CS3733: Operating Systems

Topics: Thread Synchronizations (SGG Chapter 6)

Instructor: Dr. Tongping Liu
Midterm2 Statistics

- Highest: 90
- Lowest: 42
- Average:
  - 73.5
- Average with > 60%
  - 79.4
- Average for < 60% attendance
  - 64
Few Examples

> 4 attendances
- 70.5 → 82.5
- 77 → 85

< 2 attendances
- 85 → 59.5
- 91 → 65
Lecture Outline

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

```plaintext
monitor monitor-name{
  shared variable declarations

  procedure body P1 (...) { . . . }
  procedure body P2 (...) { . . . }
  procedure body Pn (...) { . . . }

  {initialization codes; } }
```
monitors vs. semaphores

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
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**.
class Account {
    private lock myLock;

    private int balance := 0
    invariant balance >= 0

    public method boolean withdraw(int amount)
        precondition amount >= 0
        {
            int ret;
            myLock.acquire();
            if balance < amount then ret = false
            else { balance := balance - amount ; ret = true }
            myLock.release();
        }

    public method deposit(int amount)
        precondition amount >= 0
        {
            myLock.acquire();
            balance := balance + amount
            myLock.release();
        }
}
Lecture Outline

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

- Mutex variables
  - `pthread_mutex_t`

- Conditional variables
  - `pthread_cond_t`

- All POSIX thread functions have the form:
  
  \[ \textit{pthread[\_object \_operation} \]

- 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
volatile int cnt = 0;
pthread_mutex_t mutex;

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

**Entering and Exit CS**

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

```c
for (i = 0; i < niters; i++) {
    pthread_mutex_lock(&mutex);
    cnt++;
    pthread_mutex_unlock(&mutex);
}
```
Binary Semaphore and 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

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

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

unlock();
while (empty()) {}
lock();
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();
}
```

```c
Enqueue()
{
    lock();
    insert item;
    if (thread waiting)
        wake up dequeler();
    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 condition variable associated with one lock
Condition Variables

- Wait for 1 event, atomically release lock
  - `wait(Lock& l, CV& c)`
    - If queue is empty, wait
      - Atomically releases lock, goes to sleep
      - You must be holding lock!
      - May reacquire lock when awakened (pthreads do)
  - `signal(CV& c)`
    - Insert item in queue
      - Wakes up one waiting thread, if any
  - `broadcast(CV& c)`
    - Wakes up all waiting threads
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 * 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);
    return item;
}

void * consumer_dequeue (void){
    void * item = NULL;
    wait(full);
    wait(mutex);
    item = q.pop_back;
    signal(mutex);
    signal(empty);
    return item;
}

void producer_enqueue(void *item){
    wait(empty);
    wait(mutex);
    q.push_front (item);
    signal(mutex);
    signal(full);
    return item;
}
```
Lecture Outline

- High-level synchronization structure: Monitor
- Pthread mutex
- Conditional variables
- Barrier
- Threading Issues
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?
  - Watching movie.
Barrier

"Thread" 1

"Thread" 2

"Thread" 3

Epoch0  Epoch1  Epoch2

Time
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);
    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 ("main() before barrier at %s", ctime (&now));
    pthread_barrier_wait (&barrier);
    time (&now);
    printf ("T2: after barrier at %s", ctime (&now));
    time (&now);
    printf ("main() after barrier at %s", ctime (&now));
    pthread_exit( NULL );
    return (EXIT_SUCCESS);
}
```
Lecture Outline

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

\[
\begin{align*}
\text{signal}(\text{mutex}); & \quad \text{wait}(\text{mutex}); \\
\ldots & \quad \ldots \\
\text{critical section} & \quad \text{critical section} \\
\ldots & \quad \ldots \\
\text{wait}(\text{mutex}); & \quad \text{wait}(\text{mutex}); \\
\end{align*}
\]

Revise order, no mutual exclusion

Deadlocks no mutual exclusion

Critical Section

\[\text{signal}(\text{mutex});\]
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.
Atomicity Violation

```c
int counter; // shared variable
    // protected by lock L

void increment() {
    int temp;
    lock (L);
    temp = counter;
    unlock (L);
    temp++;
    lock (L);
    counter = temp;
    unlock (L);
}
```

**Figure 1.** A simple example of an atomicity violation. The read and update of `counter` from two threads may interleave such that the counter is incremented only once.
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 waken up
  - May lead to arbitrary errors, such as segmentation fault

- Forgot to unlock mutex after signal/wakeup

https://courses.engr.illinois.edu/cs241/sp2014/lecture/22-condition-deadlock.pdf
Lecture Conclusion

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