Synchronization Primitives

Synchronization Mechanisms

- Locks
  - Very primitive constructs with minimal semantics
- Semaphores
  - A generalization of locks
  - Easy to understand, hard to program with
- Condition Variables
  - Constructs used in implementing *monitors* (more on this later…)

Locks

- Synchronization mechanisms with 2 operations: acquire(), and release()
- In simplest terms: an object associated with a particular critical section that you need to "own" if you wish to execute in that region
- Simple semantics to provide mutual exclusion:

```
acquire(lock);
    //CRITICAL SECTION
release(lock);
```

- Downsides: spinlock has performance issues
  - Can cause deadlock if not careful i.e. not releasing lock properly
  - Cannot allow multiple concurrent accesses to a resourc

POSIX Locks

- POSIX locks are called mutexes (since locks provide mutual exclusion…)
- A few calls associated with POSIX mutexes:

pthread_mutex_init (mutex, attr)

  - Initialize a mutex

pthread_mutex_destroy (mutex)

  - Destroy a mutex

pthread_mutex_lock (mutex) blocked if lock not available

  - Acquire the lock

pthread_mutex_trylock(mutex) return value indicates if lock is available, up to programmer to decide what to do

  - Try to acquire the lock (more on this later…)

pthread_mutex_unlock (mutex)

  - Release the lock

Initializing & Destroying POSIX Mutexes

- POSIX mutexes can be created statically or dynamically
  - Statically, using PTHREAD_MUTEX_INITIALIZER

    pthread_mutex_t mx = PTHREAD_MUTEX_INITIALIZER;

    - Will initialize the mutex will default attributes
    - Only use for static mutexes; no error checking is performed
  - Dynamically, using the pthread_mutex_init call

    int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t * attr);

    - mutex: the mutex to be initialized
    - attr: structure whose contents are used at mutex creation to determine the mutex's attributes
      - Same idea as pthread_attr_t attributes for threads
- Destroy using pthread_mutex_destroy

  int pthread_mutex_destroy(pthread_mutex_t *mutex);

  - mutex: the mutex to be destroyed
    - Make sure it's unlocked! (destroying a locked mutex leads to undefined behaviour…)

Acquiring and Releasing POSIX Locks

- Acquire

  int pthread_mutex_lock(pthread_mutex_t *mutex);

  - mutex: the mutex to lock (acquire)
  - If mutex is already locked by another thread, the call will block until the mutex is unlocked

  int pthread_mutex_trylock(pthread_mutex_t *mutex);

  - mutex: the mutex to TRY to lock (acquire)
  - If mutex is already locked by another thread, the call will return a “busy” error code (EBUSY)

- Release

  int pthread_mutex_unlock(pthread_mutex_t *mutex);

  - mutex: the mutex to unlock (release)

Banking Example

- Bank account balance maintained in one variable int balance
- Transactions: deposit or withdraw some amount from the account (+/- balance)
- Unprotected, concurrented accesses to your balance could create <u>race conditions</u>

Banking Example

- Thread 1 withdraws 100

int new_balance = balance –
amount;

balance = new_balance;

- Thread 2 withdraws 100

int new_balance = balance –
amount;

balance = new_balance;

- End with balance – 100 instead of balance – 200
- Bank error in your favour? Cold be the other way around!
- Idea: put a lock around the code that modifies balance so only a single thread accesses it at any given time

Banking Example

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS200
int balance=0;
pthread_mutex_t bal_mutex;

int main (int argc, char *argv[]){
  pthread_t thread[NUM_THREADS];
  int rc;
  long t;
  void *status;

  pthread_mutex_init(&bal_mutex, NULL);
  for(t=0; t<NUM_THREADS; t+=2) {
    rc = pthread_create(&thread[t], NULL, deposit, (void *)10);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
     }
    rc = pthread_create(&thread[t+1], NULL, widthdraw, (void *)10);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
     }
  }
(...)
```

thread array for concurrent access

Banking Example

```
(…)
  for(t=0; t<NUM_THREADS; t++) {
    rc = pthread_join(thread[t], &status);
    if (rc) {
      printf("ERROR; return code from pthread_join() is %d\n", rc);
      exit(-1);
    }
  }
  printf("Final Balance is %d.\n", balance);
  pthread_exit(NULL);
}
```

Banking Example - Transactions

```
void *deposit(void *amt){

pthread_mutex_lock(&bal_mutex);

  //CRITICAL SECTION
  int amount = (int)amt;
  int new_balance = balance +
amount;
  balance = new_balance;

pthread_mutex_unlock(&bal_mutex)
;

  pthread_exit((void *)0);
}
```

```
void *withdraw(void *amt){

pthread_mutex_lock(&bal_mutex);

  //CRITICAL SECTION
  int amount = (int)amt;
  int new_balance = balance -
amount;
  balance = new_balance;

pthread_mutex_unlock(&bal_mutex)
;

  pthread_exit((void *)0);
}
```

Semaphore

- Synchronization mechanism that generalizes locks to more than just "acquired" and "free" (or "released")
- A semaphore provides you with:
  - An integer count accessed through 2 atomic operations
  - Wait - aka: down, decrement, P (for proberen)

    taking a spot
    - Block until semaphore is free, then decrement the variable
  - Signal - aka: up, post, increment, V (for verhogen)

    giving up a spot
    - Increment the variable and unblock a waiting thread (if there are any)

mutex = binary semaphore !

- A mutex was just a binary semaphore (remember pthread_mutex_lock blocked if another thread was holding the lock)
- A queue of waiting threads

POSIX Semaphores

- Declared in semaphore.h
- A few calls associated with POSIX semaphores:

  sem_init
    - Initialize the semaphore

  sem_wait
    - Wait on the semaphore (decrement value)

  sem_post
    - Signal (post) on the semaphore (increment value)

  sem_getvalue
    - Get the current value of the semaphore

  sem_destroy
    - Destroy the semaphore

Initializing & Destroying POSIX Semaphores

- Initialize semaphores using sem_init

  int sem_init(sem_t *sem, int pshared, unsigned int value);

  - sem: the semaphore to initialize
  - pshared: non-zero to share between processes

    defaults to be shared between threads
  - value: initial count value of the semaphore

    the integer count

- Destroy semaphores using sem_destroy

  int sem_destroy(sem_t *sem);

  - sem: semaphore to destroy
- Semaphore must have been created using sem_init
- Destroying a semaphore that has threads blocked on it is undefined.

every threads has to be unblocked

Decrementing & Incrementing POSIX Semaphores

- Decrement semaphores using sem_wait

  int sem_wait(sem_t *sem);

  - sem: the semaphore to decrement (wait on)

- Increment semaphores using sem_post

  int sem_post(sem_t *sem);

  - sem: semaphore to increment

- Let's look at an example of a very simple server simulation…

Server Example

```
(...)
#define NUM_THREADS200
#define NUM_RESOURCES10sem_t resource_sem; //Sempahore declaration

int main (int argc, char *argv[])
{   pthread_t thread[NUM_THREADS];
   int rc;
   int i;
   void *status;
      sem_init(&resource_sem, 0, NUM_RESOURCES); //Resource Semaphore

   for(i=0; i<NUM_THREADS; i++) {

      rc = pthread_create(&thread[i], NULL, handle_connection, (void *)i);
      if (rc) {
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1);
        }
   }
(…)
   for(i=0; i<NUM_THREADS; i++) {
      rc = pthread_join(thread[i], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join() is %d\n", rc);
         exit(-1);
      }
   }
  return 0;
 } //End of main
```

Server Example – Connection Handler

```
void *handle_connection(void *client){
  printf ("Handler for client %d created!\n", (int)client);

  sem_wait(&resource_sem);

  //DO WORK TO HANDLE CONNECTION HERE
  sleep(1);
  printf ("Done servicing client %d\n", (int) client);

  sem_post(&resource_sem);

  pthread_exit((void *)0);
}
```

Condition Variables

- Another useful synchronization construct used in implementing monitors - only a single process execute inside the monitor
- Locks control thread access to data; condition variables allow threads to synchronize based on the value of the data.
- Alternative to condition variables is to constantly poll the variable (from the critical section)
  - BAD!
  - Ties up a lot of CPU resources
  - Could potentially lead to synchronization problems
- Monitors support suspending execution within the monitor
  - wait() (suspend the invoking process and release the lock)
  - signal() (resume exactly one suspended process)
  - broadcast() (resumes all suspended processes)
  - If no process is suspended, signal/broadcast has no effect (in contrast to semaphores, where signal always changes state of the semaphore)

POSIX Condition Variables

- POSIX condition variables: pthred_cond_t
- A few calls associated with POSIX CVs:

int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);

  - Initialize a condition variable

int pthread_cond_destroy(pthread_cond_t *cond);

  - Destroy a condition variable

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

  - Wait on a condition variable

int pthread_cond_signal(pthread_cond_t *cond);

  - Wake up one thread waiting on this condition variable

int pthread_cond_broadcast(pthread_cond_t *cond);

  - Wake up all threads waiting on this condition variable

Using Condition Variables (from LLNL tutorial)

<table>
<tr><td colspan="2">Main Thread
- Declare and initialize global data/variables which require synchronization (such as "count")
- Declare and initialize a condition variable object
- Declare and initialize an associated mutex
-Create threads A and B to do work</td></tr>
<tr><td>Thread A
- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call pthread_cond_wait() to perform a blocking wait for signal from Thread-B. Note that a call to pthread_cond_wait()automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue</td><td>Thread B
- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue</td></tr>
<tr><td colspan="2">Main Thread: Join / Continue</td></tr>
</table>

Monitors

- Locks
  - Provide mutual exclusion
  - 2 operations: acquire() and release()
- Semaphores
  - Generalize locks with an integer count variable and a thread queue
  - 2 operations: wait() and signal()
  - If the integer count is negative, threads wait in a queue until another thread signals the semaphore
- Monitors
  - An abstraction that encapsulates shared data and operations on it in such a way that only a single process at a time may be executing “in” the monitor

More on Monitors

- Programmer defines the scope of the monitor
  - ie: which data is “monitored”
- Local data can be accessed only by the monitor’s procedures (not by any external procedures)
- Before any monitor procedure may be invoked, mutual exclusion must be guaranteed
  - There is often a lock associated with each monitored object
- Other processes that attempt to enter the monitor are blocked. They must first acquire the lock before becoming active in the monitor

Complications With Monitors

- Complication
  - A process may need to wait for something to happen
    - Input from another thread might be necessary for example
  - The other thread may require access to the monitor to produce that event
- Solution?
  - Monitors support suspending execution within the monitor
    - wait() (suspend the invoking process and release the lock)
    - signal() (resume exactly one suspended process)
    - broadcast() (resumes allsuspended processes)
      - If no process is suspended, signal/broadcast has no effect (in contrast to semaphores, where signal always changes state of the semaphore)

Monitor signal() ; who goes first?

- Suppose P executes a signal operation that would wake up a suspended process Q
  - Either process can continue execution, but both cannot simultaneously be active in the monitor
- Who goes first?
  - Hoare monitors: waiter first
    - signal() immediately switches from the caller to a waiting thread
    - Condition that the waiter was blocked on is guaranteed to hold when the waiter resumes
  - Mesa monitors: signaler first
    - signal() places a waiter on the ready queue, but signaler continues inside the monitor
    - Condition that the waiter was blocked on is not guaranteed to hold when the waiter resumes (must check again...)

Hoare vs. Mesa Monitors

- Hoare monitor wait

```
if(...){
    wait(cv, lock);
}
```

- Mesa monitor wait

```
while(...){
    wait(cv, lock);
}
```

- Tradeoffs
  - Hoare monitors are easier to reason with, but hard to implement
  - Mesa monitors are easier to implement, and support additional operations like broadcast()

Monitor Example - Bounded Buffers

- We have a buffer of limited size N
  - Producers add to the buffer if it is not full
  - Consumers remove from the buffer if it is not empty
- Want to control buffer as a monitor

  i.e. read/write buffer are procedures inside of monitor
  - Buffer can only be accessed by methods that are "part of" the monitor, that only give one producer or consumer access to the buffer at a time
- Need 2 functions
  - add_to_buffer()
  - remove_from_buffer()
- Need
  - One lock
  - Two conditions
    - One for producers to wait
    - One for consumers to wait

Monitor Example - Bounded Buffers

```
#define N 100
typedef struct buf_s {
        int data[N];
        int inpos; /* producer inserts here */
        int outpos; /* consumer removes from here */
        int numelements; /* # items in buffer */
        struct lock *mylock; /* access to monitor */
        struct cv *notFull; /* for producers to wait */
        struct cv *notEmpty; /* for consumers to wait */
} buf_t;

buf_t buffer;
void add_to_buff(int value);
int remove_from_buff();
```

Monitor Example - Bounded Buffers

What kind of monitor is this?

mesa monitor

```
void add_to_buf(int value) {
        lock_acquire(buffer.mylock);
        while (nelements == N) {
                /* buffer is full, wait */
                cv_wait(buffer.notFull, buffer.mylock);
        }
        buf.data[inpos] = value;
        inpos = (inpos + 1) % N;
        nelements++;
        cv_signal(buffer.notEmpty, buffer.mylock);
        lock_release(buffer.mylock);
}
```

Monitor Example - Bounded Buffers

```
int remove_from_buf() {
        int val;
        lock_acquire(buffer.mylock);
        while (nelements == 0) {
                /* buffer is empty, wait */
                cv_wait(buffer.notEmpty, buffer.mylock);
        }
        val = buf.data[outpos];
        outpos = (outpos + 1) % N;
        nelements--;
        cv_signal(buffer.notFull, buffer.mylock);
        lock_release(buffer.mylock);
}
```