Concurrency: Common Problems

CS 537: Introduction to Operating Systems

Louis Oliphant & Tej Chajed

University of Wisconsin - Madison

Spring 2024

Administrivia

- Project 5 due Tue Apr 2 @ 11:59pm
- Exam 2, Wed, Mar 20 5:45-7:30pm
  - Bring ID and #2 Pencil, same format as Exam 1
  - Lec 001 (**1pm**) – Humanities 3650
  - Lec 002 (**9:30am**) – Humanities 2650
  - McBurney – 5:45-8pm, CS 1325

Review: Semaphores

A **semaphore** is an object with an integer value that must be initialized and can be manipulated with two routines:

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); //initializes to 1 (3rd arg)
```

```
int sem_wait(sem_t *s) {
   decrement the value of semaphore s by one
   wait if value of semaphore s is negative
}
```

```
int sem_post(sem_t *s) {
   increment the value of semaphore s by one
   if there are threads waiting, wake one
}
```

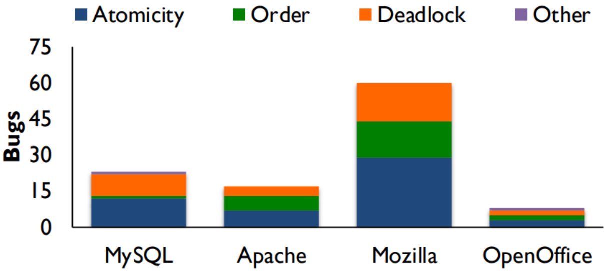
Quiz: Semaphores

https://tinyurl.com/cs537-sp24-q13

Concurrency Problems Agenda

- Non-Deadlock Bugs
  - Atomicity Violation
  - Order Violation
- Deadlock Bugs
  - Why they occur
  - How to prevent them

Concurrency Study



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 Violation

Thread 1:

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

Thread 2:

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

Fix Atomicity Violations with Locks

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);
```

Order Violation

Thread 1:

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

Thread 2:

```
void mMain(...) {
 ...
 mState = mThread->State;
 ...
}
```

Fix Order Violations with Condition Variables

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;
 ...
}
```

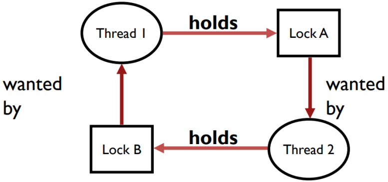
Why Deadlocks Occur

Thread 1:

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

Thread 2:

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

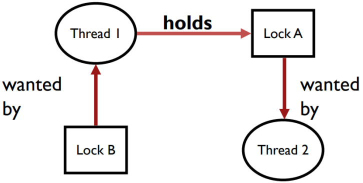


No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does (**Circular Dependency**).

Fix by Removing Circular Dependencies

Have an order that locks are obtained:

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



Conditions for Deadlock

1. **Mutual Exclusion** – Threads claim exclusive control of resources that they require (e.g. a thread grabs a lock)
2. **Hold-and-wait** – Threads hold resources allocated to them while waiting for additional resources
3. **No preemption** – Resources cannot be forcibly removed from threads
4. **Circular wait** – Circular chain of threads hold resources that other threads need

Remove *any one* of these criteria and deadlock cannot occur.

Problem: encapsulation

Solving deadlocks becomes trickier with **encapsulated** code, e.g. Vector class in Java:

Thread 1

```
v1.addAll(v2);
```

Thread 2

```
v2.addAll(v1);
```

Need to know that `v1.addAll(v2)` acquires locks on `v1` and `v2`.

Can't control how Vector acquires locks.

Prevention Technique – Lock ordering (condition 4)

Create a **total** or **partial** lock ordering

- can order by lock address in memory
- requires programmer discipline

Problem: might not know what locks are needed upfront

Prevention Technique – Hold-and-wait (condition 2)

- Acquire all locks at once:

```
pthread_mutex_lock(prevention);
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention);
```

- Can be problematic:
  - Encapsulation (must know what locks are required for each function call and get them)
  - Decreases concurrency since all locks must be acquired at once

Prevention Technique – No Preemption (condition 3)

- Stop holding onto lock if you can't acquire the other needed locks:

```
top:
  pthread_mutex_lock(L1);
  if (pthread_mutex_trylock(L2) != 0) {
     pthread_mutex_unlock(L1);
     goto top;
  }
```

- New Problem: **Livelock** – two threads can repeatedly attempt this sequence and repeatedly fail to acquire both locks.
- Encapsulation still a problem (if a lock acquisition is buried in some routine, difficult to jump back).

Final Prevention Technique – Avoid Mutual Exclusion (condition 1)

- Avoid needing mutual exclusion by using thread safe, **lock-free** data structures. These use the hardware atomic instructions.

```
// hardware guarantees atomicity
int CompareAndSwap(int *a, int e, int new) {
  if (*a == e) {
    *a = new;
    return 1; //success
  }
  return 0; //failure
}
```

Insert into a List:

```
void insert(int value) {
   node_t *n = malloc(sizeof(node_t));
   n->value = value;
   do {
      n->next = head;
   } while (CompareAndSwap(&head,
                    n->next, n)==0);
}
```

Other Strategies

Deadlock Avoidance

A smart scheduler that is aware of which threads require which locks can schedule threads such that deadlock cannot occur.

Deadlock Recovery

Allow deadlocks to occur (hopefully occasionally), have process to detect a deadlock, and then take some action to fix it.

Concurrency Summary

- **Threads** and shared memory
- **Locks** and protection surrounding critical code sections
  - Use of Locks to create **thread-safe data structures**
- **Condition Variables** controlling thread execution / sleeping on some program state.
- **Semaphores** are flexible primitives that can replace locks and condition variables
- Use concurrency primitives to prevent common concurrency problems like **deadlock**, **starvation**, guarantee **atomicity** and thread order.