OS LAB 2 (Group 35)

Team Member	Roll Number	branch
Mohan Kumar	180101042	CSE
Saurabh Baranwal	180101072	CSE
Rahul Krishna	180123035	MNC
Pankaj Kumar	180123031	MNC

PART A

Q1). For adding **getNumProc**() and **getMaxPid**() system call we need to follow these process.

- We need to map the function with an integer in the **syscall.h** file.
  - #define SYS_numproc 23
  - #define SYS_maxpid 24
- Now we need to add a pointer to the system call in the **syscall**.c file.
  - [SYS_numproc] sys_numproc,
  - [SYS_maxpid] sys_maxpid,
- We only add the function prototype in **syscall**.c and we define the function implementation in a different file.
  - extern int sys_numproc(void);
  - extern int sys_maxpid(void);
- Now we will add function declaration in **sysproc**.c

```c
//numproc will return number of currently running process.
int
sys_numproc(void)
{
  return numberOfRunningProcess();
}

//maxpid will return the maximum pid of currently running processes.

int
sys_maxpid(void)
{
return getMaxPid();
}
```

  - Here function **numberOfRunningProcess**() and **getMaxPid**() is declared in **defs**.h header file.
- Add the function definition in **defs**.h file (we can't write code related to the process in the **sysproc**.c file, we have to write that code in the process file **proc**.c and then call that function from the **sysproc**.c file). Add these two lines in the **proc**.c section in **defs**.h file.
  - int numberOfRunningProcess(void);
  - int getMaxPid(void);
- Now we have to write the function details in **proc**.c file.

```c
int
getMaxPid(void)
{
  struct proc *p;
  int mx = 0;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
     if(p->pid > mx && p->state != UNUSED){
        mx = p->pid;
        //whenever state of process p is active than we
     }
  }

  release(&ptable.lock);

 return mx;
}
```

```c
int
numberOfRunningProcess(void)
{
  struct proc *p;
  int count = 0;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
     if(p->state != UNUSED){
        count++;
        //whenever state of process p is active than w
     }
  }

  release(&ptable.lock);

 return count;
}
```

- Add the interface of the system call in the usys.S file.
  - `SYSCALL(numproc)`
  - `SYSCALL(maxpid)`
- Now user.h file is need to be edited.
  - `int numproc(void);`
  - `int maxpid(void);`
- Now create **getNumProc**.c file and **getMaxPid**.c file.

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

int
main(int argc, char *argv[])
{
  printf(1,"Currently %d processes are running in your system\n",numproc());
  exit();
}
```

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

int
main(int argc, char *argv[])
{
  printf(1,"Out of all of the processes running currently %d is the maximum process ID\n",maxpid());
  exit();
}
```

- Now make the appropriate change in Makefile for user call and than you are all set for this system call.

Q2). As per given in the question include the header file processInfo.h in **user**.h and **proc**.c file.

- Now in the proc.c file create a function **getProcInfo**(int pid, struct processInfo *process_info) which returns int value.
- Here is the implementation of the function getProcInfo();
- We have added an extra variable in **Switch_count** in the proc structure.
- Since all of the process starts running
  In the procalloc() function, we have Assigned **switch_count** = 0, in the procAlloc() function.
  p->switch_count = 0;
- Now each time scheduler has been Called, we are incrementing the **switch_count** value by 1.
  p->switch_count = p->switch_count + 1;

```c
int
getProcInfo(int pid,struct processInfo *process_info){
  struct proc *p;
  int flag = 0;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
     if(p->pid == pid){
        flag = 1;
        process_info->ppid = p->parent->pid;
        process_info->psize = p->sz;
        process_info->numberContextSwitches = p->switch_count;
        break;
     }
  }
  release(&ptable.lock);
  if(flag == 0){
        return -1;
  }else{
        return 0;
  }
}
```

Q3). In the file proc.c create a function set_burst_time() and get_burst_time(). Now we have to add a new variable burst_time in the proc structure.

```c
int set_burst_time(int pid, int n)
{
  struct proc *p;
  int flag = 0;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
     if(p->pid == pid){
        flag = 1;
        p->burst_time = n;
        break;
     }
  }
  release(&ptable.lock);
  if(flag == 0){
        return -1;
  }else{
        return 0;
  }
}
```

```c
int get_burst_time(int pid)
{
  struct proc *p;
  int flag = 0;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
     if(p->pid == pid){
        return p->burst_time;
     }
  }
  release(&ptable.lock);
  return -1;
}
```

PART B

Scheduler function (round-robin + SJF scheduler):-

WE have modified the scheduler function and added SJF (short job first) algorithm. In the scheduler SJF function we are choosing the job having shortest burst time. We have also created the set_burst_time function to manually set the burst time of the process.

The time complexity of modified scheduler to pick a job having minimum burst time is O(N). For each process it is taking O (N) time. Below is the snapshot of the modified scheduler function.

```c
highP = p;
for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
  if(p1->state != RUNNABLE){
          continue;
  }
  if( p1->burst_time < highP->burst_time){
          highP = p1;
  }
}
```

Above is the extra code we have added in the round robin algorithm to choose a process having small burst time.

Set_burst_time:

```
int set_btime(int pid, int n)
{
  struct proc *p;
  int flag = 0;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
     if(p->pid == pid){
        flag = 1;
        p->burst_time = n;
        break;
     }
  }
  release(&ptable.lock);
  yield();
  if(flag == 0){
        return -1;
  }else{
        return 0;
  }
  return 0;
}
```

Yield() function is being called after the execution of set_burst_time to handle some corner cases. Suppose if a process is running having burst_time high now in the mean time if a process from the witing queue had changed its burst_time to lower value from that of running than the new process with lower burst time must execute before one which is running.

Get Burst time:-

```
int get_btime(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
     if(p->pid == pid){
        release(&ptable.lock);
        return p->burst_time;
     }
  }
  release(&ptable.lock);
  return -1;
}
```

We have also done some modification in the allocproc() function (when the process is allocated for the first time). We set the value of burst_time to be 10 as a default value.

```c
sp -= sizeof *p->context;
p->context = (struct context*)sp;
memset(p->context, 0, sizeof *p->context);
p->context->eip = (uint)forkret;
p->switch_count = 0;
p->burst_time = 10;   //default burst time is set to 10;
return p;
```

Below is the modification in the exec.c file where are assigning burst_time = 7 as we need to give some priority to the child process and hence the burst_time is set to be lower than parent process.

```c
curproc->tf->esp = sp;
curproc->burst_time = 7;      //changing the burst time of child process to be 5
switchuvm(curproc);
freevm(oldpgdir);
return 0;
```

In the robin algorithm after completion of time slice it results in a interrupt which later on call yield() in trap.c. We don’t want our process to be primitive so we need to remove that

```c
// 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();
```

Below is the test_scheduler

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
int main(int argc, char *argv[])
{
     printf(1,"==========================test1==============================\n");
     int pid=-1;
     for(int k=0;k<5;k++){
      pid=fork();
      if(pid<0){
        printf(1, "%d failed in fork!\n", getpid());
        exit();
      }
      else if(pid==0)
      {
        set_btime(getpid(),50-5*k);
        for (int ind = 0; ind < 10; ind++)
        {
          int i = 0;
          while (i < 1000000) i++;
        }
        exit();
      }
     }

     for(int k=0;k<5;k++)
     {
      pid=wait();
      if(pid!=-1){
                printf(1,"\n=====================Process Id: %d completed=================================\n\n", pid,get_btime(pid));
                printproc();
        }
     }

  exit();
}
```

Here is the observation after running the test_scheduler on qemu

Test case:- In this case we are creating 5 fork() (child processes) and also making the parent process to wait for the execution of child processes. Now In each child process we are using some dummy arithmetics for consuming cpu time.
Output: order of process are 6->7->5->8->9 which is not the same order of process creation.

```
--------------------Now we are running test1 -------------------------------------------

--------------------Process Completed with PID :6 and burst time : -1 -------------------------------------------

name     pid     state   burst_time
init     1       SLEEPING        7
 sh      2       SLEEPING        7
 test_scheduler          4       RUNNING         7
 test_scheduler          5       RUNNING         19
 test_scheduler          7       RUNNABLE        100
 test_scheduler          8       RUNNABLE        100
 test_scheduler          9       RUNNABLE        100

--------------------Process Completed with PID :7 and burst time : -1 -------------------------------------------

name     pid     state   burst_time
init     1       SLEEPING        7
 sh      2       SLEEPING        7
 test_scheduler          4       RUNNING         7
 test_scheduler          8       RUNNING         16
 test_scheduler          9       RUNNABLE        100

--------------------Process Completed with PID :5 and burst time : -1 -------------------------------------------

name     pid     state   burst_time
init     1       SLEEPING        7
 sh      2       SLEEPING        7
 test_scheduler          4       RUNNING         7
 test_scheduler          8       RUNNING         16
 test_scheduler          9       RUNNABLE        100

--------------------Process Completed with PID :8 and burst time : -1 -------------------------------------------

name     pid     state   burst_time
init     1       SLEEPING        7
 sh      2       SLEEPING        7
 test_scheduler          4       RUNNING         7

--------------------Process Completed with PID :9 and burst time : -1 -------------------------------------------

name     pid     state   burst_time
init     1       SLEEPING        7
 sh      2       SLEEPING        7
 test_scheduler          4       RUNNING         7
```

Below is the default round-robin result. Here each child processes are created and executed at the same time. It is executed in the order. As expected it is giving order as 4->6->7->8.

```
======================Process Id: 5 completed==================================

name     pid     state   burst_time
init     1       SLEEPING        7
 sh      2       SLEEPING        7
 test_scheduler          3       RUNNING         7

======================Process Id: 4 completed==================================

name     pid     state   burst_time
init     1       SLEEPING        7
 sh      2       SLEEPING        7
 test_scheduler          3       RUNNING         7

======================Process Id: 6 completed==================================

name     pid     state   burst_time
init     1       SLEEPING        7
 sh      2       SLEEPING        7
 test_scheduler          3       RUNNING         7

======================Process Id: 7 completed==================================

name     pid     state   burst_time
init     1       SLEEPING        7
 sh      2       SLEEPING        7
 test_scheduler          3       RUNNING         7

======================Process Id: 8 completed==================================

name     pid     state   burst_time
init     1       SLEEPING        7
 sh      2       SLEEPING        7
 test_scheduler          3       RUNNING         7
 $ test_scheduler
```