CS202 (003): Operating Systems
Concurrency IV

Instructor: Jocelyn Chen

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

Last Time

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.

```
// 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);
    while ((AW + AR) > 0) { // check if safe to write.
                            // if any readers or writers, wait
        WW++;
        wait(&okToWrite, &mutex);
        WW--;
    }
    AW++;
    release(&mutex);
}

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

- 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!

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

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

```c
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;
};
```

```c
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_swtch();
        // 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

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

```c
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;
};
```

```c
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.

Another implementation is covered in the textbook (https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf)

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

Next lecture: reading is required!

(yes, we will quiz you about it at the beginning of the Thursday class)