CS 344: OPERATING SYSTEMS LAB-2

GROUP NO: 14

GROUP MEMBERS:
NAMAN GOYAL (180123029)
KARTIKEYA KUMAR GUPTA (180123020)
DHAWAL BADI (180101020)
RATHOD VIJAY MAHENDRA (180123037)

GitHub-Link for Patch files

EXERCISE A

1.) Implementing a system call for the total number of processes and maxpid among all PID's.

```
int main()
{
  printf(1 , "Number of Processes are %d\n" , getnumproc());
  exit();
}
```

- Above is the user-program `getNumProc.c` to call the system call `getnumproc()` that will return the number of currently running processes.

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

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
      if(p->state != UNUSED)
          count++;
  }

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

- This is the system call named `getnumproc()` defined in file *proc.c*. A locking discipline is followed by first acquiring the lock before accessing this structure. Traversing the ptable and maintaining counter variable count to count the number of active processes. If a current process state is *UNUSED (active process)* increment the value of the count variable. The lock is released before returning the count variable.

- Navigate to the xv6 directory, then run **make clean** command, thereafter **make** command, then **make qemu** command. A QEMU terminal till init is shown. Then we run the file named *getNumProc* whose code is shown in the first figure. The following is the generated output :

```
xv6 — qemu-system-i386
getMaxPid       2 19 14676
getProcInfo     2 20 15464
test_a3         2 21 14896
test_b1         2 22 18472
testcase_A      2 23 18384
testcase_B      2 24 17844
testcase_C      2 25 18452
console         3 26 0
Exit Process with ID: 4

$ getNumProc
Name = init
Id = 1
Burst time = 0
Name = sh
Id = 2
Burst time = 0
Name = getNumProc
Id = 5
Burst time = 0
Number of Processes are 3
Exit Process with ID: 5
```

- The output of the `getNumProc()` syscall is 3 as we can see above. Since there are 3 processes currently running namely the *parent* process, *shell*, and the above *getNum* process.

```c
int main()
{
  int maxpid = getmaxpid();
  if(maxpid!=-1)
  printf(1 , "Maximum PID among all PID's is %d\n" , maxpid);
  else
  printf(1,"No running process\n");
  exit();
}
```

- Above is the user-defined new file named get*MaxPid.c* to call `getmaxpid()` system call which returns the maximum PID among all the currently active processes.

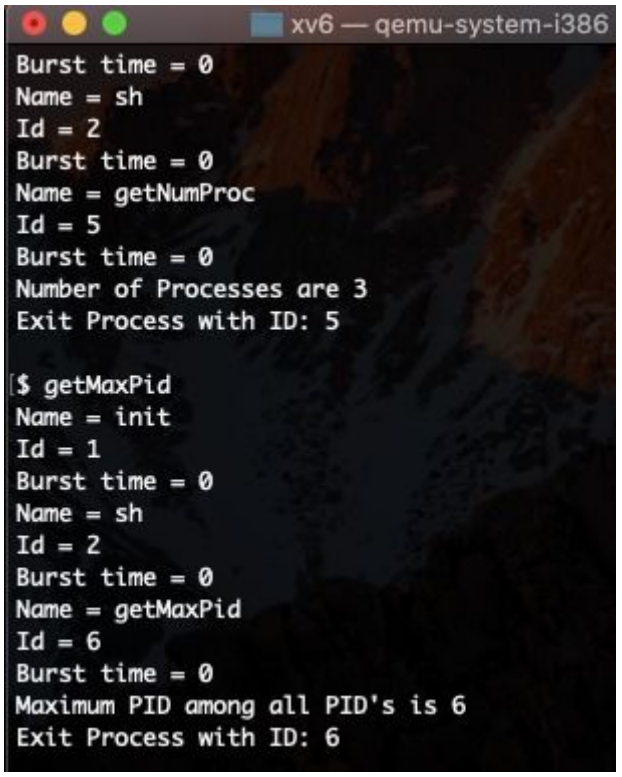
```c
int getmaxpid(void){
  struct proc *p;
  int maxpid = -1;

  acquire(&ptable.lock);

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

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

- This is the system call named `sys_getMaxPid()` defined in *sysproc.c*. Traversing all PID's through ptable and maintaining a max counter (here *ans*) and calculating max at each iteration like if the current ans variable has value less than the current process's PID then change the ans variable to the current PID else no change.
- Navigate to the xv6 directory, then run **make clean** command, thereafter **make** command, then **make qemu** command. A QEMU terminal till init is shown. Then we run
  $ `getMaxPid`
- The following output is generated from the QEMU Emulator:



- The output for the above **`getMaxPid()`** call is 6. Since the process calling the *MaxPid* function is having the PID = 5. The *init* is for the parent process, *sh* is for shell and *getmaxPid* is our implemented process. *Init* process has PID = 1 , *sh* process has PID = 2 , *getMaxPid* has PID = 6.

2.)

Makefile × sysproc.c × proc.h × getMaxPid.c × proc.c × init.c × test_b1.c × test_a3.c × getNumProc.c × getProcInfo.c ×

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
int main(int argc,char *argv[])
{
  int pid = 0;
  if(argc<2){
    printf(1,"Please provide PID: getProcInfo <PID>\n");
    exit();
  }
  struct processInfo *processinfo = (struct processInfo*)malloc(sizeof(struct processInfo));
  pid = atoi(argv[1]);
  int flag = getprocinfo(pid,processinfo);
  if(flag!=-1){
  printf(1,"Parent ID of process with ID %d is %d\n",pid ,processinfo->ppid);
  printf(1,"Size of process with ID %d is %d bytes\n",pid ,processinfo->psize);
  printf(1,"Number of context switches for process with ID %d is %d\n",pid ,processinfo->numberContextSwitches);
  }
  else
  printf(1,"No running process with id %d\n",pid);
  exit();
}
```

Above is the user program for System Call `getProcInfo(PID,&processInfo)`.

- *Argc* and *Argv* denotes the number of arguments provided in the kernel and arguments provided respectively,
- A struct pointer (*processinfo*) is declared to hold the process information.
- *PID* and *processinfo* is passed as a parameter to function `getprocessinfo()`
- `getprocessinfo()` returns -1 if the process with supplied PID is not found and 0 if the process is found. `getprocessinfo() is shown below:`

```c
int getprocinfo(int pid,struct processInfo *processinfo){

  struct proc *p;
  int flag = -1;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
      if(p->state != UNUSED&& p->pid==pid){
          flag=0;
          processinfo->ppid = -1;
          if(p->parent!=0){
            processinfo->ppid = p->parent->pid;
          }
          processinfo->psize = p->sz;
          processinfo->numberContextSwitches = p->numberOfContextSwitches;
      }
  }
  release(&ptable.lock);
  return flag;

}
```

Above is the function `getprocinfo()` from file *proc.c*

- A locking discipline is followed as ptable lock is acquired before accessing the structure and then released.
- Traverse the ptable contents and look for the process with the supplied PID.
- If the process with PID equal to the supplied PID is found a check is performed to look for the parent PID. If the parent is not present ppid is assigned value -1 else ppid is assigned PID of the parent process.
- The size of the process is stored by *struct proc* only.
- The number of Context Switches is evaluated by adding an integer variable in *struct proc* in file *proc.h* is shown below:

```
// Per-process state
struct proc {

  int numberOfContextSwitches; // Counter of Context Switches for this process
  int burst_time;              // Burst Time of this Process
  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)
};
```

- When a process is initiated for the first time in `allocproc()the variable numberofContextSwitches` is set to 0.

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->numberOfContextSwitches=0;
  p->burst_time = 0;
  release(&ptable.lock);
```

- In `scheduler()` we increment the value of `numberofContextSwitches` as a process is done running and the scheduler is switched with another process if there's any context switching takes place.

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

- In *sysproc.c* `argint` and `argptr` are used to send and receive data between the userspace program and proc.c(Kernel space).

```
int
sys_getprocinfo(int pid,struct processInfo *processinfo){

  argint(0,&pid);
  argptr(1,(char**)&processinfo,sizeof(processinfo));
  return getprocinfo(pid,processinfo);

}
```

`$ getProcInfo`

- Case-I: If PID is not provided, the kernel would ask the user to give PID as input.
- Case-II: If a PID is provided which doesn't match with the PID of an ACTIVE process system call return -1 and shown on the QEMU Emulator.
- Case-III: If the PID supplied by the user matches with an Active process PID the number of Context Switches is returned by a system call and printed on QEMU Emulator.

```
$ getProcInfo
-1, Please provide PID: getProcInfo <PID>
$ getProcInfo 45
-1, No running process with id 45
$ getProcInfo 2
Parent ID of process with ID 2 is 1
Size of process with ID 2 is 16384 bytes
Number of context switches for process with ID 2 is 18
```

3.)

Makefile | sysproc.c | proc.h | getMaxPid.c | proc.c | init.c | test_b1.c | test_a3.c

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

int main(int argc,char* argv[])
{
    if(argc<2){
        printf(1,"Please provide Burst Time for process: test_a3 <BURST_TIME>\n");
        exit();
    }
    int burst_time = atoi(argv[1]);
    setbursttime(burst_time);
    printf(1,"Burst Time of the process is set to: %d\n",getbursttime());
    exit();

}
```

Above user program is for system call set_burst_time(n) and get_burst_time()

- Kernel asks for the ***burst_time*** from the user and then calls the function `setbursttime(int)`.
- Burst_time variable is added to *struct proc* and initialized to 0 in *allocproc()* function.
- *setbursttime(int n)* sets the ***burst_time*** to the value given by the user.

```c
int getbursttime(){
  return myproc()->burst_time;
}
```

```c
void setbursttime(int n){
  argint(0,&n);
  myproc()->burst_time = n;
  yield();
}
```

- **myproc()** returns the currently running process. argint is used to share *burst_time* supplied by the user between the user-space program and kernel.

```
$ test_a3
Please provide Burst Time for process: test_a3 <BURST_TIME>
$ test_a3 56
Current running process PID: 4, Burst Time: 56
Burst Time of the process is set to: 56
```

- In the above example, *set_burst_time* sets *burst_time* to 56, and *get_burst_time* prints the **burst_time** of the current process.

EXERCISE B:

- By Default, xv6 uses Round-Robin Scheduling in which a time interrupt is generated at regular intervals which holds the currently running process, changes its state, and chooses a RUNNABLE process from the proc table, and starts its execution.

- Following is the code snippet in trap.c responsible for generating a time interrupt, the value of `TIRQ0+IRQ_TIMER` is 32 and as soon as clock ticker `tf->trapno` becomes equal to 32, `yield()` function is called which starts scheduling.

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

- **Shortest Job Scheduling** (SJF) is a scheduling algorithm by which the CPU schedules the shortest burst_time processes first.
- Time Complexity of SJF scheduling is O(NPROC) which is O(1) in this case(NPROC=64).
- In the case of SJF, the process having *min burst_time* is selected and executed first.
- The number of CPU used is 1 (default 8) by updating the makefile.

```
ifndef CPUS
CPUS := 1
```

- Following is the code snippet of core functioning of SJF Scheduler:

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

  //process with minimum bursttime be infinite
  int minbt = 100000000;

  // Loop over process table looking for process to run.
  acquire(&ptable.lock);
  struct proc *highP = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

    //continue to next iteration if state is not runnable
    if(p->state!=RUNNABLE)
      continue;

    //if found a process with shorter burst time than current we take that process
    if(minbt > p->burst_time){
              //updating current process
              highP = p;
              //updating current minimum burst time
              minbt = p->burst_time;
    }
    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
  }

  p = highP;

  if(p==0){
      release(&ptable.lock);
      continue;
    }
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    if(p->burst_time>1)
    cprintf("Current running process PID: %d, Burst Time: %d\n", p->pid,p->burst_time)
    swtch(&(c->scheduler), p->context);
    switchkvm();

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    p->numberOfContextSwitches++;
    c->proc = 0;
  release(&ptable.lock);
```

Explanation:

1. We initialise minimum_burst_time as a variable minbt and set it to a large value(10e^9).
2. Again a locking discipline is ensured.

3. highP : Stores the process with highest priority which is the shortest burst time process in SJF scheduling.
4. Traverse the ptable and compare the burst time for every runnable process with the value stored in integer variable minbt.
5. Case 1 : If burst_time of process is greater than minbt we ignore that process as we already have a process with shorter burst time.
6. Case 2 : If burst_Time of process is shortest among the processes encountered we set this process as highP (Process with highest priority).
7. After the traversal of processes if we didn't encounter a runnable process (ready queue is empty) we release the lock.
8. On the other hand cpu schedules the highP process and sends it from ready to running state for it's execution by setting the state of process as RUNNING.

- Exit function defined in proc.c closes the process and prints the pid of the process ended.

```
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");
  cprintf("Exit Process with ID: %d\n\n",curproc->pid);
  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }
```

TESTING:

We executed three types of test cases, both with Round-Robin as well with SJF algorithm. The code of test cases can be seen in the uploaded files.

1. testcase_A: Takes integer argument N as number of childrens to be produced of the parent program. Generates N random numbers and sets i^{th} random numbers to i^{th} child's burst time.
2. testcase_B: Takes integer argument N as number of childrens to be produced of the parent program. Generates N numbers in strictly decreasing burst time and sets i^{th} random numbers to i^{th} child's burst time.
3. testcase_C: Takes integer argument N as number of childrens to be produced of the parent program. Generates N random numbers and sets i^{th} random numbers to i^{th} child's burst time. Along with it, this program calls I/O bound process upto 10^8 times, The I/O process used in this program is `printf`.

We will look over test cases and would compare the outcome for the algorithm:
After Qemu get booted:

Test case A:

- Running command: `$ testcase_A 5`
  The outcome for the **SJF algorithm** is shown below.

- 5 children processes are created using fork() sequentially, child 0, 1, 2, 3, 4 with pid 6, 7, 8, 9, 10 respectively have random burst time 36, 16, 27, 8, 24 respectively.
- We can clearly see the order of execution of child processes and its completion is in increasing order of burst times i.e. process with minimum burst time(pid = 9, burst-time = 8) is executed and completed first and so on.
- Order of execution and completion of processes:

Exe/Com order	PID - 9	PID - 7	PID - 10	PID - 8	PID - 6
Burst Time	8	16	24	27	36

```
$ testcase_A 5
Current running process PID: 5, Burst Time: 1000
Current running process PID: 9, Burst Time: 8
Exit Process with ID: 9

Current running process PID: 7, Burst Time: 16
Exit Process with ID: 7

Current running process PID: 10, Burst Time: 24
Exit Process with ID: 10

Current running process PID: 8, Burst Time: 27
Exit Process with ID: 8

Current running process PID: 6, Burst Time: 36
Exit Process with ID: 6

Current running process PID: 5, Burst Time: 1000

All children completed
Child 0.    pid 6, BT: 36
Child 1.    pid 7, BT: 16
Child 2.    pid 8, BT: 27
Child 3.    pid 9, BT: 8
Child 4.    pid 10, BT: 24
Exit Process with ID: 5
```

- **Round Robin Scheduling** outcome is shown below.
- Again 5 children processes were generated with pids 4, 5, 6, 7 and 8 with burst time 36, 16, 27, 8 and 24 respectively.
- Order of Execution and completion of processes -

Exec/Comp	Pid- 8	Pid- 5	Pid- 6	Pid- 7	Pid- 4
Burst Time	24	16	27	8	36

```
$ testcase_A 5
Exit Process with ID: 8

Exit Process with ID: 5

Exit Process with ID: 6

Exit Process with ID: 7

Exit Process with ID: 4


All children completed
Child 0.    pid 4, BT: 36
Child 1.    pid 5, BT: 16
Child 2.    pid 6, BT: 27
Child 3.    pid 7, BT: 8
Child 4.    pid 8, BT: 24
```

Comparison

- We see a difference between the execution order in SJF and RR scheduling algorithms.
- In SJF scheduling, the process with the shortest burst time was executed before the execution of other processes(Increasing order of burst time).
- In RR scheduling there is FIFO scheduling with each process assigned a time quantum.
- A process in RR scheduling which was executing late, executed earlier in SJF scheduling when it's burst time was decreased.

Test case B:

- Running command: `$ testcase_B 5`
- The outcome for the **SJF algorithm** is shown below.
  5 children processes are created using fork() sequentially, child 0, 1, 2, 3, 4 with pid 13, 14, 15, 16, 17 respectively with strictly decreasing burst time 800, 799, 798, 797, 796 respectively.
- We can clearly see the order of execution of child processes and its completion is in increasing order of burst times i.e. process with minimum burst time(pid = 17, burst-time = 796) is executed and completed first and so on.
- Order of execution and completion of processes:

Exec/Com order	PID - 17	PID - 16	PID - 15	PID - 14	PID - 13
Burst Time	796	797	798	799	800

```
$ testcase_B 5
Current running process PID: 12, Burst Time: 10000
Current running process PID: 17, Burst Time: 796
Exit Process with ID: 17

Current running process PID: 16, Burst Time: 797
Exit Process with ID: 16

Current running process PID: 15, Burst Time: 798
Exit Process with ID: 15

Current running process PID: 14, Burst Time: 799
Exit Process with ID: 14

Current running process PID: 13, Burst Time: 800
Exit Process with ID: 13

Current running process PID: 12, Burst Time: 10000

All children completed and were generated in following order:
Child 0.    pid 13, BT: 800
Child 1.    pid 14, BT: 799
Child 2.    pid 15, BT: 798
Child 3.    pid 16, BT: 797
Child 4.    pid 17, BT: 796
Exit Process with ID: 12
```

- **Round Robin Scheduling** outcome is shown below.
- Again 5 children processes were generated with pids 17, 18, 19, 20 and 21 with decreasing burst time 800, 799, 798, 797, 796 respectively.
- Order of Execution and completion of processes -

Exec/Comp	Pid- 17	Pid- 20	Pid - 18	Pid- 21	Pid- 19
Burst Time	800	797	799	796	798

```
[$ testcase_B 5
Exit Process with ID: 17

Exit Process with ID: 20

Exit Process with ID: 18

Exit Process with ID: 21

Exit Process with ID: 19


All children completed and w
Child 0.     pid 17, BT: 800
Child 1.     pid 18, BT: 799
Child 2.     pid 19, BT: 798
Child 3.     pid 20, BT: 797
Child 4.     pid 21, BT: 796
Exit Process with ID: 16
```

Test case C:

- Running command: `$ testcase_C 5`
- The outcome for the **SJF algorithm** is shown below.
- 5 children processes are created using fork() sequentially, child 0, 1, 2, 3, 4 with pid 20, 21, 22, 23, 24 respectively have random burst time 36, 16, 27, 8, 24 respectively.
- Upto 10^8 I/O bound process were called, which generated upto 10^8 I/O interrupts between the process and clearly it doesn't affect the scheduling and completion of the program.
- We can clearly see the order of execution of child processes and its completion is in increasing order of burst times i.e. process with minimum burst time(pid = 23, burst-time = 8) is executed and completed first and so on.
- Order of execution and completion of processes:

Exe/Com order	PID - 23	PID - 21	PID - 24	PID - 22	PID - 20
Burst Time	8	16	24	27	36

```
$ testcase_C 5
Current running process PID: 19, Burst Time: 1000
Current running process PID: 23, Burst Time: 8
Exit Process with ID: 23

Current running process PID: 21, Burst Time: 16
Exit Process with ID: 21

Current running process PID: 24, Burst Time: 24
Exit Process with ID: 24

Current running process PID: 22, Burst Time: 27
Exit Process with ID: 22

Current running process PID: 20, Burst Time: 36
Exit Process with ID: 20

Current running process PID: 19, Burst Time: 1000

All children completed
Child 0.     pid 20, BT: 36
Child 1.     pid 21, BT: 16
Child 2.     pid 22, BT: 27
Child 3.     pid 23, BT: 8
Child 4.     pid 24, BT: 24
Exit Process with ID: 19
```

- **Round Robin Scheduling** outcome is shown below.
- Again 5 children processes were generated with pids 10, 11, 12, 13 and 14 with burst time 36, 16, 27, 8 and 24 respectively.
- Order of Execution and completion of processes -

Exec/Comp	Pid- 12	Pid- 10	Pid- 14	Pid- 11	Pid- 13
Burst Time	24	16	27	8	36

```
$ testcase_C 5
Exit Process with ID: 12

Exit Process with ID: 10

Exit Process with ID: 14

Exit Process with ID: 11

Exit Process with ID: 13


All children completed
Child 0.    pid 10, BT: 36
Child 1.    pid 11, BT: 16
Child 2.    pid 12, BT: 27
Child 3.    pid 13, BT: 8
Child 4.    pid 14, BT: 24
Exit Process with ID: 9
```