CS 423
Operating System Design: Semaphores and Deadlocks
Apr 2

Ram Kesavan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

Logistics

Restarted C4 paper reading

MP3 due by Apr 15, 11:59 CT

MP2 grades have been released

- Any one of MP2 or MP3 can be resubmitted
- Deadline for MP2 resubmission: Apr 7 11:59 CT

MP2 interviews requests have been emailed out

Midterm regrade deadline: Apr 7 11:59 CT

AGENDA / LEARNING OUTCOMES

Semaphores

Equivalence: Build…

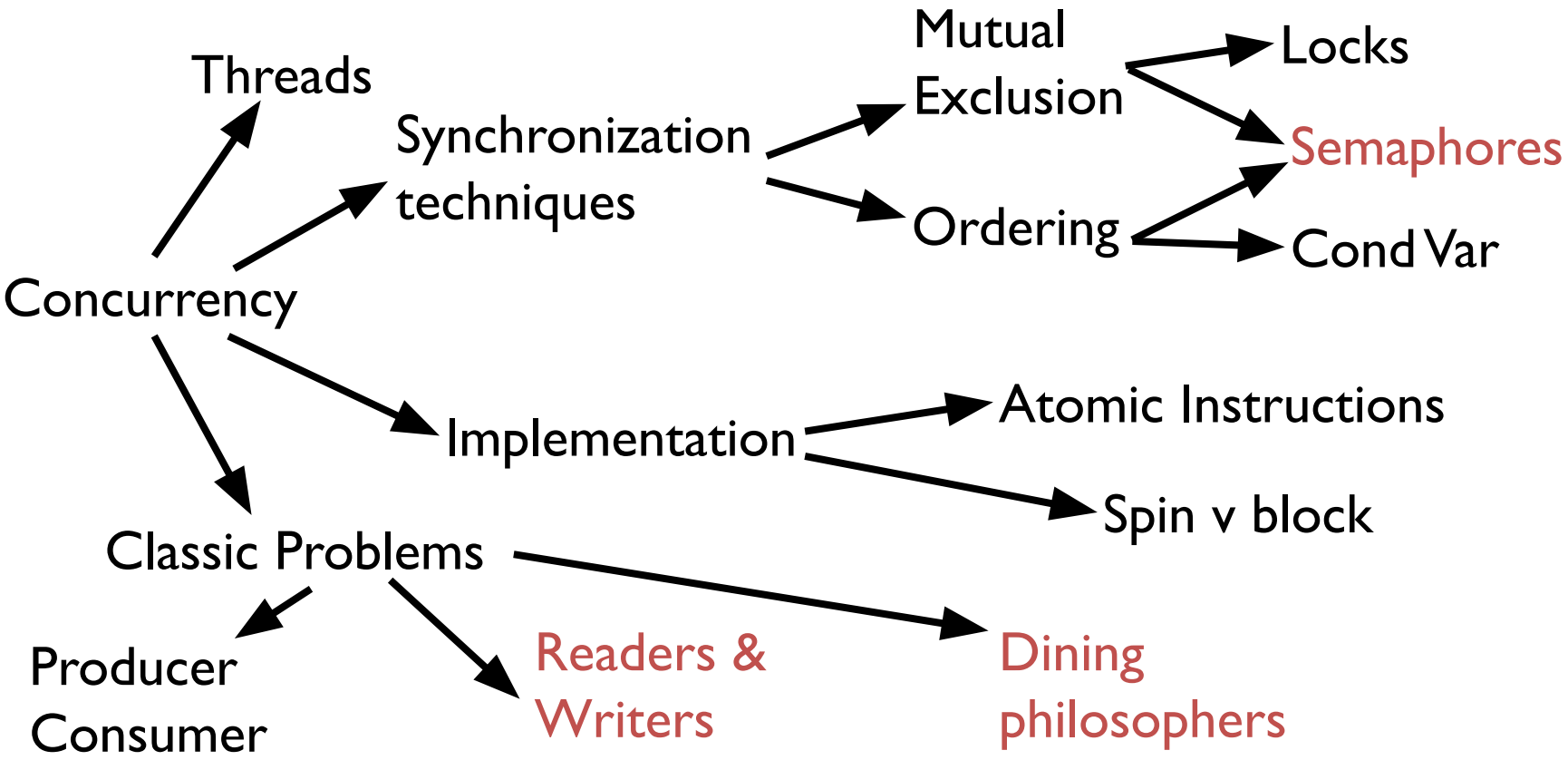
- mutex using semaphore
- cv using semaphore
- semaphore using mutex & cv

Producer/Consumer with semaphores

Concurrency Bugs

RECAP

CONCURRENCY: EASY PIECE 2



Semaphores

Condition variables have no state (other than waiting queue)

- Programmer must track additional state

Semaphores have state: integer value

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

Equivalence

Semaphores are equally powerful to Locks+CVs

- what does this mean?

One might be more convenient, but that's not relevant

Equivalence means each can be built from the other

END RECAP

SEMAPHORE OPERATIONS

Allocate and Initialize

```
#include <semaphore.h>
sem_t s;
sem_init(&s, xx, 1);        // ignore 2nd arg
sem_init(sem_t *s, int initval) {
  s->value = initval;
}
```

User cannot read or write value directly after initialization

SEMAPHORE OPERATIONS

sem_wait(sem_t*)

Decrement sem value. Wait if value of sem is negative

If value < 0, then equals #waiters

sem_post(sem_t*)

Increment sem value. If waiters exist, wake one of them up

Wait and Post are atomic

aka P & V, down & up, respectively

BINARY Semaphore (MUTEX)

```
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_value)
sem_wait(sem_t*): decrement, wait if value < 0
sem_post(sem_t*): increment, wake up one waiter

BINARY Semaphore (LOCK)

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

void init(lock_t *lock) {
    sem_init(&lock_t->sem, 1);
}

void acquire(lock_t *lock) {
    sem_wait(&lock_t->sem);
}

void release(lock_t *lock) {
    sem_post(&lock_t->sem);
}
```

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

thread_join using Semaphores

```
void thread_join() {
    mutex_lock(&m);       // w
    while (done == 0)     // x
      cond_wait(&c, &m); // y
    mutex_unlock(&m);     // z
}
```

```
void thread_exit() {
    mutex_lock(&m);       // a
    done = 1;             // b
    cond_signal(&c);      // c
    mutex_unlock(&m);     // d
}
```

---

sem_t s;
sem_init(&s, _0_);

sem_wait(): decrement, wait if value < 0
sem_post(): increment, wake a single waiter

```
void thread_join() {
        sem_wait(&s);
}
```

```
void thread_exit() {
        sem_post(&s)
}
```

Producer/Consumer: ATTEMPT#1

Single producer thread, single consumer thread
Single shared buffer between producer and consumer

Use 2 semaphores

- sem_init(&emptyBuffer, __1__);
- sem_init(&fullBuffer, __0__);

```
Producer
while (1) {
    sem_wait(&emptyBuffer);
    produce(&buffer);
    sem_post(&fullBuffer);
}
```

```
Consumer
while (1) {
    sem_wait(&fullBuffer);
    consume(&buffer);
    sem_post(&emptyBuffer);
}
```

Producer/Consumer: ATTEMPT#2

Single producer thread, single consumer thread
Shared (circular) buffer with **N** elements between producer and consumer
Use 2 semaphores

- sem_init(&emptyBuffer, __N__);
- sem_init(&fullBuffer, __0__);

```
Producer
i = 0;
while (1) {
    sem_wait(&emptyBuffer);
    produce(&buffer[i]);
    i = (i+1)%N;
    sem_post(&fullBuffer);
}
```

```
Consumer
j = 0;
While (1) {
    sem_wait(&fullBuffer);
    consume(&buffer[j]);
    j = (j+1)%N;
    sem_post(&emptyBuffer);
}
```

Producer/Consumer: ATTEMPT#3

Final case:

- Multiple producer threads, multiple consumer threads
- Shared buffer with N elements between producer and consumer

Requirements

- Each C thread must consume from a unique filled element
- Each P thread must produce into a unique empty element

P/C Multiple Threads

```
Producer
while (1) {
    sem_wait(&emptyBuffer);
    myi = findempty(&buffer);
    produce(&buffer[myi]);
    sem_post(&fullBuffer);
}
```

```
Consumer
while (1) {
    sem_wait(&fullBuffer);
    myj = findfull(&buffer);
    consume(&buffer[myj]);
    sem_post(&emptyBuffer);
}
```

We need private `myi` and `myj` for each thread
Where is mutual exclusion needed?

P/C Multiple Threads: ATTEMPT#1

Consider 3 possible locations for mutual exclusion

```
Producer
    sem_wait(&mutex);
    sem_wait(&emptyBuffer);
    myi = findempty(&buffer);
    produce(&buffer[myi]);
    sem_post(&fullBuffer);
    sem_post(&mutex);
```

```
Consumer
    sem_wait(&mutex);
    sem_wait(&fullBuffer);
    myj = findfull(&buffer);
    consume(&buffer[myj]);
    sem_post(&emptyBuffer);
    sem_post(&mutex);
```

QUESTION: Is it correct? Is it efficient?

P/C Multiple Threads: ATTEMPT#2

```
Producer
    sem_wait(&emptyBuffer);
    sem_wait(&mutex);
    myi = findempty(&buffer);
    produce(&buffer[myi]);
    sem_post(&mutex);
    sem_post(&fullBuffer);
```

```
Consumer
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    myj = findfull(&buffer);
    consume(&buffer[myj]);
    sem_post(&mutex);
    sem_post(&emptyBuffer);
```

QUESTION: Is it correct? Is it efficient?

P/C Multiple Threads: ATTEMPT#2

```
Producer
    sem_wait(&emptyBuffer);
    sem_wait(&mutex);
    myi = findempty(&buffer);
    produce(&buffer[myi]);
    sem_post(&mutex);
    sem_post(&fullBuffer);
```

```
Consumer
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    myj = findfull(&buffer);
    consume(&buffer[myj]);
    sem_post(&mutex);
    sem_post(&emptyBuffer);
```

Correct but limits concurrency

Only 1 thread at a time can be producing or consuming a buffer

P/C Multiple Threads: ATTEMPT#3

```
Producer #3
    sem_wait(&emptyBuffer);
    sem_wait(&mutex);
    myi = findempty(&buffer);
    sem_post(&mutex);
    produce(&buffer[myi]);
    sem_post(&fullBuffer);
```

```
Consumer #3
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    myj = findfull(&buffer);
    sem_post(&mutex);
    consume(&buffer[myj]);
    sem_post(&emptyBuffer);
```

Correct and increases concurrency; shrunk critical section to finding buffer
Producing or consuming different buffers can run in parallel

General Rule

What should we initialize a semaphore to?

General Rule

What should we initialize a semaphore to?

Set it to the #resources being handed out

- Lock - 1: there's only 1 resource being acquired
- CV - 0: there's nothing to acquire
- Producer/Consumer - set to N: #buffers to fill or consume

Reader/Writer Locks

Let multiple reader threads grab lock (shared)

Only one writer thread can grab lock (exclusive)

- No reader threads
- No other writer threads

OSTEP Ch31: (sub-optimal) implementation using semaphores

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?

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: release_readlock()
// what happens?
// what's the problem?

Build Zemaphore With LOCK + CV

```
typedef struct {
    int value;
    cond_t cond;
    lock_t lock;
} sem_t;

void sem_init(sem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    lock_init(&s->lock);
}
```

OSTEP: **Zemaphore**!

Build Zemaphore With LOCK + CV

```
sem_wait(sem_t *s) {
    lock_acquire(&s->lock);
    while (s->value <= 0)
        cond_wait(&s->cond,
                  &s->lock);
    s->value--;
    lock_release(&s->lock);
}
```

```
sem_post(sem_t *s) {
    lock_acquire(&s->lock);
    s->value++;
    cond_signal(&s->cond);
    lock_release(&s->lock);
}
```

sem_wait(): Wait while value <= 0, decrement

Zemaphore: num-waiters != value

sem_post(): Increment value; wake a single waiter

Semaphores

Equivalence: Semaphore, lock, & cv

- Semaphore count initialized to #resources being controlled

Can use semaphores in producer/consumer, reader/writer locks, etc.

Linux kernel: Uses semaphore as a cv

/include/linux/semaphore.h, /kernel/locking/semaphore.c

```
struct semaphore {
    raw_spinlock_t      lock;
    unsigned int        count;
    struct list_head    wait_list;
    …
};
```

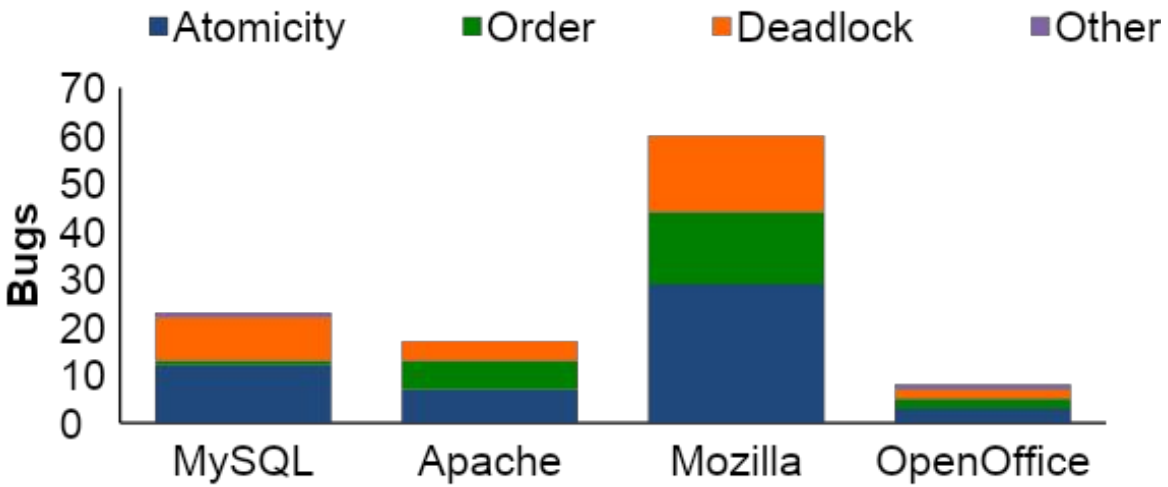
CONCURRENCY: PERFORMANCE

Spinlock vs. mutex, fairness, etc. already covered

- **Lock cost:** uncontended vs. contended acquisition
- **Lock granularity:** Fine-grained vs. coarse-grained locking
  - Ideally, lock acquisition < critical section time
- **CPU cache effects:**
  - High contention causes frequent cache invalidations (coherency traffic)
  - False sharing (lock & other data in the same cacheline) effects
    - Linux provides macros in linux/cache.h to pad & align data structures to cacheline
- **Mutex as Barrier:** Acts as a memory fence
  - Prevents compilers from optimizing code across that fence

CONCURRENCY BUGS

"The only thing worse that a bug that is hit all the time is a bug that isn't hit all the time"



Lu *et al.* [ASPLOS'08]:

Analyzed concurrency bugs in 4 major projects

FIX ATOMICITY BUGS WITH LOCKS

Thread 1:

```
if (thd->proc_info) {
   …
   fputs(thd->proc_info, …);
   …
}
```

Thread 2:

```
thd->proc_info = NULL;
```

FIX ORDERING BUGS WITH CONDITION VARIABLES

Thread 1:

```
void init() {
    …

    mThread =
        PR_CreateThread(mMain, …);

    …
}
```

Thread 2:

```
void mMain(…) {
  …

  mState = mThread->State;
  …
}
```

FIX ORDERING BUGS WITH CONDITION VARIABLES

Thread 1:

```
void init() {
    …
    mThread =
        PR_CreateThread(mMain, …);

    mutex_lock(&m);
    mtInit = 1;
    cond_signal(&cv);
    mutex_unlock(&m);
    …
}
```

Thread 2:

```
void mMain(…) {
  …

  mutex_lock(&m);
  while (mtInit == 0)
    cond_wait(&cv, &m);
  mutex_unlock(&m);

  mState = mThread->State;
  …
}
```

DEADLOCK

No progress possible because 2+ threads each waiting for another to take some action

CODE EXAMPLE

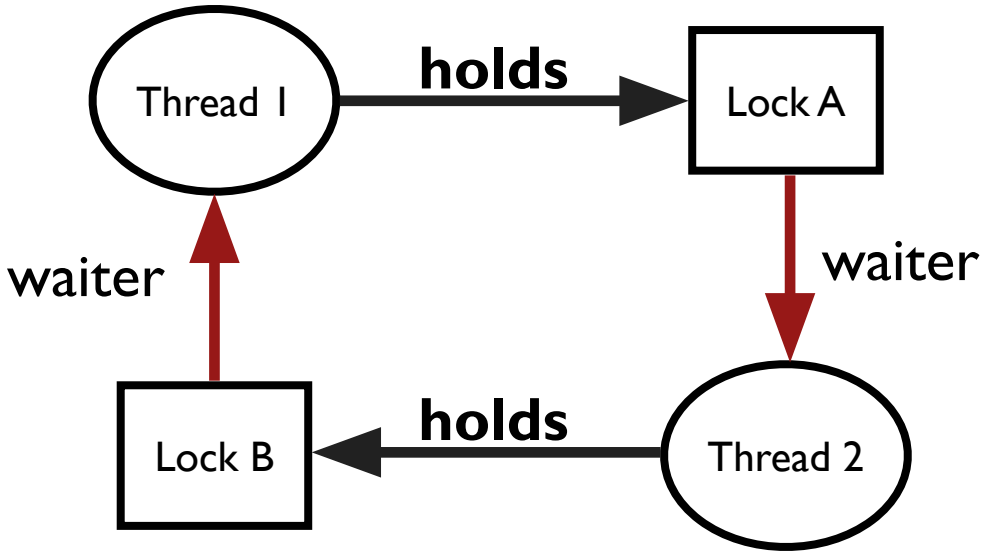
Thread 1:

```
lock(&A);
lock(&B);
```

Thread 2:

```
lock(&B);
lock(&A);
```

CIRCULAR DEPENDENCY



ENCAPSULATION/MODULARITY

```
set_t *set_intersection (set_t *s1, set_t *s2) {
     set_t *rset= malloc(sizeof(*rset));
     mutex_lock(&s1->lock);
     mutex_lock(&s2->lock);
     for (int i = 0; i < s1->len; i++) {
          if (set_contains(s2, s1->items[i])
               set_add(rset, s1->items[i]);
     mutex_unlock(&s2->lock);
     mutex_unlock(&s1->lock);
     return rset;
}
```

Can there be a deadlock?

DEADLOCK THEORY

Deadlocks require all four conditions:

1. Thread get exclusive access to resource
2. Thread holds resource & wait on other resource
3. Thread cannot be forced to give up a resource
4. Circular wait of threads on resources

Can avoid deadlock by eliminating one of them

1. EXCLUSIVE ACCESS

Strategy: Eliminate locks!

Try to replace locks with atomic H/W primitive:

```
int CompareAndSwap(int *address, int expected, int new) {
    if (*address == expected) {
        *address = new;
        return 1;  // success
    }
    return 0;      // failure
}
```

WAIT-FREE ADDITION

```
void add(*int value, int incr_amt) {
    mutex_lock(&m);
    *value += incr_amt;
    mutex_unlock(&m);
}
```

```
int CompareAndSwap(int *address, int
                   expected, int new) {
      if (*address == expected) {
            *address = new;
            return 1;    // success
   }
   return 0;        // failure
}
```

```
void add(*int value, int incr_amt) {
    do {
        int old = *value;
    } while (!CompareAndSwap(value, __, __));
}
```

WAIT-FREE ADDITION

```
void add(*int value, int incr_amt) {
    mutex_lock(&m);
    *value += incr_amt;
    mutex_unlock(&m);
}
```

```
int CompareAndSwap(int *address, int
                   expected, int new) {
      if (*address == expected) {
            *address = new;
            return 1;    // success
   }
   return 0;        // failure
}
```

```
void add(*int value, int incr_amt) {
    do {
        int old = *value;
    } while (!CompareAndSwap(value, old, old + incr_amt));
}
```

WAIT-FREE: LINKED LIST INSERTION

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

```
int CompareAndSwap(int *address, int
                   expected, int new) {
      if (*address == expected) {
            *address = new;
            return 1;    // success
   }
   return 0;        // failure
}
```

```
void enqueue(int val) {
  node_t *n = malloc(sizeof(*n));
  n->val = val;
  do {
    n->next = head;
  } while (!CompareAndSwap(&head,___,___));
}
```

WAIT-FREE: LINKED LIST INSERTION

```
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 (!CompareAndSwap(&head, n->next, n));
}
```

```
int CompareAndSwap(int *address, int
                   expected, int new) {
    if (*address == expected) {
        *address = new;
        return 1;    // success
    }
    return 0;        // failure
}
```

2. HOLD-AND-WAIT

Problem: Thread holds resources while waiting on additional resources

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

How to do this? 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);
```

3. THREADS DON'T GIVE UP LOCKS

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

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?

Solution?

4. CIRCULAR WAIT

Circular dependency of threads and resources being held & requested

Strategy:

- decide which locks should be acquired before others
- if A before B, can never wait on A after acquiring B
- document this, and write code accordingly

Many systems use this strategy; Linux as well

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…

CONCURRENCY SUMMARY

Motivation: Parallel programming patterns, multi-core machines

Abstractions, Mechanisms:

- Spin Locks, Ticket locks
- Condition variables
- Semaphores

Concurrency is difficult to get right!

Lock ordering to avoid deadlocks

Other systems use deadlocks detection + recovery (eg. databases)

Distributed systems add a different dimension to this problem

NEXT

Persistence!