Odds and Ends

• Quiz 3: next Tuesday (12/5)
  • 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
4\textsuperscript{th} Candidate: Test and Test & Set

- \texttt{tts\_xchg\_lock}
- Algorithm
  - Wait until lock becomes 0
  - After lock == 0
    - \texttt{xchg\ (lock, 1)}
    - This only updates lock = 1 if lock was 0

- Why \texttt{xchg}, why not \texttt{*lock = 1} directly
  - \texttt{while} and \texttt{xchg} are not atomic
  - Load/Store must happen at
    - The same time!
4\textsuperscript{th} Candidate TTS Result

- Consistent!

- A little less cache misses but

- Faster (~500ms vs. 900 ~ 1200 ms)
Still Slow and Many Cache Misses..

- Can we do better? Why we still have too many misses?
  - A thread acquires the lock (update 0 -> 1)
    - Invalidate caches in 29 other cores
  - A thread releases the lock (update 1 -> 0)
    - Invalidate caches in 29 other cores
  - 29 other cores are all reading the variable lock
    - Immediately after invalidate, it loads data to cache
    - Then invalidated again by either lock/release...
    - This happens in every 3~4 cycles...
5\textsuperscript{th} Candidate: Backoff Lock

- Too many contention on reading lock while only 1 can run critical sec.
  - All other 29 cores running while (*lock == 1);
  - This is the slow down factor

- Idea: can we slow down that check?
  - Let’s set a wait time if CPU checked the lock value as 1

- Something like, exponential backoff
  - After checking lock == 1,
    - Wait 1 cycle
  - After checking lock == 1 again,
    - Wait 2 cycles
    - Wait 4 cycles
    - Wait 8 cycles
    - ...

```
void

void xchg_lock(volatile uint32_t *lock) {
    while (1) {
        while(*lock == 1);
        if (xchg(lock, 1) == 0) {
            break;
        }
    }
}
```
5th Candidate: Backoff Lock

- backoff_cmpxchg_lock(lock)
- Try cmpxchg
  - If succeeded, acquire the lock.
  - If failed
    - Wait 1 cycle (pause) for 1st trial
    - Wait 2 cycles for 2nd trial
    - Wait 4 cycles for 3rd trial
    - ...
    - Wait 65536 cycles for 17th trial..
    - Wait 65536 cycles for 18th trial..
5\textsuperscript{th} Candidate: Backoff Result

• **Consistent!**

```
os2 ~/cs444/s21/lock-example-master 168% ./perf-lock.sh backoff
Counting 10000 with 30 threads using BACKOFF_LOCK...
Count: 300000, elapsed Time: 210.387 ms

Performance counter stats for './lock backoff':
  196,227   L1-dcache-load-misses:u

0.214007977 seconds time elapsed
4.405105000 seconds user
0.112746000 seconds sys
```

• **Much lower cache miss**

• **Faster! (~200ms!)**
Even Faster Than pthread_mutex

Performance counter stats for './lock backoff':

196,227 L1-dcache-load-misses:u
0.214007977 seconds time elapsed
4.405105000 seconds user
0.112746000 seconds sys

Performance counter stats for './lock mutex':

1,656,537 L1-dcache-load-misses:u
0.477209142 seconds time elapsed
0.519430000 seconds user
12.487676000 seconds sys
Summary

• Mutex is implemented with Spinlock
  • Waits until lock == 0 with a while loop (that’s why it’s called spin)
• Naïve code implementation never works
  • Load/Store must be atomic
• `xchg` is a “test and set” atomic instruction
  • Consistent, however, many cache misses, slow! (950ms)
• Lock `cmpxchg` is a ”test and test&set” atomic instruction
  • But Intel implemented this as `xchg`... slow! (1150ms)
• We can implement test-and-test-and-set (tts) with while + `xchg`
  • Faster! (500ms)
• We can also implement exponential backoff to reduce contention
  • Much faster! (200ms)
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
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

```c
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);

    #define atomic_spin_nop () __asm ("pause")
```

If the lock variable is not 0

Check if the lock variable is 0...

Spins * 2 + 10... exp backoff!
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
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.

Wasting time for waiting for all
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!
Atomicity: Use 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..

```c
1 pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3 Thread 1::
4 pthread_mutex_lock(&proc_info_lock);
5 if (thd->proc_info) {
6     ...
7     fputs(thd->proc_info, ...);
8     ...
9 }
10 pthread_mutex_unlock(&proc_info_lock);
11
12 Thread 2::
13 pthread_mutex_lock(&proc_info_lock);
14 thd->proc_info = NULL;
15 pthread_mutex_unlock(&proc_info_lock);
```
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 }
Thread 1::
void init() {
  ...
  mThread = PR_CreateThread(mMain, ...);
  ...
}

Thread 2::
void mMain(...) {
  ...
  mState = mThread->State;  // Not Initialized...
  ...
}
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
  - `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);
  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;
  ...
}
```
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);

Thread 1 holds Lock L1
Thread 2 holds Lock L2
Lock L1 is wanted by Lock L2