Group no C38
Vatsal Gupta 200101105
Sweeya Reddy 200101079
Pranshu Kandoi 200101086

Assignment-2 OS344

PART A

Here following system calls need to be implemented **getNumProc(), getMaxPid(), getProcInfo(pid, &processInfo), set_burst_time(n)** and **get_burst_time().** Note defs and processInfo.h files are provided later containing the processInfo data structure. Also in params.h the number of cpu is also set to 1. Also note that to get current process info and pid 2 system calls are also added which will be used later in the assignment.

So for the purpose of implementing these system calls following files are changed:

1. **We will create system calls** and make their corresponding links. Indexes for the system calls that we will implement are defined in **syscall.h.** Thus it contains the mapping from system call name to system call number.

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_getNumProc 22
#define SYS_getMaxPID 23
#define SYS_getProcInfo 24
#define SYS_set_burst_time 25
#define SYS_get_burst_time 26
```

2. **syscall.c** contains helper functions to parse call arguments and pointers to the actual system call implementations.

```
133   [SYS_close]   sys_close,
134   [SYS_getNumProc] sys_getNumProc,
135   [SYS_getMaxPID] sys_getMaxPID,
136   [SYS_getProcInfo] sys_getProcInfo,
137   [SYS_set_burst_time] sys_set_burst_time,
138   [SYS_get_burst_time] sys_get_burst_time,
139   };
140
```

```
104   extern int sys_write(void);
105   extern int sys_uptime(void);
106   extern int sys_getNumProc(void);
107   extern int sys_getMaxPID(void);
108   extern int sys_getProcInfo(void);
109   extern int sys_set_burst_time(void);
110   extern int sys_get_burst_time(void);
111
```

3. **usys.S** contains a list of system calls exported by the kernel.

```
31    SYSCALL(uptime)
32    SYSCALL(getNumProc)
33    SYSCALL(getMaxPID)
34    SYSCALL(getProcInfo)
35    SYSCALL(set_burst_time)
36    SYSCALL(get_burst_time)
37
```

4. **Proc.h** contains the struct proc structure. In this number of context fields, burst time and running time fields have been added. Also system call has been defined in proc.h.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int nocs;                    //Number of context switches
  int burst_time;              //Burst time duhhh
  int rt;                      //running time
};

int getNumProc_system(void);
int getMaxPID_system(void);
struct processInfo getProcInfo_system(int pid);
int set_burst_time_system(int burst_time);
int get_burst_time_system();

```

5. **user.h** contains system call definitions in xv6.

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

```

6. In makefile, in **EXTRA** the following test file name is added and also in **UPROGS** for linking our test files and corresponding system calls.

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _getNumProcTest\
    _getMaxPIDTest\
    _getProcInfoTest\
    _burstTimeTest\
    _ioProcTester\
    _cpuProcTester\
    _test_scheduler\
```

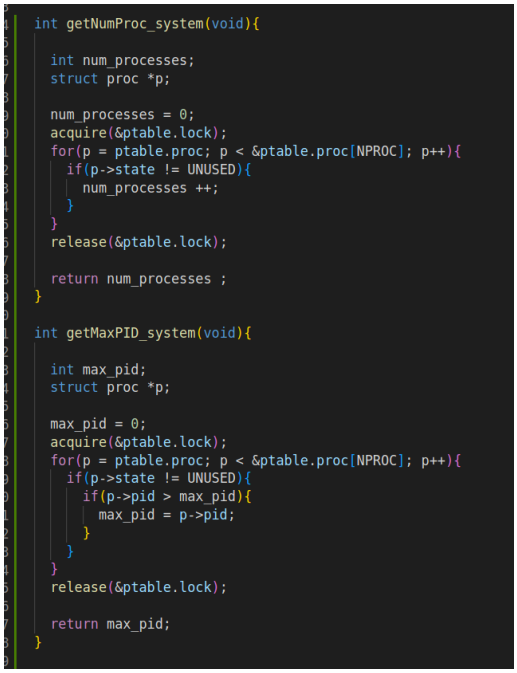
```
ll.c\
bie.c getNumProcTest.c getMaxPIDTest.c getProcInfoTest.c burstTimeTest.c test_scheduler.c ioProcTester.c cpuProcTester.c\
```

7. **sysproc.c** contains the implementations of process related system calls.

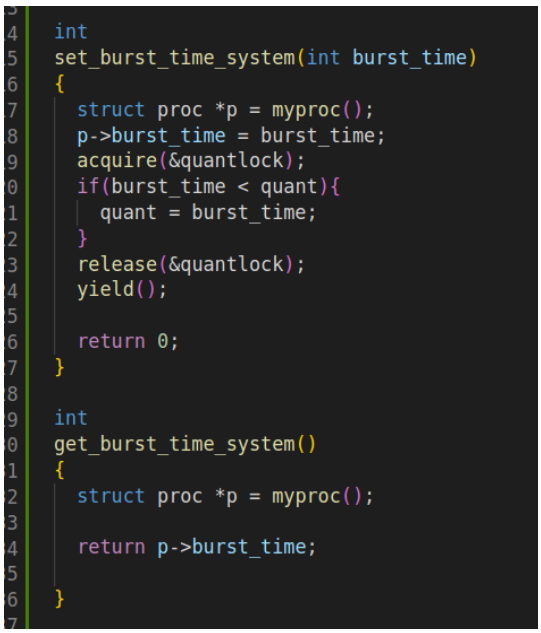
```
94  int
95  sys_getNumProc(void)
96  {
97    return getNumProc_system();
98  }
99
100 int
101 sys_getMaxPID(void)
102 {
103   return getMaxPID_system();
104 }
105
106 int
107 sys_getProcInfo(void){
108   int pid;
109   struct processInfo *info;
110   argptr(0,(void *)&pid, sizeof(pid));
111   argptr(1,(void *)&info, sizeof(info));
112
113   struct processInfo tempInfo = getProcInfo_system(pid);
114
115   if(tempInfo.ppid == -1) return -1;
116   info->ppid = tempInfo.ppid;
117   info->psize = tempInfo.psize;
118   info->numberContextSwitches = tempInfo.numberContextSwitches;
119   return 0;
120 }
121
```

```
int
sys_set_burst_time(void)
{
  int burst_time;
  argptr(0,(void *)&burst_time, sizeof(burst_time));

  return set_burst_time_system(burst_time);
}

int
sys_get_burst_time(void)
{
  return get_burst_time_system();
}
```

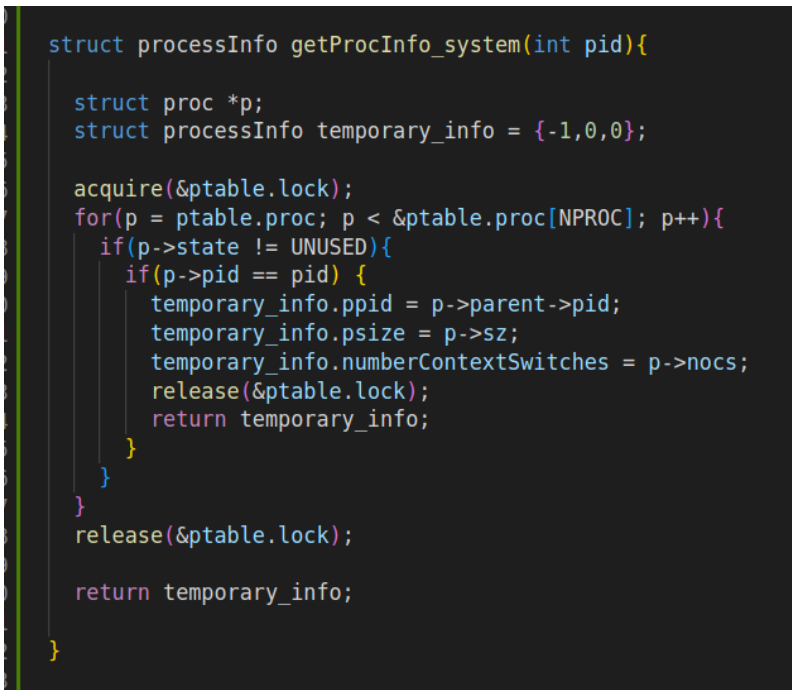
8. **proc.c,** the assist functions used by system calls are defined in the proc.c



```c
int getNumProc_system(void){

  int num_processes;
  struct proc *p;

  num_processes = 0;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != UNUSED){
      num_processes ++;
    }
  }
  release(&ptable.lock);

  return num_processes ;
}

int getMaxPID_system(void){

  int max_pid;
  struct proc *p;

  max_pid = 0;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != UNUSED){
      if(p->pid > max_pid){
        max_pid = p->pid;
      }
    }
  }
  release(&ptable.lock);

  return max_pid;
}
```



```c
int
set_burst_time_system(int burst_time)
{
  struct proc *p = myproc();
  p->burst_time = burst_time;
  acquire(&quantlock);
  if(burst_time < quant){
    quant = burst_time;
  }
  release(&quantlock);
  yield();

  return 0;
}

int
get_burst_time_system()
{
  struct proc *p = myproc();

  return p->burst_time;

}
```



```c
struct processInfo getProcInfo_system(int pid){

  struct proc *p;
  struct processInfo temporary_info = {-1,0,0};

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != UNUSED){
      if(p->pid == pid) {
        temporary_info.ppid = p->parent->pid;
        temporary_info.psize = p->sz;
        temporary_info.numberContextSwitches = p->nocs;
        release(&ptable.lock);
        return temporary_info;
      }
    }
  }
  release(&ptable.lock);

  return temporary_info;

}
```

Part 1

For getting the number of processes and the maximum pid we will simply acquire a lock on the process table and loop through the table and for each process which is not unused we will add to the number of processes and also see if its the maximum pid process. This whole thing is done in the assist function in proc.c in **getNumProc_system** and **getMaxPID _system** and then in sysproc.c the following function is used. These function implementations are shown above.

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ getNumProcTest
The total number of active processes in the system is 3
$ getMaxPIDTest
The maximum PID of all active processes in the system is 4
$
```

We can note that since 2 system processes are always running thus total process running are 3 as 2 system processes are there and one the current **getNumProcTest** and also since the **getMaxPIDTest** is the process with PID 4 and is the max among all active process as right now this one is also running thus maxPID is 4.

Part 2

For implementing **getProcInfo** we are providing 2 arguments one is the pid of the required process and other is the space where we need to store that info thus similar to previous a lock is maintained on the process table and is searched for the current process if the current process is found then lock is released and the process info is saved to the location and if not found then default value is returned by the helper function in proc.c also system call in sysproc.c uses this to get the required info. Also the process parent is set to default parent and number of content switches and burst time are set to 0 by default in proc.c. Also we set the **default parent pid to -2** which helps us to determine whether the current process has a parent or not. For calculating the number of context switches we will simply **increase the value of the nocs variable when the process is being scheduled by the scheduler**.

```
Booting from Hard Disk...
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ getProcInfoTest 1
PPID: No Parent Process
Size: 16384
Number of Context Switches: 17
$ getProcInfoTest 2
PPID: 1
Size: 20480
Number of Context Switches: 22
```

We can see from the output that the process with pid 1 has no parent and the process with pid 2 has a parent with pid 1. Also the context switches are shown in the above picture.

Part 3

For this part we simply need to add a burst time field in the proc data structure and created two system call to set this value and get this value as shown above in sysproc.c and proc.c . sysproc.c take what value to be set as burst time and helper function in proc.c set that burst time. Note after burst time is set yield() is called as we need to remove the process from running as its burst time has changed thus it needs to be evaluated by the scheduling algorithm again. Also to get the current process we can simply use myproc() function.

burstTimeTest.c > ...

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

int
main(void){
    printf(1,"Burst time is set to 10\n");
    set_burst_time(10);
    printf(1, "Getting the value of burst time %d\n", get_burst_time());
    exit();
}
```

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ burstTimeTest
Burst time is set to 10
Getting the value of burst time 10
$
```

From the output we clearly see that the current process burst time is set to 10.

PART B

SJF (Shortest Job First) Scheduler

Firstly for implementing the shortest job scheduler without the round robin scheduler in trap.c the following line needs to be commented as to prevent any process to be evicted based on time quantum. Thus commenting this line will prevent forced preemption of the process and scheduling is done based on the shortest job first not round robin.

```
103  // Force process to give up CPU on clock tick.
104  // If interrupts were on while locks held, would need to check nlock.
105  if(myproc() && myproc()->state == RUNNING &&
106     tf->trapno == T_IRQ0+IRQ_TIMER){
107    yield();
108     }
109
```

For the purpose of implementing the scheduling algorithm a **min heap** is maintained using an array of all the processes whose state is RUNNABLE and proper locking is handled to add and remove processes from the queue. Thus functions are also written to handle various min heap operations. Also whenever a function is made runnable then we add this to our min heap or priority queue.

```
// //new SJF SCHEDULER
acquire(&ptable.lock);
if((p = extractMin()) == 0){
  release(&ptable.lock);
  continue;
}

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

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

p->state = RUNNING;

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

switchkvm();

c->proc = 0;
release(&ptable.lock);
```

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

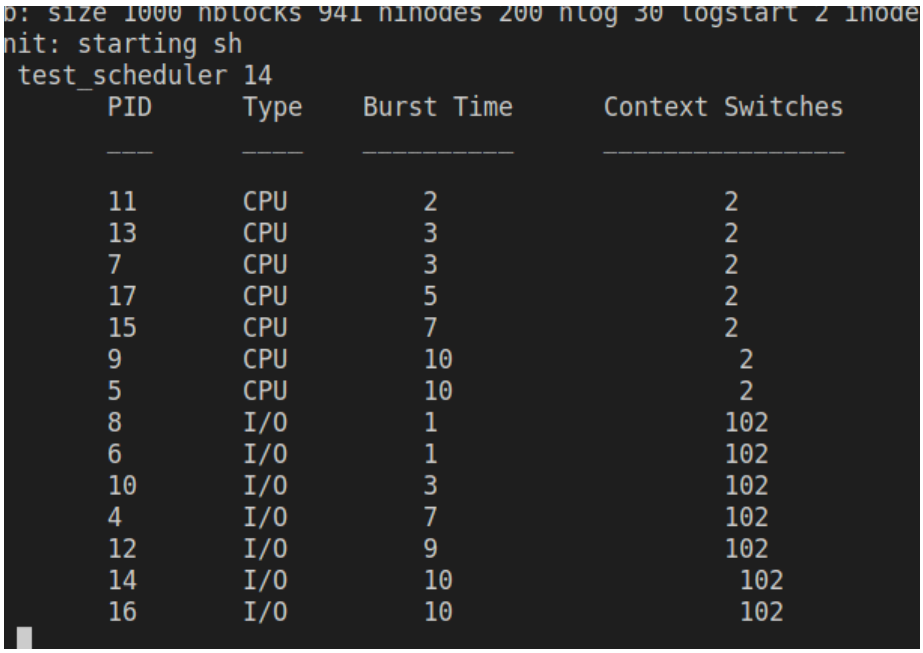
Note here firstly using extract min the minimum burst time process is extracted and then scheduled thus a shortest job first scheduler. Also note that in the pqueue structure spinlock is there, size of the queue is there and an array is used which act as a min heap.

Time complexity and corner cases

Running complexity of the algorithm is O(logn) as the extract min operation in the priority queue is of O(logn) and then the remaining operation is of order 1. Note for handling corner cases firstly if the queue is full no new process is inserted into the queue and it returns as it is. If the queue is empty then extract min returns 0 and thus the scheduler does not schedule any processes. Also if the process state is already runnable then it is already in the queue and thus it is not added in the queue. Also if a process is finished that is when it's state is ZOMBIE then also it is removed from the queue. Also when using priority queue a locking is done properly and also only those processes whose state are runnable are scheduled otherwise is not scheduled.

Testing

For testing purposes a test_scheduler.c file is created which take an integer n as input and for half of that uses for io bound processes and half of that uses as cpu bound processes. For a cpu bound process a double for loop is there and for a io bound process sleep(1) is written in for loop for 50 times. Thus calling test_scheduler gives following output



Note here firstly that the processes are scheduled based on the burst time and also on the type of the processes. Note firstly that the CPU bound processes completes first and then the IO bound processes as in IO bound processes because of sleep it takes more time to complete. Also note in CPU and also IO bound processes processes are completed based on the burst time. Also note that there are 2 context switches in case of CPU bound processes as 1st for when the process is scheduled and when its burst time is set. Note in case of IO bound processes because of 100 sleep additional 100 context switches are there.

Given below is that of the default round robin scheduler and we can see that the scheduling happens mostly on the basis PID values of the processes and also context switches in this case are very high as compared to the shortest job first.

```
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inode
init: starting sh
$ test_scheduler 14
        PID      Type    Burst Time      Context Switches
        ___      ____    __________      ________________

        5        CPU         10                  34
        7        CPU         3                  33
        9        CPU         10                  33
        11       CPU         2                  33
        15       CPU         7                  34
        13       CPU         3                  35
        17       CPU         5                  35
        4        I/O         7                  102
        6        I/O         1                  103
        8        I/O         1                  102
        10       I/O         3                  102
        12       I/O         9                  102
        14       I/O         10                  102
        16       I/O         10                  102
$
```

For another program we can see the same difference

Round robin

```
nit: starting sh
test_scheduler 14
      PID      Type    Burst Time      Context Switches
      ___      ____    __________      ________________

      7        CPU         3             73
      5        CPU         10             74
      11       CPU         2             73
      9        CPU         10             75
      15       CPU         7             73
      13       CPU         3             75
      17       CPU         5             75
      4        I/O         7                   202
      6        I/O         1                   202
      8        I/O         1                   202
      10       I/O         3                   202
      12       I/O         9                   202
      14       I/O         10                   202
      16       I/O         10                   202
```

shortest job first

```
size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodes
: starting sh
st_scheduler 14
    PID      Type    Burst Time      Context Switches
    ___      ____    __________      ________________

    11       CPU         2             2
    13       CPU         3             2
    7        CPU         3             2
    17       CPU         5             2
    15       CPU         7             2
    9        CPU         10             2
    5        CPU         10             2
    6        I/O         1                   202
    8        I/O         1                   202
    10       I/O         3                   202
    4        I/O         7                   202
    12       I/O         9                   202
    14       I/O         10                   202
    16       I/O         10                   202
```

Hybrid Scheduler

For the purpose of a hybrid scheduler we firstly need to maintain 2 priority queues. Firstly job are scheduled on the basis of the first queue in which all the runnable processes are there and then after a fixed time quantum it is preempted and put to the second queue. After the 1st queue is empty it means that the 1st round is completed and all the remaining processes in the second are again put to the first queue and again scheduled based on the shortest burst time. And thus we are able to achieve a hybrid scheduler. Also note that the time quantum variable is modified

based on the burst time, that is if the burst time is less than the time quantum its value is decreased to that value.

Finally in trap.c interrupt code need to be modified as shown below

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to ch
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();
        // yield();
  }
```

Note here rt is the running time variable keeping track of how long the process is executed and if it becomes equal to burst time then that process is completed exit command is executed. If not then new_yield is executed and the current process is put to the other queue.

Thus the scheduler code looks like

```
  // Enable interrupts on this processor.
  sti();
  acquire(&ptable.lock);
  //Hybrid scheduler
  if(isEmpty()){
    if(isEmpty2()){
      goto label;
    }
    while(!isEmpty2()){
      if((p = extractMin2()) == 0){release(&ptable.lock);break;}
      insert(p,0);
    }
  }
  label:
    if((p = extractMin()) == 0){
      release(&ptable.lock);
      continue;
    }
    if(p->state!=RUNNABLE){
      release(&ptable.lock);
      continue;
    }

    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    (p->nocs)++;

    swtch(&(c->scheduler), p->context);
    switchkvm();
    c->proc = 0;
  release(&ptable.lock);
```

Note firstly that the main queue is empty then all the remaining processes from the other queue are put into the main queue then similar to the previous shortest job first scheduler, jobs are scheduled. Other functions of the other priority queue are identical to that of the main priority queue. Also a proper locking system is used.

```
nit: starting sh
 test_scheduler 14
       PID      Type    Burst Time      Context Switches
       ___      ____    __________      ________________

       6        I/O         11                  52
       8        I/O         36                  52
       10       I/O         126                  52
       7        CPU         132             10
       13       CPU         135             10
       17       CPU         222             10
       15       CPU         338             10
       9        CPU         493             10
       5        CPU         500             10
       4        I/O         318                 52
       12       I/O         411                 52
       16       I/O         466                 52
       14       I/O         481                 52
```

From the hybrid scheduler we can clearly see that the number of content switches have increased in case of cpu bound processes as compared to shortest job scheduling as process are now been preempted more due to round robin scheduling but in case of IO bound processes since most of the time they are in sleep thus difference is not well observed and context switches are thus same as that of sjf scheduler also note that process are completed based on the burst time of the cpu and io type processes.