Lecture 8 - Threads (Chapters 11 & 12)

- Overview

Threads are often considered "lightweight" processes.

A typical UNIX process contains a single thread of execution.

A thread is essentially a program counter, a stack, and a set of registers.

Threads are attractive from a programming standpoint since they are very small (i.e. consume few CPU resources to create and manage), and they require very little memory.

Threads provide a concurrent programming model without the expense of creating new processes.

Threads also allow us to program asynchronous programs easier (i.e. we can create a thread for each async event, and deal with each event as synchronous code).

- POSIX threads

There are many implementations of threads across the different versions of UNIX, but one of the more popular implementations is known as POSIX threads or pthreads.

Nearly all modern versions of UNIX support pthreads in some form…

- Thread identification

Just like processes have unique PIDs, each thread has a unique thread ID.

Although PIDs are unique across the whole system, thread IDs are unique only in the context of the process to which they belong.

Thread IDs are represented by type pthread_t (could be a structure).

```
int             pthread_equal(pthead_t tid1, pthead_t tid2);
pthread_t       pthread_self(void);
```

- Thread creation

  int pthread_create(thread, attr, start_routine, arg)

  This routine creates a new thread and makes it executable.

  The thread ID of the new thread is returned via the "thread" argument (i.e. somewhat analogous to a PID for processes).

  "attr" is used to set thread attributes; you can either specify a thread attributes object or NULL to use the defaults.

  "start_routine" is the C function that the thread will execute once it is created.

  "arg" is the single argument that may be passed to "start_routine" (must be passed by reference as a cast to pointer to void).

  Note: The max number of threads per process is implementation dependent and finite.

  Example:

```
void *thr_fn(void *arg)
{
        printf("%d\n", pthread_self());
        return ((void *) 0);
}

int main(void)
{
        pthread_t       ntid;
        int             err;

        if (pthread_create(&ntid, NULL, thr_fn, NULL) != 0)
                perror("pthread_create");

        sleep(1);
        exit(0);
}
```

- Thread Terminatation

  There are several ways in which a pthread may terminate:

  1. The thread returns from the "start_routine"
  2. The thread makes a call to pthread_exit
  3. The thread is canceled by another thread via pthread_cancel
  4. The entire process is terminated due to a call to exec or exit.

  ```
  void    pthread_exit (void *status)
  ```

  "status" will be available to any thread that joins this thread.

  ```
  int     pthread_join(pthead_t thread, void **status);
  ```

  If main() finishes before the other threads, and exits with pthread_exit, the other threads will continue to execute. Otherwise they will automatically be terminated when main() finishes.

  Note: pthread_exit does not close any open files.

- Example: thread creation and termination

```
#define NUM_THREADS     5

void *print_hello(void *threadid) {
    printf("\n%d: Hello World\n", threadid);
    pthread_exit(NULL);
}

int main() {
    pthread_t  threads[NUM_THREADS];
    int        t, rc;

    for (t = 0; t < NUM_THREADS; t++)      {
        printf("Creating thread %d\n", t);

        if (pthread_create(&threads[t], NULL, print_hello, (void *) t)) {
            printf("ERROR: return code is %d\n", rc);
            return(1);
        }
    }

    pthread_exit(NULL);
}
```

- Passing arguments to threads

  The "pthread_create" function permits us to pass one arg to the thread start routine.

  To pass multiple arguments, we must create a structure containing all of the arguments, and pass a pointer to that structure.

  All arguments must be passed by reference and cast to (void *).

  Example:

```
int     *taskids[NUM_THREADS];

for (t = 0; t < NUM_THREADS; t++)
{
        taskids[t] = (int *) malloc(sizeof(int));
        *taskids[t] = t;

        printf("Createing thread %d\n", t);

        rc = pthread_create(&threads[t], NULL, print_hello, (void *) taskids[t]);

        …
}
```

  Example:

```
struct tdata {
        int     thread_id;
        int     sum;
        char    *message;
};

struct tdata tdata_array[NUM_THREADS];

void    *print_hello(void       *arg)
{
        struct tdata    *data;

        ...
        data = (struct tdata *) arg;
        taskid = data->thread_id;
        sum = data->sum;
        hello_msg = data->message;
        ...
}
```

```
        int main()
        {

                tdata_array[t].thread_id = t;
                tdata_array[t].sum = sum;
                tdata_array[t].message = messages[t];

                rc = pthread_create(&threads[t], NULL, print_hello,
                        (void *) &tdata_array[t]);

                ...
        }
```

- Detaching / joining threads

  When a thread is created, one of its attributes defines if it is joinable or detached.

  pthread_attr_init(attr)
  pthread_attr_setdetachstate(attr, detachstate)
  pthread_attr_getdetachstate(attr, detachstate)
  pthread_attr_destroy(attr)
  pthread_detach(threadid, status)

  To explicitly create a thread of joinable or detached, we use the attr argument to pthread_create. The process would be like this:

  1. Declare a pthread attribute variable of pthread_attr_t data type.
  2. Initialize the attribute variable with pthread_attr_init()
  3. Set the attribute detached status with pthread_attr_setdetachstate()
  4. When done, free up resources with pthread_attr_destroy()

  Note: pthread_detach() can be used to explicitly detach a thread even if it was created as joinable.

- Join example

```
#define NUM_THREADS     3

void    *busy_work(void *null)
{
    int     i;
    double  result = 0.0;

    for (i = 0; i < 1000000; i++)
        result = result + (double) random();
    printf("result = %d\n", result);
    pthread_exit((void *) 0);
}


int main()
{
    pthread_t       thread[NUM_THREADS];
    pthread_attr_t  attr;
    int             rc, t, status;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for (t = 0; t < NUM_THREADS; t++)
    {
        printf("Creating Thread %d\n", t);
        rc = pthread_create(&thread[t], &attr, busy_work, NULL);

        if (rc)
        {
            printf("ERROR: return code is %d\n", rc);
            exit(1);
        }
    }

    pthread_attr_destroy(&attr);

    for (t = 0; t < NUM_THREADS; t++)
    {
        rc = pthread_join(thread[t], (void **) &status);

        if (rc)
        {
            printf("ERROR: return code is %d\n", rc);
            exit(1);
        }

        printf("Completed join with thread = %d, status = %d\n",
            t, status);
    }

    pthread_exit(NULL);
}
```

- Mutex variables

  Mutex is an abbreviation for "mutual exclusion", and is one of the primary mechanisms for implementing thread synchronization and to protect shared data resources.

  Mutex variables acts as a "lock" protecting access to a shared data resource (i.e. a binary semaphore).

  Typical sequence:

  1. Create and initialize a mutex variable
  2. Several threads attempt to lock the mutex.
  3. One of those threads succeeds, and that thread owns the mutex
  4. The owner performs some set of actions
  5. The owner unlocks the mutex
  6. Another thread acquires the mutex and repeats the process
  7. Finally, the mutex is destroyed

- Creating / destroying mutexes

  pthread_mutex_init(mutex, attr)
  pthread_mutex_destroy(mutex)
  pthread_mutexattr_init(attr)
  pthread_mutexattr_destroy(attr)

  Mutex variables must be declared with type pthread_mutex_t, and must be initialized before use. They can be initialized by:

  1. When declared, pthread_mutuex_t m = PTHREAD_MUTEX_INITIALIZER;
  2. By calling pthread_mutex_init().

  Note: Mutex variables are initially unlocked.

  The "attr" object is used to establish properties for the mutex object and it must be of type pthread_mutexattr_t.

- Locking / unlocking mutexes

  pthread_mutex_lock(mutex)
  pthread_mutex_trylock(mutex)
  pthread_mutex_unlock(mutex)

  pthread_mutex_lock() tries to acquire lock on "mutex" and blocks until it is available

  pthread_mutex_trylock(mutex) tries to acquire lock on "mutex", but will return "busy" error code if the mutex is already locked.

- Example: Mutexes

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

#define NUM_THREADS     4
#define VECLEN          100

typedef struct {
        double  *a;
        double  *b;
        double  sum;
        int     veclen;
} DOTDATA;

DOTDATA         dotstr;
pthread_t       tCall[NUM_THREADS];
pthread_mutex_t mSum;

void    *dotprod(void *arg)
{
        int     i, start, end, offset, len;
        double  mysum, *x, *y;

        offset = (int) arg;

        len = dotstr.veclen;
        start = offset * len;
        end = start + len;
        x = dotstr.a;
        y = dotstr.b;

        mysum = 0;

        for (i = start; i < end; i++)
        {
                mysum += (x[i] * y[i]);
        }

        pthread_mutex_lock(&mSum);

        dotstr.sum += mysum;

        pthread_mutex_unlock(&mSum);
        pthread_exit((void *) 0);
}
```

```
int main(int argc, char **argv)
{
        int             i, status;
        double          *a, *b;
        pthread_attr_t  attr;

        a = (double *) malloc(NUM_THREADS * VECLEN * sizeof(double));
        b = (double *) malloc(NUM_THREADS * VECLEN * sizeof(double));

        for (i = 0; i < VECLEN * NUM_THREADS; i++)
        {
                a[i] = 1;
                b[i] = a[i];
        }

        dotstr.veclen = VECLEN;
        dotstr.a = a;
        dotstr.b = b;
        dotstr.sum = 0;

        pthread_mutex_init(&mSum, NULL);

        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);


        for (i = 0; i < NUM_THREADS; i++)
        {
                pthread_create(&tCall[i], &attr, dotprod, (void *) i);
        }

        pthread_attr_destroy(&attr);

        for (i = 0; i < NUM_THREADS; i++)
        {
                pthread_join(tCall[i], (void **) &status);
        }

        printf("Sum = %f\n", dotstr.sum);
        free(a);
        free(b);

        pthread_mutex_destroy(&mSum);
        pthread_exit(NULL);
}
```

- Condition Variables

  Condition variables provide another way for threads to synchronize. While mutexes control access to data, condition variables allow threads to synchronize based upon the actual value of the data.

  This mechanism allows threads to avoid a "busy" loop that locks a resource, checks its value, and unlocks it until some condition occurs.

  Condition variables are always used in conjunction with mutex locks.

- Creating / Destroying condition variables

  pthread_cond_init(condition, attr)
  pthread_cond_destroy(condition)
  pthread_condattr_init(attr)
  pthread_condattr_destroy(attr)

  Condition variables must be declared with type pthread_cond_t and must be initialized before use either when declaring it or by calling pthread_cond_init().

- Waiting / signaling on condition variables

  pthread_cond_wait(condition, mutex)
  pthread_cond_signal(condition)
  pthread_cond_broadcast(condition)

  pthread_cond_wait() blocks the calling thread until the specified condition is signaled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits. Note: you should also unlock the mutex after signal has been received.

  pthread_cond_signal() is used to signal (wake up) another thread which is waiting on the condition variable.

  pthread_cond_broadcast() should be used instead of pthread_cond_signal() if more than one thread is blocked on this condition.

  Note: Proper locking / unlocking of the associated mutex variable is essential to the functionality of these routines. For example, failing to lock mutex before calling pthread_cond_wait() may cause it NOT to block or failing the unlock the mutex after calling pthread_cond_signal() may not allow a matching pthread_cond_wait() to complete.

- Example: condition variables

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

#define NUM_THREADS     3
#define TCOUNT          10
#define COUNT_LIMIT     12

int             count = 0;
int             tids[3] = { 0, 1, 2};
pthread_mutex_t cmutex;
pthread_cond_t  ccond;

void    *inc_count(void *idp)
{
        int     i, j;
        double  result = 0.0;
        int     *myid = idp;

        for (i = 0; i < TCOUNT; i++)
        {
                pthread_mutex_lock(&cmutex);

                count++;


                if (count == COUNT_LIMIT)
                {
                        pthread_cond_signal(&ccond);
                        printf("inc_count: thread %d, count %d, threshold\n",
                                *myid, count);
                }

                printf("inc_count: thread %d, count %d, unlock\n",
                        *myid, count);

                pthread_mutex_unlock(&cmutex);

                for (j = 0; j < 1000; j++)
                        result = result + (double) random();
        }

        pthread_exit(NULL);
}
```

```
void *watch_count(void *idp)
{
	int     *myid = idp;

	printf("watch_count: starting thread %d\n", *myid);

	pthread_mutex_lock(&cmutex);

	while (count < COUNT_LIMIT)
	{
		pthread_cond_wait(&ccond, &cmutex);

		printf("watch_count: thead %d, condition signal recv\n", *myid);
	}

	pthread_mutex_unlock(&cmutex);
	pthread_exit(NULL);
}

int main(int argc, char **argv)
{
	int             i, rc;
	pthread_t       threads[3];
	pthread_attr_t  attr;

	pthread_mutex_init(&cmutex, NULL);
	pthread_cond_init(&ccond, NULL);

	pthread_attr_init(&attr);
	pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

	pthread_create(&threads[0], &attr, inc_count, (void *) &tids[0]);
	pthread_create(&threads[1], &attr, inc_count, (void *) &tids[1]);
	pthread_create(&threads[2], &attr, watch_count, (void *) &tids[2]);

	for (i = 0; i < NUM_THREADS; i++)
		pthread_join(threads[i], NULL);

	printf("main: waited on %d threads.\n", NUM_THREADS);

	pthread_attr_destroy(&attr);
	pthread_mutex_destroy(&cmutex);
	pthread_cond_destroy(&ccond);
	pthread_exit(NULL);
}
```