pthreads

CS 537 – Introduction to Operating Systems

What are pthreads?

- Posix 1003.1c defines a thread interface
  - pthreads
  - defines how threads should be created, managed, and destroyed
- Unix provides a pthreads library
  - API to create and manage threads
  - you don't need to worry about the implementation details
    - this is a good thing

Creating Threads

- Prototype:
  - int pthread_create(pthread_t *tid, const pthread_attr_t *tattr,
    void*(*start_routine)(void *), void *arg);
    - *tid*: an unsigned long integer that indicates a threads id
    - *tattr*: attributes of the thread – usually NULL
    - *start_routine*: the name of the function the thread starts executing
    - *arg*: the argument to be passed to the start routine – only one
  - after this function gets executed, a new thread has been created and is executing the function indicated by *start_routine*

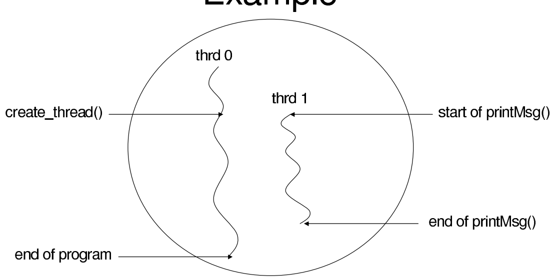
Waiting for a Thread

- Prototype:
  - int pthread_join(thread_t tid, void **status);
    - *tid*: identification of the thread to wait for
    - *status:* the exit status of the terminating thread – can be NULL
  - the thread that calls this function blocks its own execution until the thread indicated by *tid* terminates its execution
    - finishes the function it started with or
    - issues a *pthread_exit()* command – more on this in a minute

Example

```
#include <stdio.h>
#include <pthread.h>

void printMsg(char* msg) {
   printf(“%s\n”, msg);
}

int main(int argc, char** argv) {
   pthread_t thrdID;

   printf(“creating a new thread\n”);
   pthread_create(&thrdID, NULL, (void*)printMsg, argv[1]);
   printf(“created thread %d\n”. thrdID);
   pthread_join(thrdID, NULL);

   return 0;
}
```

Example



Note: thrd 0 is the function that contains *main()* – only one *main()* per program

Exiting a Thread

- pthreads exist in user space and are seen by the kernel as a single process
  - if one issues and *exit()* system call, all the threads are terminated by the OS
  - if the *main()* function exits, all of the other threads are terminated
- To have a thread exit, use *pthread_exit()*
- Prototype:
  - void pthread_exit(void *status);
    - *status:* the exit status of the thread – passed to the *status* variable in the *pthread_join()* function of a thread waiting for this one

Example Revisited

```
#include <stdio.h>
#include <pthread.h>

void printMsg(char* msg) {
   int status = 0;
   printf("%s\n", msg);
   pthread_exit(&status);
}

int main(int argc, char** argv) {
   pthread_t thrdID;
   int* status = (int*)malloc(sizeof(int));

   printf("creating a new thread\n");
   pthread_create(&thrdID, NULL, (void*)printMsg, argv[1]);
   printf("created thread %d\n". thrdID);
   pthread_join(thrdID, &status);
   printf("Thread %d exited with status %d\n", thrdID, *status);

   return 0;
}
```

Synchronizing Threads

- Three basic synchronization primitives
  1. mutex locks
  2. condition variables
  3. semaphores
- Mutexes and condition variables will handle most of the cases you need in this class
  - but feel free to use semaphores if you like

Mutex Locks

- A Mutex lock is created like a normal variable
  - *pthread_mutex_p mutex;*
- Mutexes must be initialized before being used
  - a mutex can only be initialized once
  - prototype:
    - int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *mattr);
      - *mp:* a pointer to the mutex lock to be initialized
      - *mattr:* attributes of the mutex – usually NULL

Locking a Mutex

- To insure mutual exclusion to a critical section, a thread should lock a mutex
  - when locking function is called, it does not return until the current thread owns the lock
  - if the mutex is already locked, calling thread blocks
  - if multiple threads try to gain lock at the same time, the return order is based on priority of the threads
    - higher priorities return first
    - no guarantees about ordering between same priority threads
  - prototype:
    - int pthread_mutex_lock(pthread_mutex_t *mp);
      - *mp:* mutex to lock

Unlocking a Mutex

- When a thread is finished within the critical section, it needs to release the mutex
  - calling the unlock function releases the lock
  - then, any threads waiting for the lock compete to get it
  - very important to remember to release mutex
  - prototype:
    - int pthread_mutex_unlock(pthread_mutex_t *mp);
      - *mp:* mutex to unlock

Example

```
#include <stdio.h>
#include <pthread.h>

#define MAX_SIZE  5
pthread_mutex_t bufLock;
int count;

void producer(char* buf) {
  for(;;) {
    while(count == MAX_SIZE);
    pthread_mutex_lock(bufLock);
    buf[count] = getChar();
    count++;
    pthread_mutex_unlock(bufLock);
  }
}
```

```
void consumer(char* buf) {
  for(;;) {
    while(count == 0);
    pthread_mutex_lock(bufLock);
    useChar(buf[count-1]);
    count--;
    pthread_mutex_unlock(bufLock);
  }
}

int main() {
  char buffer[MAX_SIZE];
  pthread_t p;
  count = 0;
  pthread_mutex_init(&bufLock);
  pthread_create(&p, NULL, (void*)producer, &buffer);
  consume(&buffer);
  return 0;
}
```

Condition Variables (CV)

- Notice in the previous example a *spin-lock* was used wait for a condition to be true
  - the buffer to be full or empty
  - spin-locks require CPU time to run
    - waste of cycles
- Condition variables allow a thread to block until a specific condition becomes true
  - recall that a blocked process cannot be run
    - doesn't waste CPU cycles
  - blocked thread goes to wait queue for condition
- When the condition becomes true, some other thread signals the blocked thread(s)

Condition Variables (CV)

- A CV is created like a normal variable
  - *pthread_cond_t condition;*
- CVs must be initialized before being used
  - a CV can only be initialized once
  - prototype:
    - int pthread_cond_init(pthread_cond_t *cv, const pthread_condattr_t *cattr);
      - *cv:* a pointer to the conditon variable to be initialized
      - *cattr:* attributes of the condition variable – usually NULL

Blocking on CV

- A wait call is used to block a thread on a CV
  - puts the thread on a wait queue until it gets signaled that the condition is true
    - even after signal, condition may still not be true!
  - blocked thread does not compete for CPU
  - the wait call should occur under the protection of a mutex
    - this mutex is automatically released by the wait call
    - the mutex is automatically reclaimed on return from wait call
- prototype:
  - int pthread_cond_wait(pthread_cond_t *cv,pthread_mutex_t *mutex);
    - *cv*: condition variable to block on
    - *mutex:* the mutex to release while waiting

Signaling a Condition

- A signal call is used to “wake up” a single thread waiting on a condition
  - multiple threads may be waiting and there is no guarantee as to which one wakes up first
  - thread to wake up does not actually wake until the lock indicated by the wait call becomes available
  - condition thread was waiting for may not be true when the thread actually gets to run again
    - should always do a wait call inside of a while loop
  - if no waiters on a condition, signaling has no effect
  - prototype:
    - int pthread_cond_signal(pthread_cond_t *cv);
      - *cv:* condition variable to signal on

```
#include <stdio.h>
#include <pthread.h>

#define MAX_SIZE  5
pthread_mutex_t lock;
pthread_cond_t notFull, notEmpty;
int count;

void producer(char* buf) {
  for(;;) {
    pthreads_mutex_lock(lock);
    while(count == MAX_SIZE)
          pthread_cond_wait(notFull, lock);
    buf[count] = getChar();
    count++;
    pthread_cond_signal(notEmpty);
    pthread_mutex_unlock(lock);
  }
}
```

```
void consumer(char* buf) {
  for(;;) {
    pthread_mutex_lock(lock);
    while(count == 0)
          pthread_cond_wait(notEmpty, lock);
    useChar(buf[count-1]);
    count--;
    pthread_cond_signal(notFull);
    pthread_mutex_unlock(lock);
  }
}

int main() {
  char buffer[MAX_SIZE];
  pthread_t p;
  count = 0;
  pthread_mutex_init(&bufLock);
  pthread_cond_init(&notFull);
  pthread_cond_init(&notEmpty);
  pthread_create(&p, NULL, (void*)producer, &buffer);
  consume(&buffer);
  return 0;
}
```

More on Signaling Threads

- The previous example only wakes a single thread
  - not much control over which thread this is
- Perhaps all threads waiting on a condition need to be woken up
  - can do a broadcast of a signal
  - very similar to a regular signal in every other respect
- Prototype:
  - int pthread_cond_broadcast(pthread_cond_t *cv);
    - *cv:* condition variable to signal all waiters on

Semaphores

- pthreads allows the specific creation of semaphores
  - can do increments and decrements of semaphore value
  - semaphore can be initialized to any value
  - thread blocks if semaphore value is less than or equal to zero when a decrement is attempted
  - as soon as semaphore value is greater than zero, one of the blocked threads wakes up and continues
    - no guarantees as to which thread this might be

Creating Semaphores

- Semaphores are created like other variables
  - sem_t semaphore;
- Semaphores must be initialized
  - Prototype:
    - int sem_init(sem_t *sem, int pshared, unsigned int value);
      - *sem:* the semaphore value to initialize
      - *pshared:* share semaphore across processes – usually 0
      - *value:* the initial value of the semaphore

Decrementing a Semaphore

- Prototype:
  - int sem_wait(sem_t *sem);
    - *sem:* semaphore to try and decrement
- If the semaphore value is greater than 0, the *sem_wait* call return immediately
  - otherwise it blocks the calling thread until the value becomes greater than 0

Incrementing a Semaphore

- Prototype:
  - int sem_post(sem_t *sem);
    - *sem:* the semaphore to imcrement
- Increments the value of the semaphore by 1
  - if any threads are blocked on the semaphore, they will be unblocked
- Be careful
  - doing a post to a semaphore always raises its value – even if it shouldn't!

```
#include <stdio.h>
#include <semaphore.h>

#define MAX_SIZE  5
sem_t empty, full;

void producer(char* buf) {
  int in = 0;
  for(;;) {
    sem_wait(&empty);
    buf[in] = getChar();
    in = (in + 1) % MAX_SIZE;
    sem_post(&full);
  }
}
```

```
void consumer(char* buf) {
  int out = 0;
  for(;;) {
    sem_wait(&full);
    useChar(buf[out]);
    out = (out + 1) % MAX_SIZE;
    sem_post(&empty);
  }
}

int main() {
  char buffer[MAX_SIZE];
  pthread_t p;
  sem_init(&empty, 0, MAX_SIZE);
  sem_init(&full, 0, 0);
  pthread_create(&p, NULL, (void*)producer, &buffer);
  consume(&buffer);
  return 0;
}
```

Parting Notes

- Very important to get all the ordering right
  - one simple mistake can lead to problems
    - no progress
    - mutual exclusion violation
- Comparing primitives
  - Using mutual exclusion with CV's is faster than using semaphores
  - Sometimes semaphores are intuitively simpler