Introduction to Multithreading

DEPARTMENT OF COMPUTER ENGINEERING

In this lab we will look at multi-threading. For this task we will use POSIX `pthread` library.

Processes and Threads
Threads are separate, independent sequences of commands that could belong to the same process. Threads could execute in parallel. Threads inside the same process share the resources that belong to the process.

Since the memory space is one of those resources they share, there is the possibility of race conditions when two threads access the same variables.

Process is the 'unit of protection' provided by the operating system. The resources are set aside for a process (disk access, processor time, memory space etc.). Other processes cannot tamper with these resources without the owning processes' permission.

pthread Library
`pthread` library is the POSIX standard for multi-threading. In this lab session we will be exploring and exploiting its power.

Thread Creation and Destruction
In a previous lab, we created a process and then replaced its image with a new process. i.e. we created a new process and ran a program in it. In threads, we create a new thread and run a function in it.

To create a thread use `pthread_create()` function and `pthread_exit()` function to destroy a thread. Here is an example:

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
void *thread_function(void *arg)
{
    int a;
    for(a = 0;a < 10; a++)
    {
        printf("Thread says hi!\n");
        sleep(2);
    }
    return NULL;
}

int main(void)
{
    pthread_t mythread;
    if ( pthread_create( &mythread, NULL, thread_function, NULL) )
    {
        printf("error creating thread.");
        abort();
    }
    if ( pthread_join ( mythread, NULL ) )
    {
        printf("error joining thread.");
        abort();
    }
    printf("Main thread says hi!\n");
    exit(0);
}
```

Exercise1

1. Save the above program as `thread.c` and compile it as:

```
gcc -pthread -o thread thread.c -Wall
```

   a) Why do you need '`-pthread`' flag when you compile?

`pthread_t` type

`pthread_t` is of an unsigned integer type. In 64bit Linux systems, it is an unsigned long integer. The value in *`pthread_t`* represents the thread (something similar to a `pid` for a process).

`pthread_create()` call

This function creates and runs the new thread. This takes few arguments and returns an integer. The first argument taken is the pointer for the *`pthread_t`* type variable. The `tid` value is stored by this function in this pointer.

The next argument is a struct of type *`pthread_attr_t`* which specifies the attributes that we have to use when creating a thread. We are not going to discuss these attributes. Therefore we set this value to `NULL`, allowing the attributes to be in their default values.

The next argument is the *function pointer* to the function that the thread should execute. This function could take one argument of type `void*` and returns a value of type `void*` (`void*` is the generic type in C that represents any type of pointer)

The last argument is the *argument* that should be passed to the function that the thread should execute. This argument should be of type `void*`. We have set this value to `NULL` since we're not passing any arguments to our function.

`pthread_create()` returns 0 in success and returns an non zero integer representing the error code otherwise.

Exercise2

1. Consider the following piece of code from multiprocessing lab session:

```
int i;
for (i = 0; i < 3; i++)
     fork();
```

   a) how many new processes did it create?

Now consider this piece of code:

```
int i;
for(i = 0; i < 3; i++)
     pthread_create(&myThread,NULL,function,NULL);
```

b) how many threads will it create? Compare this with a) above. Why is there a difference?

Passing parameters to the function that the thread executes and using those within the thread function is a bit tricky. The following example passes a `count` variable to the *`thread_function`*, and update its value within the function call (Note: the Line Numbers are provided to assist you in the next exercise).

```
 1      #include <pthread.h>
 2      #include <stdlib.h>
 3      #include <unistd.h>
 4      #include <stdio.h>
 5
 6      void *thread_function(void *arg)
 7      {
 8              int a;
 9              for(a = 1; a <= 3; a++)
10              {
11                      printf("Thread %d:%d says hi!\n", *(int *)arg, a);
12      //              sleep(1);
13                      }
14
15                      (*(int *)arg)++;
16                      return NULL;
17              }
18
19              int main(void)
20              {
21                      pthread_t mythread;
22                      int i, count = 1;
23
24                      for (i = 0; i < 5; i++){
25                              if (pthread_create( &mythread, NULL, thread_function, &count))
26                              {
27                                      printf("error creating thread.");
28                                      abort();
29                              }
30                              if ( pthread_join ( mythread, NULL ) )
31                              {
32                                      printf("error joining thread.");
33                                      abort();
34                              }
35              //              sleep(1);
36                      }
37              //      sleep(1);
38                      printf("Main thread says hi!\n");
39                      exit(0);
40              }
```

The *`pthread_join`* call

If one thread calls join on another, the first thread waits until the other threads finishes execution to continue. In a case where the second one has already finished, the first thread continues as if nothing happened.

`pthread_join()` takes two arguments. The first one is the `tid` value stored in the *`pthread_t`* variable that represents the thread we want to join with and the second one is a variable that is used to store the exit status of the thread we're joining.

Here in our example, the join call is used to stop the main thread from being exited before the other threads exits. If the main thread exits before the others, then the rest is forced to terminate with the main thread, leaving the work unfinished.

Exercise 3

1. Compile and run the above program as is.
   a) Can you explain the results?
   b) What is the objective of the code in line 15?
   c) Deconstruct this statement, and explain how the objective you mentioned is achieved.
2. Comment out the code segment for the join call (lines 30 – 34). Can you explain the result?
   a) Now, comment out the *`sleep()`* statements at lines 12, 35 and 37 (only one at a time) and explain each result.
   b) Now, comment out pairs of *`sleep()`* statements as follows, and explain the results.
      i. lines 35 & 37
      ii. lines 12 & 35 (run the program multiple times, and explain changes in the results, if any)
      iii. lines 12 & 37 (increase the sleep time of line 37 gradually to 5)
   c) Now, uncomment all *`sleep()`* statements.
      i. Can you explain the results?
      ii. Does the output change if you revert the sleep value in line 37 back to 1? If so, why?
3. Consider the following statement: "you use *`sleep()`* statements instead of join calls to get the desired output of a multithreaded program."
   a) Write a short critique of this statement expressing your views and preferences, if any.

Exercise 4

1. Use the following skeleton code to implement a multi-threaded server.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>

void* handle_client(void*);

int main()
{
     int listenfd;
     int* connfd;
     struct sockaddr_in servaddr,cliaddr;
     socklen_t clilen;
     listenfd = socket(AF_INET,SOCK_STREAM,0);
     servaddr.sin_family = AF_INET;
     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
     servaddr.sin_port = htons(32000);
     bind(listenfd,(struct sockaddr *)&servaddr,sizeof(servaddr));
     listen(listenfd,5);
     clilen = sizeof(cliaddr);
     while(1)
     {
          connfd = malloc(sizeof(int));
          *connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &clilen);

          /* now create a new thread, pass it the socket and run the thread.  */
     }
}

void* handle_client(void* connfd)
{
       /*read a string sent by the client,
         print it and then send the string
         "Hello from the server" to the client*/
       free(connfd);
       return NULL;
}
```

2. Why do you need to declare `connfd` as a variable in the heap? What problem might occur if it was declared as a local variable?