CS_344 Assignment 2

Group C21: Satvik Tiwari, 200101091
Pranjal Baranwal, 200101083
Akshat Mittal, 200101011

Part A: Implementing System Calls

In this part, we have to implement following system calls in xv6:

1. ***getNumProc()***

   Returns the total number of active processes in the system (either in embryo, running, runnable, sleeping, or zombie states).
   To do so, we simply looped over the process table to get all the active processes i.e. any process which is not in *UNUSED* state and returned the count.

```
592  // get total number of active processes in system
593  int
594  getNumProc(void)
595  {
596    struct proc *p;
597
598    int count = 0;
599    acquire(&ptable.lock);
600    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
601      if(p->state == UNUSED)
602        continue;
603      else
604        count++;
605      // cprintf("%d\n",p->pid);
606    }
607    release(&ptable.lock);
608    return count;
609  }
```

2. ***getMaxPid()***

   Returns the maximum process ID (*pid*) out of all currently active processes in the system. We simply looped over the process table to get the maximum *pid*.

```
611  // get the maximum PID among the PIDs of all currently active processes
612  int
613  getMaxPid(void)
614  {
615    struct proc *p;
616
617    int max = -1;
618    acquire(&ptable.lock);
619    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
620      if(p->state == UNUSED)
621        continue;
622      if(p->pid > max)
623        max = p->pid;
624    }
625    release(&ptable.lock);
626    return max;
627  }
```

3. ***getProcInfo(pid, &processInfo)***

   This function takes two arguments, ID of the process and a *processInfo struct*. It gives the *parentID*, the number of times context switched and the process size in bytes of the process with given ID.
   For this, we added *contextswitches* field in our process structure in ***proc.h*** and simply looped over the process table. Upon finding the given ID, copy all the required data into our *processInfo struct*.
   It returns -1 if this ID is not found in the process table.

```c
// get info about the process having given PID
int
getProcInfo(int pid, struct processInfo* st)
{
  struct proc *p;
  int flag = -1;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid)
    {
      st->ppid = 0;
      // check if parent exists
      if(p->parent != 0)
        st->ppid = p->parent->pid;
      st->psize = p->sz;
      st->numberContextSwitches = p->contextswitches;
      flag = 0;
      break;
    }
  }
  release(&ptable.lock);
  return flag;
}
```

4. ***set_burst_time(n)***

   This function takes an integer input n and set the burst time of current process equal to n.
   For this, we added burst field in our process structure. Using *myproc()* function we simply get instance to our current process and then set the burst time. We used yield to skip one CPU scheduling as scheduling might change due to this change.

```c
653  // set burst time for a process
654  int
655  set_burst_time(int bt)
656  {
657    myproc()->burst = bt;
658    // cprintf("%d\n",myproc()->pid);
659    // skip one CPU scheduling round.
660    yield();
661    return 0;
662  }
```

5. ***get_burst_time()***

   It simply returns the burst time of current process.

```c
664  int
665  get_burst_time(void)
666  {
667    return myproc()->burst;
668  }
```

Files updated to add system calls and user programs:

1. ***proc.c*** - All the above codes are written in this file.
2. ***proc.h*** - The proc structure is modified to include fields *contextswitches* and *burst*.

```c
int contextswitches;          // store number of context switches
int burst;                    // store approximate burst time
```

3. ***defs.h*** - This header file is included in ***sysproc.c*** to allow it to call the functions implemented in ***proc.c***, so we declare our functions here.

```c
// assignmnet 2A
int             getNumProc(void);
int             getMaxPid(void);
int             getProcInfo(int, struct processInfo*);
int             set_burst_time(int);
int             get_burst_time(void);
```

4. ***sysproc.c*** - We need to pick the arguments provided by the user from stack using argptr, since the process related to system call must have a void argument. Here we call the actual functions defined in ***proc.c***.

```
// assignment 2A
int
sys_getNumProc(void)
{
  return getNumProc();
}

int
sys_getMaxPid(void)
{
  return getMaxPid();
}

int
sys_getProcInfo(void)
{
  int pid;
  struct processInfo* st;
  if(argint(0, &pid) < 0)
    return -1;
  if(argptr(1, (void*)&st, sizeof(st)) < 0)
    return -1;
  return getProcInfo(pid, st);
}
```

```
int
sys_set_burst_time()
{
  int bt;
  if(argint(0, &bt) < 0)
    return -1;
  if(bt <= 0)
    return -1;
  set_burst_time(bt);
  return 0;
}

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

5. ***syscall.h*** - Map the system call names to system call numbers.

```
#define SYS_getNumProc  22
#define SYS_getMaxPid  23
#define SYS_getProcInfo 24
#define SYS_set_burst_time  25
#define SYS_get_burst_time  26
```

6. ***syscall.c*** - Add function pointers to the actual system call implementations and export it. The variables declared in ***syscall.h*** are used to index into the array of function pointers.

```
// assignment 2A
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);
```

```
// assignment 2A
[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
```

7. ***user.h*** - System call definitions are added here to make them available to the user program.

```
// Assignment 2A
int getNumProc(void);
int getMaxPid(void);
int getProcInfo(int pid, struct processInfo*);
int set_burst_time(int n);
int get_burst_time(void);
```

8. ***usys.S*** - The list of system calls which we want to export by kernel are added here.

```
SYSCALL(getNumProc)
SYSCALL(getMaxPid)
SYSCALL(getProcInfo)
SYSCALL(set_burst_time)
SYSCALL(get_burst_time)
```

9. ***Makefile*** - Makefile needs to be edited before our user program is available for xv6 source code for compilation. We included *_<userprogram_name>*\ in the UPROGS list to add this as a command line instruction in XV6 kernel which will execute corresponding test files.

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _getNumProc\
    _getMaxPid\
    _getProcInfo\
    _set_burst_time\
```

10. We also added ***getNumProc.c***, ***getMaxPid.c***, ***getProcInfo.c*** and ***set_burst_time.c*** to test the system calls we made.

Part B: Scheduling

Shortest Job First (SJF) Scheduling

The SJF scheduling algorithm schedules the process with lowest burst time first. It is implemented in the *scheduler()* function in ***proc.c*** file.

- We initialize *burst* and *contextswitches* field in *allocproc()* function in **proc.c**.

```
p->contextswitches = 0;
p->burst = 0;
```

- As we wanted to simulate this on a single CPU, the NCPU parameter is changed to 1 in ***param.h***.

```
#define NCPU          1  // maximum number of CPUs
```

- In ***trap.c***, lines 107 to 109 are commented to turn off the preemption in scheduler as SJF is non-preemptive.

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

- Changes in scheduler function of ***proc.c*** for SJF scheduling:

1. We iterate over all the processes which are *RUNNABLE* and chose the one with the smallest burst time for scheduling.
2. If there is no process then we simply continue. Otherwise, we context-switch to this chosen process.
3. On line 403 we increased number of context switches by 1 for current process.

- If we have n processes then our scheduler has a time complexity of ***O(n)***.

```
361     // Shortest job first scheduling algorithm
362     while(1){
363       // Enable interrupts on this processor.
364       sti();
365       // To store the job with least burst time
366       struct proc *shortest_job = 0;
367
368       // Acquire process table lock
369       acquire(&ptable.lock);
370       // Find the job with least burst time
371       for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
372       {
373         if (p->state == RUNNABLE)
374         {
375           if (!shortest_job)
376           {
377             shortest_job = p;
378           }
379           else if (p->burst < shortest_job->burst)
380           {
381             shortest_job = p;
382           }
383         }
384       }
385
386       if (!shortest_job)
387       {
388         release(&ptable.lock);
389         continue;
390       }
```

```
392       p = shortest_job;
393
394       // Switch to chosen process.  It is the process's job
395       // to release ptable.lock and then reacquire it
396       // before jumping back to us.
397       c->proc = p;
398       switchuvm(p);
399       p->state = RUNNING;
400
401       swtch(&(c->scheduler), p->context);
402       // increment number of context switches
403       p->contextswitches = p->contextswitches + 1;
404       switchkvm();
405
406       // Process is done running for now.
407       // It should have changed its p->state before coming back.
408       c->proc = 0;
409       release(&ptable.lock);
410     }
```

Testcases SJF Scheduler:

- We have our testcase in ***testSched1.c*** and ***testSched2.c*** in which we fork 10 processes one by one with different burst times. We make 6 CPU bound processes and 4 I/O bound processes.

```
$ testSched1
(10) CPU Bound(931036809)       Burst Time: 10  Context Switches: 5
(9) CPU Bound(874574840)        Burst Time: 20  Context Switches: 10
(8) CPU Bound(745846863)        Burst Time: 30  Context Switches: 15
(7) CPU Bound(467847273)        Burst Time: 40  Context Switches: 20
(6) CPU Bound(1284847454)       Burst Time: 50  Context Switches: 25
(5) CPU Bound(1130519139)       Burst Time: 60  Context Switches: 34
(4) IO Bound    Burst Time: 70  Context Switches: 701
(3) IO Bound    Burst Time: 80  Context Switches: 801
(2) IO Bound    Burst Time: 90  Context Switches: 901
(1) IO Bound    Burst Time: 100 Context Switches: 1001
$
$
$
$ testSched2
(3) CPU Bound(491195140)        Burst Time: 10  Context Switches: 6
(6) CPU Bound(1872725030)       Burst Time: 30  Context Switches: 16
(5) CPU Bound(-2147483648)      Burst Time: 60  Context Switches: 34
(2) CPU Bound(487655483)        Burst Time: 70  Context Switches: 39
(8) CPU Bound(981612159)        Burst Time: 80  Context Switches: 45
(9) CPU Bound(-2147483648)      Burst Time: 100 Context Switches: 56
(7) IO Bound    Burst Time: 20  Context Switches: 201
(1) IO Bound    Burst Time: 40  Context Switches: 401
(10) IO Bound   Burst Time: 50  Context Switches: 501
(4) IO Bound    Burst Time: 90  Context Switches: 901
```

Fig- Output of Round Robin scheduling

```
$ testSched1
(10) CPU Bound(174984499)       Burst Time: 10  Context Switches: 1
(9) CPU Bound(349968999)        Burst Time: 20  Context Switches: 1
(8) CPU Bound(524953499)        Burst Time: 30  Context Switches: 1
(7) CPU Bound(699938000)        Burst Time: 40  Context Switches: 1
(6) CPU Bound(874922500)        Burst Time: 50  Context Switches: 1
(5) CPU Bound(1049907000)       Burst Time: 60  Context Switches: 1
(4) IO Bound    Burst Time: 70  Context Switches: 701
(3) IO Bound    Burst Time: 80  Context Switches: 801
(2) IO Bound    Burst Time: 90  Context Switches: 901
(1) IO Bound    Burst Time: 100 Context Switches: 1001
$
$
$
$ testSched2
(3) CPU Bound(174984499)        Burst Time: 10  Context Switches: 1
(6) CPU Bound(524953499)        Burst Time: 30  Context Switches: 1
(5) CPU Bound(1049907000)       Burst Time: 60  Context Switches: 1
(2) CPU Bound(1224891500)       Burst Time: 70  Context Switches: 1
(8) CPU Bound(1399876000)       Burst Time: 80  Context Switches: 1
(9) CPU Bound(1749845000)       Burst Time: 100 Context Switches: 1
(7) IO Bound    Burst Time: 20  Context Switches: 201
(1) IO Bound    Burst Time: 40  Context Switches: 401
(10) IO Bound   Burst Time: 50  Context Switches: 501
(4) IO Bound    Burst Time: 90  Context Switches: 901
```

Fig- Output of SJF scheduling

- For testcase1, we have forked 10 processes.
  The first 6 are CPU bound processes with burst time of 100, 90, 80, 70, 60 and 50 units respectively while the last 4 are IO bound process having burst time of 40, 30, 20 and 10 units respectively.

- For testcase2, we have forked 10 processes and for every IO bound process, we fork two CPU bound processes except the 4th IO bound process (10th process).
  The burst time of the processes are: 40 (IO), 70 (CPU), 10 (CPU), 90 (IO), 60 (CPU), 30 (CPU), 20 (IO), 80 (CPU), 90 (CPU) and 50 (IO).

- In normal round robin, we can see that the order of completion depends on burst time, shorter jobs are getting completed first but there are several preemptions for all processes before they complete.
  IO processes took more time to complete (though they have less burst time). This is due to a greater number of IO operations; they are pushed from running to waiting state and thus, the number of context switches also increases.

- While in SJF, we do not preempt the process until completed and hence only 1 context switch is observed in case of CPU bound processes while context switches still occur in IO bound processes due to IO operation.
- When we execute testcase1 and testcase2, we see that all CPU bound processes terminate before IO bound processes and both CPU bound and IO bound processes terminate in ascending order of burst times among themselves, showcasing a successful SJF scheduling algorithm. IO bound processes have a large number of context switches because of lots of IO delays.

Hybrid Scheduler (SJF + Round Robin)

The hybrid scheduling algorithm is similar to round-robin but the processes are selected not on basis of first-come-first serve but on shortest burst time. It is implemented in the *scheduler()* function in ***proc.c*** file.

- The ***struct proc*** was also modified wherein additional members such as *time_slice* and *first_proc* were included to keep track of the time slice taken by a process and the shortest process so as to change the *time_quanta* variable as per the *time_slice* required for the *first_proc*, i.e., the shortest burst time process.

```
int time_slice;    // for time quanta
int first_proc;    // to indicate shortest process
```

- We initialize burst, contextswitches, time_slice and first_proc field in *allocproc()* function in ***proc.c***

```
98    // For hybrid scheduling
99    p->burst = 1;
100   p->time_slice = 1;
101   p->first_proc = 0;
```

- In ***trap.c***, for round-robin, the preemption happens when time-quantum expires for a process. But for hybrid scheduling, we have to consider the burst time as well since we are using SJF.
  Hence, for the *first_proc*, i.e., the shortest process, we increment the *time slice* and update *time_quanta* for the round.
  For all remaining processes, if they have not run for *time_quanta* amount of time, we simply increment the *time_slice*, else as soon as their *time_quanta* expire, we preempt the current process.

```
109   if(myproc() && myproc()->state == RUNNING &&
110      tf->trapno == T_IRQ0+IRQ_TIMER)
111   {
112     // yield();
113     // For hybrid scheduling algoritm
114     if(myproc()->first_proc && (first_pid == -1 || first_pid == myproc()->pid))
115     {
116       myproc()->time_slice++;
117       time_quanta = myproc()->time_slice + 1;
118       first_pid = myproc()->pid;
119     }
120     else
121     {
122       if(myproc()->time_slice < time_quanta)
123       {
124         myproc()->time_slice++;
125       }
126       else {
127         myproc()->time_slice = 0;
128         yield();
129       }
130     }
131   }
```

- Changes in scheduler function of ***proc.c*** for hybrid scheduling:

  1. First, a ready queue is created wherein all the *RUNNABLE* processes will be pushed.
  2. Then, we sort the processes in ready queue using standard **merge sort algorithm** based on their burst time and mark first process of queue as *first_proc*.

3. Then from line 397 to 421, we schedule the processes one by one with increasing burst time and doing round robin at the same time.

- If we have n processes then our scheduler has a time complexity of O(n*logn) which is time complexity to sort the processes.

```
369   // Hybrid scheduling algorithm
370   int flag = 1;
371   while(1){
372     // Enable interrupts on this processor.
373     sti();
374
375     // Acquire process table lock
376     acquire(&ptable.lock);
377     // Set up Ready Queue
378     struct proc* ready_queue[NPROC];
379     int k = 0;
380     for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
381     {
382       if(p->state == RUNNABLE)
383       {
384         ready_queue[k++] = p;
385       }
386     }
387
388     // Sort Ready Queue
389     merge_sort(ready_queue, 0, k-1);
390
391     if(k && flag)
392     {
393       ready_queue[0]->first_proc = 1;
394       flag = 0;
395     }
```

```
396
397     // Schedule the jobs with increasing burst time
398     for (int i = 0; i <= k-1; i++)
399     {
400       p = ready_queue[i];
401       if (p->state == RUNNABLE)
402       {
403         // Switch to chosen process.  It is the process's job
404         // to release ptable.lock and then reacquire it
405         // before jumping back to us.
406         c->proc = p;
407         switchuvm(p);
408         p->state = RUNNING;
409
410         swtch(&(c->scheduler), p->context);
411
412         // increment number of context switches
413         p->contextswitches = p->contextswitches + 1;
414
415         switchkvm();
416
417         // Process is done running for now.
418         // It should have changed its p->state before coming back.
419         c->proc = 0;
420       }
421     }
422     release(&ptable.lock);
423   }
```

Testcases Hybrid Scheduler:

- We have our testcase in ***testSched1.c*** and ***testSched2.c*** in which we fork 10 processes one by one with different burst times. We make 6 CPU bound processes and 4 I/O bound processes.

```
$ testSched1
(10) CPU Bound(663776823)       Burst Time: 10  Context Switches: 2
(9) CPU Bound(966320054)        Burst Time: 20  Context Switches: 3
(8) CPU Bound(818218982)        Burst Time: 30  Context Switches: 4
(7) CPU Bound(241819969)        Burst Time: 40  Context Switches: 5
(6) CPU Bound(1242780935)       Burst Time: 50  Context Switches: 7
(5) CPU Bound(1166756679)       Burst Time: 60  Context Switches: 8
(4) IO Bound    Burst Time: 70  Context Switches: 701
(3) IO Bound    Burst Time: 80  Context Switches: 801
(2) IO Bound    Burst Time: 90  Context Switches: 901
(1) IO Bound    Burst Time: 100 Context Switches: 1001
$
$
$
$ testSched2
(3) CPU Bound(203214434)        Burst Time: 10  Context Switches: 2
(6) CPU Bound(1015743411)       Burst Time: 30  Context Switches: 4
(5) CPU Bound(-2147483648)      Burst Time: 60  Context Switches: 8
(2) CPU Bound(1068254894)       Burst Time: 70  Context Switches: 9
(8) CPU Bound(803224280)        Burst Time: 80  Context Switches: 10
(9) CPU Bound(1454928048)       Burst Time: 100 Context Switches: 13
(7) IO Bound    Burst Time: 20  Context Switches: 201
(1) IO Bound    Burst Time: 40  Context Switches: 401
(10) IO Bound   Burst Time: 50  Context Switches: 501
(4) IO Bound    Burst Time: 90  Context Switches: 901
```

Fig- Output of Round Robin scheduling

- For testcase1, we have forked 10 processes.
  The first 6 are CPU bound processes with burst time of 100, 90, 80, 70, 60 and 50 units respectively while the last 4 are IO bound process having burst time of 40, 30, 20 and 10 units respectively.

- For testcase2, we have forked 10 processes and for every IO bound process, we fork two CPU bound processes except the 4^{th} IO bound process (10^{th} process).
  The burst time of the processes are: 40 (IO), 70 (CPU), 10 (CPU), 90 (IO), 60 (CPU), 30 (CPU), 20 (IO), 80 (CPU), 90 (CPU) and 50 (IO).

- When we execute testcase1 and testcase2, we see that all CPU bound processes terminate before IO bound processes and both CPU bound processes and I/O bound processes terminate in ascending order of burst times among themselves, showcasing a successful hybrid scheduling algorithm.
  The only difference between hybrid scheduler and SJF is that SJF was non preemptive but hybrid scheduler is preemptive and hence we can see some context switches for CPU bound process too unlike SJF.
  IO bound processes have a large number of context switches because of lots of IO delays.

Submission:

C21.zip contains following:

1. **Report.pdf**
2. **partAB** – Folder containing all modified files for part A and SJF scheduling
3. **partAB.patch**
4. **hybrid** – Folder containing all modified files for hybrid scheduling
5. **hybrid.patch**
6. **xv6-public-assign2-sjfs** – xv6 folder corresponding to partAB.patch
7. **xv6-public-assign2-hybrid** – xv6 folder corresponding to hybrid.patch