CSC369 Tutorial 4
Synchronization Primitives
(continued)

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)

Semaphores

- Synchronization mechanism that generalizes locks to more than just “acquired” and “free” (or “released”)
- A semaphore provides you with:
  - An integer counter accessed through 2 atomic operations
  - Wait - aka: down, decrement, P (for proberen)
    - Block until semaphore is free (counter > 1), then decrement the counter
  - Signal - aka: up, post, increment, V (for verhogen)
    - Increment the counter and unblock one waiting thread (if there are any)
  - A queue of waiting threads
- A mutex is just a binary semaphore
  - remember pthread_mutex_lock() blocks if another thread is holding the lock

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 (e.g. after fork)
  - value: initial count value of the semaphore

- 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

- No static initializer. Why?

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

Semaphores: (main) use case

- Allow only up to N threads to access a resource at any point in time

```
sem_init(&semaphore, N);
…
sem_wait(&semaphore);
// Only up to N threads can be executing here
sem_post(&semaphore);
```

  - Example: server accepting ≤ N clients simultaneously

- After N threads have executed sem_wait(), thread # N+1 will block until one of them releases the semaphore

Server Example

```
//...
#define NUM_THREADS 200
#define NUM_RESOURCES 10
sem_t resource_sem; // Semaphore declaration

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

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

Server Example – Connection Handler

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

    sem_wait(&resource_sem);

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

    sem_post(&resource_sem);

    return NULL;
}
```

Condition Variables

- Another useful synchronization construct used in implementing monitors; only a single process executes inside the monitor
- Locks control thread access to data; condition variables allow threads to synchronize based on the value of the data
  - a thread can notify another thread that a condition is now satisfied
- Alternative to condition variables is to constantly poll the data (from the critical section); it’s a bad idea:
  - Ties up a lot of CPU resources
  - Could potentially lead to synchronization problems
- Conditional variable operations:
  - 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: pthread_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
  - Also **PTHREAD_COND_INITIALIZER**

  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; *mutex* is released during wait

  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

POSIX Condition Variables

- int **pthread_cond_wait**(pthread_cond_t *cond,
  *pthread_mutex_t *mutex*);
  - Two steps:
    1. **Atomically** release *mutex* and start waiting for *cond* to be signaled
    2. When *cond* is signaled by another thread, **atomically** acquire *mutex* and stop waiting, continue execution

- Signaling thread must release the mutex after calling *pthread_cond_signal() / pthread_cond_broadcast()* before *pthread_cond_wait()* can return

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”
  - basically, “monitor” == “synchronized object”
- 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 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)

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()
  - Pthread conditional variables implement Mesa semantics

Monitors implementation

- Implemented (e.g.) using pthreads conditional variables and mutexes
- One mutex (to protect access to data)
- Possibly multiple condition variables
  - one per each condition to be signaled/waited on

- Note: mutex necessary for pthread conditional variables even if not used for monitors
  - mutex protects the conditional variable itself;
    race between waiting thread and signaling thread
    - signal/broadcast do nothing if no waiting threads

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
  - 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;// # of items in buffer
    struct lock *lock;// 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

```
void add_to_buf(int value)
{
    lock_acquire(buffer.lock);

    while (buffer.numelements == N) {
        // buffer is full, wait
        cv_wait(buffer.notFull, buffer.lock);
    }

    buffer.data[buffer.inpos] = value;
    buffer.inpos = (buffer.inpos + 1) % N;
    buffer.numelements++;

    cv_signal(buffer.notEmpty);
    lock_release(buffer.lock);
}
```

What kind of monitor is this?

Monitor Example - Bounded Buffers

```
int remove_from_buf()
{
    lock_acquire(buffer.lock);

    while (buffer.numelements == 0) {
        // buffer is empty, wait
        cv_wait(buffer.notEmpty, buffer.lock);
    }

    int val = buffer.data[buffer.outpos];
    buffer.outpos = (buffer.outpos + 1) % N;
    buffer.numelements--;

    cv_signal(buffer.notFull);
    lock_release(buffer.lock);
    return val;
}
```