```
//#####################################################
//Date: 5th Feb 2018                                  #
//Programming Assignment 1                              #
//Name: Sushant Trivedi                                 #
//GROUP 20                                                    #
//#####################################################
#define MLOCA "/dev/input/event4"
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/input.h>
#include <sys/timerfd.h>
#include <stdbool.h>
#include <sched.h>

//PTHREAD INITIALIZATIONS + GLOBAL VARIABLES
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_cond_t leftclick_cond = PTHREAD_COND_INITIALIZER;
pthread_cond_t rightclick_cond = PTHREAD_COND_INITIALIZER;
pthread_mutexattr_t mutex_attr[11];
pthread_mutex_t cond_mutex = PTHREAD_MUTEX_INITIALIZER;
      //Sync Mutex
pthread_mutex_t mutex[11] = {PTHREAD_MUTEX_INITIALIZER};              //Task
Mutex
pthread_mutex_t ccond_mutex = PTHREAD_MUTEX_INITIALIZER;
      //Periodic Mutex
pthread_mutex_t quitmutex = PTHREAD_MUTEX_INITIALIZER;
      //Termination Mutex
pthread_mutex_t mouse_mutex = PTHREAD_MUTEX_INITIALIZER;              //Mouse
Mutex
pthread_mutex_t aperiodic_mutex = PTHREAD_MUTEX_INITIALIZER;
      //Aperiodic Mutex
pthread_mutex_t busy_mutex = PTHREAD_MUTEX_INITIALIZER;
      //Busy Mutex
pthread_mutex_t left_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t right_mutex = PTHREAD_MUTEX_INITIALIZER;


int flag_exit = 0;
int left_flag = 0;
int right_flag = 0;
int lc = 0;
int rc = 0;
int lp = 0;
int rp = 0;
int mouseopen = 0;
struct input_event mouse_event;
int flag = 1;
int UserInput = 0;

//STRUCT DEFINITION
struct node
{
      char command;
      int ano;
      struct node *next;
};
```

```
//########################################FUNCTION
DEFINITION#####################################################
//DELAY FUNCTION
void delay(unsigned int msec)
{
    clock_t t1 = msec + clock();
    while (t1 > clock());
}

//BUSY LOOP
void BusyLoop(int count)
{
      int r=0;
      for(;r<count;r++);
}

//THREAD TERMINATION RELATED FUNCTIONS
int shouldQuit(void)
{
    int temp;
    pthread_mutex_lock( &quitmutex );
    temp = flag_exit;
    pthread_mutex_unlock( &quitmutex );
    return temp;
}
void ToExit()
{
    pthread_mutex_lock( &quitmutex );
    flag_exit = 1;
    pthread_mutex_unlock( &quitmutex );
}

//TASK BODY FUNCTIONS
int task(struct node *current)
{
      int tracker = 0;
      while(current != NULL)
      {
            switch(current->command)
            {
                  case 'P':
                        tracker=1;
                        break;
                  case 'C':
                        printf("C%d ", current->ano);
                        BusyLoop(current->ano);
                        tracker=1;
                        break;
                  case 'L':
                        printf("L%d ", current->ano);
                        pthread_mutex_lock( &mutex[current->ano]);
                        break;
                  case 'U':
                        printf("U%d ", current->ano);
                        pthread_mutex_unlock( &mutex[current->ano]);
                        tracker=1;
                        break;
                  default:
                        printf("No condition selected");
            }
            current = current->next;
      }
      printf("\n");
      if(tracker==1)
```

```
            {return 1;}
      else
            {return 0;}
}

//MONITOR MOUSE FUNCTIONS
void * monitor_mouse()
{
      int s;
      left_flag = 0;
      right_flag = 0;
      cpu_set_t cpuset;
      pthread_t pid = pthread_self();
      CPU_ZERO(&cpuset);
      CPU_SET(1,&cpuset);
      s = pthread_setaffinity_np(pid, sizeof(cpu_set_t), &cpuset);
        if (s != 0)
           printf("CPU BIND SET ERROR");

  /*  //SYNC LOCK
      pthread_mutex_lock(&cond_mutex);
      while(flag)
      {
            pthread_cond_wait(&cond, &cond_mutex);
      }
      pthread_mutex_unlock(&cond_mutex);
*/
      while(!shouldQuit())
      {
            if((read(mouseopen, &mouse_event, sizeof(struct input_event))))
            {
                  pthread_mutex_lock(&mouse_mutex);
            if(mouse_event.value==0 &&  mouse_event.code==272)
             {
                  printf("\nLEFT CLICK");
                  lp = lc;
                  pthread_cond_broadcast(&leftclick_cond);
                  left_flag = 1;
            }
            else if(mouse_event.value==0 && mouse_event.code==273)
            {
                  printf("\nRIGHT CLICK");
                  rp = rc;
                  pthread_cond_broadcast(&rightclick_cond);
                  right_flag = 1;
            }
                  pthread_mutex_unlock(&mouse_mutex);
            }
      }
    pthread_exit(NULL);
}

//APERIODIC TASK
//NOTE: 0 = LEFT BUTTON; 1 = RIGHT BUTTON
void * aperiodic (void * arg)
{

      int f;
      int eventtype;
      int period;
      struct node *current = arg;
      eventtype = current->ano;
      current = current->next;
      period = current->ano;
```

```
	cpu_set_t cpuset;
	pthread_t pid = pthread_self();
	CPU_ZERO(&cpuset);
	CPU_SET(1,&cpuset);
	f = pthread_setaffinity_np(pid, sizeof(cpu_set_t), &cpuset);
		if (f != 0)
		printf("CPU BIND SET ERROR");

	//SYNC LOCK
	pthread_mutex_lock(&aperiodic_mutex);
	while(flag)
	{
		pthread_cond_wait(&cond, &cond_mutex);
	}
	pthread_mutex_unlock(&aperiodic_mutex);
	printf("TASK BODY STARTED %d\n", eventtype);

	left_flag = 0;
	right_flag = 0;
	while(!shouldQuit())
	{
		switch(eventtype)
		{
			//LEFT CLICK CODE
			case 0:
					pthread_mutex_lock( &left_mutex);
					while(lp == 0)
					{
						pthread_cond_wait( &leftclick_cond,
&left_mutex );
						if(!shouldQuit())
					{
						BusyLoop(period);
						printf(" BUSY LOOP\n");
					}
			lp-=1;
			pthread_cond_broadcast(&leftclick_cond);
				}
					pthread_mutex_unlock( &left_mutex);
					break;
			//RIGHT CLICK CODE
			case 1:
					pthread_mutex_lock( &right_mutex);
					while(rp == 0)
					{
						pthread_cond_wait( &rightclick_cond,
&right_mutex);
					if(!shouldQuit())
					{
						BusyLoop(period);
						printf(" BUSY LOOP\n");
					}
			rp-=1;
			pthread_cond_broadcast(&rightclick_cond);
				}
					pthread_mutex_unlock( &right_mutex);
					break;
		}

	}

	printf("THREAD ENDED\n");
	//pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
```

```
	pthread_exit(NULL);

}


//PERIODIC TASK
void * periodic (void * arg)
{
	int per;
	struct node *LLP = arg;
	int status;
	struct timespec pott;
	int s;

	//CPU BINDING
	cpu_set_t cpuset;
	pthread_t pid = pthread_self();
	CPU_ZERO(&cpuset);
	CPU_SET(1,&cpuset);
	s = pthread_setaffinity_np(pid, sizeof(cpu_set_t), &cpuset);
	  if (s != 0)
	     printf("CPU BIND SETERROR");

	//SYNC LOCK
	pthread_mutex_lock(&cond_mutex);
	while(flag)
	{
		pthread_cond_wait(&cond, &cond_mutex);
	}
	pthread_mutex_unlock(&cond_mutex);

	per = LLP->ano;
	LLP = LLP->next;
	while(!shouldQuit())
	{

		pthread_mutex_lock(&ccond_mutex);
		status = clock_gettime(CLOCK_MONOTONIC, &pott);
		if (status == -1)
		{printf("Time Error");exit(1);}

		pott.tv_sec += (per/1000);
		pott.tv_nsec += (per%1000) * 1000000;
		if (pott.tv_nsec >= 1000000000)
		{
			pott.tv_nsec -= 1000000000;
			pott.tv_sec ++;
		}
		pthread_mutex_unlock(&ccond_mutex);
		s = task(LLP);
		if(s==0)
			{printf("no task body found");}
		clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &pott, NULL);
	}

pthread_exit(NULL);
}

//#########################################LINKED LISTS
CREATION#####################################################
// CREATE A NODE AS PER THE INFO IN THE ARRAY AND RETURN THE ADDRESS POINTER TO
IT
struct node * NNodeFunc ( char *metadata)
{
```

```
	struct node *newnode = NULL;
	newnode = (struct node *)malloc(sizeof(struct node));

	switch(metadata[0])
	{
		case'L':
			newnode->command = 'L';
			newnode->ano = metadata[1] - '0';
			break;
		case'U':
			newnode->command = 'U';
			newnode->ano = metadata[1] - '0';
			break;
		default:
			newnode->command = 'C';
			newnode->ano = atoi(metadata);
	}
return newnode;
}

//CREATING A LINKED LIST AS PER THE MATRIX ARRAY AND RETURNS A POINTER TO THE
START OF LL
struct node * createLL( char (*fdata)[600])
{
	int count = 0;
	char *token1;
	struct node *front = (struct node *)malloc(sizeof(struct node));
	struct node *acurrent = (struct node  *)malloc(sizeof(struct node));
	struct node *temp = (struct node *)malloc(sizeof(struct node));
	front = acurrent = temp;
	token1 = strtok(*fdata, " ");
	while(token1 != NULL)
	{
		switch(count)
		{
			case 0:
			case 1:

			break;
			case 2:
				front->command = 'P';
				front->ano = atoi(token1);
				break;
			default:
				temp = NNodeFunc (token1);
				acurrent->next = temp;
				acurrent = temp;
		}
		token1 = strtok(NULL, " ");
		count++;
	}
	acurrent->next = NULL;
	return front;
}

// CREATES A LINKED LIST FOR APERIODIC TASK
// 0 -LEFT CLICK , 1 - RIGHT CLICK
struct node * create_aperiodicLL (char (*str)[600])
{
	struct node *front = (struct node*) malloc(sizeof(struct node));
	struct node *first = (struct node*) malloc(sizeof(struct node));
	struct node *second = (struct node*) malloc(sizeof(struct node));
	int aperiod = 0;
	int eventno = 0;
```

```
	int apcount = 0;
	char *ap_token;
	ap_token = strtok(*str, " ");
	while(ap_token != NULL)
	{
		switch(apcount)
		{
			case 2: eventno = atoi(ap_token);
				if(eventno == 0){lc += 1;}
				else if(eventno == 1){rc += 1;}
				//printf("%d ", eventno);
				break;
			case 3: aperiod = atoi(ap_token);
				//printf("%d\n", aperiod);
				break;
			default: break;
		}
		apcount++;
		ap_token = strtok(NULL, " ");
	}
	first = front;
	first->command = 'A';
	first->ano = eventno;
	first->next = second;
	second->command = 'C';
	second->ano = aperiod;
	second->next = NULL;

	return first;
}

//#########################################MAIN
FUNCTION#####################################################
int main()
{

	//INTIALIZATIONS
	char temp[600];
	char str[600];
	int priority[600];
	memset(priority,'\0',600);
	memset(temp,'\0',600);
	memset(str,'\0',600);
	int i = 0;
	int ptrack = 0;
	int no_of_tasks = 0;
	int task_period = 0;
	//int input = 1;
	const char s[2] = " ";
	char *ptoken;
	struct node *ListLL = NULL;
	char *string,*found;

	//USER MAKES A CHOICE

			printf("PLEASE CHOOSE\n1. Dont set Protocol as
PTHREAD_PRIO_INHERIT\n2. Set Protocol as PTHREAD_PRIO_INHERIT\nChoice: ");
			scanf("%d", &UserInput);

			switch(UserInput)
		{
			case 1:
				for (i=0; i<11; i++)
					{ pthread_mutex_init(&mutex[i], NULL); }
```

```
                              //input = 0;
                              break;
                        case 2:
                                    for(i=0; i<11; i++)
                                          {

      pthread_mutexattr_setprotocol(&mutex_attr[i], PTHREAD_PRIO_INHERIT);
                                                pthread_mutex_init(&mutex[i],
&mutex_attr[i]);
                                          }
                                    //input = 0;
                                    break;
                        default:
                                    printf("Incorrect Choice\n");
                                    exit(-1);

                  }

    //MOUSE CODE
    mouseopen = open(MLOCA, O_RDONLY|O_NONBLOCK);
    if(mouseopen==-1)
    {printf("Mouse open failed\n");  exit(EXIT_FAILURE);}

      //MOUSE THREAD CREATE
      pthread_t m;
      pthread_attr_t matt;
      struct sched_param mparam;
      pthread_attr_init(&matt);
      pthread_attr_getschedparam(&matt, &mparam);
      mparam.sched_priority = 1;
      pthread_attr_setschedpolicy(&matt, SCHED_FIFO);
      pthread_attr_setschedparam(&matt, &mparam);
      pthread_create(&m, &matt, monitor_mouse, NULL);

      pthread_mutex_init(&mouse_mutex, NULL);
      pthread_mutex_init(&left_mutex, NULL);
      pthread_mutex_init(&right_mutex, NULL);
      pthread_mutex_init(&aperiodic_mutex, NULL);


      printf("MOUSE PORT OPENED AND THREAD CREATED\n");


      //FILE OPEN
      FILE *in;
      in = fopen("Specification.txt","r+");
      if (in == NULL)
      {printf("FILE NOT FOUND");}

      //FIRST LINE READ
      if( fgets (str, 600, in)!=NULL );

    i=0;
      string = strdup(str);
    while( (found = strsep(&string," ")) != NULL )
    {
            switch(i)
            {
                  case 0: no_of_tasks = atoi(found);
                  break;
                  case 1: task_period = atoi(found);
                  break;
            }i++;
```

```
    }
      printf("%d %d", no_of_tasks,task_period);
      //PTHREAD ATTRIBUTES;
      pthread_t tid[no_of_tasks];
      pthread_attr_t attr [no_of_tasks];
      struct sched_param param[no_of_tasks];
      pthread_cond_init(&cond, NULL);
      pthread_cond_init(&leftclick_cond,NULL);
      pthread_cond_init(&rightclick_cond,NULL);
      pthread_mutex_init(&cond_mutex, NULL);
      pthread_mutex_init(&ccond_mutex, NULL);
      pthread_mutex_init(&quitmutex, NULL);
      pthread_mutex_init(&busy_mutex, NULL);



      //CREATE LINKEDLIST
      ListLL = (struct node *)malloc((sizeof(struct node)*no_of_tasks));
      i=0;
      memset(str,'\0',600);

      while(i<no_of_tasks)
      {

            if( fgets(str, 600, in)==NULL )
            {
                  printf("No Data\n");
                  break;
            }
            printf("STRING: %s\n", str);
            strcpy(temp, str);

            //CREATING PRIORITY ARRAY
            ptrack = 0;
            ptoken = strtok(temp, s);
            while( ptoken != NULL )
            {
                  if(ptrack == 1)
                  {
                        priority[i] = atoi(ptoken);
                  }
                  ptrack++;
                  ptoken = strtok(NULL, s);
            }

            //SETTING PRIORITY ATTRIBUTES
            pthread_attr_init(&attr[i]);
            pthread_attr_getschedparam(&attr[i], &param[i]);

            param[i].sched_priority = priority[i];
            pthread_attr_setschedpolicy(&attr[i], SCHED_FIFO);
            pthread_attr_setschedparam(&attr[i],&param[i]);

            if (str[0] == 'P')
            {
                  //str has one line of data in it
                  struct node *start = (struct node *)malloc(sizeof(struct
node));
                  start = createLL((&str));
                  //printf("The START ELEMENT IS: %c%d\n", start->command,start-
>ano);
                  ListLL[i] = *start;
                  pthread_create(&tid[i], &attr[i], periodic, &ListLL[i]);
            }
```

```
            else if (str[0] == 'A')
            {

                  struct node *header = (struct node *)malloc(sizeof(struct
node));
                  header = create_aperiodicLL(&str);
                  ListLL[i] = *header;

                  //printf("Current: %c%d\n", header->command, header->ano);
                  pthread_create(&tid[i], &attr[i], aperiodic, header);
            }
                        // PRINTING THE CURRENT STRING THAT IS BEING PROCESSED
            i++;
      }

      //THREAD STARTED
    printf("--------------------STARTING THREADS WITH TOTAL PERIOD:
%d-----------------------------\n", task_period);
      flag = 0;
      delay(500);
      pthread_cond_broadcast(&cond);

      //WAITING FOR A TIME PERIOD OF THE TASK TO TERMINATE TASK
      sleep(task_period/1000);
      printf("TASK TIME COMPLETE: TERMINATING\n");
      ToExit();
      pthread_cond_broadcast(&rightclick_cond);
    pthread_cond_broadcast(&leftclick_cond);
      for(i=0;i<no_of_tasks;i++)
      {
            pthread_join(tid[i], NULL);
      }
      pthread_cancel(m);
      pthread_join(m, NULL);
      printf("ALL THREADS TERMINATED\n");

      //DESTROY ALL MUTEX, COND
      pthread_mutex_destroy(&mouse_mutex);
      pthread_mutex_destroy(&aperiodic_mutex);
      pthread_mutex_destroy(&cond_mutex);
      pthread_mutex_destroy(&ccond_mutex);
      pthread_mutex_destroy(&quitmutex);
      pthread_mutex_destroy(&busy_mutex);
      pthread_mutex_destroy(&right_mutex);
      pthread_mutex_destroy(&left_mutex);
      for(i=0;i<11;i++)
      {pthread_mutex_destroy(&mutex[i]);}
      pthread_cond_destroy(&cond);
      pthread_cond_init(&leftclick_cond,NULL);
      pthread_cond_init(&rightclick_cond,NULL);
      pthread_attr_destroy(&matt);
/*  for(i=0;i<no_of_tasks;i++)
      {pthread_attr_destroy(&attr[i]);}

*/
return 0;
}
```