

# CONCURRENCY: DEADLOCK + REVIEW

Andrea Arpaci-Dusseau  
CS 537, Fall 2019

# ADMINISTRIVIA

Project 5 available now (xv6 Memory)

- Topic of Discussion Sections tomorrow
- Due Monday 11/4 (5 pm)
- Request new project partner if desired (web form)
- Turn in any of 3 versions:
  - v0 (alloc alternating pages, all marked as UNKNOWN PID)
  - v1 (alternating pages, some marked UNKNOWN, most known PIDs)
  - v2 (contiguous allocations when possible, some marked UNKNOWN, most known PIDs)

Midterm 2: Nov 11/6 (Wed) from 7:30-9:30pm

- Practice exams available
- Rooms on Canvas
- Mostly Concurrency
  - + Some Virtualization (usually repeated from Midterm I)

**RECAP**

# CONCURRENCY OBJECTIVES

**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*

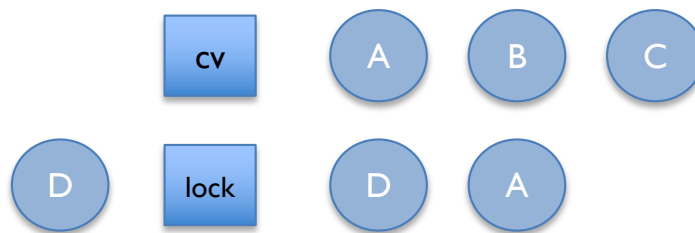
# CONDITION VARIABLES

**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  $\geq 1$  thread is waiting)
- if there is no waiting thread, just return, doing nothing



signal(cv) - what happens?

release(lock) - what happens?

# MONITORS

## Motivation

- Users can inadvertently misuse locks and semaphores (e.g., never unlock a mutex)

## Idea

- Provide language support to automatically lock and unlock monitor lock when in critical section
  - Lock is added implicitly; never seen by user
- Provide condition variables for scheduling constraints (zero or more)

## Examples

- Mesa language from Xerox
- Java: Acquire monitor lock when call **synchronized** methods in class

```
synchronized deposit(int amount) {  
    // language adds lock.acquire()  
    balance += amount;  
    // language adds lock.release()  
}
```

# INTRODUCING SEMAPHORES

Condition variables have no **state** (other than waiting queue)

- Programmer must track additional state

Semaphores have state: **track integer value**

- State cannot be directly accessed by user program,  
but state determines behavior of semaphore operations

# SUMMARY: SEMAPHORES

Semaphores are equivalent to locks + condition variables

- Can be used for both mutual exclusion and ordering

Semaphores contain **state**

- How they are initialized depends on how they will be used
- Init to 0: Join (1 thread must arrive first, then other)
- Init to N: Number of available resources

`sem_wait()`: Waits until value  $> 0$ , decrement value

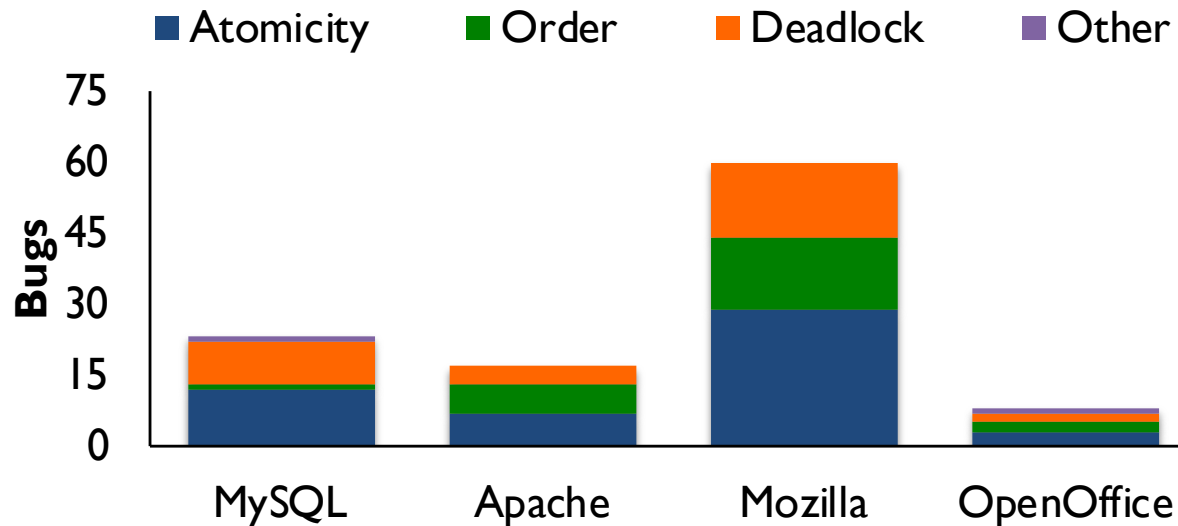
`sem_post()`: Increment value, if value  $> 0$ , wake a single waiter (atomic)

Can use semaphores in producer/consumer and for reader/writer locks



# CONCURRENCY BUGS

# CONCURRENCY STUDY



## **Lu *etal.* [ASPLOS 2008]:**

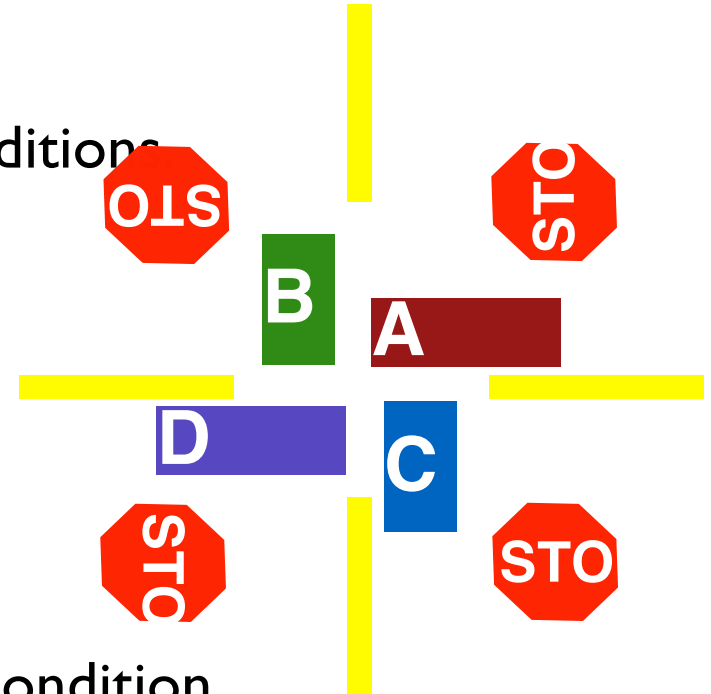
For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

# 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



# 1. MUTUAL EXCLUSION

Problem: Threads claim exclusive control of resources that they require

Strategy: Eliminate locks!

Try to replace locks with atomic primitive:

```
int CompAndSwap(int *addr, int expected, int new)  
Returns 0 fail, 1 success
```

# WAIT-FREE ALGORITHMS

```
void add (int *val, int amt)
{
    Mutex_lock(&m);
    *val += amt;
    Mutex_unlock(&m);
}
```

```
void add (int *val, int amt) {
    do {
        int old = *val;
    } while(!CompAndSwap(val, , old+amt));
}
```

# WAIT-FREE ALGORITHMS

```
void add (int *val, int amt) {  
    do {  
        int old = *val;  
    } while(!CompAndSwap(val, old, old+amt);  
}
```

**Val = 10;**

**T1: add(&val, 2);**

Old = 10;

\*Val != 10 → return false;

Old = 13;

\*Val == 13 → \*val=15; return true

**T2: add(&val, 3);**

Old = 10;

\*Val == 10 → \*val = 13; return true;

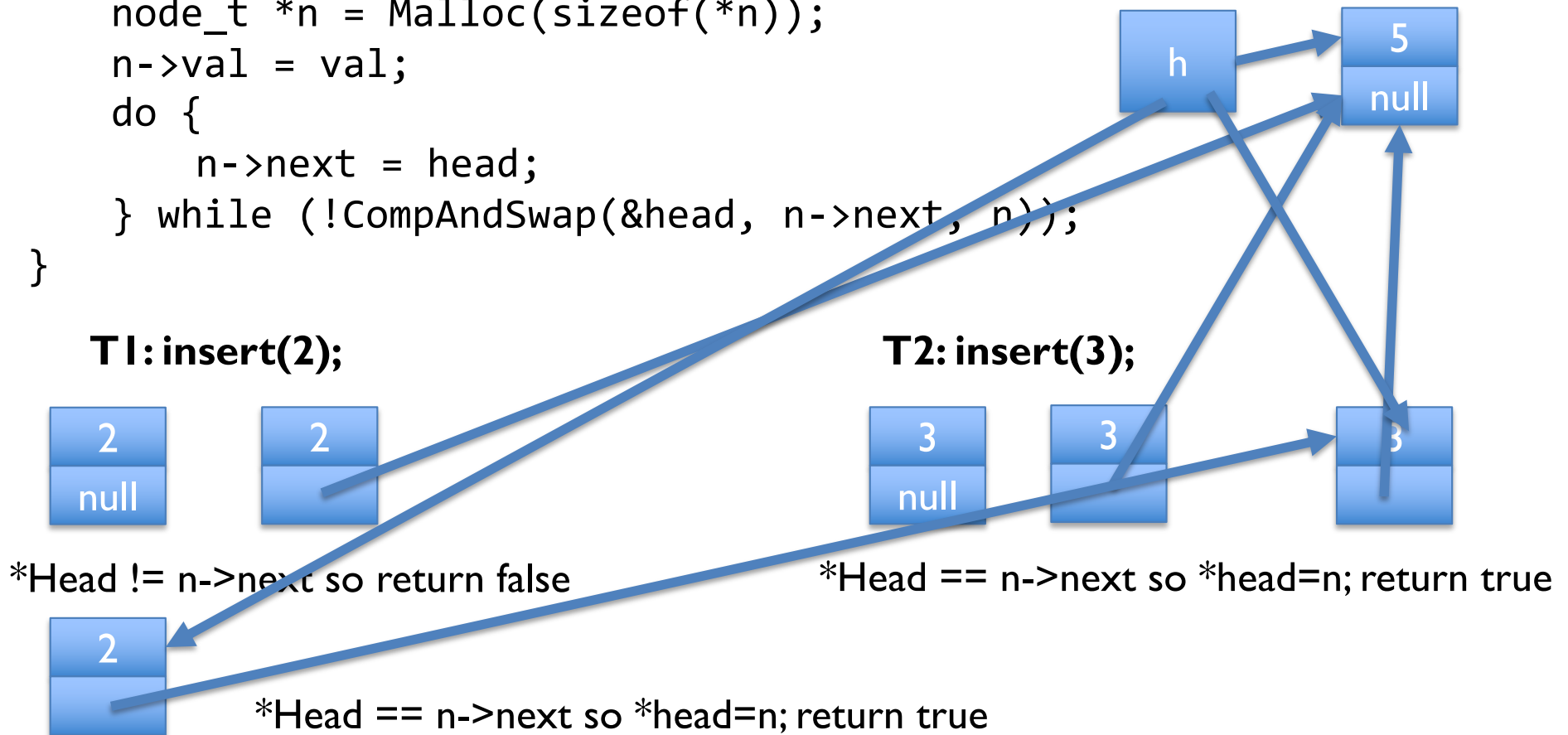
# WAIT-FREE ALGORITHM: LINKED LIST INSERT

```
void insert (int val) {  
    node_t *n = Malloc(sizeof(*n));  
    n->val = val;  
    lock(&m);  
    n->next = head;  
    head = n;  
    unlock(&m);  
}
```

```
void insert (int val) {  
    node_t *n = Malloc(sizeof(*n));  
    n->val = val;  
    do {  
        n->next = head;  
    } while (!CompAndSwap(&head,  
                           n->next, n));  
}
```

# WAIT-FREE ALGORITHM: LINKED LIST INSERT

```
void insert (int val) {  
    node_t *n = Malloc(sizeof(*n));  
    n->val = val;  
    do {  
        n->next = head;  
    } while (!CompAndSwap(&head, n->next, n));  
}
```





## 2. HOLD-AND-WAIT

Problem: Threads hold resources while waiting for additional resources

Strategy: Acquire all locks atomically **once**. Can release locks over time, but cannot acquire any again until all have been released

How? Use a meta lock:

```
lock(&meta);  
lock(&L1);  
lock(&L2);  
lock(&L3);  
...  
unlock(&meta);  
// CS1  
unlock(&L1);  
// CS 2  
Unlock(&L2);
```

```
lock(&meta);  
lock(&L2);  
lock(&L1);  
unlock(&meta);  
  
// CS1  
unlock(&L1);  
  
// CS2  
Unlock(&L2);
```

```
lock(&meta);  
lock(&L1);  
unlock(&meta);  
  
// CS1  
unlock(&L1);
```

Disadvantages?

## 2. HOLD-AND-WAIT

Must know ahead of time which locks will be needed

Must be conservative (acquire any lock possibly needed)

Degenerates to just having one big lock

```
lock(&meta);  
lock(&L1);  
lock(&L2);  
lock(&L3);  
...  
unlock(&meta);  
// CS1  
unlock(&L1);  
// CS 2  
Unlock(&L2);
```

```
lock(&meta);  
lock(&L2);  
lock(&L1);  
unlock(&meta);  
  
// CS1  
unlock(&L1);  
  
// CS2  
Unlock(&L2);
```

```
lock(&meta);  
lock(&L1);  
unlock(&meta);  
  
// CS1  
unlock(&L1);
```

# 3. NO PREEMPTION

Problem: Resources (e.g., locks) cannot be forcibly removed from threads that are

Strategy: if thread can't get what it wants, release what it holds

top:

```
lock(A);
```

```
if (trylock(B) == -1) {
```

```
    unlock(A);
```

```
    goto top;
```

```
}
```

```
...
```

Disadvantages?

Livelock:

No processes make progress, but the state of involved processes constantly changes

Classic solution: Exponential back-off

## 4. CIRCULAR WAIT

Circular chain of threads such that each thread holds a resource (e.g., lock) being requested by next thread in the chain.

Strategy:

- decide which locks should be acquired before others
- if A before B, never acquire A if B is already held!
- document this, and write code accordingly

Works well if system has distinct layers

# LOCK ORDERING IN XV6

Creating a file requires simultaneously holding:

- a lock on the directory,
- a lock on the new file's inode,
- a lock on a disk block buffer,
- idelock,
- ptable.lock

Always acquires locks in order listed

Linux has similar rules...

# SUMMARY

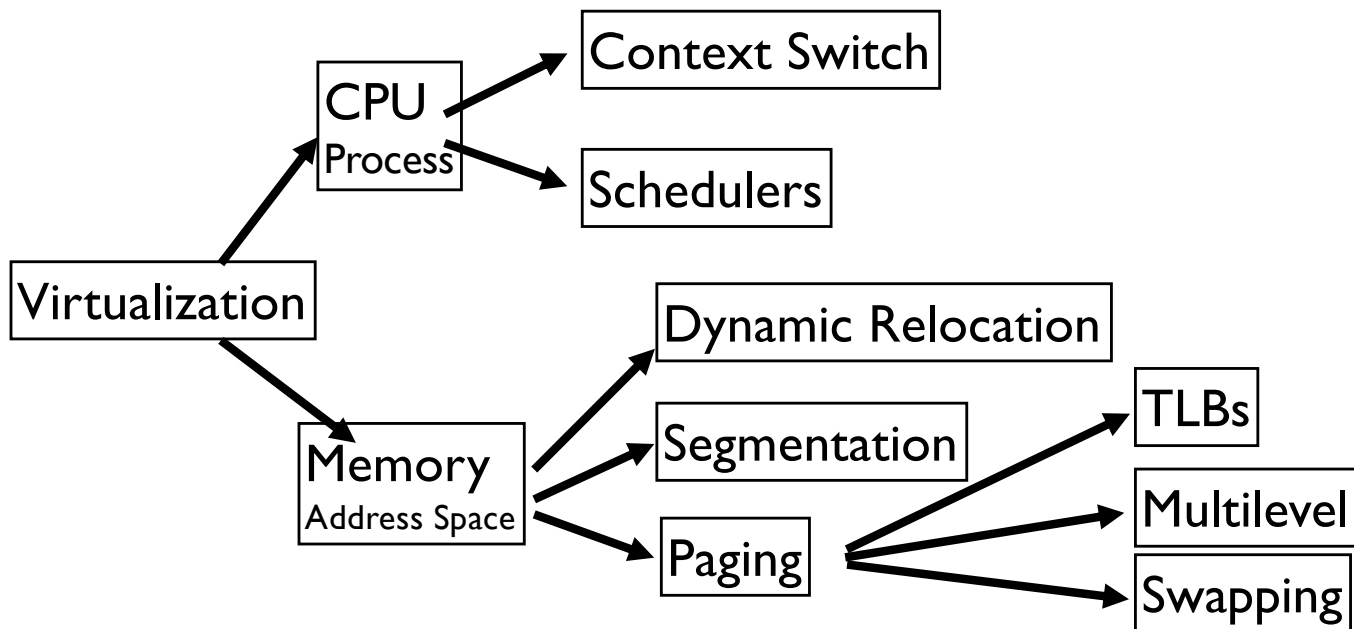
When in doubt about **correctness**, better to limit concurrency  
(i.e., add unnecessary locks, one big lock)

Concurrency is hard, encapsulation makes it harder!

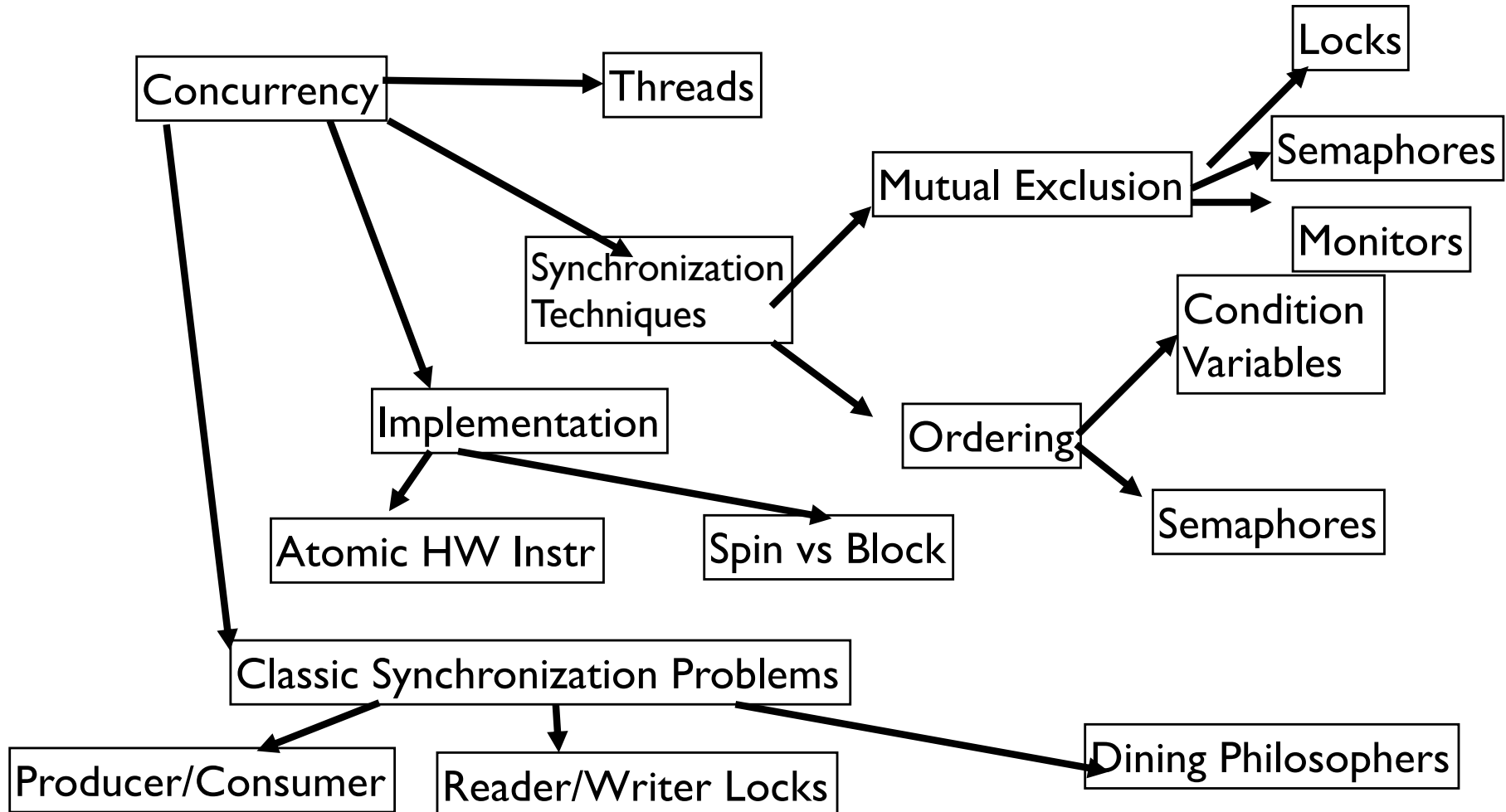
Have a strategy to avoid deadlock and stick to it

Choosing a lock order is probably most practical for reasonable performance

# 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?

4

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

3

# 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?

3

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

*200. 'result' is a local variable allocated on the stack; each thread has its own private copy which only it increments, therefore there are no race conditions.*

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

*Unknown. balance is allocated on the heap and shared between the two threads that are each accessing it without locks; there is a race condition.*


# SAMPLE HOMEWORK: HW-THREADSINTRO


```
./x86.py -p looping-race-nolock.s -t 2 -r -i 3
# assumes %bx has loop count in it
.main
.top
mov 2000, %ax # get the value at the address
add $1, %ax   # increment it
mov %ax, 2000 # store it back


# see if we're still looping
sub $1, %bx
test $0, %bx
jgt .top
halt
```


# LOOPING-RACE-NOLOCKS.S (ADDR 2000 HAS 0)


Thread 0	Thread 1
1000 mov 2000, %ax	
1001 add \$1, %ax	
----- Interrupt -----	----- Interrupt -----
	1000 mov 2000, %ax
----- Interrupt -----	----- Interrupt -----
1002 mov %ax, 2000	
----- Interrupt -----	----- Interrupt -----
	1001 add \$1, %ax
	1002 mov %ax, 2000
	1003 sub \$1, %bx
----- Interrupt -----	----- Interrupt -----
1003 sub \$1, %bx	
----- Interrupt -----	----- Interrupt -----
	1004 test \$0, %bx
----- Interrupt -----	----- Interrupt -----
1004 test \$0, %bx	
----- Interrupt -----	----- Interrupt -----
	1005 jgt .top
----- Interrupt -----	1000 mov 2000, %ax
	----- Interrupt -----
1005 jgt .top	
1000 mov 2000, %ax	
1001 add \$1, %ax	
----- Interrupt -----	----- Interrupt -----
	1001 add \$1, %ax
----- Interrupt -----	1002 mov %ax, 2000
	----- Interrupt -----
1002 mov %ax, 2000	


Contents of ax = 


Contents of ax = 


Contents of ax = 


50) Contents of addr = 


Contents of ax = 


51) Contents of addr = 

Contents of ax = 

Contents of ax = 

Contents of ax = 

52) Contents of addr = 

53) Contents of addr = 

# WALKING THROUGH CODE BEHAVIOR

The scheduler runs each thread in the system such that each **line** of the given C-language code executes in one scheduler tick, or interval. For example, if there are two threads in the system, T and S, we tell you which thread was scheduled in each tick by showing you either a “T” or a “S” to designate that **one line** of C-code was scheduled for the corresponding thread; for example, TTTSS means that 3 lines of C code were run from thread T followed by 2 lines from thread S.

Some lines of C code require special rules, as follows.

Assume each **test** of a `while()` loop or an `if()` statement requires one scheduler tick. Assume jumping to the correct code does not take an additional tick (e.g., jumping either inside or outside the while loop or back to the while condition does not take an extra tick; jumping to the **then** or the **else** branch of an **if** statement does not take an extra tick).

Assume **function calls** whose internals are not shown and that do not require synchronization, such as `qadd()`, `qremove()`, `qempty()`, and `malloc()`, require one scheduling tick.

Function calls that may need to **wait for another thread** to do something (e.g., `mutex_lock()` and `cond_wait()`) may consume an arbitrary number of scheduling ticks and are treated as follow.

For **`mutex_lock()`**, assume that the function call to `mutex_lock()` requires one scheduling interval if the lock is available. If the lock is not available, assume the call spin-waits until the lock is available (e.g., you may see a long instruction stream TTTTTTT that causes no progress for this thread). Once the lock is available, the next scheduling of the acquiring thread causes that thread to obtain the lock (e.g., after a thread S releases the lock, the next scheduling of the waiting thread T will complete `mutex_lock()`; note that T does need to be scheduled for one tick with the lock released for `mutex_lock()` to complete).

The rules for **`cond_wait()`** and **`sema_wait()`** are similar. When a thread calls one of these versions of `wait()`, if the work has not yet been done to complete the `wait()`, then no matter how long the scheduler runs this thread (e.g., TTTTT), this thread will remain waiting in the `wait()` routine. After another thread runs and does the work necessary for the `wait()` routine to complete, then the next scheduling of thread T will cause the `wait()` line to complete; again, note that T does need to be scheduled for one tick with the work completed for `wait()` to complete).

```

void thread_join() {
    Mutex_lock(&m);           // p1
    while (done == 0)         // p2
        Cond_wait(&c, &m);    // p3
    Mutex_unlock(&m);         // p4
}
void thread_exit() {
    Mutex_lock(&m);           // c1
    done = 1;                 // c2
    Cond_signal(&c);          // c3
    Mutex_unlock(&m);         // c4
}

```

After the instruction stream “P” (i.e., after scheduler runs one line from parent), which line of the parent’s will execute when it is scheduled again?

**p2.** *P will have acquired lock and finished mutex\_lock().*

Assume the scheduler continues on with “C” (the full instruction stream is PC). Which line will child execute when it is scheduled again?

*C1. C must wait to acquire the lock since it is currently held by P.*

After PPP (full is PCPPP), which line for parent next?

*p3. Since done = 0, P will execute p2 and p3; it is stuck in p3 until signaled.*

After CCC (full is PCPPPCCC), which line for child next?

*c4. C finishes c1, c2, c3. Next time it executes c4. CV signaled but mutex still locked*

After PP (full is PCPPPCCCPP), which line for parent next?

*p3. P cannot return from cond\_wait() until it acquires lock held by C; P stays in p3*

After CC (full is PCPPPCCCPPCC), which line for child next?

*Beyond. C executes c4 and then code beyond c4.*

After PPP (full PCPPPCCCPPCCPPP), which line for parent next?

*Beyond. P finishes p3, rechecks p2, then p4. Next time beyond p4.*

# TICKET LOCK WITH YIELD

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

Remember: `yield()` voluntarily relinquishes CPU for remainder of timeslice, but process remains READY

# BLOCK WHEN WAITING: FINAL CORRECT LOCK

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

setpark() fixes race condition

Park() does not block if unpark()  
occurred after setpark()

```
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;  
}
```



```

sem_t mayEat[5]; // how to initialize?
sem_t mutex;     // how to init?
int state[5] = {THINKING};
take_chopsticks(int i) {
    wait(&mutex); // enter critical section
    state[i] = HUNGRY;
    testSafetyAndLiveness(i); // check if I can run
    signal(&mutex); // exit critical section
    wait(&mayEat[i]);
}
put_chopsticks(int i) {
    wait(&mutex); // enter critical section
    state[i] = THINKING;
    test(i+1 %5); // check if neighbor can run now
    test(i+4 %5);
    signal(&mutex); // exit critical section
}
testSafetyAndLiveness(int i) {
    if(state[i]==HUNGRY&&state[i+4%5]!=EATING&&state[i+1%5]!=EATING) {
        state[i] = EATING;
        signal(&mayEat[i]);
    } }

```

# 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()  
T4: acquire\_readlock()  
// what happens?

```

Acquire_readlock() {
    Sem_wait(&mutex);
    If (ActiveWriters +
        WaitingWriters==0) {
        sem_post(OKToRead);
        ActiveReaders++;
    } else WaitingReaders++;
    Sem_post(&mutex);
    Sem_wait(OKToRead);
}

Release_readlock() {
    Sem_wait(&mutex);
    ActiveReaders--;
    If (ActiveReaders==0 &&
        WaitingWriters > 0) {
        ActiveWriters++;
        WaitingWriters--;
        Sem_post(OKToWrite);
    }
    Sem_post(&mutex);
}

```

```

Acquire_writelock() {
    Sem_wait(&mutex);
    If (ActiveWriters + ActiveReaders + WaitingWriters==0) {
        ActiveWriters++;
        sem_post(OKToWrite);
    } else WaitingWriters++;
    Sem_post(&mutex);
    Sem_wait(OKToWrite);
}

Release_writelock() {
    Sem_wait(&mutex);
    ActiveWriters--;
    If (WaitingWriters > 0) {
        ActiveWriters++;
        WaitingWriters--;
        Sem_post(OKToWrite);
    } else while(WaitingReaders>0) {
        ActiveReaders++;
        WaitingReaders--;
        sem_post(OKToRead);
    }
    Sem_post(&mutex);
}

```

T1: acquire\_readlock()  
 T2: acquire\_readlock()  
 T3: acquire\_writelock()  
 T4: acquire\_readlock()  
 // what happens?  
 ... release\_readlock() x 2  
 T3: release\_writelock()