CS 360 Internet Programming

Concurrent Programming

Shared Memory Synchronization with Mutexes and Condition Variables

Daniel Zappala
Computer Science
Brigham Young University
1 Mutual Exclusion
   - The Need for Mutual Exclusion
   - Mutual Exclusion

2 Mutexes and Condition Variables
   - Mutexes
   - Condition Variables
A Simple Example

```c
void echo()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

- shared method among multiple threads
- unpredictable execution sequence
  - Thread1 executes up to conclusion of input function, then interrupted: Thread1 reads 'x'
  - Thread2 executes completely through the procedure: Thread2 reads and prints 'y'
  - Thread1 starts again: Thread1 input variable has 'y'!
Concurrency Problems

- demonstration
- problems
  - can’t predict the speed with which threads will execute and therefore when a resource will be accessed
  - if synchronization is not used, errors will be rare but they will occur
  - errors are hard to duplicate and debug since they are nondeterministic
Concurrency Problems

- demonstration
  - problems
    - can’t predict the speed with which threads will execute and therefore when a resource will be accessed
    - if synchronization is not used, errors will be rare but they will occur
    - errors are hard to duplicate and debug since they are nondeterministic
Mutual Exclusion

- need to protect shared resources (e.g. global variable, shared data structures) among multiple processes or threads
  - atomic access to method
  - when Thread2 tries to enter method, block it until Thread1 is finished
- may involve processes or threads interleaved in time on a single processor or running in parallel on a multiprocessor machine
- result of process or thread must be independent of the speed of execution of other concurrent processes
Mutual Exclusion

- **critical section**: shared portion of code that must be executed by one thread at a time
  - thread must mark the critical section because OS doesn’t know where it is
- **starvation**: one or more threads are prevented from ever executing critical section
- **deadlock**: situation in which no thread can make progress because they are all waiting for a critical section
- must ensure data coherence, e.g. atomic access to a database
Mutual Exclusion Requirements

- only one thread has access to critical section at a time
- halting in non-critical section must not interfere with other threads
- no indefinite wait for critical section, i.e. no starvation or deadlock
- if no thread in critical section, then no wait to enter
- no assumptions about process speeds or number of processors
- thread may only spend finite time within critical section
Solutions

- software
  - assume no support from OS, hardware, or language
  - historic algorithms: Dekker, Peterson, Lamport
  - difficult to get right, to generalize

- hardware
  - disable interrupts: single processor machines
    - no other process can run until they are re-enabled
    - limits flexibility of OS to schedule threads, doesn’t work for multiprocessors
  - atomic machine instructions: test-and-set

- operating system support
Test-and-Set Example

```java
1  boolean test_and_set(int i) {
2       if (i == 0) {
3           i = 1;
4           return true;
5       } else {
6           return false;
7       }
8  }

9  int bolt = 0;
10 void method(int i) {
11     while (true) {
12         while (!test_and_set(bolt))
13             /* do nothing */
14             /* critical section */
15         bolt = 0;
16         /* remainder */
17     }
18 }
```

Daniel Zappala
Computer Science
Brigham Young University
Test-and-Set Pros and Cons

- advantages
  - applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - it is simple and therefore easy to verify
  - it can be used to support multiple critical sections: each section gets its own variable

- disadvantages
  - busy-waiting consumes processor time
  - starvation is possible when more than one process waits
  - deadlock
    - low priority process has the critical region
    - higher priority process needs it
    - higher priority process obtains the processor and waits for the critical region
Operating System Support

- **mutex and condition variable**
  - **mutex**: lock that allows only one thread into a critical section
  - **condition variable**: signal conditions between threads

- **semaphore**
  - when one thread is in the critical section, others may **wait** by sleeping
  - when thread is done with critical section, it wakes one other thread with a **signal**

- **monitor**
  - programming language construct that makes it easier to declare and use a critical section
  - construct a class with methods, only one thread may access a method of the class at a time

- **message passing**
  - synchronization by explicitly exchanging messages
  - define a mailbox and enter critical section when a message is waiting
Mutex

- lock that allows only one thread into a critical section

```c
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- must initialize the mutex first
- `pthread_mutex_lock()` will block if mutex is already locked
- `pthread_mutex_trylock()` will return EBUSY if mutex is locked
- `demonstration`
Busy Waiting

```
while running {
  c = NULL;
pthread_mutex_lock(&mutex);
  if queue.not_empty() {
    c = queue.dequeue();
  }
pthread_mutex_unlock(&mutex);
  if c {
    /* handle connection */
  }
}
```

- must busy wait until a connection is available
- wastes CPU time on a server that does not handle many connections
Condition Variables

```c
#include <pthread.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_wait(pthread_cond_t *cond,
                       pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t);
```

- must initialize the condition variable first
- `pthread_cond_wait()` will block until the condition is signaled; the thread now owns the mutex as well
- need a corresponding `pthread_cond_signal()` to wake up
Using Condition Variables

```c
while running {
    c = NULL;
    pthread_mutex_lock(&mutex);
    while queue.empty() {
        pthread_cond_wait(&cond,&mutex);
    }
    c = queue.dequeue();
    pthread_mutex_unlock(&mutex);
    /* handle connection */
}
```

- process inserting into queue should signal condition when queue goes from empty to having at least one item
- **must re-check queue status when conditional wait returns**
- no guarantee that queue will be empty when you return
Timed Wait and Broadcast Signals

```c
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

- `pthread_cond_timedwait()` needs an absolute time; use `clock_gettime()` and add the length of time you want to wait.
- `pthread_cond_broadcast()` wakes up all threads waiting for a signal.