

CS344
Lab Assignment-2

Group No- M23

Group members -

- 1- Harshul Gupta (200123023)
- 2- Arti Sahu (200123011)
- 3- Sunny Narzary(200123062)

Part A:

(1) To create the system calls getNumProc() and getMaxPID().

We create user programs named getNumProcTest and getMaxPIDTest to use the above system calls.

Two functions namely, getNumProcAssist and getMaxPIDAssist are implemented in proc.c which help us in achieving the desired functionalities. They are attached below.

```
//total number of active processes in the system
int getNumProcAssist(void)
{
    int ans=0;
    struct proc *p;

    //process table structure ptable is protected by a lock.
    acquire(&ptable.lock);
    //define NPROC -- > 64-maximum number of processes(param.h)
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if(p->state != UNUSED) //state which is not is unused
            ans++;
    }
    release(&ptable.lock);

    return ans;
}

//returns the maximum PID amongst the PIDs of all currently active processes
int getMaxPIDAssist(void)
{
    int max=0;
    struct proc *p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if(p->state != UNUSED)
        {
            if(p->pid > max)
                max=p->pid;    // obtain max pid
        }
    }
    release(&ptable.lock);

    return max;
}
```

In order to define our all system calls given in Part A in xv6, we changed some files mentioned below.

1) syscall.h

We added the below new system calls in syscall.h:

```
#define SYS_getNumProc 22
#define SYS_getMaxPID 23
#define SYS_getProcInfo 24
#define SYS_set_burst_time 25
#define SYS_get_burst_time 26
#define SYS_test_scheduler 27
#define SYS_getCurrentInfo 28
#define SYS_getCurrentPID 29
```

21 positions were already occupied by the inbuilt system calls in syscall.h.

2) syscall.c

We added the below pointers to in order to add our custom system call.

```
[SYS_getNumProc] sys_getNumProc,
[SYS_getMaxPID] sys_getMaxPID,
[SYS_getProcInfo] sys_getProcInfo,
[SYS_set_burst_time] sys_set_burst_time,
[SYS_get_burst_time] sys_get_burst_time,
[SYS_getCurrentInfo] sys_getCurrentInfo,
[SYS_getCurrentPID] sys_getCurrentPID,
```

Then below function prototypes are added in syscall.c file to be called by system call numbers:

```
extern int sys_getNumProc(void);
extern int sys_getMaxPID(void);
extern int sys_getProcInfo(void);
extern int sys_set_burst_time(void);
extern int sys_get_burst_time(void);
extern int sys_getCurrentInfo(void);
extern int sys_getCurrentPID(void);
```

3) sysproc.c

Below system call functions are implemented in sysproc.c. All the below function definitions are given along with the function body in sysproc.c with explanations.

```
Int sys_getNumProc(void)
int sys_getMaxPID(void)
Int sys_getProcInfo(void)
Int sys_set_burst_time(void)
Int sys_get_burst_time(void)
Int sys_getCurrentInfo(void)
int sys_getCurrentPID(void)
```

4) usys.s

For creating an interface for our user programs to access system call from user.h we added the following lines in usys.S.

```
SYSCALL(getNumProc)
SYSCALL(getMaxPID)
SYSCALL(getProcInfo)
SYSCALL(set_burst_time)
SYSCALL(get_burst_time)
SYSCALL(getCurrentInfo)
SYSCALL(getCurrentPID)
```

5) user.h

We added the below function declarations that the respective user programs will be calling in user.h for invoking system calls.

```
int getNumProc(void);
int getMaxPID(void);
int getProcInfo(int, struct processInfo*);
int set_burst_time(int);
int get_burst_time();
int getCurrentInfo(struct processInfo *);
int getCurrentPID();
```

6) proc.h

```
int nocs;           //Number of context switches
int burst_time;     //Burst time
int rt;             //running time
```

7) proc.c

The above functions are defined in proc.c along with their function body so that the kernel can access and execute the code in prog.c.

8)MakeFile:

```
_getNumProcTest\
_getMaxPIDTest\
_getProcInfoTest\
_set_burst_time\
_test_scheduler\
_burstTimeTest\
```

As can be seen, the functions access ptable by acquiring its lock and then loop through it to carry out their respective tasks.

The output obtained on calling the user programs is attached below:

```
harshul@harshul: ~/xv6-public
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 19312
echo       2 4 18188
forktest   2 5 9036
grep       2 6 22156
init       2 7 18764
kill       2 8 18276
ln         2 9 18176
ls         2 10 20744
ioProcTester 2 11 20244
rm         2 12 18280
sh         2 13 32288
stressfs   2 14 19208
usertests  2 15 66616
cpuProcTester 2 16 20464
zombie     2 17 17860
getNumProcTest 2 18 17964
getMaxPIDTest 2 19 17924
getProcInfoTes 2 20 18700
set_burst_time 2 21 18280
test_scheduler 2 22 21212
burstTimeTest 2 23 18080
console    3 24 0
$ getNumProcTest
The total number of active processes in the system are 3
$ getMaxPIDTest
The maximum PID is 5
$ getNumProcTest
The total number of active processes in the system are 3
```

First, a `ls` command is run which shows the list of user programs available. This process is run with process ID 3. Because process ID 1 and 2 are allotted to system processes because of which the next available process ID is 3 which is allotted to `ls`.

After completion of `ls` process, `getNumProcTest` is run. The next available process ID is 4 which is allotted to it. At this time, 3 processes (`ls` has already terminated) are currently running on the xv6 OS, the two system processes with PID 1 and PID 2 and `getNumProcTest`. Hence, the output is 3

Similarly, when `getMaxPIDTest` is run, it gets a PID of 5. Hence the output is 5.

Then again `getNumProcTest` is run so the output will be 3 as `getMaxPIDTest` is terminated.

(2) We solve the problem of passing parameters to syscall using argptr which is a predefined system call which serves our purpose.

As for the context switch part, we will solve the problem by modifying struct proc to include one additional member named nocs which will store the number of context switches.

Now we need to initialize noc for every process. We do this by setting p->nocs to 0 in allocproc() since before any process is run, it is allocated (assigned a place in ptable) using allocproc.

The next thing we do is add the statement (p->nocs)++ to scheduler(). This ensures that every time a process is scheduled, the number of context switches are updated accordingly.

We create a dummy process defaultParent and set it as parent of every process using p->parent=&defaultParent in allocproc(). fork() replaces it with the original parent after the process is allocated. We set the PID of defaultParent to -2 in scheduler(). Using defaultParent, we instantly know if the process has a parent or not and if it has, we get its PID using p->parent->pid. The implementations are given below:

```
//tells info about the process
struct processInfo getProcInfoAssist(int pid)
{
    struct proc *p;
    struct processInfo temp = {-1,0,0}; //temp (a dummy variable of type processInfo)
                                         //to indicate that there exists no process with the given PID.

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if(p->state != UNUSED){
            // printf(1, "%d\n", p->pid);
            if(p->pid == pid)
            {
                //find pid
                temp.ppid = p->parent->pid;           //parent pid
                temp.psize = p->sz;                   //size
                temp.numberContextSwitches = p->nocs;  //nocs->no. of contest switching
                release(&ptable.lock);
                return temp;
            }
        }
    }
    release(&ptable.lock);
    return temp;
}
```

```

int
sys_getProcInfo(void){
    int pid; //process id

    struct processInfo *info;
    // argptr to pass a pointer-to-struct to my system call
    argptr(0,(void *)&pid, sizeof(pid));
    argptr(1,(void *)&info, sizeof(info));

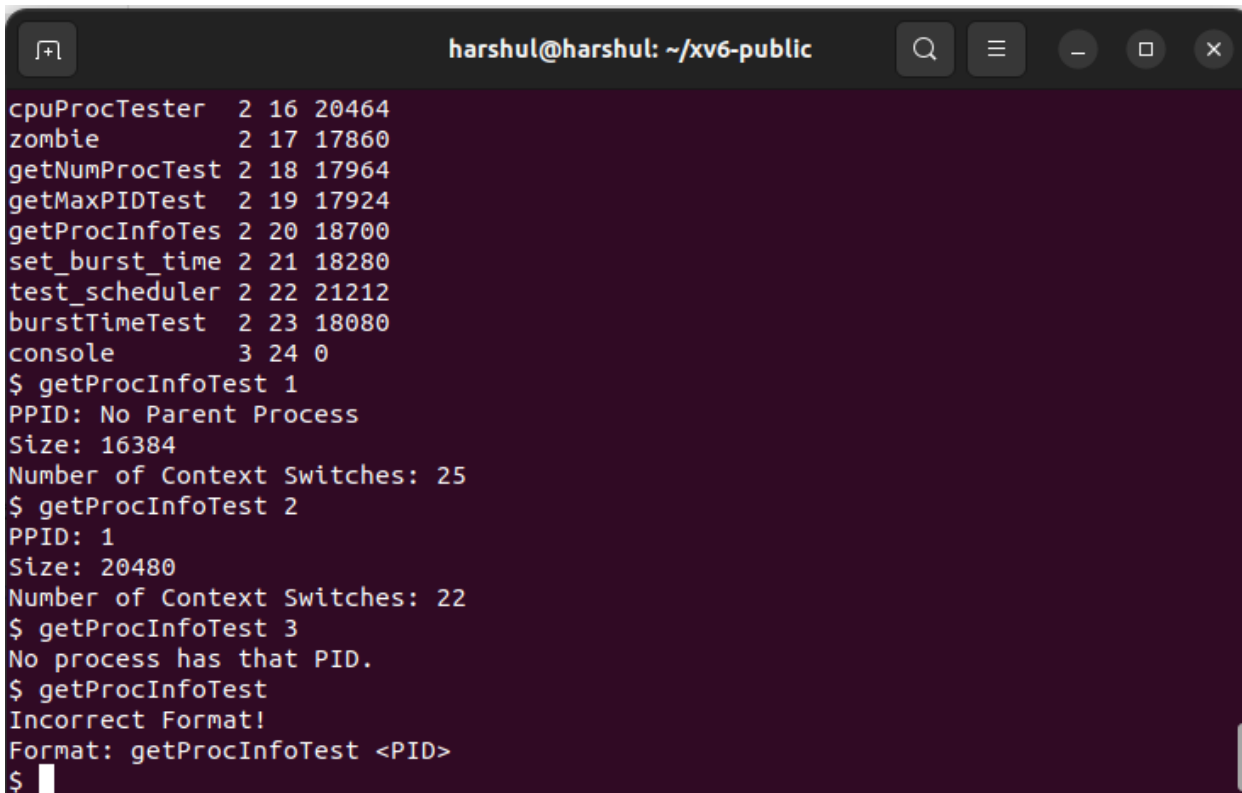
    struct processInfo temporaryInfo = getProcInfoAssist(pid);

    if(temporaryInfo.ppid == -1)
        return -1;

    info->ppid = temporaryInfo.ppid;
    info->psize = temporaryInfo.psize;
    info->numberContextSwitches = temporaryInfo.numberContextSwitches;
    return 0;
}

```

The output obtained on running getProcInfoTest is given below:
getProcInfoTest takes process ID as a command line
parameter which it then passes to getProcInfo which then passes it to
getProcInfoAssist.



```

harshul@harshul: ~/xv6-public
cpuProcTester  2 16 20464
zombie         2 17 17860
getNumProcTest 2 18 17964
getMaxPIDTest  2 19 17924
getProcInfoTes 2 20 18700
set_burst_time 2 21 18280
test_scheduler 2 22 21212
burstTimeTest  2 23 18080
console        3 24 0
$ getProcInfoTest 1
PPID: No Parent Process
Size: 16384
Number of Context Switches: 25
$ getProcInfoTest 2
PPID: 1
Size: 20480
Number of Context Switches: 22
$ getProcInfoTest 3
No process has that PID.
$ getProcInfoTest
Incorrect Format!
Format: getProcInfoTest <PID>
$

```

3) For this part, an additional attribute namely ***burst_time*** has to be added to proc structure. We also have to implement 2 system calls namely, `set_burst_time` and `get_burst_time` which will set and get the burst time of the current process to a given value respectively.

We have already defined an attribute named `burst_time` but we need to set it to a default value before any changes are made. We do this by setting `p->burst_time` to 0 (we took the default value of burst time to be 0) in `allocproc()`.

Now, the next problem we face is how to access the current process without any given info such as process ID etc. The solution is to use a predefined function in xv6 namely `myproc()` which returns the pointer to the proc structure of the current process. Using this, we can easily access and change the burst times of the current process.

The implementation for both the functions are given below.

```
//get burst time assist function
int get_burst_timeAssist()
{
    struct proc *p = myproc();
    return p->burst_time;
}

//set burst time assist function
int set_burst_timeAssist(int burst_time)
{
    struct proc *p = myproc(); // function in xv6 namely myproc() which returns the pointer to proc structure of current process.
    p->burst_time = burst_time; //set burst time
    yield(); //yield function in xv6, the process that wishes to give up the CPU after a timer interrupt.
            //yield first locks the global lock protecting the process table, before marking itself as RUNNABLE and invoking the scheduler.
            // added for sjf sheduler , not needed in round robin
    //yield switches the state of the current process to RUNNABLE, inserts it into the priority queue and switches the context to the scheduler.
    return 0;
}
```

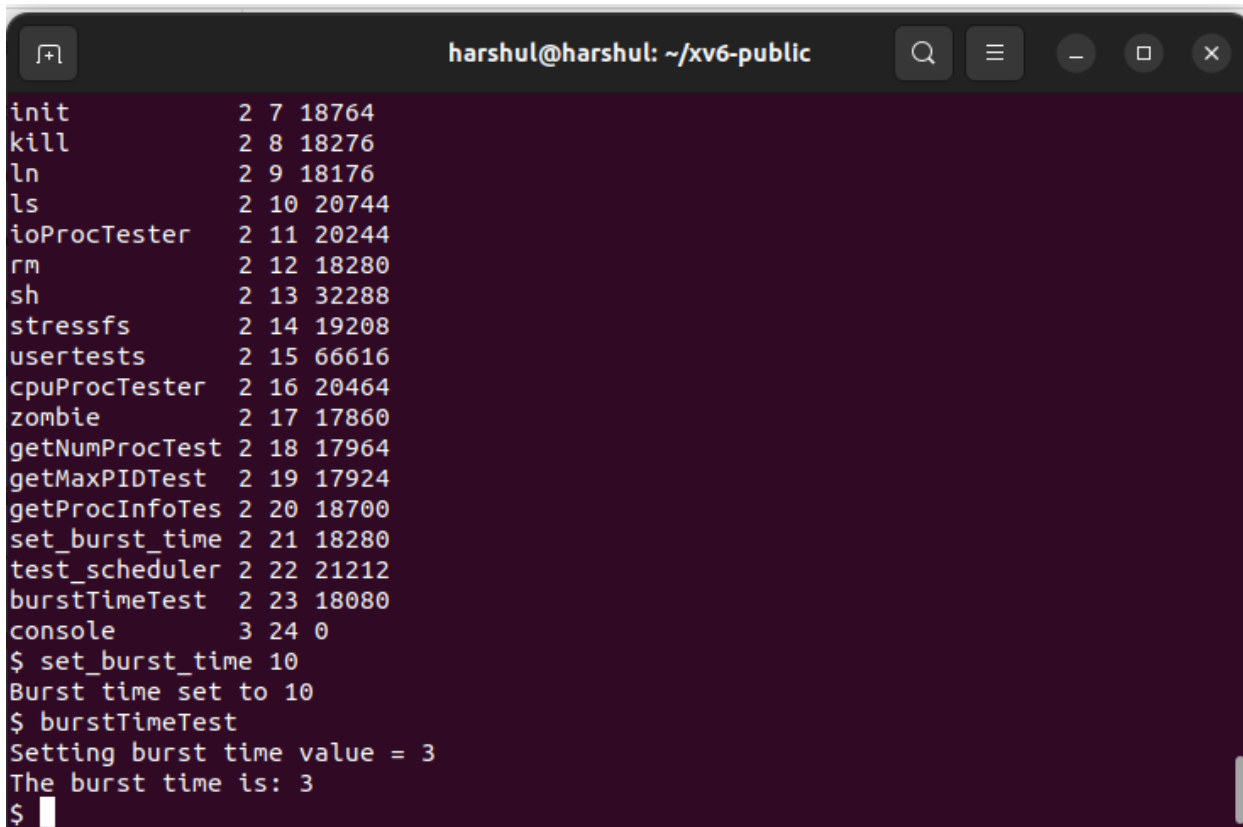
To test the above processes, we create a user program named `burst_time_test`.

It is shown below.

```
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(void){
    //int a=3;
    printf(1, "Setting burst time value = 3 \n");

    set_burst_time(3);
    printf(1, "The burst time is: %d \n", get_burst_time());
    exit();
}
```


A terminal window titled 'harshul@harshul: ~/xv6-public' with standard window controls. It displays a list of processes with their names, PPIDs, and PIDs. The 'console' process is highlighted in blue. Below the list, the user runs 'set_burst_time 10', which outputs 'Burst time set to 10'. Then, the user runs 'burstTimeTest', which outputs 'Setting burst time value = 3' and 'The burst time is: 3'.

```
init          2 7 18764
kill          2 8 18276
ln            2 9 18176
ls            2 10 20744
ioProcTester  2 11 20244
rm            2 12 18280
sh            2 13 32288
stressfs      2 14 19208
usertests     2 15 66616
cpuProcTester 2 16 20464
zombie        2 17 17860
getNumProcTest 2 18 17964
getMaxPIDTest 2 19 17924
getProcInfoTes 2 20 18700
set_burst_time 2 21 18280
test_scheduler 2 22 21212
burstTimeTest 2 23 18080
console       3 24 0
$ set_burst_time 10
Burst time set to 10
$ burstTimeTest
Setting burst time value = 3
The burst time is: 3
$
```

As we can see, the burst time is set to 3. It changes from the default burst time (0) to 3 indicating that the above functions are working correctly.

PART B:

The current scheduler in xv6 is an unweighted round robin scheduler:

```
//THE OLD ROUND ROBIN SCHEDULER

// Loop over process table looking for process to run.
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
    while(!isEmpty())
    {
        p=extractMin();
        struct proc *temp = p;
        if(isEmpty())
            break;

        p=extractMin();

        if(p->state != RUNNABLE)
            continue;

        // Switch to chosen process. It is the process's job
        // to release ptable.lock and then reacquire it
        // before jumping back to us.
        c->proc = p;
        switchvm(p);
        p->state = RUNNING;
        (p->nocs)++;                //nocs increment

        swtch(&(c->scheduler), p->context);
        switchkvm();

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
        p=temp;
    }
}
```

Keeping the burst times in mind, we have implemented a ‘**Shortest Job First**’ (SJF) scheduler to replace the previously used ‘Round Robin’ scheduler. In order to do this, we had to do two things:

- Remove the preemption of the current process (yield) on every OS clock tick so the current process completely finishes first. In the given round robin scheduler, the forced preemption of the current process with every clock tick is being handled in the trap.c file. We simply remove the following lines from trap.c to fix this issue:

```
//Remove these lines

// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
// if(myproc() && myproc()->state == RUNNING &&
//     tf->trapno == T_IRQ0+IRQ_TIMER)
//     yield();
```

- Change the scheduler so that processes are executed in the increasing order of their burst times. In order to do this, we implemented a priority queue (min heap) using a simple array which sorts processes by burst time . Of course, this heap is locked. The queue at any particular time would contain all the ‘RUNNABLE’ processes on the system. When the scheduler needs to pick

the next process, it simply chooses the process at the front of the priority queue by calling `extract min`. We had to make the following changes (in `proc.c`):

- Declare priority queue:

```
struct {  
    struct spinlock lock;           //lock  
    int siz;                       //size  
    struct proc* proc[NPROC+1];  
} pqueue;                          //min heap(priority queue)
```

- `insertIntoHeap` (Inserts a given process into the priority queue):

```
//Inserts a given process into the priority queue  
void insertIntoHeap(struct proc *p)  
{  
    if(isFull())  
        return;  
  
    acquire(&pqueue.lock);  
  
    pqueue.siz++;  
    pqueue.proc[pqueue.siz]=p;  
    int curr=pqueue.siz;  
    while(curr>1 && ((pqueue.proc[curr]->burst_time)<(pqueue.proc[curr/2]->burst_time))) //sort  
    {  
        struct proc* temp=pqueue.proc[curr];  
        pqueue.proc[curr]=pqueue.proc[curr/2];  
        pqueue.proc[curr/2]=temp;  
        curr/=2;  
    }  
    release(&pqueue.lock);  
}
```

- `isEmpty` (Checks if the priority queue is empty or not):

```
//Checks if the priority queue is empty or not  
int isEmpty(){  
    acquire(&pqueue.lock);  
    if(pqueue.siz == 0){  
        release(&pqueue.lock);  
        return 1;  
    }  
    else{  
        release(&pqueue.lock);  
        return 0;  
    }  
}
```

- **isFull** (Checks if the priority queue is full or not):

```
//Checks if the priority queue is full or not
int isFull()
{
    acquire(&pqueue.lock);
    if(pqueue.siz==NPROC){
        release(&pqueue.lock);
        return 1;
    }
    else{
        release(&pqueue.lock);
        return 0;
    }
}
```

- **extractMin** (removes the process at the front of the queue and returns it):

```
//removes the process at the front of the queue and returns it
struct proc * extractMin(){

    if(isEmpty())
        return 0;

    acquire(&pqueue.lock);
    struct proc* min=pqueue.proc[1];
    if(pqueue.siz==1)
    {
        pqueue.siz=0;
        release(&pqueue.lock);
    }
    else{
        pqueue.proc[1] = pqueue.proc[pqueue.siz];
        pqueue.siz--;
        release(&pqueue.lock);

        fix(1);
    }
    return min;
}
```

- **changeKey** (Changes the burst time of a process with a given PID in the priority queue and updates the queue accordingly):

```
//Changes the burst time of a process with a given PID in the priority queue and updates the queue accordingly
void changeKey(int pid, int x){

    acquire(&pqueue.lock);

    struct proc* p;
    int curr=-1;
    for(int i=1;i<=pqueue.siz;i++){
        if(pqueue.proc[i]->pid == pid){
            p=pqueue.proc[i];
            curr=i;
            break;
        }
    }

    if(curr==-1){
        release(&pqueue.lock);
        return;
    }

    if(curr==pqueue.siz){
        pqueue.siz--;
        release(&pqueue.lock);
    }
    else{
        pqueue.proc[curr]=pqueue.proc[pqueue.siz];
        pqueue.siz--;
        release(&pqueue.lock);

        fix(curr);                // fix the order of pqueue
    }

    p->burst_time=x;
    insertIntoHeap(p);
}
}
```

- **fix** (performs Heapify on priority queue):

```
//HEAPIFY
void fix(int curr){

    acquire(&pqueue.lock);
    while(curr*2<=pqueue.siz){
        if(curr*2+1<=pqueue.siz){
            if((pqueue.proc[curr]->burst_time)<=(pqueue.proc[curr*2]->burst_time)&&(pqueue.proc[curr]->burst_time)<=(pqueue.proc[curr*2+1]->burst_time)){
                break;
            }
            else{
                if((pqueue.proc[curr*2]->burst_time)<=(pqueue.proc[curr*2+1]->burst_time)){
                    struct proc* temp=pqueue.proc[curr*2];
                    pqueue.proc[curr*2]=pqueue.proc[curr];
                    pqueue.proc[curr]=temp;
                    curr*=2;
                }
                else {
                    struct proc* temp=pqueue.proc[curr*2+1];
                    pqueue.proc[curr*2+1]=pqueue.proc[curr];
                    pqueue.proc[curr]=temp;
                    curr*=2;
                    curr++;
                }
            }
        }
        else {
            if((pqueue.proc[curr]->burst_time)<=(pqueue.proc[curr*2]->burst_time))
                break;
            else{
                struct proc* temp=pqueue.proc[curr*2];
                pqueue.proc[curr*2]=pqueue.proc[curr];
                pqueue.proc[curr]=temp;
                curr*=2;
            }
        }
    }
    release(&pqueue.lock);
}
}
```

- Change the **scheduler** so that it uses the priority queue to schedule the next process :

```
//Change the scheduler so that it uses the priority queue to schedule the next process
void scheduler(void)
{
    defaultParent.pid = -2;    //default ppid
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;){
        // Enable interrupts on this processor.
        sti();

        acquire(&ptable.lock);

        //NEW SJF SCHEDULER

        if((p = extractMin()) == 0)
        {
            release(&ptable.lock);
            continue;
        }

        if(p->state!=RUNNABLE)
        {
            release(&ptable.lock);
            continue;
        }

        c->proc = p;
        switchvm(p);

        p->state = RUNNING;
        (p->nocs)++;                //nocs increment for each scheduling

        swtch(&(c->scheduler), p->context);

        switchkvm();

        c->proc = 0; |

        release(&ptable.lock);
    }
}
```

Insert processes into the priority queue as and when their state becomes **RUNNABLE**. This happens in five functions - **yield**, **kill**, **fork**, **userinit** and **wakeup1**. The code from the **fork** function is given below. The rest of the instances are identical. The variable check is created to check if the process was already in the **RUNNABLE** state in which case it is already in the priority queue and shouldn't be inserted again:

```

385 pid = np->pid;
386
387 acquire(&ptable.lock);
388
389 short check = (np->state!=RUNNABLE);
390 np->state = RUNNABLE;
391
392 //Insert Process Into Queue while checking
393 if(check)
394     insertIntoHeap(np);
395
396 release(&ptable.lock);
397
398 return pid;

```

Insert a **yield** call into **set_burst_time**. This is because when the burst time of a process is set, its scheduling needs to be done on the basis of the new burst time. **yield** switches the state of the current process to **RUNNABLE**, inserts it into the priority queue and switches the context to the scheduler:

```

//set burst time assist function
int set_burst_timeAssist(int burst_time)
{
    struct proc *p = myproc(); // function in xv6 namely myproc() which returns the pointer to proc structure of current process.
    p->burst_time = burst_time; //set burst time
    yield(); //yield function in xv6, the process that wishes to give up the CPU after a timer interrupt.
            //yield first locks the global lock protecting the process table, before marking itself as RUNNABLE and invoking the scheduler.
            // added for sjf sheduler , not needed in round robin
    //yield switches the state of the current process to RUNNABLE, inserts it into the priority queue and switches the context to the scheduler.

    return 0;
}

```

Runtime complexity:

The runtime complexity of the scheduler is $O(\log n)$ because **extractMin** has a $O(\log n)$ time complexity and that is the dominating part of the scheduling process. The rest of the statements run in $O(1)$ time..

Corner case handling and safety:

- If the queue is empty, **extractMin** returns 0 after which the scheduler doesn't schedule any process. (See scheduler function)
- If the priority queue is full, **insertIntoHeap** rejects the new process and simply returns so no new process is inserted into the queue by removing an older process.
- When inserting a process into the priority queue, it is always checked whether the element was already in the priority queue or not. This is done by checking the state of the process prior to it becoming **RUNNABLE**. If it was already runnable, it was already in the queue.
- The priority queue functions are robust and don't lead to situations where a segmentation fault would occur.

- Although the priority queue is expected to have only RUNNABLE processes, our scheduler checks if the process at the front is RUNNABLE or not. If not, the scheduler doesn't schedule this process. If the process somehow changes state, this measure protects the operating system.
- When ZOMBIE child processes are freed, the priority queue is also checked for the processes and these processes are removed from there too using `changeKey` and `extractMin`.
- In order to maintain data consistency, a lock is always used when accessing *pqueue*. This lock is created specially for *pqueue* and is initialized in `pinit`:

```
//remove corner cases -- extra lock for pqqueue|
void pinit(void)
{
    initlock(&ptable.lock, "ptable");
    initlock(&pqueue.lock, "pqueue");
}
```

Testing

Testing was done to make sure our new scheduler is robust and works correctly in every case. In order to do this, we forked multiple processes and gave them different burst times. Roughly half of the processes are CPU bound processes and the other half are I/O bound processes.

- CPU bound processes consist of loops that run for many iterations (10^8). An interesting fact we learned was that the loops are ignored by the compiler if the information computed in the loop isn't used later. Hence, we had to use the information computed in the loop later.
- I/O bound processes were simulated by calling `sleep(1)` 100 times. 'sleep' changes the state of the current process to sleeping for a given number of clock ticks, which is something that happens when processes wait for user input. When one I/O bound process is put to sleep, the context is switched to another process that is decided by the scheduler.

We first made a program called `test_scheduler` to check if the SJF scheduler is working according to burst times. It takes an argument equal to the number of forked processes and returns stats of each executed process:


```
harshul@harshul: ~/xv6-public
getMaxPIDTest 2 19 17924
getProcInfoTes 2 20 18700
set_burst_time 2 21 18280
test_scheduler 2 22 21212
burstTimeTest 2 23 18080
console 3 24 0
$ test_scheduler 20
```

PID	Type	Burst Time	Context Switches
22	CPU	2	2
12	CPU	3	2
8	CPU	6	2
14	CPU	6	2
18	CPU	9	2
20	CPU	14	2
16	CPU	14	2
24	CPU	17	2
10	CPU	20	2
6	CPU	20	2
7	I/O	1	102
9	I/O	2	102
11	I/O	6	102
21	I/O	9	102
5	I/O	13	102
23	I/O	15	102
13	I/O	17	102
19	I/O	18	102
17	I/O	19	102
15	I/O	20	102

```
$
```

As you can see, all CPU bound processes and I/O bound processes are sorted by their burst times and CPU bound processes finish first. The CPU bound processes finish first because I/O bound processes are blocked by the 'sleep' system call. Since the processes are sorted by burst time, we can say that the SJF scheduler is working perfectly.

The context switches are also as expected. In the case of CPU bound processes, first the process is switched in after which set_burst_time is called because of which the process is yielded and the next process is brought in. Finally, when the earlier process is chosen again by the scheduler, it finishes. In the case of I/O bound processes, they are also put to sleep 100 times. Hence, the processes have 100 additional context switches (they are brought back in 100 more times).

This is in contrast to the default Round Robin scheduler. We created two special programs called cpuProcTester and ioProcTester to compare the Round Robin scheduler with the

SJF Scheduler . cpuProcTester runs CPU processes to simplify the comparison. ioProcTester only runs I/O bound processes:

This is the outputs with the **Round Robin scheduler**:

```
$ cpuProcTester 4
```

PID	Type	Burst Time	Context Switches
4	CPU	13	17
5	CPU	20	18
6	CPU	1	18
7	CPU	6	19

```
$
```

```
$ ioProcTester 4
```

PID	Type	Burst Time	Context Switches
12	I/O	13	22
13	I/O	20	22
14	I/O	1	22
15	I/O	6	22

```
$
```

This is the output with the SJF scheduler (cpuProcTester):

```
$ ioProcTester 4
```

PID	Type	Burst Time	Context Switches
29	I/O	1	22
30	I/O	6	22
27	I/O	13	22
28	I/O	20	22

```
$ cpuProcTester 4
```

PID	Type	Burst Time	Context Switches
34	CPU	1	2
35	CPU	6	2
32	CPU	13	2
33	CPU	20	2

As you can see, since the Round Robin scheduler uses an FCFS queue, the order of execution is highly related to the PID of the process whereas in the SJF scheduler, the scheduling is happening by the burst times. Also, the number of context switches in the RR scheduler is very high. This is because of forced preemption on every clock tick.

Some notes regarding testing:

- Burst times are generated by the random number generator created in the file random.c. We made this file a user library.
 - IMPORTANT: We removed wc and mkdir from the Makefile because we couldn't have more than 20 user programs in UPROGS. The OS wasn't compiling with a large number of user programs due to some virtual hard drive issue.
 - set_burst_time yields the current process as mentioned in the above point. This is so that the new burst times are used in scheduling.
-

For hybrid scheduling algorithm:

We implemented a scheduler that behaves as a hybrid between Round Robin and SJF schedulers. We did this by modifying the trap.c, proc.c and defs.h files.

We first create the logic for setting up a time quantum. In order to do this, we declare an extern int variable in defs.h so it can be initialized in proc.c. This variable is called *quant*. It is initialized to 1000 by default as the burst times are between 1 and 1000. The assignment asks us to make the time quantum equal to the burst time of the process with the shortest burst time in the priority queue. However, the default burst time is zero. If the burst time of a process is zero, we do not know how long the process will run and hence we assign a default burst time of zero. Therefore, the quant variable is modified in the set_burst_time function. If the burst time to be set is less than the quant value, quant's value is set to burst time.

Next, we create another priority queue called *pqueue2* and the corresponding functions for this priority queue. Functions like insertIntoHeap, fix, extractMax, etc were created corresponding to pqueue. The corresponding functions for pqueue2 are insertIntoHeap2, extractMax2, fix2 etc. The functions are the same as the original ones except they modify pqueue2 instead of pqueue so you can refer to the previous section to get details about the functions.

```
int quant = 1000;          //quant

struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;

struct {
    struct spinlock lock;
    int siz;
    struct proc* proc[NPROC+1];
} pqueue;

struct {
    struct spinlock lock;
    int siz;
    struct proc* proc[NPROC+1];
} pqueue2;                //priority queue 2
```

We then modified the clock tick interrupt handler in the trap.c file. At every clock tick, we increment the running time (added parameter in struct proc - rt which is initialized to zero when proc is created in allocproc) in the current process (myproc()). By default, processes have burst time zero. If the burst time of a process isn't manually set, we don't want to kill the process as soon as its first clock tick is observed since that may seriously affect the functioning of the OS.

So, before we check if the current running time is equal to the burst time of the process, we check if the burst time is zero. We only make the equivalence check between rt and burst_time if the burst_time value is non zero. If the equivalence is true, we exit() the current process. Otherwise, we make the next check - check if the running time of the current process is divisible by the time quantum, quant. If true, we preempt the current process and insert into the other priority queue pqueue2.

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
    tf->trapno == T_IRQ0+IRQ_TIMER)
{
    (myproc()->rt)++;
    if(myproc()->burst_time != 0){
        if(myproc()->burst_time == myproc()->rt)
            exit();
    }
    if((myproc()->rt)%quant == 0)
        new_yield();
}

void new_yield(void){
    acquire(&ptable.lock);

    myproc()->state = RUNNABLE;

    insertIntoHeap2(myproc());

    sched();
    release(&ptable.lock);
}
```

Next, we modified the scheduler. Basically, in the new scheduler while deciding on which process to run next, we check if the original priority queue, **pqueue** is empty or not. If not, we **extractMin** from **pqueue** run the extracted process. If **pqueue** is empty, we remove all processes from **pqueue2** and insert them into **pqueue** (The processes that were preempted because of time quantum) and then we normally pick the front element from **pqueue** using **extractMin**.

```
void
scheduler(void)
{
    defaultParent.pid = -2;
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;){
        // Enable interrupts on this processor.
        sti();
        acquire(&ptable.lock);

        if(isEmpty()){
            if(isEmpty2()){
                goto label;
            }
            while(!isEmpty2()){
                if((p = extractMin2()) == 0){release(&ptable.lock);break;}
                insertIntoHeap(p);
            }
        }
        label:
            if((p = extractMin()) == 0){release(&ptable.lock);continue;}
            if(p->state!=RUNNABLE)
                {release(&ptable.lock);continue;}
            c->proc = p;
            switchvm(p);
            p->state = RUNNING;
            (p->nocs)++;
            swtch(&(c->scheduler), p->context);
            switchkvm();
            c->proc = 0;

            release(&ptable.lock);
    }
}
```

Testing:

The final part is the testing which is done using three user programs - test_scheduler, cpuProcTester and ioProcTester. In the code files corresponding to these programs, we did the following:

- In test_scheduler.c, half of the forked processes are I/O bound processes and the other half are CPU bound processes. All forked processes are assigned a random burst time between 1 and 1000. CPU bound processes consist of loops that run for 10^9 iterations and I/O bound processes consist of calling sleep(1) 10 times.
- In cpuProcTester.c, we just made all the forked processes CPU bound. Burst times are assigned randomly between 1 and 1000. A loop of 10^9 iterations is run (takes some time - approximately 200 ticks).
- In ioProcTester.c, every forked process is I/O bound. In order to simulate I/O, we are simply calling sleep(1) 10 times. Burst times are assigned at random with values between 1 and 1000.

The output obtained from the above tests are shown below.

```
harshul@harshul: ~/xv6-public
$ cpuProcTester 20
```

PID	Type	Burst Time	Context Switches
45	CPU	738	12
35	CPU	822	12
46	CPU	846	12
41	CPU	889	12
39	CPU	932	12
37	CPU	962	12
32	CPU	985	12
28	CPU	999	12
33	CPU	252	13
30	CPU	264	13
36	CPU	270	13
43	CPU	411	13
40	CPU	443	13
27	CPU	635	13
38	CPU	675	13
42	CPU	677	13

```
$
```

```
harshul@harshul: ~/xv6-public
$ ioProcTester 20
```

PID	Type	Burst Time	Context Switches
50	I/O	21	12
65	I/O	67	12
52	I/O	71	12
55	I/O	126	12
54	I/O	252	12
51	I/O	264	12
57	I/O	270	12
64	I/O	411	12
61	I/O	443	12
48	I/O	635	12
59	I/O	675	12
63	I/O	677	12
66	I/O	738	12
56	I/O	822	12
67	I/O	846	12
62	I/O	889	12
60	I/O	932	12
58	I/O	962	12
53	I/O	985	12
49	I/O	999	12

```
$
```

```
harshul@harshul: ~/xv6-public
$ test_scheduler 30
```

PID	Type	Burst Time	Context Switches
71	I/O	21	12
73	I/O	71	12
95	I/O	145	12
89	I/O	179	12
75	I/O	252	12
85	I/O	411	12
97	I/O	423	12
69	I/O	635	12
87	I/O	738	12
91	I/O	805	12
77	I/O	822	12
83	I/O	889	12
93	I/O	914	12
81	I/O	932	12
79	I/O	962	12
72	CPU	264	12
78	CPU	270	12
92	CPU	418	12
82	CPU	443	12
96	CPU	457	12
98	CPU	571	13
90	CPU	624	13
80	CPU	675	13
84	CPU	677	13
88	CPU	846	13
74	CPU	985	13
70	CPU	999	13

```
$
```


Results:

Observations:

- Not all CPU bound processes were actually completed. This is because some of them had a burst time lower than their actual execution time and were killed before they printed anything. This proves that when the `rt` value of the process becomes equal to the `burst_time` value of that process, the process is actually being excited.
 - There was a considerably higher number of context switches with this new hybrid scheduling in the CPU bound processes. This is because the CPU processes are being preempted every quant clock ticks.
 - The I/O bound processes weren't affected by the preemption and they behaved just like they did in SJF scheduling. This is because they are sleeping most of the time. Their actual execution time is very low. The likelihood of them experiencing quant clock ticks is very low since quant is expected to be a 2-3 digit number (burst times are chosen randomly between 1 and 1000 which affects quant). Hence, they didn't get forcefully preempted at regular intervals. They just went to sleep repeatedly. Hence, their number of context switches remained the same as they would be in SJF scheduling.
 - Note: When we are using a combination of CPU and I/O bound processes, some CPU bound processes are getting preempted more than others since I/O bound processes are returning from the SLEEPING state and forcing the currently running CPU bound processes to get preempted. This leads to out of order execution of some CPU bound processes.
-