CS444/544
Operating Systems II

Lecture 15
Deadlock (cont.)
Prep. for Quiz 3
5/29/2024

Acknowledgement: Slides drawn heavily from Yeongjin Jiang
Recap: Concurrency Bugs

• Code does not have a bug when it runs with single thread could have a bug when it runs with multiple threads
  • Multiple cores, etc.

• What are the types of concurrency bugs?
  • Atomicity
  • Ordering
  • Deadlock
Recap: Atomicity: Use Lock

1. `pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;`
2. Thread 1:
3. `pthread_mutex_lock(&proc_info_lock);`
4. `if (thd->proc_info) {
   ...
   fputs(thd->proc_info, ...);
   ...
   }
5. `pthread_mutex_unlock(&proc_info_lock);`
6. Thread 2:
7. `pthread_mutex_lock(&proc_info_lock);`
8. `thd->proc_info = NULL;`
9. `pthread_mutex_unlock(&proc_info_lock);`

**Time of check**

**Time of use**

**Update!**

In critical section, **NO UPDATE**

Do not have TOCTTOU!

This will also block other threads that run line 5 while thread 2 updates thd->proc_info.

---

Time of check

Time of use

Update!
Recap: How Can We Resolve the Ordering Issue?

- Use locks and conditional variables to force a specific ordering...

```
Thread 1::
void init() {
    ...
    mThread = PR_CreateThread(mMain, ...);
    // signal that the thread has been created...
    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);
    ...
}

Thread 2::
void mMain(...) {
    ...
    // wait for the thread to be initialized...
    pthread_mutex_lock(&mtLock);
    while (mtInit == 0)
        pthread_cond_wait(&mtCond, &mtLock);
    pthread_mutex_unlock(&mtLock);
    mState = mThread->State;
    ...
}
```
Recap: Deadlock

- Two or more threads are waiting for the other to take some actions thus neither makes any progress.

Thread 1:
```c
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

Thread 2:
```c
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```
Recap: Circular Dependency

Thread 1:
```c
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

Thread 2:
```c
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```
Recap: Non-Circular Dependency

Thread 1:
`pthread_mutex_lock(L1);`
`pthread_mutex_lock(L2);`

Thread 2:
`pthread_mutex_lock(L1);`
`pthread_mutex_lock(L2);`
Thread-safe Data structure

```c
set_t *set_intersection (set_t *s1, set_t *s2) {
    set_t *rv = new set_t();
    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);
    return rv;
}
```
Thread-safe Data structure

Thread 1:

rv = set_intersection(setA, setB);

Thread 2:

rv = set_intersection(setA, setB);

set_t *set_intersection (set_t *s1, set_t *s2) {

...  
Mutex_lock(&s1->lock);
Mutex_lock(&s2->lock);

...

}
Thread-safe Datastructure

Thread 1:

rv = set_intersection(setA, setB);

Mutex_lock(&setA->lock);
Mutex_lock(&setB->lock);

...
Mutex_unlock(&setB->lock);
Mutex_unlock(&setA->lock);

Thread 2:

rv = set_intersection(setA, setB);

Mutex_lock(&setA->lock);
Mutex_lock(&setB->lock);

...
Mutex_unlock(&setB->lock);
Mutex_unlock(&setA->lock);
Is This a Thread-safe Datastructure?

```c
set_t *set_intersection (set_t *s1, set_t *s2) {
    set_t *rv = new set_t();
    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);
    return rv;
}
```
Thread 1:

rv = set_intersection(setA, setB);

set_t *set_intersection(set_t *s1, set_t *s2) {
    ...
    Mutex_lock(&s1->lock);
    Mutex_lock(&s2->lock);
    ...
}

Thread 2:

rv = set_intersection(setB, setA);
Find a Problem..

Thread 1:
rv = set_intersection(setA, setB);
Mutex_lock(&setA->lock);
Mutex_lock(&setB->lock);

Thread 2:
rv = set_intersection(setB, setA);
Mutex_lock(&setB->lock);
Mutex_lock(&setA->lock);

Deadlock!
Deadlock Theory

• Deadlocks can only happen if threads are having
  • Mutual exclusion
  • Hold-and-wait
  • No preemption
  • Circular wait

• We can eliminate deadlock by removing such conditions...
Mutual Exclusion

• Definition
  • Threads claims an exclusive control of a resource
  • E.g., Threads grabs a lock
How to Remove Mutual Exclusion

• Do not use lock
  • What???

• Replace locks with atomic primitives
  • compare_and_swap(uint64_t *addr, uint64_t prev, uint64_t value);
  • if *addr == prev, then update *addr = value;
  • lock cmpxchg in x86..

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

```c
Mutex_lock(&setA->lock);
Mutex_lock(&setB->lock);
```
How to Remove Hold-and-Wait

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

• How to do this? Use a meta lock, like this:

  lock(&meta);
  lock(&L1);
  lock(&L2);

  ...
  unlock(&meta);

  // Critical section code
  unlock(...);
Remove Hold-and-Wait

```c
set_t *set_intersection (set_t *s1, set_t *s2) {
    Mutex_lock(&meta_lock)
    Mutex_lock(&s1->lock);
    Mutex_lock(&s2->lock);
    ...
    Mutex_unlock(&s2->lock);
    Mutex_unlock(&s1->lock);
    Mutex_unlock(&meta_lock);
}
```
Remove Hold-and-Wait

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

Mutex_lock(&meta_lock);
Mutex_lock(&setA->lock);
Mutex_lock(&setB->lock);
...
Mutex_unlock(&setB->lock);
Mutex_unlock(&setA->lock);
Mutex_unlock(&meta_lock);

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

Mutex_lock(&meta_lock);
Mutex_lock(&setB->lock);
Mutex_lock(&setA->lock);

Will wait until Thread 1 finishes (release meta_lock)!
No Preemption

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

```c
lock(A);
lock(B);
...
```

In case if B is acquired by other thread

All other threads must wait for acquiring A
How to Remove No Preemption

Release the lock if obtaining a resource fails...

top:

lock(A);

if (trylock(B) == -1) {
    unlock(A);
    goto top;
}

...
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.
How to Remove Circular Wait

Thread 1:
    pthread_mutex_lock(L1);
    pthread_mutex_lock(L2);

Thread 2:
    pthread_mutex_lock(L2);
    pthread_mutex_lock(L1);
How to Remove Circular Wait

Lock variable is mostly a pointer, then provide a correct order of having a lock e.g.,
if(l1 > l2) {
    Mutex_lock(l1);
    Mutex_lock(l2);
}
else {
    Mutex_lock(l2);
    Mutex_lock(l1);
}

Thread 1:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);

Thread 2:
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
Deadlock Theory

• Deadlocks can only happen if threads are having
  • Mutual exclusion
  • Hold-and-wait
  • No preemption
  • Circular wait

• We can eliminate deadlock by removing such conditions...
Quiz 3

• Next Tuesday (6/3 from 8:00 am to 11:59 pm)
  • Open materials (slides, videos, code, and textbook)

• You will have 2 attempts for the quiz
Quiz 3 Coverage

• Lab 3 (User/Kernel, System Call and Interrupt Handling)
• Lab 4 (Preemptive Multitasking & Copy-on-write Fork)
• Lecture 12: Multithreading and Synchronization
• Lecture 13-14: Lock and Thread Synchronization
• Lecture 14-15: Concurrency Bugs and Deadlock
Sample Questions

• In x86, which of the following instruction runs atomically?
  • cmpxchg
  • popa
  • lea
  • xchg
  • mov
Sample Questions

- In x86, which of the following instruction runs atomically?
  - cmpxchg
  - popa
  - lea
  - xchg
  - mov
Sample Questions

• In x86, which of the following instruction runs atomic test and test-and-set?
  • cmpxchg
  • int $0x30
  • lock cmpxchg
  • lock
  • xchg
Sample Questions

• In x86, which of the following instruction runs atomical test and test-and-set?
  • cmpxchg
  • int $0x30
  • lock cmpxchg
  • lock
  • xchg

cmpxchg in x86 is not a hardware atomic instruction. However, when used with the lock prefix, the instruction will be an atomic test and test-and-set instruction.
Sample Questions

• In x86, which register is being used for storing “compare” value when running the cmpxchg instruction?
  • CR3
  • EAX
  • EBX
  • ESP
  • EIP
Sample Questions

• In x86, which register is being used for storing “compare” value when running the cmpxchg instruction?
  • CR3
  • EAX
  • EBX
  • ESP
  • EIP
Sample Questions

• T/F: Page table is not relevant to data racing / thread synchronization.
Sample Questions

• T/F: Page table is not relevant to data racing / thread synchronization.

   True. Page table is for virtual memory, and thus is not relevant to thread sync.
Sample Questions

• In JOS lab, which value will the fork() returns to the child environment if the function has been executed successfully?
  • 0
  • 1
  • The envid of the parent env
  • The envid of the child env
  • The address of the page table of the child env
Sample Questions

• In JOS lab, which value will the fork() returns to the child environment if the function has been executed successfully?
  • 0
  • 1
  • The envid of the parent env
  • The envid of the child env
  • The address of the page table of the child env

Fork returns:
Parent: child envid
Child: 0
Sample Questions

• Which of the following stores the information about the reason of a page fault?
  • EAX
  • CR2
  • CR3
  • eflags
  • Trapframe
Sample Questions

• Which of the following stores the information about the reason of a page fault?
  • EAX
  • CR2
  • CR3
  • eflags
  • Trapframe

Error code in trapframe
Sample Questions

• Will this implementation cause deadlock (assuming no infinite loop in the critical section)?

Thread 1:
```c
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

Yes

Thread 2:
```c
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```
Sample Questions

• Will this implementation cause deadlock (assuming no infinite loop in the critical section)?

Thread 1:
spin_lock(&meta);
spin_lock(&l1);
spin_lock(&l2);
spin_unlock(&meta);
...
spin_unlock(&l2);
spin_unlock(&l1);

Thread 2:
spin_lock(&meta);
spin_lock(&l2);
spin_lock(&l1);
spin_unlock(&meta);
...
spin_unlock(&l1);
spin_unlock(&l2);

No