AIM: To write a program that demonstrates the interrupt handling mechanism

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/tqueue.h>
#include <linux/interrupt.h>
#include <asm/io.h>
static void got_char(void *scancode){
printk("Scan Code %x %s.\n",
(int) *((char *) scancode) & 0x7F, *((char *) scancode) & 0x80 ? "Released" : "Pressed");
}
void irq_handler(int irq, void *dev_id, struct pt_regs *regs){
static unsigned char scancode;
static struct tq_struct task = {NULL, 0, got_char, &scancode
};
unsigned char status;
status = inb(0x64);
scancode = inb(0x60);
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
queue_task(&task, &tq_immediate);
#else
queue_task_irq(&task, &tq_immediate);
#endif
mark_bh(IMMEDIATE_BH);}
int init_module(){
free_irq(1, NULL);
return request_irq(1, irq_handler,SA_SHIRQ,"test_keyboard_irq_handler", NULL);
}
void cleanup_module(){
```

```
free_irq(1, NULL);}
```

AIM: To write a program that allows sharing of resource using MUTEX lock

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
pthread_t tid[2];
int counter;
pthread_mutex_t lock;
void* doSomeThing(void *arg){
pthread_mutex_lock(&lock);
unsigned long i = 0;
counter += 1;
printf("\n Job %d started\n", counter);
for(i=0; i<(0xFFFFFFFF);i++);
printf("\n Job %d finished\n", counter);
pthread_mutex_unlock(&lock);
return NULL;}
int main(void){
int i = 0;
int err;
if (pthread_mutex_init(&lock, NULL) != 0){
printf("\n mutex init failed\n");
return 1;}
while(i < 2){
err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
if (err != 0)
printf("\ncan't create thread :[%s]", strerror(err));
i++;
}
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);
pthread_mutex_destroy(&lock);
return 0;
}
```

AIM: To write a program to create a new task/process

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
int global;
int main(){
pid_t child_pid;
int status;
int local = 0;
child_pid = fork();
if (child_pid >= 0) {
if (child_pid == 0){
printf("child process!\n");
local++;
global++;
printf("child PID = %d, parent pid = %d\n", getpid(), getppid());
printf("\n child's local = %d, child's global = %d\n",local,global);
char *cmd[] = {"whoami",(char*)0};
return execv("/usr/bin/",cmd); }
else {
printf("parent process!\n");
printf("parent PID = %d, child pid = %d\n", getpid(), child_pid);
wait(&status);
printf("Child exit code: %d\n", WEXITSTATUS(status));
printf("\n Parent'z local = %d, parent's global = %d\n",local,global);
printf("Parent says bye!\n");
exit(0); }}
else {
perror("fork");
exit(0);}}
```

AIM: To write a program that demonstrates reader’s and writer’s problem

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
sem_t w;
sem_t m;
int rc=0;
int writersCount;
int readersCount;
pthread_t writersThread[10], readersThread[10];
int writeCount[10], readCount[10];
int i;
void *writer(void *i) {
int a = *((int *) i);
sem_wait(&w);
printf("Writer %d writes to DB.\n",a+1);
writeCount[a+1]++;
```

```
sem_post(&w); // V(w)
free(i);
}
void *reader(void *i) {
int a = *((int *) i);
sem_wait(&m);
rc++;
if (rc == 1) {
sem_wait(&w);
}
sem_post(&m);
printf("Reader %d reads from DB.\n",a+1);
readCount[a+1]++;
sem_wait(&m);
if (rc == 0) {
sem_post(&w); }
sem_post(&m);
free(i);}
int main() {
sem_init(&w,0,1);
sem_init(&m,0,1);
printf("Enter count of writers:");
scanf("%d",&writersCount);
printf("Enter count of readers:");
scanf("%d",&readersCount);
for (i=0; i<readersCount; i++) {
int *arg = malloc(sizeof(*arg));
*arg = i;
pthread_create(&readersThread[i], NULL, reader, arg);
}
for (i=0; i<writersCount; i++) {
int *arg = malloc(sizeof(*arg));
*arg = i;
pthread_create(&writersThread[i], NULL, writer, arg);}
for (i=0; i<writersCount; i++) {
pthread_join(writersThread[i], NULL);}
for (i=0; i<readersCount; i++) {
pthread_join(readersThread[i], NULL);}
printf("--------------\n");
for (i=0; i<readersCount; i++) {
printf("Reader %d read %d times\n",i+1,readCount[i+1]);}
for (i=0; i<writersCount; i++) {
printf("Writer %d wrote %d times\n",i+1,writeCount[i+1]);}
sem_destroy(&w);
sem_destroy(&m);
return 0;
}
```

AIM: To write a program to that allocates a resource using a semaphore

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <stdlib.h>
#include <semaphore.h>
#define THREADS 20
#define RESOURCES 4
int resourceTable[RESOURCES];
void initResourceTable(){
for(int i = 0; i < RESOURCES; i++) resourceTable[i] = 1;
```

```
}

int allocateResource(){

int id;

id = 0;

while((id < RESOURCES) && (resourceTable[id] != 1)) id++;

if(id >= RESOURCES){

printf("**** error in allocation!\n");

exit(-1);

}

resourceTable[id] = 0;

return id;

}

void releaseResource(int id){

resourceTable[id] = 1;

}

void printResourceTable(){

printf("-- resource table --\n");

for(int i = 0; i < RESOURCES; i++){

printf(" resource #%d: %d\n", i, resourceTable[i]);

}

printf("--------------------\n");}

void* worker(void *threadId){

int resourceId;

resourceId = allocateResource();

printf("thread #%ld uses resource %d\n", (long)threadId, resourceId);

releaseResource(resourceId);

return NULL;

}

int main(int argc, char **argv){
```

pthread_t threads[THREADS];

sem_t * sem = NULL;

sem_init(sem,0,1);

int k;

initResourceTable();

printResourceTable();

for(int i = 0; i < THREADS; i++){

if(pthread_create(&threads[i], NULL, &worker, (void *)((long)i))){

printf("** could not create thread %d\n", i);**

return -1;}}

for(int i = 0; i < THREADS; i++){

if(pthread_join(threads[i], NULL)){

printf("** could not join thread %d\n", i);**

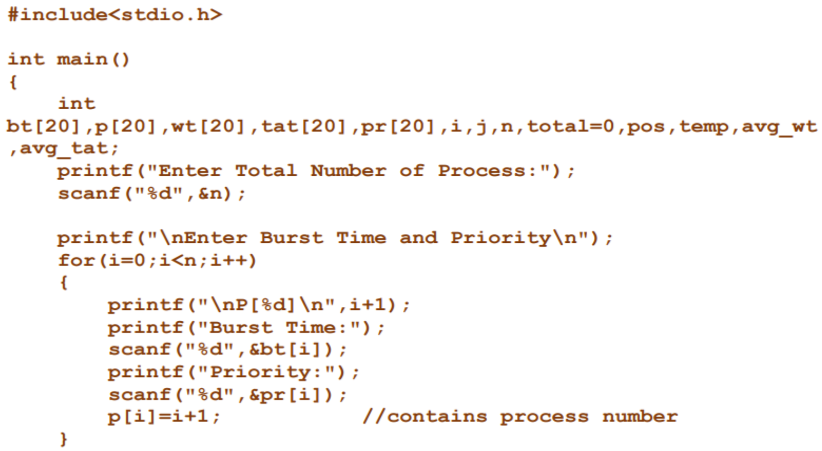
return -1;}}

printResourceTable();

return 0;

}

Aim: Priority-based Non-Preemptive Scheduling



```
#include<stdio.h>

int main()
{
    int
bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt
,avg_tat;
    printf("Enter Total Number of Process:");
    scanf("%d",&n);

    printf("\nEnter Burst Time and Priority\n");
    for(i=0;i<n;i++)
    {
        printf("\nP[%d]\n",i+1);
        printf("Burst Time:");
        scanf("%d",&bt[i]);
        printf("Priority:");
        scanf("%d",&pr[i]);
        p[i]=i+1;           //contains process number
    }
    //sorting burst time, priority and process number in
ascending order using selection sort
    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(pr[j]<pr[pos])
                pos=j;
        }

        temp=pr[i];
        pr[i]=pr[pos];
        pr[pos]=temp;

        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;

        temp=p[i];
        p[i]=p[pos];
        p[pos]=temp;
    }

    wt[0]=0;    //waiting time for first process is zero

    //calculate waiting time
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];

        total+=wt[i];
    }

    avg_wt=total/n;      //average waiting time
    total=0;

    printf("\nProcess\t    Burst Time    \tWaiting
Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];     //calculate turnaround time
        total+=tat[i];
        printf("\nP[%d]\t\t  %d\t\t
%d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);

    }

    avg_tat=total/n;     //average turnaround time
    printf("\n\nAverage Waiting Time=%d",avg_wt);
    printf("\nAverage Turnaround Time=%d\n",avg_tat);

    return 0;
}
```