CS240: Programming in C

Lecture 18: PThreads

Thread Creation

- Initially, your `main()` program comprises a single, default thread.
- `pthread_create` creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.
- `pthread_create` arguments:
  - thread: An opaque, unique identifier for the new thread returned by the subroutine.
  - attr: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
  - start_routine: the C routine that the thread will execute once it is created.
  - arg: A single argument that may be passed to *start_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

- The maximum number of threads that may be created by a process is implementation dependent.

Example

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Passing Arguments

```
long *taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
   taskids[t] = (long *) malloc(sizeof(long));
   *taskids[t] = t;
   printf("Creating thread %ld\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello,
                       (void *) taskids[t]);
   ...
}
```

Multiple Arguments

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

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}

int main (int argc, char *argv[])
{
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
   ...
}
```

Example

What's wrong with the following?

```
int rc;
long t;

for(t=0; t<NUM_THREADS; t++)
{
   printf("Creating thread %ld\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
   ...
}
```

Identity

```
pthread_self()

pthread_equal(thread1,thread2)


pthread_once(once_control, init_routine)
```

Mutexes

- Mutual Exclusion
  - At most one thread can “acquire” a mutex at any given time.
    - Once the acquiring thread “releases” the mutex, another thread waiting for it can acquire it
    - Threads waiting for a mutex are blocked from performing any other work
  - What are some of the logical errors that can occur when mutexes are used incorrectly
    - Not used when they should be
    - Used when they shouldn’t be

Example: parallel dot product

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

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


#define NUMTHRDS 4
#define VECLEN 100
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;
```

Example

```
void *dotprod(void *arg)
{

   /* Define and use local variables for convenience */

   int i, start, end, len ;
   long offset;
   double mysum, *x, *y;
   offset = (long)arg;

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

   /*
   Perform the dot product and assign result
   to the appropriate variable in the structure.
   */

   mysum = 0;
   for (i=start; i<end ; i++)
    {
      mysum += (x[i] * y[i]);
    }

   /*
     Lock a mutex prior to updating the value
     in the shared
     structure, and unlock it upon updating.
     */
     pthread_mutex_lock (&mutexsum);
     dotstr.sum += mysum;
     pthread_mutex_unlock (&mutexsum);

     pthread_exit((void*) 0);
}
```

Example (cont)

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

   /* Assign storage and initialize values */
   a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

   for (i=0; i<VECLEN*NUMTHRDS; i++)
    {
     a[i]=1.0;
     b[i]=a[i];
    }

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

Example (cont)

```
    pthread_mutex_init(&mutexsum, NULL);

    /* Create threads to perform the dotproduct  */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

     for(i=0; i<NUMTHRDS; i++)
         {
     /*
     Each thread works on a different set of data.
     The offset is specified by 'i'. The size of
     the data for each thread is indicated by VECLEN.
     */
     pthread_create(&callThd[i], &attr, dotprod, (void *)i);
     }

     pthread_attr_destroy(&attr);

         /* Wait on the other threads */
     for(i=0; i<NUMTHRDS; i++)
         {
       pthread_join(callThd[i], &status);
     }

    /* After joining, print out the results and cleanup */
    printf ("Sum =  %f \n", dotstr.sum);
    free (a);
    free (b);
    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
}
```