CS444/544
Operating Systems II

Lecture 15
Lock and Synchronization (cont.)
Concurrency Bugs and Deadlock
5/30/2023

Acknowledgement: Slides drawn heavily from Yeongjin Jiang
Odds and Ends

• Quiz 3: next Tuesday (6/6)
  • More info later

• Lab 3 will be graded by this weekend
Recap: Data Race Example

- counter += value
  - edx = value;
  - eax = counter;
  - eax = edx + eax;
  - counter = eax;

- Assume counter = 0 at start

Counter = 1, not 2!!!
Recap: Mutex

• **Lock**
  • Prevent others enter the critical section

• **Unlock**
  • Release the lock, let others acquire the lock

• **counter += value**
  • **lock()**
  • edx = value;
  • eax = counter;
  • eax = edx + eax;
  • counter = eax;
  • **unlock()**
Spinlock Examples

• unzip lock-example-master.zip
• Run 30 threads, each count upto 10000

• Build code
  • $ make

```bash
os2 ~/cs444/s21/lock-example-master 146% make
gcc -o lock lock.c -std=c99 -g -Wno-implicit-function-declaration -O2 -lpthread
```
Summary

• 5 Lock implementations
  • Naïve lock (*bad_lock*, *not working*)
  • xchg lock (*test-and-set*, *slow*)
  • cmpxchg lock (*a fake test and test-and-set, still slow*)
  • Software test and hardware test-and-set (*fast!*)
  • Hardware test-and-set with exponential backoff (*faster!*)

• Performance check
  • Total execution time
  • L1-dcache-load-misses
  • Compare the performance to pthread_mutex
os2 ~/cs444/s21/lock-example-master 172% ./lock
Counting 10000 with 30 threads using NO_LOCK...
Count: 37484, elapsed Time: 37.261 ms
Counting 10000 with 30 threads using BAD_LOCK...
Count: 45567, elapsed Time: 43.420 ms
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time: 908.793 ms
Counting 10000 with 30 threads using CMPXCHG_LOCK...
Count: 300000, elapsed Time: 956.066 ms
Counting 10000 with 30 threads using TTS_LOCK...
Count: 300000, elapsed Time: 465.198 ms
Counting 10000 with 30 threads using BACKOFF_LOCK...
Count: 300000, elapsed Time: 142.791 ms
Counting 10000 with 30 threads using MUTEX_LOCK...
Count: 300000, elapsed Time: 428.405 ms
lock-example

if (LLL_MUTEX_TRYLOCK (mutex) != 0)
{
    int cnt = 0;
    int max_cnt = MIN (max_adaptive_count (),
                        mutex->___data.__spins * 2 + 10);
    do
    {
        if (cnt++ >= max_cnt)
        {
            LLL_MUTEX_LOCK (mutex);
            break;
        }
        atomic_spin_nop ();
    }
    while (LLL_MUTEX_TRYLOCK (mutex) != 0);

    if the lock variable is not 0
    Spins * 2 + 10... exp backoff!

    #define atomic_spin_nop() __asm ("pause")

    Check if the lock variable is 0...
Lock is Slow

• Run While() internally

• Can block other threads

• We need to determine when and where to use lock
When Do We Need to Use a Lock?

• Write must be finished before the next load
  • Many writers and one reader
    • Yes... many writers..
  • Two writers and two readers
    • Yes, two writers...
  • One writer and many readers
    • Not always if there is only one writer

Thread 1

- `eax = counter`
- `eax = edx + eax`
- `counter = eax`

Thread 2

- `eax = counter`
- `eax = edx + eax`
- `counter = eax`
Where Do We Need to Put a Lock?

- What will happen if a critical section is too big?

Four independent variables
Require a lock on updating these..
Small Critical Sections

Thread 1
- CS A1
- Update A
- CS B1
- Update B
- CS D1
- Update D
- CS C1
- Update C

Thread 2
- CS B2
- Update B
- CS A2
- Update A
- CS C2
- Update C
- CS D2
- Update D

Fast, but developer must take care of splitting critical sections..
General Practice

• Use lock only if it is required
  • Determine the case when you do not need a lock
    • Atomic read
    • Only one writer

• Use a small critical section
  • Critical section prohibits concurrent execution
  • Determine where do we share a variable
  • Wrap only the code that updates the shared variable

• Looks simple, but sometimes it’s difficult
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
Atomicity

```
Read
1   Thread 1::
2     if (thd->proc_info) { Time of check
3         ...
4         fputs(thd->proc_info, ...); Time of use
5         ...
6     }
7
8   Thread 2::
9     thd->proc_info = NULL;  Time-of-check-to-time-of-use bug

Write!
```

TOCTTOU
Atomicity: Use Lock

Time of check

Time of use

Update!

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) {
5. ...}
6. `fputs(thd->proc_info, ...);`
7. ...}
8. `pthread_mutex_unlock(&proc_info_lock);`
9. Thread 2:
10. `pthread_mutex_lock(&proc_info_lock);`
11. thd->proc_info = NULL;
12. `pthread_mutex_unlock(&proc_info_lock);`

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.
Ordering: Mozilla – Order 1

1  Thread 1::
2    void init() {
3        ...
4        mThread = PR_CreateThread(mMain, ...);
5        ...
6    }
7
8  Thread 2::
9    void mMain(...) {
10       ...
11       mState = mThread->State;
12       ...
13    }
Ordering: Mozilla – Order 2

8   Thread 2::
9    void mMain(...) {
10      ...
11      mState = mThread->State;       Not Initialized...
12      ...
13   }

1   Thread 1::
2    void init() {
3       ...
4       mThread = PR_CreateThread(mMain, ...);
5       ...
6    }
How Can We Resolve the Ordering Issue?

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

• `pthread_cond_wait(cond, lock)`
  • Set `cond = 0`
  • You will release the lock
  • Wait until `cond == 1`
  • Acquire the lock again

• `pthread_cond_signal(cond)`
  • `cond = 1`

```c
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); // Sends Signal..
    pthread_mutex_unlock(&mtLock);
    ...
}

Thread 2::
void mMain(...) {
    ...
    // wait for the thread to be initialized...
    pthread_mutex_lock(&mtLock);
    while (mtInit == 0) { // Waits condition..
        pthread_cond_wait(&mtCond, &mtLock);
        pthread_mutex_unlock(&mtLock);
        mState = mThread->State;
        ...
    }
    ...
    mThread->State = ...
    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);
    ...
}
```
Deadlock

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

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

Thread 2:
- `pthread_mutex_lock(L2);`
- `pthread_mutex_lock(L1);`
Deadlock!
Deadlock: Example

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

Thread 2:
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
How Can We Resolve Circular Dependency

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

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

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

Thread 2:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
Circular Dependency

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

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

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

Thread 2:
`pthread_mutex_lock(L1);`
`pthread_mutex_lock(L2);`