Threads in Unix

What are threads?

- A thread is a single sequential flow of control within a program.
- They allow it to appear as if a program is performing multiple tasks at once through a series of rapid switches between the tasks.
- Unlike the multitasking that happens between processes (separate programs or fork()), threads in the same program share many of their resources, which allows variables to be shared between threads.

Implementing Threads

- There are several ways to implement threads, but the most convenient way to implement them in UNIX is through a package called pthread.

The pthread Library

- #include <pthread.h>
- Creating Threads: the pthread function pthread_create can be used as follows to create a new thread:

  int pthread_create (pthread_t *thread_id, const pthread_attr_t *attributes, void *(*thread_function)(void *), void *arguments);

- Exiting Threads: A thread exits when the function it was running returns or if the thread explicitly calls pthread_exit:

  int pthread_exit (void *status);

The pthread Library (cont.)

- One thread can wait for another to terminate by using the join function:

```
int pthread_join (pthread_t thread, void *status_ptr);
```

Example One

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

int totalcount = 0;

/*Function called by threads*/
void *printstring(void *string){
   char* print = (char*)string;
   int i, j;
   for (i=0;i<10;i++){
      for (j=0;j<10000000;j++);
      printf("%s\n",print);
      totalcount++;
   }
   return NULL;
}
```

```
int main(){
   pthread_t thread1,thread2,thread3;
   char* string1 = "one";
   char* string2 = "two";
   char* string3 = "three";

   /*Create Threads*/
   pthread_create(&thread1,NULL,printstring,string1);
   pthread_create(&thread2,NULL,printstring,string2);
   pthread_create(&thread3,NULL,printstring,string3);

    /*Catch up to all threads*/
    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    pthread_join(thread3,NULL);
    printf("totalcount: %d\n",totalcount);
    return 0;
}
```

Problems with Threads

- Threads can present problems when multiple threads are using the same memory location (global variables)
- pthread solves this problem by implementing mutexes.
- Mutex stands for mutual exclusion.
- A simple example can show how mutexes work, and why they are needed.

The Reason for Mutex

THREAD 1	THREAD 2
a = data;	b = data;
a++;	b--;
data = a;	data = b;

Now, this is perfectly fine as long as one thread runs, and then the other. However, threads execute in an arbitrary order, interrupting one to run the other. Consider the following orders of execution.

THREAD 1	THREAD 2
a = data;	
	b = data;
a++;	
	b--;
data = a;	
	data = b;

[data = data - 1]

THREAD 1	THREAD 2
	b = data;
a = data;	
	b--;
a++;	
	data = b;
data = a;	

[data = data + 1]

Implementing Mutexes

- Initializing Mutexes: the pthread function pthread_mutex_init can be used to initialize a mutex as follows:

```
int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutex_attr *attr);
```

- Locking and Unlocking Mutexes: Mutexes can be locked and unlocked as follows using the pthread_mutex_lock and pthread_mutex_unlock functions respectively:

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

Condition Variables

- Condition Variables are similar to mutexes, except that they allow a thread to give up control of the process while it waits for some condition to become true.

Implementing Condition Variables

- Initializing Condition Variables: the pthread function pthread_cond_init can be used to initialize a mutex as follows:

```
int pthread_cond_init(pthread_cond_t* cond, const pthread_cond_att *attr);
```

- Waiting and Signaling: condition variables wait for and signal using the following function calls:

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

Example Two

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

#define BSIZE 4
#define NUMITEMS 25

//structure used pass parameters to threads
typedef struct {
   char buf[BSIZE];
   int occupied;
   int nextin, nextout;
   pthread_mutex_t mutex;
   pthread_cond_t more;
   pthread_cond_t less;
} buffer_t;

buffer_t buffer;

void * producer(void *);
void * consumer(void *);

//number of threads to start
#define NUM_THREADS 2

//array of thread ids
pthread_t tid[NUM_THREADS];
```

```
main( int argc, char *argv[] ){
   int i;

   pthread_cond_init(&(buffer.more),  NULL);
   pthread_cond_init(&(buffer.less), NULL);

   //create producer and consumer
   pthread_create(&tid[1], NULL, consumer, NULL);
   pthread_create(&tid[0], NULL, producer, NULL);

   //join the producer and consumer
   for ( i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
      printf("\n%d threads have terminated\n", i);
}
```

Example Two (cont.)

```
void * producer(void * parm){
  char item[NUMITEMS]="PRODUCER
     CONSUMER EXAMPLE";
  int i;
  printf("producer started.\n");
  for(i=0;i<NUMITEMS;i++){
    /* Quit if at end of string. */
    if (item[i] == '\0') break;

    pthread_mutex_lock(&(buffer.mutex));

    if (buffer.occupied >= BSIZE)
      printf("producer waiting.\n");
    while (buffer.occupied >= BSIZE)
      pthread_cond_wait(&(buffer.less),
        &(buffer.mutex) );
    printf("producer executing.\n");

    buffer.buf[buffer.nextin++] = item[i];
    buffer.nextin %= BSIZE;
    buffer.occupied++;

    pthread_cond_signal(&(buffer.more));
    pthread_mutex_unlock(&(buffer.mutex));
  }
  printf("producer exiting.\n");
  pthread_exit(0);
}
```

```
void * consumer(void * parm) {
  char item;
  int i;

  printf("consumer started.\n");

  for(i=0;i<NUMITEMS;i++){
    pthread_mutex_lock(&(buffer.mutex) );
    if (buffer.occupied <= 0)
      printf("consumer  waiting.\n");
    while(buffer.occupied <= 0)
    pthread_cond_wait(&(buffer.more),
      &(buffer.mutex) );
    printf("consumer executing.\n");

    item = buffer.buf[buffer.nextout++];
    printf("%c\n",item);
    buffer.nextout %= BSIZE;
    buffer.occupied--;
    pthread_cond_signal(&(buffer.less));
    pthread_mutex_unlock(&(buffer.mutex));
  }
  printf("consumer exiting.\n");
  pthread_exit(0);
}
```

Threads Problems (Deadlocks)

```
pthread_mutex mut1, mut2;
pthread_mutex_init(&mut1,NULL);
pthread_mutex_init(&mut2,NULL);
```

THREAD 1	THREAD 2
pthread_mutex_lock(&mut1);	
	pthread_mutex_lock(&mut2);
pthread_mutex_lock(&mut2);	
	pthread_mutex_lock(&mut1);

Both of these threads are blocked. This situation is known as a deadlock. Neither thread can progress any further, because they are both waiting for a mutex that the other one has locked.

Threads Problems (Busy Waiting or Livelock)

```
int done = 0;

void function1(){
    while(done == 0);
}
```

```
void function2(){
    int i;
    for(i=0;i<1000000000;i++);
    done = 1;
}
```

In this case the computer will switch back and fourth between the two threads as usual, but the switches are simply a waste of CPU time until thread two finishes its loop.

Starvation

Starvation is when one or more threads can not gain access to the resources they need, and therefore never get to run.