Thread Synchronizations

(SGG Chapter 6)

Instructor: Dr. Tongping Liu

Lecture Outline
- Pthread mutex
- Conditional variables
- Barrier
- Threading Issues
- High-level synchronization structure: Monitor

Synchronization in Pthread Library
- Mutex variables
  - `pthread_mutex_t`
- Conditional variables
  - `pthread_cond_t`
- All POSIX thread functions have the form:
  `pthread[ _object ]_operation`
- Most of the POSIX thread library functions return 0
  in case of success and some non-zero error-number
  in case of a failure

Mutex Variables: Mutual Exclusion
- A mutex variable can be either locked or unlocked
  - `pthread_mutex_t lock;` // lock is a mutex variable
- Initialization of a mutex variable by default attributes
  - `pthread_mutex_init( &lock, NULL );`
- Lock operation
  - `pthread_mutex_lock( &lock );`
- Unlock operation
  - `pthread_mutex_unlock( &lock );`
Semaphore vs. Mutex_lock

**Definition and initialization**

```c
volatile int cnt = 0;  
sem_t mutex = 1;  
```

```c
// Initialize to Unlocked  
pthread_mutex_init(&mutex, NULL);  
```

**Entering and Exit CS**

```c
for (i = 0; i < niters; i++) {  
    Wait(&mutex);  
    cnt++;  
    Signal(&mutex);  
}  
for (i = 0; i < niters; i++) {  
    pthread_mutex_lock(&mutex);  
    cnt++;  
    pthread_mutex_unlock(&mutex);  
}  
```

Binary Semaphore vs Mutex Lock

**Binary Semaphore:**
- No ownership

**Mutex lock**
- Only the owner of a lock can release a lock.
- Priority inversion safety: potentially promote a task
- Deletion safety: a task owning a lock can’t be deleted.

Synchronization?

- Synchronization serves two purposes:
  - Ensure safety for shared updates
  - Avoid race conditions
  - Coordinate actions of threads
  - Parallel computation
  - Event notification

- All interleavings must be correct
  - there are lots of interleavings of events
  - also constrain as little as possible

Lecture Outline

- Pthread mutex
- Conditional variables
- Barrier
- Threading Issues
- High-level synchronization structure: Monitor
Synchronization Problem: Queue

- Suppose we have a thread-safe queue
  - `insert(item)`, `remove()`, `empty()`
  - must protect access with locks

- Options for removing when queue empty:
  - Return special error value (e.g., NULL)
  - Wait for something to appear in the queue

Three Possible Solutions

- Spin lock
  - Works?
  - Could release lock
    - Works?
  - Re acquire Lock

`lock();
while(empty()) {} 
unlock();
v = remove();
lock();
while (empty()) {
  unlock();
  lock();
}
v = remove();
unlock();

Works, but lots of checking...

Solution: Sleep!

- Sleep =
  - “don’t run me until something happens”
- What about this?

```c
Dequeue(){
  lock();
  if (queue empty) {
    sleep();
  }
  take one item;
  unlock();
}
```

Enqueue()

```c
lock();
insert item;
if (thread waiting)
  wake up dequeue();
unlock();
```

Cannot hold lock while sleeping!

Synchronization Operations

- Safety
  - Locks provide mutual exclusion
  - But, we need more than just mutual exclusion of critical regions...

- Coordination
  - Condition variables provide ordering
Condition Variables

- Special pthread data structure
- Make it possible/easy to go to sleep
  - Atomically:
    - release lock
    - put thread on wait queue
    - go to sleep
- Each CV has a queue of waiting threads
- Do we worry about threads that have been put on the wait queue but have NOT gone to sleep yet?
  - No, because those two actions are atomic
- Each CV is associated with one lock

Condition Variable Exercise

- Implement “Producer Consumer”
  - One thread enqueues, another dequeues

```c
void *consumer_dequeue(){
    void * item = NULL;
    pthread_mutex_lock(&l);
    while (q.empty()) {
        pthread_cond_wait(&nempty, &l);
    }
    item = q.pop_back();
    pthread_cond_signal(&nfull);
    pthread_mutex_unlock(&l);
    return item;
}

void producer_enqueue(void *item){
    pthread_mutex_lock(&l);
    while (q.full()) {
        pthread_cond_wait(&nfull, &l);
    }
    q.push_front(item);
    pthread_cond_signal(&nempty);
    pthread_mutex_unlock(&l);
}
```

- Questions?
  - Can I use if instead of while (to check cond)?

Condition Variable vs. Semaphore

```c
void *consumer_dequeue(void){
    void * item = NULL;
    pthread_mutex_lock(&l);
    while (q.empty()) {
        pthread_cond_wait(&nempty, &l);
    }
    item = q.pop_back();
    pthread_cond_signal(&nfull);
    pthread_mutex_unlock(&l);
    return item;
}

void producer_enqueue(void *item){
    void * item = NULL;
    pthread_mutex_lock(&l);
    while (q.full()) {
        pthread_cond_wait(&nfull, &l);
    }
    q.push_front(item);
    pthread_cond_signal(&nempty);
    pthread_mutex_unlock(&l);
}
```

- void producer_enqueue: (void *) item
- void consumer_dequeue: (void *) item
- void producer_enqueue: (void *) item
- void consumer_dequeue: (void *) item
- void producer_enqueue: (void *) item
- void consumer_dequeue: (void *) item
- void producer_enqueue: (void *) item
- void consumer_dequeue: (void *) item
Lecture Outline

- Pthread mutex
- Conditional variables
- Barrier
- Threading Issues
- High-level synchronization structure: Monitor

Barrier

- A barrier is used to order different threads (or a synchronization).
  - A barrier for a group of threads means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

- Where it can be used?
  - Meeting.

Barrier Interfaces

- It is used when some parallel computations need to "meet up" at certain points before continuing.
- Pthreads extension includes barriers as synchronization objects
  - (available in Single UNIX Specification)
    - Enable by #define _XOPEN_SOURCE 600 at start of file
- Initialize a barrier for count threads
  - int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrier attr_t *attr, int count);
- Each thread waits on a barrier by calling
  - int pthread_barrier_wait(pthread_barrier_t * barrier);
- Destroy a barrier
  - int pthread_barrier_destroy(pthread_barrier_t * barrier);
Synchronization – Barrier (2)

```c
int main () // ignore arguments {
    time_t now;
    // create a barrier object with a count of 3
    pthread_barrier_init (&barrier, NULL, 3);
    // start up two threads, thread1 and thread2
    pthread_create (NULL, NULL, thread1, NULL);
    pthread_create (NULL, NULL, thread2, NULL);
    time (&now);
    printf("main() before barrier at %s", ctime (&now));
    pthread_barrier_wait (&barrier);
    // Now all three threads have completed.
    time (&now);
    printf("main() after barrier at %s", ctime (&now));
    pthread_exit (NULL);
    return (EXIT_SUCCESS);
}

void * thread1 (void * not_used) {
    time (&now);
    printf("T1 starting %s", ctime (&now));
    sleep (20);
    pthread_barrier_wait (&barrier);
    time (&now);
    printf("T1: after barrier at %s", ctime (&now));
}

void * thread2 (void * not_used) {
    time (&now);
    printf("T1 starting %s", ctime (&now));
    sleep (20);
    pthread_barrier_wait (&barrier);
    time (&now);
    printf("T1: after barrier at %s", ctime (&now));
    time (&now);
    printf("T2: after barrier at %s", ctime (&now));
    time (&now);
    printf("T2: after barrier at %s", ctime (&now));
    return (EXIT_SUCCESS);
}
```

Lecture Outline

- Pthread mutex
- Conditional variables
- Barrier
- Threading Issues
- High-level synchronization structure: Monitor

Semaphore Issues

```
signal(mutex);
    critical section
    wait(mutex);
    Deadlocks  no mutual exclusion
    "signal(mutex);
```

Lock Problems

- Race condition
- Atomicity violation
- Order violation
- Deadlock
Race Conditions

- Multiple processes/threads write/read shared data and the outcome depends on the particular order to access shared data are called race conditions.
- A serious problem for concurrent system using shared variables!

*How do we solve the problem?*

- Need to make sure that some high-level code sections are executed **atomically**
  - Atomic operation means that it completes in its entirety without worrying about interruption by any other potentially conflict-causing process.

Atomically Violation

![Figure 1. A simple example of an atomically violation. The read and update of counter from two threads may interfere such that the counter is incremented only once.](image)

Order Violation

- a program depends on a sequence of threads that the scheduler may not provide.

Deadlocks

- **Thread1:**
  - Lock L1;
  - Lock L2;
  - Unlock L2;
  - Unlock L1;

- **Thread2:**
  - Lock L2;
  - Lock L1;
  - Unlock L1;
  - Unlock L2;
Issues Related to Conditional Variables

- `signal()` before `wait()`
  - Waiting thread may miss the signal
- Fail to lock `mutex` before `wait`
  - May return error, or not blocking
- `if (!condition) wait();` instead of `while (!condition) wait();`
  - Condition may still fail when woken up
  - May lead to arbitrary errors, such as segmentation fault
- Forgot to unlock `mutex` after `signal/wakeup`
- Holding multiple locks with `wait`

Lecture Outline

- Pthread mutex
- Conditional variables
- Barrier
- Threading Issues
- High-level synchronization structure: Monitor

Monitors

- **High-level synchronization construct** (implement in different languages) that provided mutual exclusion within the monitor AND the ability to wait for a certain condition to become true

  ```
  monitor monitor-name{
    shared variable declarations
    procedure body P1 (...) { . . . }
    procedure body P2 (...) { . . . }
    procedure body Pn (...) { . . . }
    (initialization codes; }
  }
  ```

A monitor is like a building that contains one special room that can be occupied by only one thread at a time. The room usually contains some data. From the time a thread enters this room to the time it leaves, it has exclusive access to any data in the room. Entering the monitor building is called "entering the monitor." Entering the special room inside the building is called "acquiring the monitor." Occupying the room is called "owning the monitor," and leaving the room is called "releasing the monitor." Leaving the entire building is called "exiting the monitor."
Monitor construct ensures at most one thread can be active within the monitor at a given time. Shared data (local variables) of the monitor can be accessed only by local procedures.

Monitors vs. Object

A Monitor:
- An object designed to be accessed across threads
- Member functions enforce mutual exclusion

A Semaphore:
- A low-level object
- We can use semaphore to implement a monitor

class Account {
    private lock myLock;
    private int balance = 0;
    public synchronized boolean withdraw(int amount) {
        int ret;
        myLock.acquire();
        if (balance < amount) then ret = false 
        else { balance = balance - amount; ret = true; }
        myLock.release();
        return ret;
    }
    public synchronized void deposit(int amount) {
        myLock.acquire();
        balance = balance + amount;
        myLock.release();
    }
}

Java’s Monitors Support

- In the Java language, you mark critical sections in your program with the synchronized keyword.
- Generally, critical sections in Java programs are methods. You can mark smaller code segments as synchronized. However, this violates object-oriented paradigms and leads to confusing code that is difficult to debug and maintain. For the majority of your Java programming purposes, it’s best to use synchronized only at the method level.

Implement shared buffer with Monitors

class SharedBuffer {
    private Pointer[] buffer;
    private int count = 0;
    private int in = 0, out = 0;
    SharedBuffer(int size) {
        buffer = new Pointer[size];
        buffer[in] = obj;
        in = (in + 1) % buffer.length;
        count++;
        notify();
    }
}

http://www.csc.villanova.edu/~mdamian/threads/javamonitors.html
Implement shared buffer with Monitors

```java
public synchronized Pointer Get() {
    while (count == 0) {
        try { wait();  }
        catch (InterruptedException e) { XXX; }
    }
    Pointer object = buffer[out];
    out = (out + 1) % buffer.length;
    count--;
    notify();
    return object;
}
```

Summarization of Monitor

- A thread may call wait(), notify() or notifyAll() on an object, only if it owns the monitor of that object.
- If a class has one or more synchronized methods, each object of the class gets a queue that holds all threads waiting to execute one of those synchronized methods.
- There are two ways for a thread to get onto this queue, either by calling the method while another thread is using the object, or by calling wait() while using the object.

Rules of Utilizing Monitors

1. If two or more threads modify an object, declare the methods that carry out the modification as synchronized.
2. If a thread must wait for the state of an object to change, it should wait inside the object, not outside, by entering the synchronized method and calling wait().
3. Whenever a method changes the state of an object, it should call notify(). That gives the waiting threads a chance to see if circumstances have changed.
Explicit Locking vs. Implicit Locking

1. High flexibility and programmatic control:
   - **Pros:** In synchronized method and statement, JVM does automatic management of lock (release of lock) and developer does not have any control over it. Lock interface gives programmatic control and flexibility over Lock by providing API for non-blocking attempts.
   - **Cons:** Lock is not easy to use, and it is developer responsibility to acquire lock and release appropriately. For synchronized methods, JVM does automatic release of lock and optimization without forgetting.


2. Chain locking supported:
   - **Pros:** Synchronized method allows multiple locking, does not allow chain locking. That is, all locks released by JVM in the same lexical scope in which they were acquired and released in opposite order. But explicit locks supports chaining, which allows multiple locks to be acquired and released in any order.

   [Diagram: Chain locking supported]

   - **Cons:** Synchronized method makes it much easier to program with monitor locks, avoiding common programming errors involving locks, such as deadlocks. With Locks, the increased flexibility comes additional responsibility.

   ```java
   Lock lockObj = new ReentrantLock(); // or others locks
   lockObj.lock();
   try {
       // access the resource protected by this lock
   } finally {
       lockObj.unlock();
   }
   ```

3. Performance improvement:
   - **Pros:** ReentrantLock (a concrete implementation of Lock interface) offered better performance than intrinsic locking, but not any more in 6.0. But performance should not be criteria to select it over implicit locking.

   [Graph: Performance improvement]
Explicit Locking vs. Implicit Locking

4. Fairness locking support:
   - ReentrantLock offers two types of lock: a fair lock and unfair lock (default). Fair lock will be acquired in the same order as the requested. By default, implicit locking provide unfair locking.
   - Performance of non-fair lock is better than fair lock. In fair lock, pause of one thread and start another thread causes a substantial overhead and with increase threads count performance degraded.

Fair Versus Non-fair Lock Performance

Conclusion

- Implicit lock is very easy to use and JVM does lock management for it. However, ReentrantLock has an edge over implicit lock in terms of performance gain.
- It is recommended to prefer implicit locking (synchronized methods and statements) over explicit locking unless we need to use explicit locking advanced features: performance, polled, or interruptible lock acquisition, fair queueing, or non block structured locking.

Lecture Conclusion

- High-level synchronization structure: Monitor
- Pthread mutex
- Conditional variables
- Barrier
- Threading Issues