CS202 (003): Operating Systems Concurrency III

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202

Last Time

Mutex (mutual exclusion objects)

```
mutex_init(mutex_t* m)
mutex_lock(mutex_t* m)
mutex_unlock(mutex_t* m)
```

Conditional Variables

```
void cond_init(Cond *cond, ...);
void cond_wait(Cond *cond, Mutex *mutex);
void cond_signal(Cond *cond);
void cond_broadcast(Cond *cond);
```

Monitor: Mutex + Conditional Variables

All method calls are protected by a **mutex**

Synchronization happens with condition variables whose associated mutex is the **mutex that protects the method calls**

"Monitor" can be used to refer to either a *programming convention* or a *method in certain programming languages**

Please follow these conventions on Lab 3!

* https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Mike Dahlin's "Programming with Threads"

You are required to follow this document
(although we don't code in Java)

You will lose a lot of points if you don't follow
(in labs and exams)

Do not program concurrency in other ways
unless you are a concurrency guru

Standards for Programming w/ Threads

Rule I: acquire/release at beginning/end of methods

Rule II: hold lock when doing condition variable operations

Rule III: a thread that is in wait() must be prepared to be restarted at any time, not just when another thread calls "signal()"

Rule IV: don't call sleep()

Advice for concurrent programming

Top-level piece of advice: SAFETY FIRST

Locking at coarse grain is easiest to get right, so do that

Don’t worry about performance at first

Don’t view deadlock as a disaster

MAKE SURE YOU PROGRAM NEVER DOES THE WRONG THING

Advice for concurrent programming

Getting started

1. Identify unit of concurrency
2. Identify chunks of state
3. write down high-level main loop of each thread

Write down the synchronization constraints, and the type

Create a lock or CV for each constraint

Implement the methods, using the locks and CVs

CS 202, Fall 2024
Handout 5 (Class 6)

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

```
    // 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.

```
112     // assume that these variables are initialized in a constructor
113     state variables:
114         AR = 0;  // # active readers
115         AW = 0;  // # active writers
116         WR = 0;  // # waiting readers
117         WW = 0;  // # waiting writers
118
119         Condition okToRead = NIL;
120         Condition okToWrite = NIL;
121         Mutex mutex = FREE;
122
123     Database::read() {
124         startRead();  // first, check self into the system
125         Access Data
126         doneRead();   // check self out of system
127     }
128
129     Database::startRead() {
130         acquire(&mutex);
131         while((AW + WW) > 0){
132             WR++;
133             wait(&okToRead, &mutex);
134             WR--;
135         }
136         AR++;
137         release(&mutex);
138     }
139
140    Database::doneRead() {
141         acquire(&mutex);
142         AR--;
143         if (AR == 0 && WW > 0) { // if no other readers still
144           signal(&okToWrite, &mutex);   // active, wake up writer
145         }
146         release(&mutex);
147     }
148
149     Database::write(){  // symmetrical
150         startWrite();  // check in
151         Access Data
152         doneWrite();  // check out
153     }
154
155     Database::startWrite() {
156         acquire(&mutex);
157         while ((AW + AR) > 0) { // check if safe to write.
158                                 // if any readers or writers, wait
159             WW++;
160             wait(&okToWrite, &mutex);
161             WW--;
162         }
163         AW++;
164         release(&mutex);
165     }
166
167     Database::doneWrite() {
168         acquire(&mutex);
169         AW--;
170         if (WW > 0) {
171             signal(&okToWrite, &mutex); // give priority to writers
172         } else if (WR > 0) {
173             broadcast(&okToRead, &mutex);
174         }
175         release(&mutex);
176     }
177
```

- workers interact with a database
- readers never modify
- writers read an modify
- allow:
  - many readers at once
    OR
  - only one writer (no reader)

Unit of concurrency?

Shared chunks of state?

What does main function looks like?

Synchronization constraints and objects?

Implementation of mutex

Peterson's algorithm

Disable interrupts

Spinlocks

Peterson's Algorithm

```
volatile bool flag[2] = {false, false};
volatile int turn;
```

```
P0:       flag[0] = true;
P0_gate:  turn = 1;
          while (flag[1] && turn == 1)
          {
              // busy wait
          }
          // critical section
          ...
          // end of critical section
          flag[0] = false;
```

```
P1:       flag[1] = true;
P1_gate:  turn = 0;
          while (flag[0] && turn == 0)
          {
              // busy wait
          }
          // critical section
          ...
          // end of critical section
          flag[1] = false;
```

- expensive (busy waiting)
- requires number of threads to be fixed statically
- assumes sequential consistency

Disable Interrupts

- Works only on a single CPU
- Cannot expose to user processes

Spinlock

```
// Abstract Lock Interface
class Lock {
    void acquire();  // Wait until lock is available, then take it
    void release();  // Release the lock
}

// Spinlock Implementation
class Spinlock implements Lock {
    private int flag = 0;  // 0 = unlocked, 1 = locked

    void acquire() {
        …
    }

    void release() {
        …
    }
}
```

Spinlock implementation I

```
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 is the problem?

```
Thread 1 A
Thread 2 A
Thread 2 B
Thread 1 B
```

Violates mutual exclusion!

Spinlock implementation II

```
/* pseudocode */
int xchg_val(addr, value) {
    %rax = value;
    xchg (*addr), %rax
}

void acquire (Spinlock *lock) {
    pushcli(); /* what does this do? */
    while (1) {
    if (xchg_val(&lock->locked, 1) == 0)
        break;
    }
}

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

(i) freeze all CPUs' memory activity for address `addr`

(ii) `temp <- *addr`

(iii) `*addr <- %rax`

(iv) `%rax <- temp`

(v) un−freeze memory activity

Spinlock implementation II

```
/* pseudocode */
int xchg_val(addr, value) {
    %rax = value;
    xchg (*addr), %rax
}

/* optimization in acquire;
call xchg_val() less frequently */
void acquire(Spinlock* lock) {
    pushcli();
    while (xchg_val(&lock->locked, 1) == 1) {
        while (lock->locked) ;
    }
}

void release(Spinlock *lock){
    xchg_val(&lock->locked, 0);
    popcli();
}
```

Busy waits!

Starvation!

Quiz Time!

Please submit both the quiz booklet and the answer sheet (with your name on both of them!)