment can independently:

1. Be placed in physical memory
2. Grow and shrink
3. Be protected (read/write/exec)

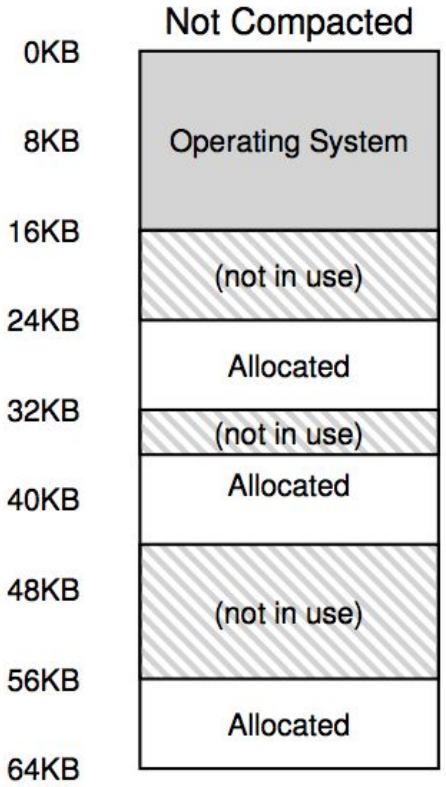


Disadvantages of Segmentation

Each segment must be allocated contiguously

May not have sufficient physical memory for large segments?

External Fragmentation

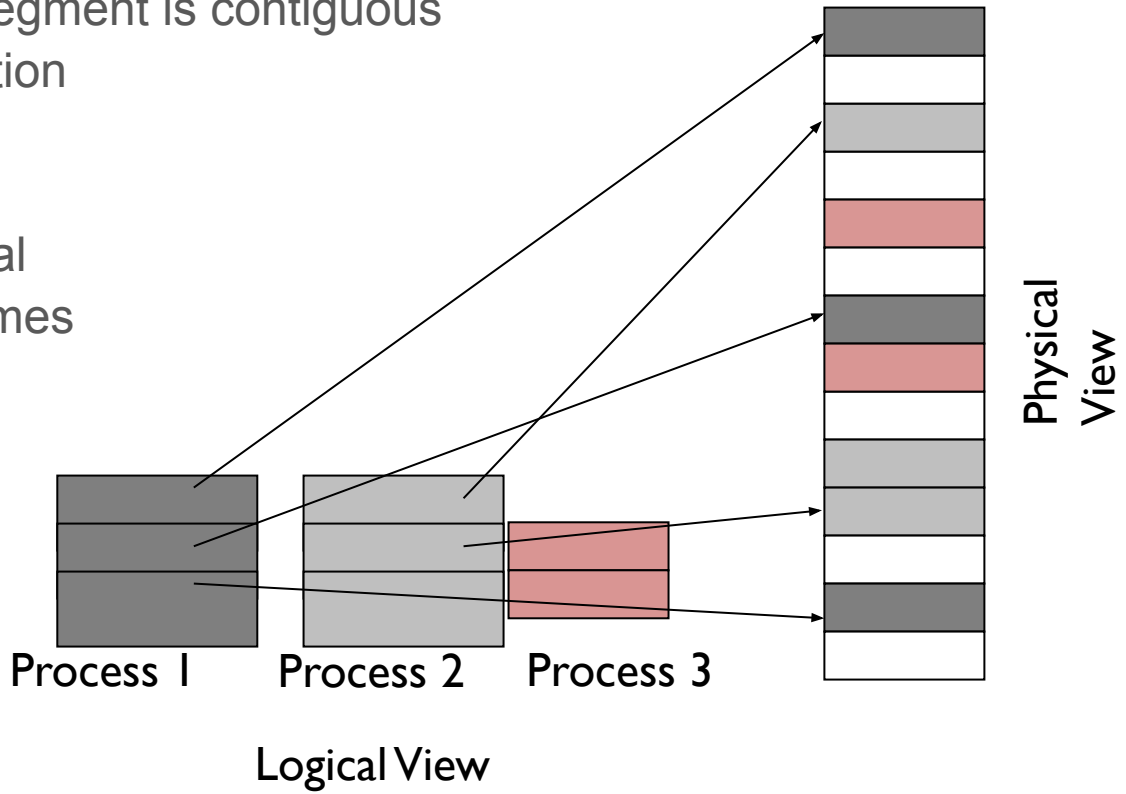


Paging

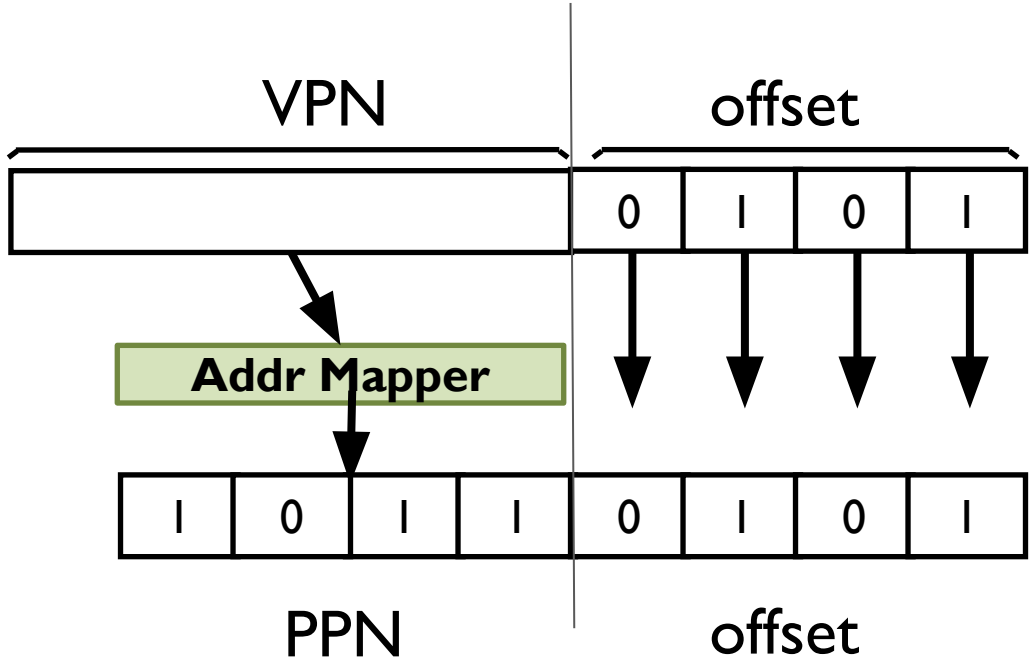
Goal: Eliminate requirement that segment is contiguous
Eliminate external fragmentation

Idea:
Divide address spaces and physical memory into fixed-sized pages/frames

Size: 2^n, Example: 4KB



VIRTUAL □ PHYSICAL PAGE MAPPING



Number of bits in virtual address

need not equal

number of bits in physical address

bits in VPN

> # bits in PPN?

What does this mean?

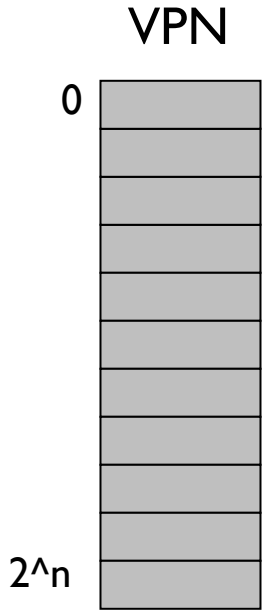
Is inverse possible?

How should OS translate VPN to PPN/PFN?

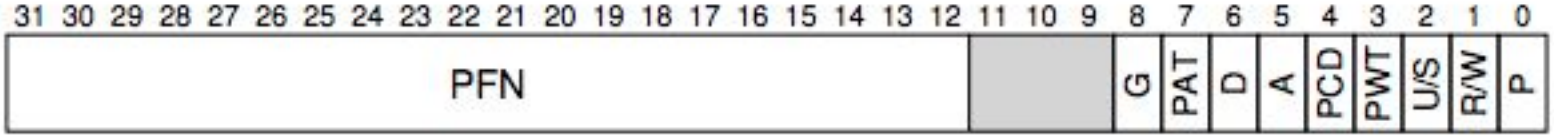
Linear PageTable

What is a good data structure ?

Simple solution: Linear page table aka *array*



A Single PTE:



Disadvantages of Paging

Internal fragmentation: Page size may not match size needed by process

- Wasted memory grows with larger pages
- **Tension? Why not make pages really small?**

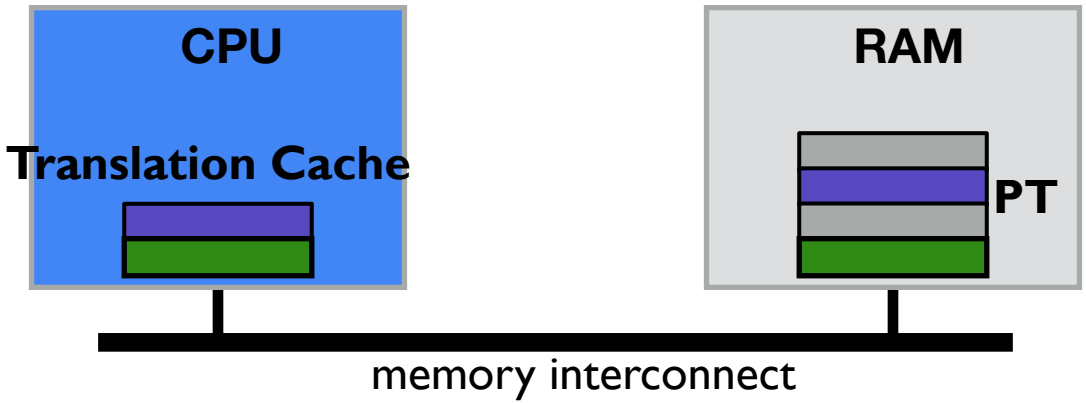
Additional memory reference to page table □ Very inefficient

- Page table must be stored in memory
- MMU stores only base address of page table

Storage for page tables may be substantial

- Simple page table: Requires PTE for all pages in address space
  Entry needed even if page not allocated
- Page tables must be contiguously allocated - fix with paging the page tables

STRATEGY: CACHE PAGE TRANSLATIONS



TLB: TRANSLATION LOOKASIDE BUFFER

HW AND OS ROLES

If H/W handles TLB Miss

CPU must know where pagetables are

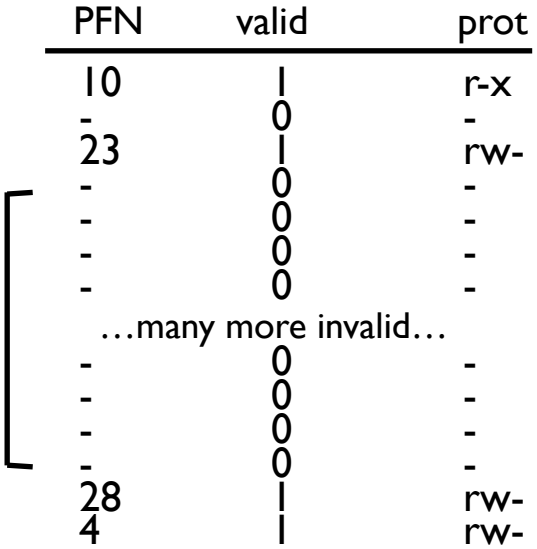
- CR3 register on x86
- Pagetable structure fixed and agreed upon between HW and OS
- HW “walks” the pagetable and fills TLB

If OS handles TLB Miss:

“Software-managed TLB”

- CPU traps into OS upon TLB miss.
- OS interprets pagetables as it chooses
- Modify TLB entries with privileged instruction

MANY INVALID PTES



how to avoid storing these?

Problem: linear PT must still allocate PTE for each page (even unallocated ones)

Combine Paging and Segmentation

Divide address space into segments (code, heap, stack)

- Segments can be variable length

Divide each segment into fixed-sized pages.

Logical address divided into three portions

Implementation

- Each segment has a page table
- Track base physical address and bounds of the **page table** per segment
  - This is different from original segmentation

Paging and Segmentation - Chat for 2 mins

seg	base	bounds	R W
0	0x002000	0xff	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f	1 1

0x002070 read:
0x202016 read:
0x104c84 read:
0x010424 write:
0x210014 write:
0x203568 read:

...
0x01f
0x011
0x003
0x02a
0x013
...
0x00c
0x007
0x004
0x00b
0x006
...

0x001000

0x002000

Bounds: # PTE entries

Advantages of Paging and Segmentation

Advantages from using Segments

- Decreases size of page tables. If segment not used, no need for page table

Advantages from using Pages

- No external fragmentation
- Segments can grow without any reshuffling
- Can run process with some pages swapped to disk

Disadvantages of Paging and Segmentation

Potentially large page tables (for each segment)

- Must allocate page table for each segment *contiguously*
- Page table size?
  - Assume 2 bits for segment, 18 bits for page number, 12 bits for offset

Each page table is:
= Number of entries * size of each entry
= Number of pages * 4 bytes
= 2^18 * 4 bytes = 2^20 bytes = 1 MB!

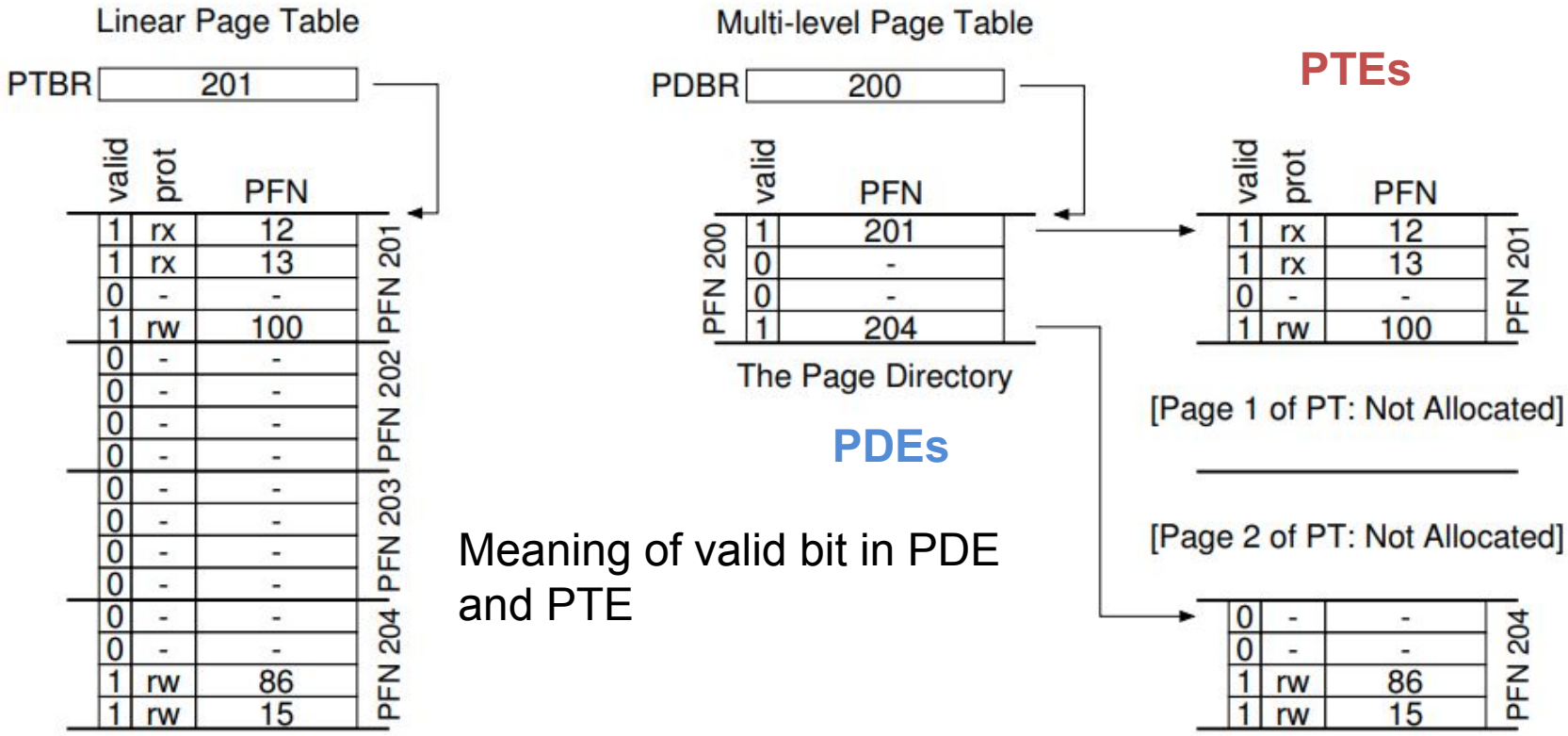
Multilevel Page Tables

Goal: Allow page table to be allocated non-contiguously

Idea: Page the page tables!

- Creates multiple levels of page tables; outer level “page directory”
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)

Multilevel Page Table – Key Idea



Meaning of valid bit in PDE and PTE

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 Memory Mechanisms

First, hardware checks TLB for virtual address

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

Else //TLB miss

- Hardware or OS walk page tables
- If PTE designates page is present, then page in physical memory

Else //Page fault

- Trap into OS (not handled by hardware)
- OS selects victim page in memory to replace
  - Write victim page out to disk if modified (use dirty bit in PTE)
- OS reads referenced page from disk into memory
- Page table is updated, present bit is set
- Process continues execution

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 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

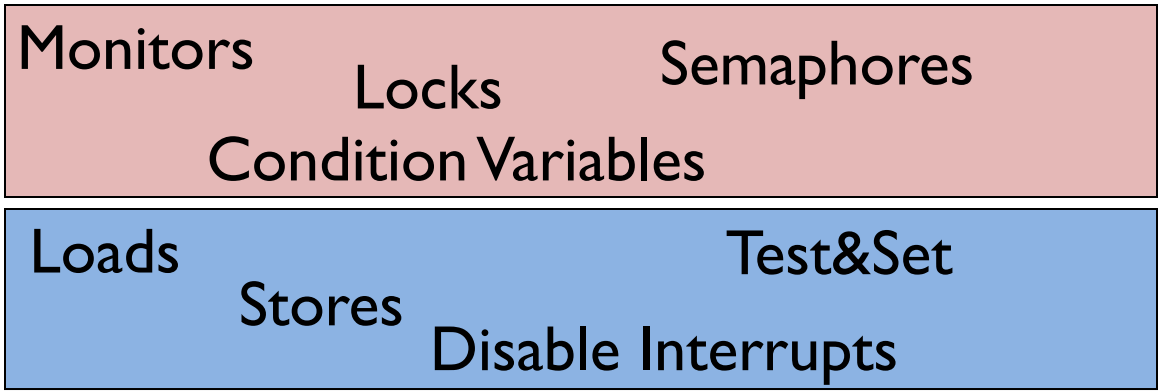
- Page selection
  When should a page (or pages) on disk be **brought into** memory?

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

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

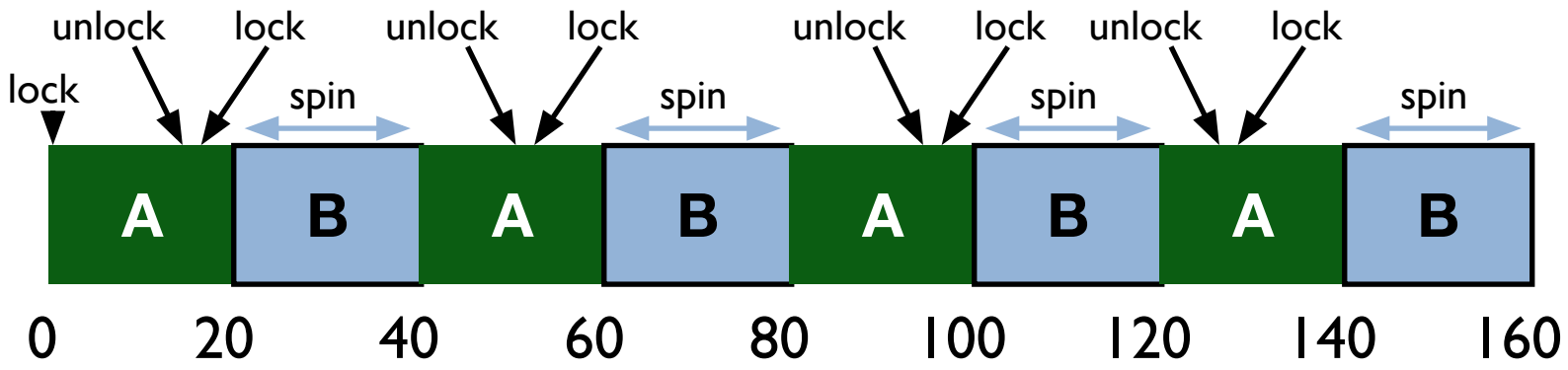


LOCK Implementation with XCHG

```
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);
}

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

int xchg(int *addr, int newval)

BASIC SPINLOCKS ARE UNFAIR



Scheduler is unaware of locks/unlocks!

B is unlucky - never is able to acquire lock

Lock Implementation: Block when Waiting

Remove waiting threads from scheduler READY queue

Move to BLOCKED state

Scheduler runs any thread that is READY

Support in Solaris OS: park(), unpark()

FINAL correct LOCK

```
typedef struct {
   bool lock = false;
   bool guard = false;
   queue_t q;
} LockT;
```

```
void acquire(LockT *l) {
   while (TAS(&l->guard, true));
   if (l->lock) {
         qadd(l->q, tid);
         setpark(); // notify of plan
         l->guard = false;
         park(); // unless unpark()
   } else {
         l->lock = true;
         l->guard = false;
   }
}
void release(LockT *l) {
   while (TAS(&l->guard, true));
   if (qempty(l->q)) l->lock=false;
   else unpark(qremove(l->q));
   l->guard = false;
}
```

setpark() fixes race condition
Park() does not block if unpark() occured after setpark()

Spin-Waiting vs Blocking

Each approach is better under different circumstances

Uniprocessor

- Waiting process is scheduled □ Process holding lock isn't
- Waiting process should always relinquish processor
- Associate queue of waiters with each lock (as in previous implementation)

Multiprocessor

- Waiting process is scheduled □ Process holding lock might be
- Spin or block depends on how long, t, before lock is released
  - Lock released quickly □ Spin-wait
  - Lock released slowly □ Block
  - Quick and slow are relative to context-switch cost, C

Producer/Consumer: Two CVs and WHILE

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

```
void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        while (numfull == 0)
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
```

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()

HOARE VS MESA SEMANTICS

- Mesa (used widely)
  - Signal puts waiter on ready list
  - Signaler keeps lock and processor
  - Not necessarily the waiter runs next
- Hoare (almost no one uses)
  - Signal gives processor and lock to waiter
  - Waiter runs when woken up by signaler
  - When waiter finishes, processor/lock given back to signaler

BINARY Semaphore (LOCK)

```
typedef struct __lock_t {
    sem_t sem;
} lock_t;

void init(lock_t *lock) {

}

void acquire(lock_t *lock) {

}

void release(lock_t *lock) {

}
```

sem_init(sem_t*, int initial)
sem_wait(sem_t*): Decrement, wait if value < 0
sem_post(sem_t*): Increment value
then wake a single waiter

Reader/Writer Locks

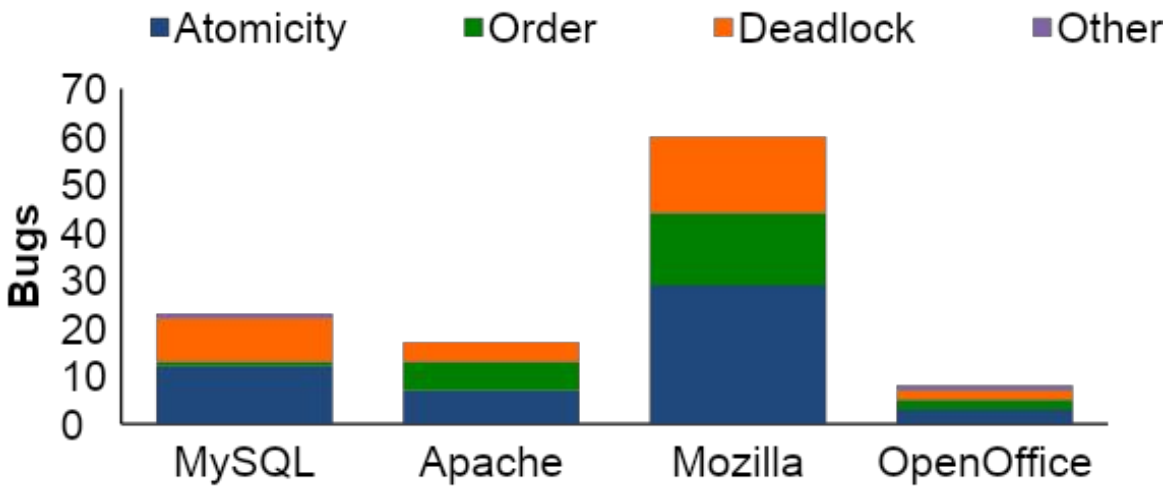
```
1 typedef struct _rwlock_t {
2     sem_t lock;
3     sem_t writelock;
4     int readers;
5 } rwlock_t;
6
7 void rwlock_init(rwlock_t *rw) {
8     rw->readers = 0;
9     sem_init(&rw->lock, 1);
10    sem_init(&rw->writelock, 1);
11 }
```

Reader/Writer Locks

```
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14       sem_wait(&rw->lock);
15       rw->readers++;
16       if (rw->readers == 1)
17           sem_wait(&rw->writelock);
18       sem_post(&rw->lock);
19 }
21 void rwlock_release_readlock(rwlock_t *rw) {
22       sem_wait(&rw->lock);
23       rw->readers--;
24       if (rw->readers == 0)
25           sem_post(&rw->writelock);
26       sem_post(&rw->lock);
27 }
29 rwlock_acquire_writelock(rwlock_t *rw) { sem_wait(&rw->writelock); }
31 rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }
```

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T1: release_readlock()
    // who runs?
T4: acquire_readlock()
    // what happens?
T5: acquire_readlock()
    // where blocked?
T3: release_writelock()
    // what happens next?

CONCURRENCY STUDY



Lu *etal.* [ASPLOS 2008]:

For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

DEADLOCK THEORY

Deadlocks can only happen with these four conditions:

1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

Can eliminate deadlock by eliminating any one condition