POSIX THREADS (Pthreads)

- Make sure you compile your code with the “-pthread” flag!
  - Eg: gcc -pthread -Wall -o example example.c
    - Use -Wall as well!

- A few important calls from the API:
  - pthread_create: create a new thread
  - pthread_exit: explicitly exit the current thread
  - pthread_join: wait for a particular thread to terminate
  - pthread_attr_init / pthread_attr_destroy: create/destroy a “thread attribute” object
  - pthread_attr_setdetachstate: set a thread’s state to “detached” or “joinable”
**PTHREAD_CREATE**

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`
  - **thread**: buffer that will contain a thread identifier returned upon successful completion of pthread_create.
  - **attr**: structure whose contents are used at thread creation to determine attributes for this thread.
  - **start_routine**: function pointer to a function of type “void *” that takes an argument of type “void *”. The execution of the new thread will begin in this function, with **arg** as an argument.
  - **arg**: the argument passed to start_routine when the new thread begins execution.
Objects of type pthread_attr_t are known as “thread attribute objects”

- Dictate the attributes that a thread will have when it is created
- Use \texttt{pthread_attr_init} and \texttt{pthread_attr_destroy} to initialize/destroy thread attribute objects
- Other methods exist to query/set specific attributes (discussed further in the LLNL tutorial…)

\begin{itemize}
  \item \texttt{int pthread_attr_init(pthread_attr_t *attr);} \\
  \item \texttt{int pthread_attr_destroy(pthread_attr_t *attr);} \\
\end{itemize}
**PTHREAD_EXIT & PTHREAD_JOIN**

- **void pthread_exit(void *value_ptr);**
  - **value_ptr**: the exit status of the calling thread.
  - The value of value_ptr will be available to any thread that does a successful pthread_join with this thread.

- **int pthread_join(pthread_t thread, void **value_ptr);**
  - **thread**: the thread identifier buffer filled by pthread_create.
  - **value_ptr**: buffer where the exit status returned by pthread_exit is stored.
JOINING & DETACHING

- Joinable threads can be “joined on” and have their exit status collected by a call to pthread_join.
- Detached threads cannot be “joined” and have their exit status discarded immediately upon exit.

To create a joinable thread:
- Use pthread_attr_init to initialize an attribute object.
- Use pthread_attr_setdetachedstate to set the state to joinable
- Use pthread_create to create the joinable thread using the previously define attributes
get & set methods for a thread’s detached state

- `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);`
  - `attr`: thread attribute object
  - `detachstate`: state of the thread to be created. Possible values:
    - PTHREAD_CREATE_DETACHED
    - PTHREAD_CREATE_JOINABLE

- `int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);`
  - `attr`: thread attribute object
  - `detachstate`: after the call, will contain the detach state of `attr`
    - PTHREAD_CREATE_DETACHED
    - PTHREAD_CREATE_JOINABLE
EXAMPLE (FROM LLNL TUTORIAL…)

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

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(t=0; t<NUM_THREADS; t++) {
        printf("Main: creating thread \%ld\n", t);
        rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
}(…)
```
/* Free attribute and wait for the other threads */
pthread_attr_destroy(&attr);
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("Main: completed join with thread %ld having a status
           of %ld\n", t,(long)status);
}

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
EXAMPLE (FROM LLNL TUTORIAL…)

```c
void *BusyWork(void *t)
{
    int i;
    long tid;
    double result=0.0;
    tid = (long)t;
    printf("Thread %ld starting...
",tid);
    for (i=0; i<1000000; i++)
    {
        result = result + i*i + i*i*i;
    }
    printf("Thread %ld done. Result = %e
",tid, result);
    pthread_exit((void*) t);
}
```
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:
  - Can cause deadlock if not careful
  - Cannot allow multiple concurrent accesses to a resource
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)`
    - Acquire the lock
  - `pthread_mutex_trylock (mutex)`
    - 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 with 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...)

Thursday, 14 February, 13
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)
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 200
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);
        }
    }
    (...)
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);

Transaction Functions:

void *deposit(void *amt){
    pthread_mutex_lock(&bal_mutex);
    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);
    int amount = (int)amt;
    int new_balance = balance - amount;
    balance = new_balance;
    pthread_mutex_unlock(&bal_mutex);
    pthread_exit((void *)0);
}
**SEMAPHORES**

- Invented by Edsger Dijkstra
- 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)
      - Block until semaphore is free, then decrement the variable
    - Signal – aka: up, post, increment, V (for verhogen)
      - Increment the variable and unblock a waiting thread (if there are any)
  - 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
    - `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.
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…
include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#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];
    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;

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
- 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
**Using Condition Variables**

*(from LLNL tutorial)*

<table>
<thead>
<tr>
<th>Main Thread</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
</table>
| - 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 | - 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 | - 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 |

**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

**Main Thread:** Join / Continue
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
  - `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