**Computer Communications and Networks**



**Lab 06**

**Multi-threaded TCP socket programming (concurrent)**

Lab Manual 05

|  |
| --- |
| Objectives  * Introduction to Thread? * Pthreads API? * Why Pthreads? * Pthreads vs fork() * Thread Creation and Termination * Malloc/Free (dynamic memory allocation or de-allocation) * File Handling * Multi-threaded TCP based concurrent server |

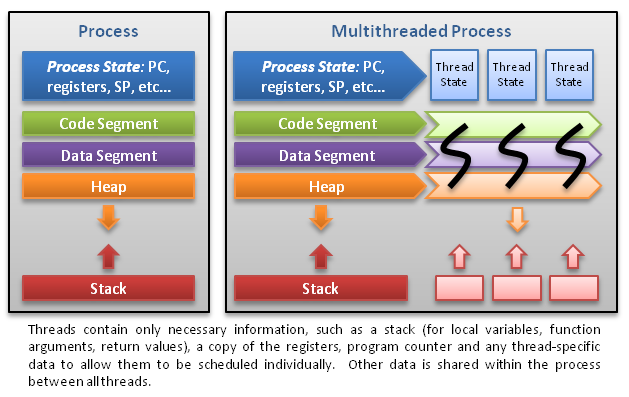
# Reference Material

**What is a Thread?**

* **Analogy**

Think of sewing needle as a processor and threads in a program as thread fiber. If you have two needles but only one thread, it would take longer to finish the job (as one needle is idle). Whereas if you split the thread into two and use both needles at same time, you will end your task quickly. Taking this analogy a little further, if one needle had to sew on a button (blocking I/O), the other needle could continue doing other useful work even if the other needle took 1 hour to sew on a single button. If you only use one needle, you would be ~1 hour behind!

A computer program becomes a **process** when it is loaded from some store into the computer's memory and begins execution. A process can be executed by a processor or a set of processors. A process description in memory contains vital information such as the program counter which keeps track of the current position in the program (i.e. which instruction is currently being executed), registers, variable stores, file handles, signals, and so forth. A **thread** is a sequence of such instructions within a program that can be executed independently of other code.



* Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.
* A thread does not maintain a list of created threads, nor does it know the thread that created it.
* All threads within a process share the same address space.

**What is pthread?**

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.

For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).

Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.

Most hardware vendors now offer Pthreads in addition to their proprietary API's.

The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.

Pthreads are defined as a set of C language programming types and procedure calls, implemented with a **pthread.h** header/include file and a thread library - though this library may be part of another library, such as **libc**, in some implementations.

**Why pthreads?**

When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

The primary motivation for considering the use of Pthreads in a high performance computing environment is to achieve optimum performance. In particular, if an application is using MPI for on-node communications, there is a potential that performance could be improved by using Pthreads instead.

MPI libraries usually implement on-node task communication via shared memory, which involves at least one memory copy operation (process to process).

For Pthreads there is no intermediate memory copy required because threads share the same address space within a single process. There is no data transfer, per se. It can be as efficient as simply passing a pointer.

In the worst case scenario, Pthread communications become more of a cache-to-CPU or memory-to-CPU bandwidth issue. These speeds are much higher than MPI shared memory communications.

**pthread vs fork()**

**pthread**

* Can specify more attributes for new threads during create
* Can specify function for new thread to start with
* Easy to grasp address space sharing

**Fork()**

* Zero parameters
* Set everything yourself after process creation
* New process begins with a virtual copy of parent at same location
* Copy on write semantics require shared memory setup

**The pthreads API**

Routines which comprise the Pthreads API can be informally grouped into four major groups:

1. **Thread management:** Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.)
2. **Mutex:** Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutex. These are supplemented by mutex attribute functions that set or modify attributes associated with mutex.
3. **Condition variables:** Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.
4. **Synchronization:** Routines that manage read/write locks and barriers.

**Creating Threads:**

* Initially, your **main()** program comprises a single, default thread. All other threads must be explicitly created by the programmer.
* **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.

#include <pthread.h>

int pthread\_create(pthread\_t \*restrict thread,

const pthread\_attr\_t \*restrict attr,

void \*(\*start\_routine)(void\*), void \*restrict arg);

The pthread\_create() function shall create a new thread, with attributes specified by attr, within a process. If attr is NULL, the default attributes shall be used. If the attributes specified by attr are modified later, the thread's attributes shall not be affected. Upon successful completion, pthread\_create() shall store the ID of the created thread in the location referenced by thread.

The thread is created executing start\_routine with arg as its sole argument. If the start\_routine returns, the effect shall be as if there was an implicit call to pthread\_exit() using the return value of start\_routine as the exit status. Note that the thread in which main() was originally invoked differs from this. When it returns from main(), the effect shall be as if there was an implicit call to exit() using the return value of main() as the exit status.

**Return Value:**

If successful, the pthread\_create() function shall return zero; otherwise, an error number shall be returned to indicate the error.

**Thread Termination and Joining**

#define \_OPEN\_THREADS

#include <pthread.h>

void pthread\_exit(void \*status);

Ends the calling thread and makes status available to any thread that calls pthread\_join() with the ending thread's thread ID.

**Dynamic Memory Allocation or De-allocation**

**malloc()**

The C library function **void \*malloc(size\_t size)** allocates the requested memory and returns a pointer to it.

* **Declaration**

Following is the declaration for malloc() function.

void \*malloc(size\_t size)

* **Parameters**
* **size** − This is the size of the memory block, in bytes.
* **Return Value**

This function returns a pointer to the allocated memory, or NULL if the request fails.

**free()**

The C library function **void free(void \*ptr)** deallocates the memory previously allocated by a call to calloc, malloc, or realloc.

* **Declaration**

Following is the declaration for free() function.

void free(void \*ptr)

* **Parameters**
* **ptr** − This is the pointer to a memory block previously allocated with malloc, calloc or realloc to be deallocated. If a null pointer is passed as argument, no action occurs.
* **Return Value**

This function does not return any value.

**File Handling**

Different operations that can be performed on a file are:

* Creation of a new file (fopen with attributes as “a” or “a+” or “w” or “w++”)
* Opening an existing file (fopen)
* Reading from file (fscanf or fgetc)
* Writing to a file (fprintf or fputs)
* Closing a file (fclose)

**Creating new file**

FILE \*filePointer;

//So, the file can be opened as

filePointer = fopen(“fileName.txt”, “w”)

**feek();**

The C library function int fseek(FILE \*stream, long int offset, int whence) sets the file position of the **stream** to the given **offset**.

* **Declaration**

Following is the declaration for fseek() function.

int fseek(FILE \*stream, long int offset, int whence)

**fgets()**

The C library function char \*fgets(char \*str, int n, FILE \*stream) reads a line from the specified stream and stores it into the string pointed to by **str**. It stops when either **(n-1)** characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

* **Declaration**

Following is the declaration for fgets() function.

char \*fgets(char \*str, int n, FILE \*stream)

**fopen()**

The C library function FILE \*fopen(const char \*filename, const char \*mode) opens the **filename** pointed to, by filename using the given **mode**.

* **Declaration**

Following is the declaration for fopen() function.

FILE \*fopen(const char \*filename, const char \*mode)

**Fclose()**

The C library function **int fclose(FILE \*stream)** closes the stream. All buffers are flushed.

* **Declaration**

Following is the declaration for fclose() function.

int fclose(FILE \*stream)

# **Lab Tasks**

**Task1 Write a client-server program with multithreaded approach, previously we have tested this program on the same machine, but this time, both client and server will be running on different machines[5 marks] (group task, max: 2 min: 2)**

**Submission checking criteria**

* Screenshot of c program **[2.5 marks]**
* Code **[2.5 marks]**

**Task2 Write a client-server program with multithreaded approach, make two seperate threads on client side to handle send and recv seperately. One thread will take care of the bytes being sent to server and other one will take care of the bytes received from the server[10 marks]**

**Submission checking criteria**

* Screenshot of c program **[5 marks]**
* Code **[10 marks]**

**Task3. Write a client-server program with multithreaded approach, in this task server will be maintaining a list of client connections accepted by the server, as server accepts the connection, server will save some information of each client in that list, list will contain each client’s port, ip and newsockfd of the client. And after saving the information server will broadcast other previously connected clients that new client has been arrived [30 Marks]**

**Sample Output:**

**Server Side:**

**Client with port 1234 connected:**

**Client with port 1235 connected:**

**Client with port 1236 connected:**

**1st Client side**

**Connected with server port 8080**

**New Connection with port 1235 has been arrived**

**New Connection with port 1236 has been arrived**

**Your Name- Registration Number // 1st client sending the data to server**

**2nd Client side**

**Connected with server port 8080**

**New Connection with port 1236 has been arrived**

**From port 1234 Your Name- Registration Number // receiving the data from client1**

**3rd Client side**

**Connected with server port 8080**

**From Port 1234 Your Name- Registration Number //receiving the data from client1**