

CS202 (003): Operating Systems

Concurrency III

Instructor: Jocelyn Chen

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

```

1 CS 202, Fall 2024
2 Handout 5 (Class 6)
3
4 The previous handout demonstrated the use of mutexes and condition
5 variables. This handout demonstrates the use of monitors (which combine
6 mutexes and condition variables).
7
8 1. The bounded buffer as a monitor
9
10 // This is pseudocode that is inspired by C++.
11 // Don't take it literally.
12
13 class MyBuffer {
14     public:
15         MyBuffer();
16         ~MyBuffer();
17         void Enqueue(Item);
18         Item = Dequeue();
19     private:
20         int count;
21         int in;
22         int out;
23         Item buffer[BUFFER_SIZE];
24         Mutex* mutex;
25         Cond* nonempty;
26         Cond* nonfull;
27 };
28
29 void
30 MyBuffer::MyBuffer()
31 {
32     in = out = count = 0;
33     mutex = new Mutex;
34     nonempty = new Cond;
35     nonfull = new Cond;
36 }
37
38 void
39 MyBuffer::Enqueue(Item item)
40 {
41     mutex.acquire();
42     while (count == BUFFER_SIZE)
43         cond_wait(&nonfull, &mutex);
44
45     buffer[in] = item;
46     in = (in + 1) % BUFFER_SIZE;
47     ++count;
48     cond_signal(&nonempty, &mutex);
49     mutex.release();
50 }
51
52 Item
53 MyBuffer::Dequeue()
54 {
55     mutex.acquire();
56     while (count == 0)
57         cond_wait(&nonempty, &mutex);
58
59     Item ret = buffer[out];
60     out = (out + 1) % BUFFER_SIZE;
61     --count;
62     cond_signal(&nonfull, &mutex);
63     mutex.release();
64     return ret;
65 }
66

```

```

67
68 int main(int, char**)
69 {
70     MyBuffer buf;
71     int dummy;
72     tid1 = thread_create(producer, &buf);
73     tid2 = thread_create(consumer, &buf);
74
75     // never reach this point
76     thread_join(tid1);
77     thread_join(tid2);
78     return -1;
79 }
80
81 void producer(void* buf)
82 {
83     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
84     for (;;) {
85         /* next line produces an item and puts it in nextProduced */
86         Item nextProduced = means_of_production();
87         sharedbuf->Enqueue(nextProduced);
88     }
89 }
90
91 void consumer(void* buf)
92 {
93     MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
94     for (;;) {
95         Item nextConsumed = sharedbuf->Dequeue();
96
97         /* next line abstractly consumes the item */
98         consume_item(nextConsumed);
99     }
100 }
101
102 Key point: *Threads* (the producer and consumer) are separate from
103 *shared object* (MyBuffer). The synchronization happens in the
104 shared object.
105

```

```

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 and modify
- allow:
 - many readers at once
 - OR
 - only one writer (no reader)

Unit of concurrency?

Shared chunks of state?

What does main function look 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  
        ...  
        turn = 0;  
        // end of critical section  
        flag[0] = false;
```

```
P1:      flag[1] = true;  
P1_gate: turn = 0;  
        while (flag[0] && turn == 0)  
        {  
            // busy wait  
        }  
        turn = 1;  
        // 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!

Mutex: spinlock + a queue

```
typedef struct thread {  
    // ... Entries elided.  
    STAILQ_ENTRY(thread_t) qlink; // Tail queue entry.  
} thread_t;
```

qlink is a field that allows each thread_t structure to be part of a singly-linked tail queue. qlink field in each thread_t is what allows these threads to be linked into that queue

```
struct Mutex {  
    // Current owner, or 0 when mutex is not held.  
    thread_t *owner;  
  
    // List of threads waiting on mutex  
    STAILQ(thread_t) waiters;  
  
    // A lock protecting the internals of the mutex.  
    Spinlock splock; // as in item 1, above  
};
```

Mutex: spinlock + a queue

```
typedef struct thread {  
    // ... Entries elided.  
    // Tail queue entry.  
    STAILQ_ENTRY(thread_t) qlink;  
} thread_t;
```

```
struct Mutex {  
    // Current owner  
    //or 0 when mutex is not held.  
    thread_t *owner;  
  
    // List of threads waiting on mutex  
    STAILQ(thread_t) waiters;  
  
    // A lock protecting  
    //the internals of the mutex.  
    Spinlock splock;  
};
```

```
void mutex_acquire(struct Mutex *m) {  
  
    acquire(&m->splock);  
  
    // Check if the mutex is held;  
    // if not, current thread gets mutex and returns  
    if (m->owner == 0) {  
        m->owner = id_of_this_thread;  
        release(&m->splock);  
    } else {  
        // Add thread to waiters.  
        STAILQ_INSERT_TAIL(&m->waiters,  
                           id_of_this_thread,  
                           qlink);  
  
        // Tell the scheduler to add  
        // current thread to the list of blocked threads.  
        sched_mark_blocked(&id_of_this_thread);  
        // Unlock spinlock.  
        release(&m->splock);  
        // Stop executing until woken.  
        sched_switch();  
        // We guaranteed to hold the mutex  
        // when we are here
```

only one thread can modify the
mutex's internal state at a time

this thread is waiting and
shouldn't be scheduled to run

allowing other threads to access
the mutex's internal state

This call switches to another
thread

This is because we can get here only if context-switched-TO, which itself can happen only if this thread is removed from the waiting queue, marked "unblocked", and set to be the owner (in mutex_release() below). However, we might have held the mutex in lines 39–42 (if we were context-switched out after the spinlock release(), followed by being run as a result of another thread's release of the mutex). But if that happens, it just means that we are context-switched out an "extra" time before proceeding.

Mutex: spinlock + a queue

```
typedef struct thread {  
    // ... Entries elided.  
    // Tail queue entry.  
    STAILQ_ENTRY(thread_t) qlink;  
} thread_t;
```

```
struct Mutex {  
    // Current owner  
    //or 0 when mutex is not held.  
    thread_t *owner;  
  
    // List of threads waiting on mutex  
    STAILQ(thread_t) waiters;  
  
    // A lock protecting  
    //the internals of the mutex.  
    Spinlock splock;  
};
```

```
void mutex_release(struct Mutex *m) {  
    // Acquire the spinlock in order to make changes.  
    acquire(&m->splock);  
  
    // Assert that the current thread  
    // actually owns the mutex  
    assert(m->owner == id_of_this_thread);  
  
    // Check if anyone is waiting.  
    m->owner = STAILQ_GET_HEAD(&m->waiters);  
  
    // If so, wake them up.  
    if (m->owner) {  
        sched_wakeone(&m->owner);  
        STAILQ_REMOVE_HEAD(&m->waiters, qlink);  
    }  
  
    // Release the internal spinlock  
    release(&m->splock);  
}
```

only one thread can modify the mutex's internal state at a time

safety check to prevent a thread from releasing a mutex it doesn't own

get the first thread from the waiters queue

If there were no waiting threads, the `m->owner` would be NULL, effectively marking the mutex as unheld.

making it ready to run.

The thread is removed from the head of the waiters queue.

What makes a good mutex implementation?

Mechanism	Pros	Cons	Best Use Case
Spinlock + Queue	- Efficient for both short and long waits - Allows context switching - Fair (FIFO ordering) - Scalable to many threads	- More complex implementation - Slightly higher overhead for uncontended case	General-purpose locking in multi-threaded environments
Pure Spinlock	- Very fast for short waits - Simple implementation	- Wastes CPU cycles for long waits - Starvation and contention	Very short-duration locks with low contention
Disabling Interrupts	- Simple to implement - Guaranteed mutual exclusion	- Only works on single-processor systems - Can increase interrupt latency - Can't be used by user-level code	Low-level OS operations on single-processor systems
Peterson's Algorithm	- Works without hardware support - Guaranteed fairness	- Limited to two threads - Busy-waiting (similar to spinlock) - Can be less efficient on modern hardware	Educational purposes, simple two-thread synchronization

Quiz Time!

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