Assignment 3

COP-701

Your own Thread Library(OTL)

Name : Sourav Sharma

Entry No. 2021MCS2154

Department of Computer Science & Engineering

Indian Institute of Technology Delhi

---

1 Software Requirements

1. GNU Compiler Collections (GCC).
2. Git Version Control System.

2 Directory Structure

1. include
2. lib
3. obj
4. src
5. Makefile

3 Introduction to Threads

Threads are used to introduce parallelism in the code. In Case if we have a multi-core system we can effectively schedule our thread over the different core and can run our code parallel. They are also referred as the lightweight process. They can follow a complete independent sequence of instructions within a running process. Multiple threads within the same process can share the same code and data section.

Each thread also have a data structure associated with it that is called TCB(Thread Control Block) which stores the state of the thread for example It stores the value of the registers that are used by the thread in case of a switch. Each thread has its local data which is generally not shared in stack space, a set of registers and a PC(Program Counter).

4 User Thread library

Threads at user levels are generally managed without any kernel supports. Kernel are not aware of these threads. Generally these are introduced by the programmer in their implementation. In our implementation of myUserThread we have implemented functions below. First we have defined a structure which which will store information about the thread.

4.1 Structure of User Thread Library

```
1         typedef struct TCB {
2             ucontext_t* context;
3             int state, id;
```

```
4           struct attr* attribute;
5           void* retval;
6       }tcb_t;
7
8       typedef struct attr {
9           size_t stackSize;
10          void* stackPtr;
11      }attr_t;
```

4.2 Methods used in the User Thread Library

1. **myThread_init()** : This is very first function which we call. This basically initialize the ready queue and the terminated queue. Along with that this function is used to push the thread of our main program into the ready queue. It also setup the setittimer function which periodically sends a **SIGALRM** signal at the period of 50ms.

2. **myThread_create(thread,attr,start_routine,arg)** : This function is used to create the user thread. It takes four(4) parameters. First one is a pointer to the thread to which memory was allocated by the user using malloc. Second is the attr which contains detail about stack size and stack pointer if user pass them else it is passed NULL. Third parameter is start_routine which is the function that this thread will be executing. And last is the arg to this functions.

3. **myThread_exit()** : It basically set the state of TCB to the TERMINATED state and also enqueue it to the terminated queue and call the myThread_yield.

4. **myThread_cancel(thread)** : It removed the thread TCB from the ready queue and push it to the terminated queue.

5. **myThread_attr_init(attr)** : It is used to initialize the thread attributes. If user has passed then it is set from that. Else if user pass NULL then we initialize it with the default values.

6. **myThread_attr_destroy(attr)** : It basically destroy the attributes which we have set in the TCB thread.

7. **myThread_self(void)** : This simply return the a pointer to the current TCB itself which is present in the front of the ready queue.

8. **myThread_yield** : Yield is used to raise the signal SIGALRM immediately and basically that then call to the scheduler which in turn either swap or set the new context.

9. **myThread_join(thread)** : This basically wait for the thread to complete.

4.3 Structure of Queue

Queue is FIFO data structure. Here in this we will store the TCB of the thread in our queue. We have some basic functionality with our queue.

```
struct container {
    struct container* next;
    void* data;
};

struct queue {
    int sz;
    struct container *start, *end;
};

typedef struct queue queue_t;
```

4.4 Methods used in the Queue

1. **createQueue()** : It initialize the queue allocate memory. And also initialize the start and end pointer to NULL.
2. **enqueue(q, data)** : It basically takes two parameter a pointer to the queue and the data which we need to enqueue to the queue. It push data at the end of the queue.
3. **dequeue(q)** : It basically dequeue the data from the front of the queue.
4. **pushBack(q)** : It basically push the front of the queue to the back of the queue.
5. **isEmpty(q)** : It returns a Boolean true or false telling whether queue is empty or not.
6. **front(q)** : It simply return the front of the queue without dequeue it.
7. **queueFind(q, tid)** : It is basically used to find whether thread having id as tid belongs to the queue q or not.
8. **destroy(q, tid)** : It basically free the data present in the queue.

4.5 Structure of User Semaphores

Semaphore is simply a variable that is shared between threads. It is used to solve the critical section problem and also provide synchronization in the code.

```
struct mySemaphore {
    struct queue* waitQ;
    int count;
};

typedef struct mySemaphore semaphore_t;
```

4.6 Methods used in the User Semaphore Library

1. **createSemaphore(val)** : It initialize the semaphore with the val provided by the user. It also allocate the queue which is waiting queue for this semaphore.
2. **down(sm)** : It basically down the semaphore count. If it is not possible then it push the thread which is calling semaphore down in it's waiting queue.
3. **up(sm)** : It basically increase the count by one. If there are still some TCB in the waiting queue then it only push the front of the waiting queue into the ready queue without incrementing.
4. **destroySemaphore** : It basically free the waiting queue allocated on the heap.

5 Use of Signals

Signals are used to generate interrupts. We can use signal with the help of signal.h header file. The main function that are used in our code are.

1. **sigemptyset(sigset_t* st)** : This basically initializes the signal set given by st to empty means no signal is included.
2. **sigaddset()** :sigaddset() This adds the signal passed to the signal set.
3. **sigprocmask(how, *newset, *oldset)** :sigprocmask() is used to change the signal mask, the set of currently blocked signals.

6 Multi threaded Matrix Multiplication with N threads

Here we have implemented matrix multiplication with the help of the user thread library that we have implemented. Along with that because our result matrix is a shared resource between these N threads we also need semaphore.

6.1 Graph Time taken versus N for userthread and pthread

Below in Figure 1.

7 Bounded Buffer with N containers and M producer and consumers

We have buffer which is of size N. And we have M producer and N consumers. We are also using two counting semaphore full and empty. And a binary semaphore which is used

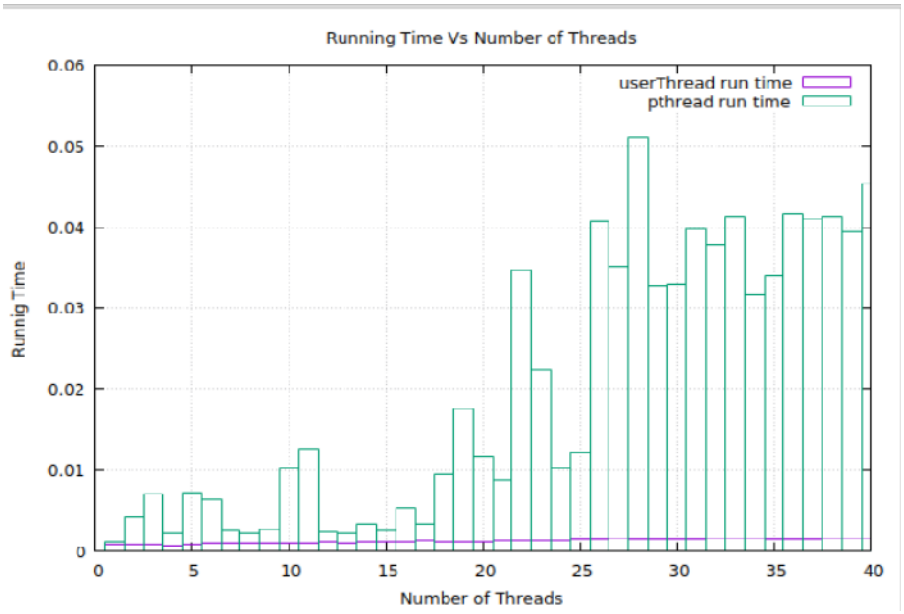

Figure 1: Graph showing Run Time for thread count up to 40

to access the common resource between producer and consumer i.e the buffer.

Producer Code:

```
while (cond) {
    down(empty);
    down(mutex);
    produce_item();
    put item to buffer;
    up(mutex);
    up(full);
}
```

Consumer Code:

```
while (cond) {
    down(full);
    down(mutex);
    consume_item();
    up(mutex);
    up(empty);
}
```

8 System Specifications

1. CPU(s) : 2
2. Socket(s) : 2
3. Core(s) per socket : 1
4. Thread(s) per core : 1

9 References

https://pubs.opengroup.org/onlinepubs/7908799/xsh/getcontext.html
https://man7.org/linux/man-pages/man3/makecontext.3.html
https://www.tutorialspoint.com/unix_system_calls/sigprocmask.htm
https://man7.org/linux/man-pages/man7/pthreads.7.html