CS202 More Concurrency

- Quick Review of Last Time

- Must protect access to shared data in concurrent programs

  → Ensure no more than one thread accesses shared data at a time
  → Critical Section: logic/code that accesses shared data

  → Mutex: Type of object used to implement critical section

    → Exactly one thread can hold a mutex at a time

    → acquire: Block until current thread holds mutex

    → release: Stop holding mutex (allowing others to make progress)

- Sometimes, threads need to wait until some condition on shared data is true
Some condition on resource is true

- Empty slots in array
- Non-empty slots in array
- ...

Signaling is a useful way to write such code

**Condition Variables**

- `cond-wait`: Block thread until C.V. is notified
- `cond-signal`: Notify 1 waiting thread
- `cond-broadcast`: Notify all waiting threads

- Must use a mutex to protect C.V.s (all calls need mutex)

  Lock mutex \rightarrow call cond-*

**Monitor**

- 1 mutex + 1 or more condition variables
class Bounded Queue {
    monitor
    public:
        void add_work(...);
        work * get_work(...);
    private:
        work * queue [SIZE];
        size_t items = 0;
        mutex m;
        cond-var empty;
        cond-var full;
}

Often implemented by languages (kind of like synchronized in Java)

Language impl's

    ➔ Acquire mutex on entry to function
    ➔ Release on return
    ➔ Inter-function synchronization using C.V.

 Manual pattern we will use in this class
Rules

- Acquire mutex on entry to func.
- Hold mutex when calling CV function
- Use loop for CV: wait

```c
while(!cond) {
    cond.wait(cv, m);
}
```

More on this next class.

Implementing Mutexes

1. Spin lock
   - 64-bit
   ```c
cas (lock, 0, TID)
```
   ```c
   void acquire (lock_t* lock)
   {
     while (TRUE) {
       if (lock == 0) {
         lock = TID; return;
       }
     }
   }
```
Atomic Instruction

**Compare-and-swap**
- address, value to compare,
- value to set

- Atomically

THE CORE OF NEARLY ALL CONCURRENCY TOOLS

\[
\text{if} (*\text{address} == \text{compare}) \exists \\
*\text{address} = \text{value}; \\
\text{return true;} \\
\text{else return false};
\]

Intel \(<\) lock cmpxchg

(switch to handout)

(2) Mutex

Spinlock is
\[
\begin{align*}
\text{holder} & \leftarrow \text{.bid} \\
\text{acquire (s)} & \\
\text{if} (\text{holder} == 0 \text{ or} \\
\text{holder} == \text{tid}) \exists
\end{align*}
\]
Deadlock

Handout

Not just for
mutex

release (m1)

acquire (m1)

release (m2)

acquire (m2)

release (m2)

acquire (m1)

release (m1)

release (m1)

release (m2)

wait until holden = 0 on
release (s)
Problem

1. At least one resource has mutual exclusion
2. Hold one resource + wait on another
3. No pre-emption: Neither thread can give up on block
4. Wait cycle

How to fix

1. Detect + fix (Databases)
2. Avoid algorithmically
3. Remove one of the conditions—

Simple fix: All code acquires locks in the same order
Matrices & Performance

- Starvation

- Priority inversion

- Wait queue traversal order

- Fine-grained vs Coarse-grained
The previous handout demonstrated the use of mutexes and condition variables. This handout demonstrates the use of monitors (which combine mutexes and condition variables).

1. The bounded buffer as a monitor

```cpp
// This is pseudocode that is inspired by C++.
// Don’t take it literally.

class MyBuffer {
public:
    MyBuffer();
    ~MyBuffer();
    void Enqueue(Item);
    Item = Dequeue();
private:
    int count;
    int in;
    int out;
    Item buffer[BUFFER_SIZE];
    Mutex* mutex;
    Cond* nonempty;
    Cond* nonfull;
};
void MyBuffer::MyBuffer()
{
    in = out = count = 0;
    mutex = new Mutex;
    nonempty = new Cond;
    nonfull = new Cond;
}
void MyBuffer::Enqueue(Item item)
{
    mutex.acquire();
    while (count == BUFFER_SIZE)
        cond_wait(&nonfull, &mutex);
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;
    cond_signal(&nonempty, &mutex);
    mutex.release();
}
Item MyBuffer::Dequeue()
{
    mutex.acquire();
    while (count == 0)
        cond_wait(&nonempty, &mutex);
    Item ret = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;
    cond_signal(&nonfull, &mutex);
    mutex.release();
    return ret;
}

int main(int, char**)
{
    MyBuffer buf;
    int dummy;
    tid1 = thread_create(producer, &buf);
    tid2 = thread_create(consumer, &buf);
    // never reach this point
    thread_join(tid1);
    thread_join(tid2);
    return -1;
}
void producer(void* buf)
{
    MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
    for (;;) {
        // next line produces an item and puts it in nextProduced */
        Item nextProduced = means_of_production();
        sharedbuf->Enqueue(nextProduced);
    }
}
void consumer(void* buf)
{
    MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
    for (;;) {
        Item nextConsumed = sharedbuf->Dequeue();
        // next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}

Key point: *Threads* (the producer and consumer) are separate from *shared object* (MyBuffer). The synchronization happens in the shared object.
2. This monitor is a model of a database with multiple readers and writers. The high-level goal here is (a) to give a writer exclusive access (a single active writer means there should be no other writers and no readers) while (b) allowing multiple readers. Like the previous example, this one is expressed in pseudocode.

// assume that these variables are initialized in a constructor
state variables:
AR = 0;  // # active readers
AW = 0;  // # active writers
WR = 0;  // # waiting readers
WW = 0;  // # waiting writers
Condition okToRead = NIL;
Condition okToWrite = NIL;
Mutex mutex = FREE;

Database::read() {
    startRead();  // first, check self into the system
    Access Data
    doneRead();   // check self out of system
}

Database::startRead() {
    acquire(&mutex);
    while((AW + WW) > 0){
        WR++;
        wait(&okToRead, &mutex);
        WR--;
    }
    AR++;
    release(&mutex);
}

Database::doneRead() {
    acquire(&mutex);
    AR--;
    if (AR == 0 && WW > 0) { // if no other readers still
        signal(&okToWrite, &mutex); // active, wake up writer
    }
    release(&mutex);
}

Database::write(){  // symmetrical
    startWrite();  // check in
    Access Data
    doneWrite();   // check out
}

Database::startWrite() {
    acquire(&mutex);
    if (AR == 0 & WW == 0) { // if no other readers still
        signal(&okToWrite, &mutex); // active, wake up writer
    }
    release(&mutex);
}

Database::doneWrite() {
    acquire(&mutex);
    if (WW > 0) { // give priority to writers
        signal(&okToWrite, &mutex);
    } else if (WR > 0) { // broadcast to okToRead, mutex;
        broadcast(&okToRead, &mutex);
    }
    release(&mutex);
}

NOTE: what is the starvation problem here?

3. Shared locks

struct sharedlock {
    int i;
    Mutex mutex;
    Cond c;
};

void AcquireExclusive (sharedlock *sl) {
    acquire(&sl->mutex);
    while (sl->i) {
        wait (&sl->c, &sl->mutex);
        sl->i = -1;
    }
    release(&sl->mutex);
}

void AcquireShared (sharedlock *sl) {
    acquire(&sl->mutex);
    while (sl->i < 0) {
        wait (&sl->c, &sl->mutex);
        sl->i++;
    }
    release(&sl->mutex);
}

void ReleaseShared (sharedlock *sl) {
    acquire(&sl->mutex);
    if (sl->i) {
        signal (&sl->c, &sl->mutex);
    }
    release(&sl->mutex);
}

void ReleaseExclusive (sharedlock *sl) {
    acquire(&sl->mutex);
    sl->i = 0;
    broadcast (&sl->c, &sl->mutex);
    release(&sl->mutex);
}

QUESTIONS:

A. There is a starvation problem here. What is it? (Readers can keep writers out if there is a steady stream of readers.)
B. How could you use these shared locks to write a cleaner version of the code in the prior item? (Though note that the starvation properties would be different.)
1. Here is a BROKEN spinlock implementation:

```c
struct Spinlock {
    int locked;
};

void acquire(Spinlock *lock) {
    while (1) {
        if (lock->locked == 0) { // A
            lock->locked = 1;      // B
            break;
        }
    }
}

void release(Spinlock *lock) {
    lock->locked = 0;
}
```

What's the problem? Two acquire()s on the same lock on different CPUs might both execute line A, and then both execute B. Then both will think they have acquired the lock. Both will proceed. That doesn't provide mutual exclusion.

2. Correct spinlock implementation

Relies on atomic hardware instruction. For example, on the x86-64, doing

```
"lock cmpxchg addr, $1, %rax"
```

does the following:

(i) freeze all CPUs' memory activity for address addr
(ii) if *addr = 1 then *addr = %rax
(iii) return *addr

/* pseudocode */

```c
int cmpxchg(addr, comprand, value) {
    if *addr == comprand {
        *addr = value;
    }
    return *addr
}
```

/* bare-bones version of acquire */

```c
void acquire(Spinlock *lock) {
    pushcli();    /* what does this do? */
    while (cmpxchg(&lock->locked, 0, TID) != TID) // yield
    }
}
```

void release(Spinlock *lock) {
    cmpxchg(&lock->locked, TID, 0);
    popcli();    /* what does this do? */
}
```

The above is called a "spinlock" because acquire() spins. The bare-bones version is called a "test-and-set (TAS) spinlock"; the other is called a "test-and-test-and-set spinlock".

The spinlock above is great for some things, not so great for others. The main problem is that it "busy waits": it spins, chewing up CPU cycles. Sometimes this is what we want (e.g., if the cost of going to sleep is greater than the cost of spinning for a few cycles waiting for another thread or process to relinquish the spinlock). But sometimes this is not at all what we want (e.g., if the lock would be held for a while: in those cases, the CPU waiting for the lock would waste cycles spinning instead of running some other thread or process).

NOTE: the spinlocks presented here can introduce performance issues when there is a lot of contention. (This happens even if the programmer is using spinlocks correctly.) The performance issues result from cross-talk among CPUs (which undermines caching and generates traffic on the memory bus). If we have time later, we will study a remediation of this issue (search the Web for "MCS locks").

ANOTHER NOTE: In everyday application-level programming, spinlocks will not be something you use (use mutexes instead). But you should know what these are for technical literacy, and to see where the mutual exclusion is truly enforced on modern hardware.
Mutex implementation

The intent of a mutex is to avoid busy waiting: if the lock is not available, the locking thread is put to sleep, and tracked by a queue in the mutex. The next page has an implementation.

```c
#include <sys/queue.h>

typedef struct thread {
    STAILQ_ENTRY(thread_t) qlink;
} thread_t;

struct Mutex {
    thread_t *owner;
    STAILQ(thread_t) waiters;
    Spinlock splock;
};

void mutex_acquire(struct Mutex *m) {
    acquire(&m->splock);
    if (m->owner == 0) {
        m->owner = id_of_this_thread;
        release(&m->splock);
    } else {
        STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);
        sched_mark_blocked(&id_of_this_thread);
        release(&m->splock);
        sched_swtch();
    }
}

void mutex_release(struct Mutex *m) {
    acquire(&m->splock);
    assert(m->owner == id_of_this_thread);
    m->owner = STAILQ_GET_HEAD(&m->waiters);
    if (m->owner) {
        sched_wakeone(&m->owner);
        STAILQ_REMOVE_HEAD(&m->waiters, qlink);
    }
    release(&m->splock);
}
```

fair_mutex.c