ILLINOIS TECH

College of Computing

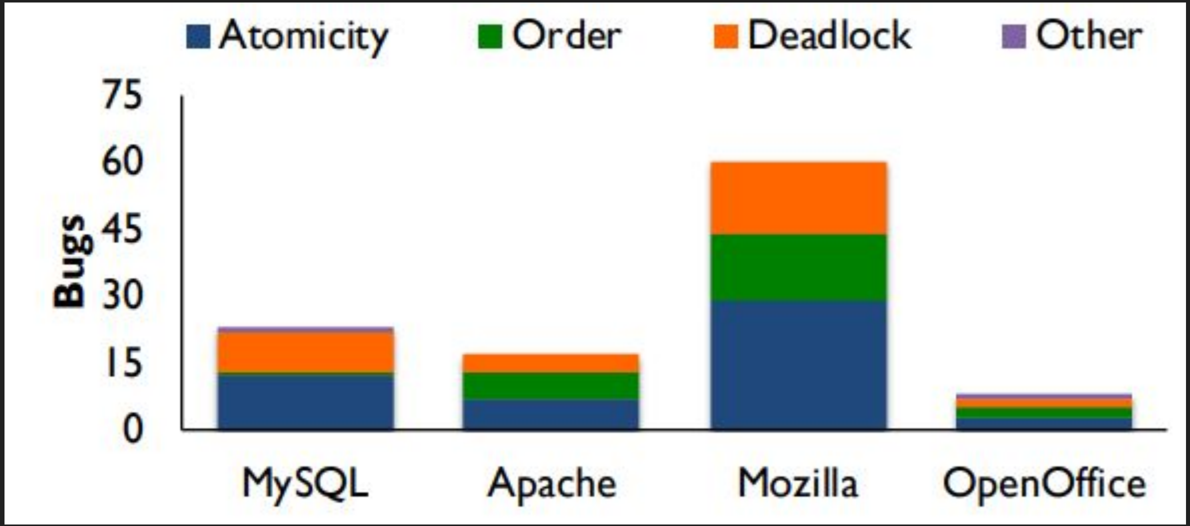
CS 450 Operating Systems
Concurrency Bugs

Yue Duan

Recap

- 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(): Decrement and waits **if** value < 0
- sem_post(): Increment value, then wake a single waiter (**atomic**)
- Use semaphores in
  - producer/consumer
  - reader/writer

Concurrency Bugs



- Lu et al. [ASPLOS 2008]:
  - For four major projects, search for concurrency bugs among >500K bug reports.
  - Analyze small sample to identify common types of concurrency bugs.

Atomicity: MySQL

Thread 1:

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

Thread 2:

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

- **Test (thd->proc_info != NULL) and set (writing to thd->proc_info) should be atomic**

Atomicity: MySQL

Thread 1:

```
pthread_mutex_lock(&lock);
if (thd->proc_info) {
    …
    fputs(thd->proc_info, …);
    …
}
pthread_mutex_unlock(&lock);
```

Thread 2:

```
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```

- Fix it with locks

Ordering: Mozilla

```
Thread 1:

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

```
Thread 2:

void mMain(…) {
  …

  mState = mThread->State;

  …
}
```

- What’s wrong?
  - Thread 1 sets value of mThread needed by Thread2
  - How to ensure that reading MThread happens after mThread initialization?

Ordering: Mozilla

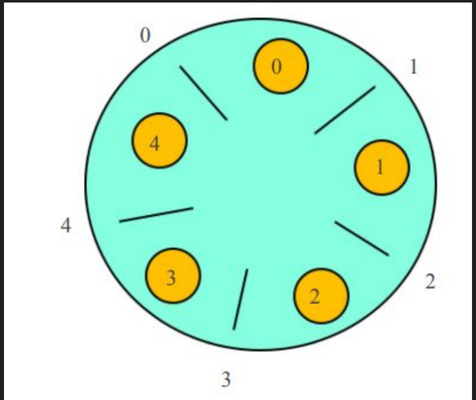
```
Thread 1:
void init() {
    …

    mThread =
    PR_CreateThread(mMain, …);

    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);

    …
```

```
Thread 2:

void mMain(…) {
  …

  mutex_lock(&mtLock);
  while (mtInit == 0)
    Cond_wait(&mtCond, &mtLock);
  Mutex_unlock(&mtLock);

  mState = mThread->State;
  …
}
```

- Fix ordering bugs with condition variables

Dining philosophers problem

- Five philosopher setting around a table
- Five forks
  - each between two philosophers
- Each philosopher will do:
  - think: no fork needed
  - get forks: two forks needed
  - eat
  - put down forks




```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

Dining philosophers problem

- Write the routines **get_forks**() and **put_forks**() such that:
  - no deadlock
  - no starvation
  - high concurrency
- Use five semaphores
  - one for each fork `sem_t forks[5]`
- Some helper functions

```
int left(int p)  { return p; }
int right(int p) { return (p + 1) % 5; }
```

Dining philosophers problem

- First attempt
  - initialize each semaphore to a value of **1**
  - assume that each philosopher knows its own number (**p**)

```
void get_forks(int p)  {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}

void put_forks(int p)  {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```

- Does it work?
  - no, but what's wrong?

Dining philosophers problem

- A solution: breaking the dependency
    - change how forks are acquired by **at least one** of the philosophers

```
void get_forks(int p) {
    if (p == 4) {
        sem_wait(&forks[right(p)]);
        sem_wait(&forks[left(p)]);
    } else {
        sem_wait(&forks[left(p)]);
        sem_wait(&forks[right(p)]);
    }
}
```

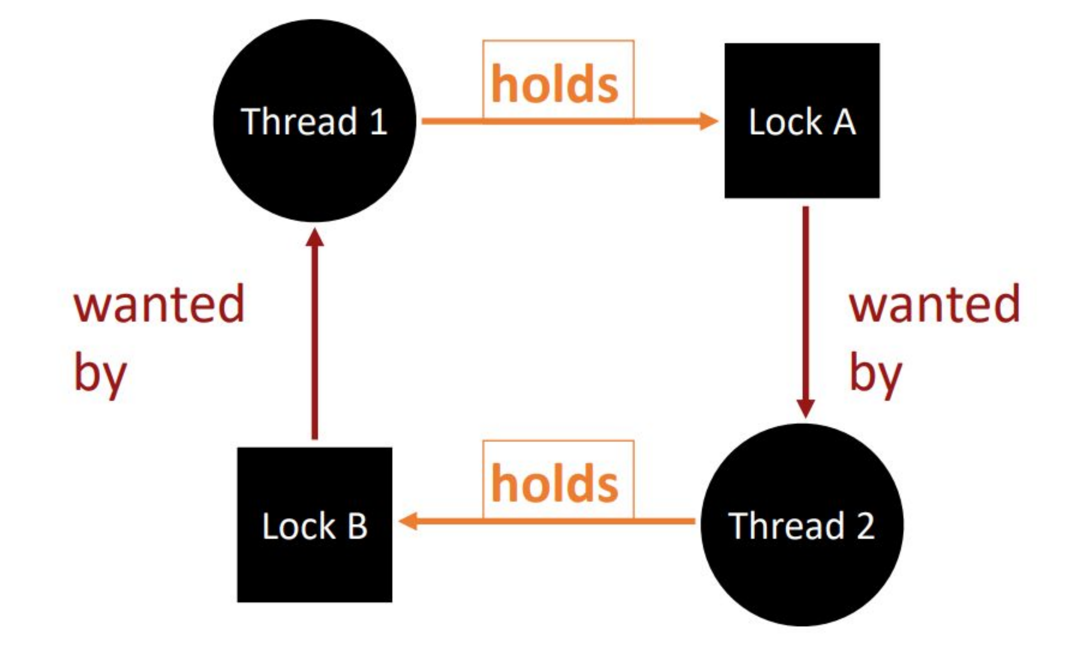
Deadlock

- No progress can be made because **two or more threads are waiting for the other** to take some action and thus neither ever does

```
Thread 1:                    Thread 2:

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

Deadlock

- Circular Dependency

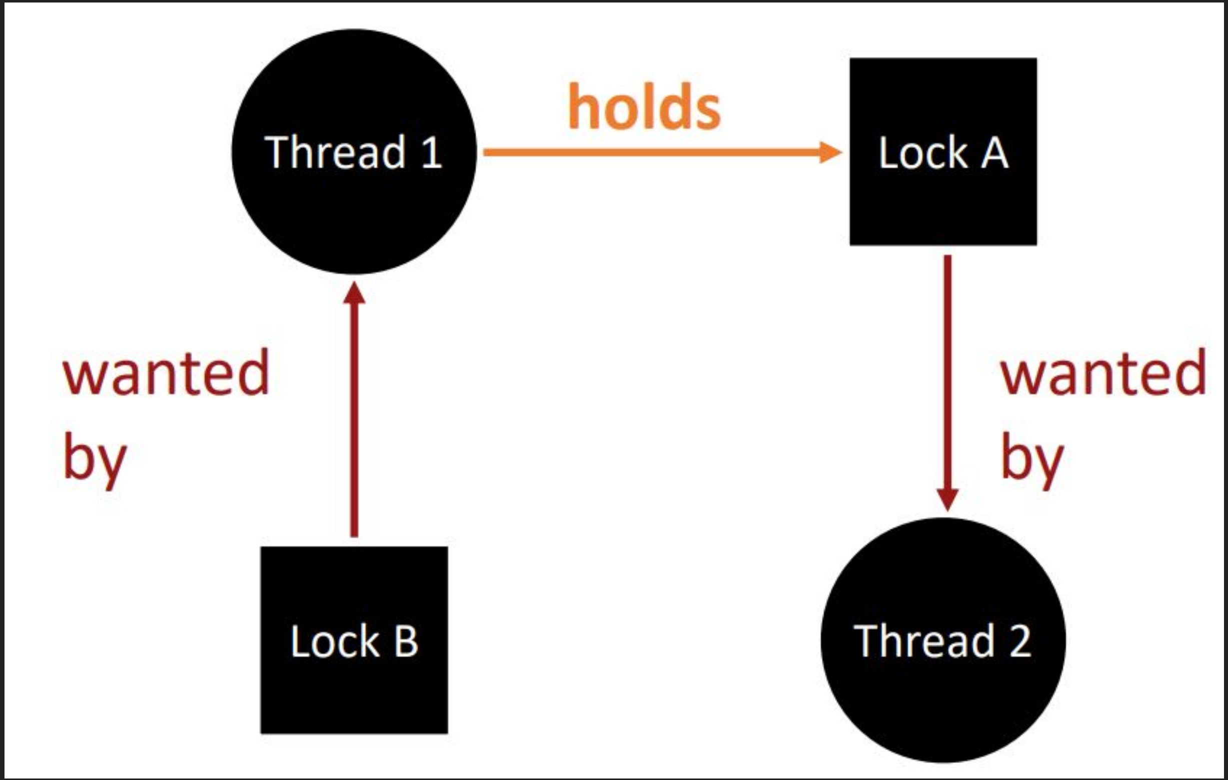


Fix Deadlock Code

Thread 1:	Thread 2:
lock(&A);	lock(&B);
lock(&B);	lock(&A);

Thread 1	Thread 2
lock(&A);	lock(&A);
lock(&B);	lock(&B);

Non-circular Dependency (fine)



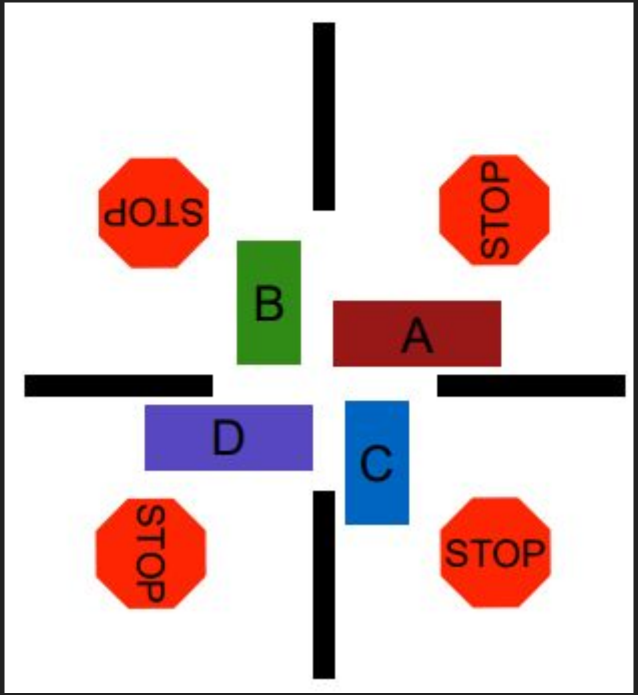
Deadlock

```
set_t *set_intersection (set_t *s1, set_t *s2) {
    set_t *rv = malloc(sizeof(*rv));
    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(rv, s1->items[i]);
    mutex_unlock(&s2->lock);
    mutex_unlock(&s1->lock);
}
```

```
Thread 1: rv = set_intersection(setA, setB);
Thread 2: rv = set_intersection(setB, setA);
```

Deadlock Theory

- Deadlocks can only occur when **all four** conditions are true:
  - 1) Mutual exclusion
  - 2) Hold-and-wait
  - 3) Circular wait
  - 4) No preemption
- Eliminate deadlock by eliminating **any one** condition



1). Mutual Exclusion

- Definition: “Threads claim **exclusive control of resources** that they require (e.g., thread grabs a lock)”
- Strategy: eliminate locks!
  - try to replace locks with atomic instructions

```
int CompAndSwap(int *addr, int expected, int new);
```

Returns 0: fail, 1: success

Code with locks

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

Code with Compare-and-Swap (CAS)

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

Example: Lock-Free Linked List Insert

Code with locks

```
void insert (int val)
{
  node_t *n =
    malloc(sizeof(*n));
  n->val = val;
  mutex_lock(&m);
  n->next = head;
  head = n;
  mutex_unlock(&m);
}
```

Code with Compare-and-Swap (CAS)

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

2). Hold-and-Wait

- Definition: “Threads **hold** resources allocated to them (e.g., locks they have already acquired) **while waiting for additional resources** (e.g., locks they wish to acquire).”
- Strategy: release **currently held resources** when waiting for new ones

2). Hold-and-Wait

- How? Use a meta lock

```
lock(&meta);
lock(&L1);
lock(&L2);
…
unlock(&meta);

// Critical section code

unlock(…);
```

2). Hold-and-Wait

- Disadvantages?
    - 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

3). No preemption

- 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;
    }
```

3). No preemption

- Potential issue:
    - **livelock**: no process makes forward progress, but the state of involved processes constantly changes
    - Can happen if all processes release resources and then try to re-acquire, fail, and keep doing this
    - Classic solution: back-off techniques
        - **Random back-off**: wait for a random amount of time before retrying
        - **Exponential back-off**: wait for exponentially increasing amount of time before retrying

4). Circular Wait

- Definition: “There exists a **circular chain of threads** such that each thread holds a resource (e.g., lock) being requested by next thread in the chain.”
- Usually the easiest deadlock requirement to attack
- Strategy: impose a well-documented order of acquiring locks
  - 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

4). Circular Wait

Thread 1

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

Thread 2

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

How would you fix this code?

Thread 1

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

Thread 2

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

Lock Ordering in Linux

- **Works well if system has distinct layers**

In linux-3.2.51/include/linux/fs.h

```
/* inode->i_mutex nesting subclasses for the lock
 * validator:
 * 0: the object of the current VFS operation
 * 1: parent
 * 2: child/target
 * 3: quota file
 * The locking order between these classes is
 * parent -> child -> normal -> xattr -> quota
 */
```

THANK YOU!