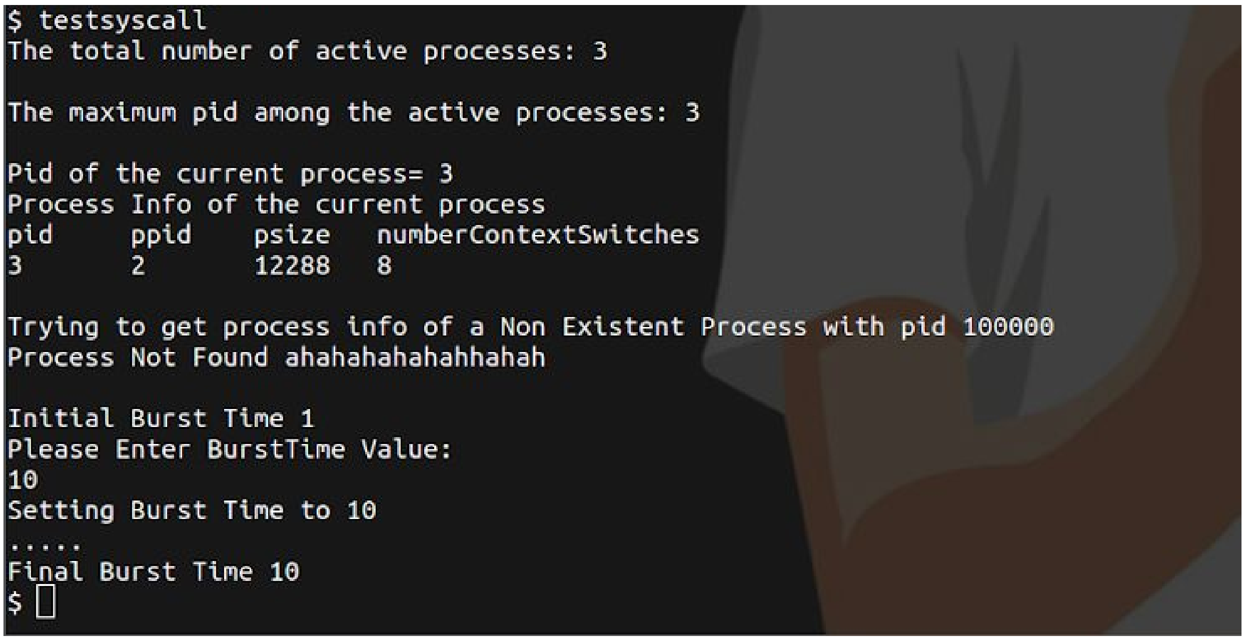
Assignment-2

G-11

Kartikeya Saxena 180101034
Kushal Sangwan 180101096
Milind Prabhu 180101091
Aditya Patil 180101004

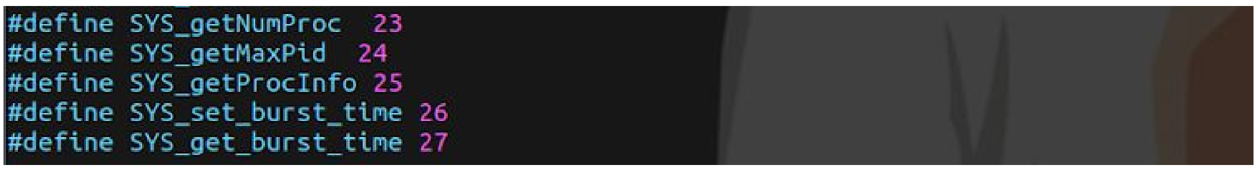
Ans. A- Files modified for this question: syscall.h, syscall.c, user.h, usys.S, defs.h, proc.h, proc.c and sysproc.c.

testsyscall.c
A simple user program to test the system calls we made in Part - A.



```
$ testsyscall
The total number of active processes: 3

The maximum pid among the active processes: 3

Pid of the current process= 3
Process Info of the current process
pid     ppid    psize   numberContextSwitches
3       2       12288   8

Trying to get process info of a Non Existent Process with pid 100000
Process Not Found ahahahahahahhahah

Initial Burst Time 1
Please Enter BurstTime Value:
10
Setting Burst Time to 10
.....
Final Burst Time 10
$ 
```

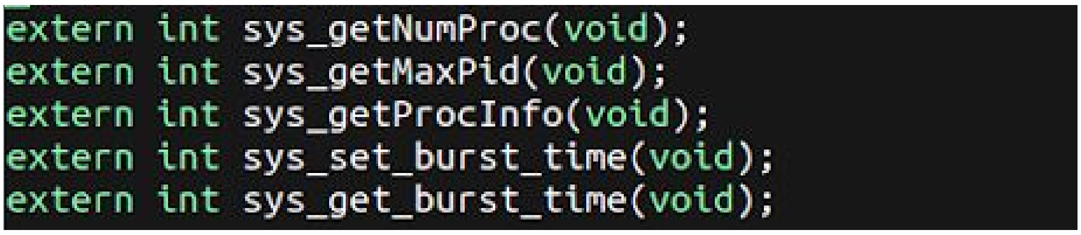
syscall.h
Map the system call names to system call number.



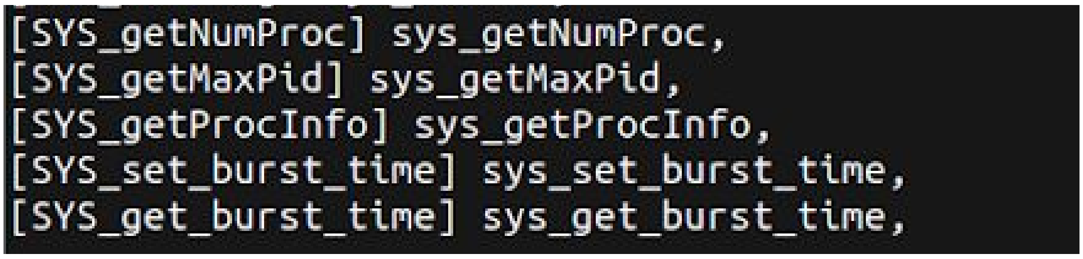
```
#define SYS_getNumProc  23
#define SYS_getMaxPid  24
#define SYS_getProcInfo 25
#define SYS_set_burst_time 26
#define SYS_get_burst_time 27
```

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.



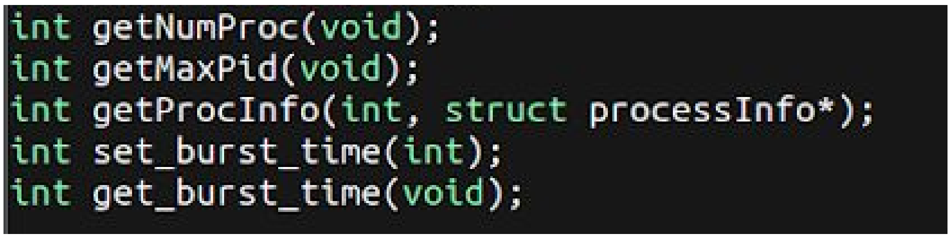
```
extern int sys_getNumProc(void);
extern int sys_getMaxPid(void);
extern int sys_getProcInfo(void);
extern int sys_set_burst_time(void);
extern int sys_get_burst_time(void);
```



```
[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,
```

user.h

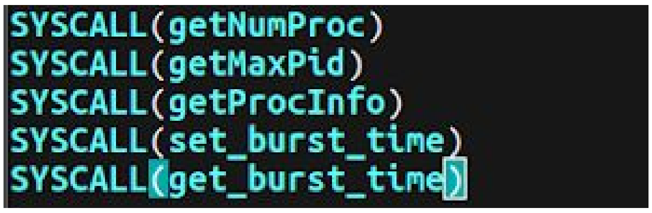
System call definitions are added here to make them available to the user program.



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

usys.S

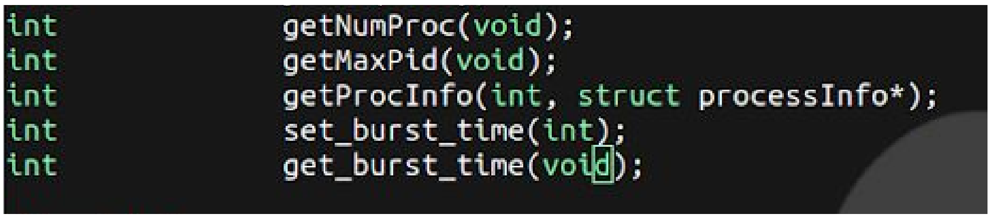
The list of system calls which we want to export by kernel are added here.



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

defs.h

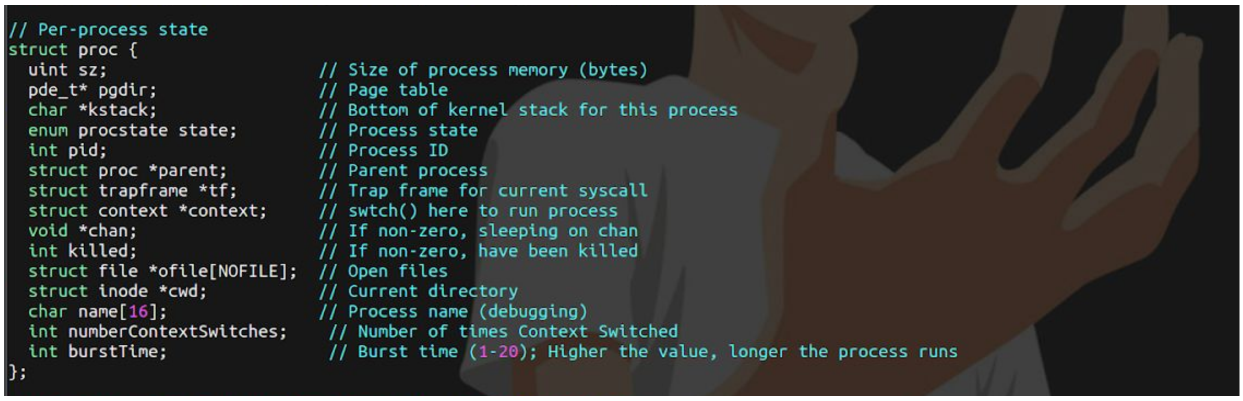
This header file is included in sysproc.c to allow it to call the functions implemented in proc.c (like one lines calls getNumProc() returned in sysproc system call implementations)



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

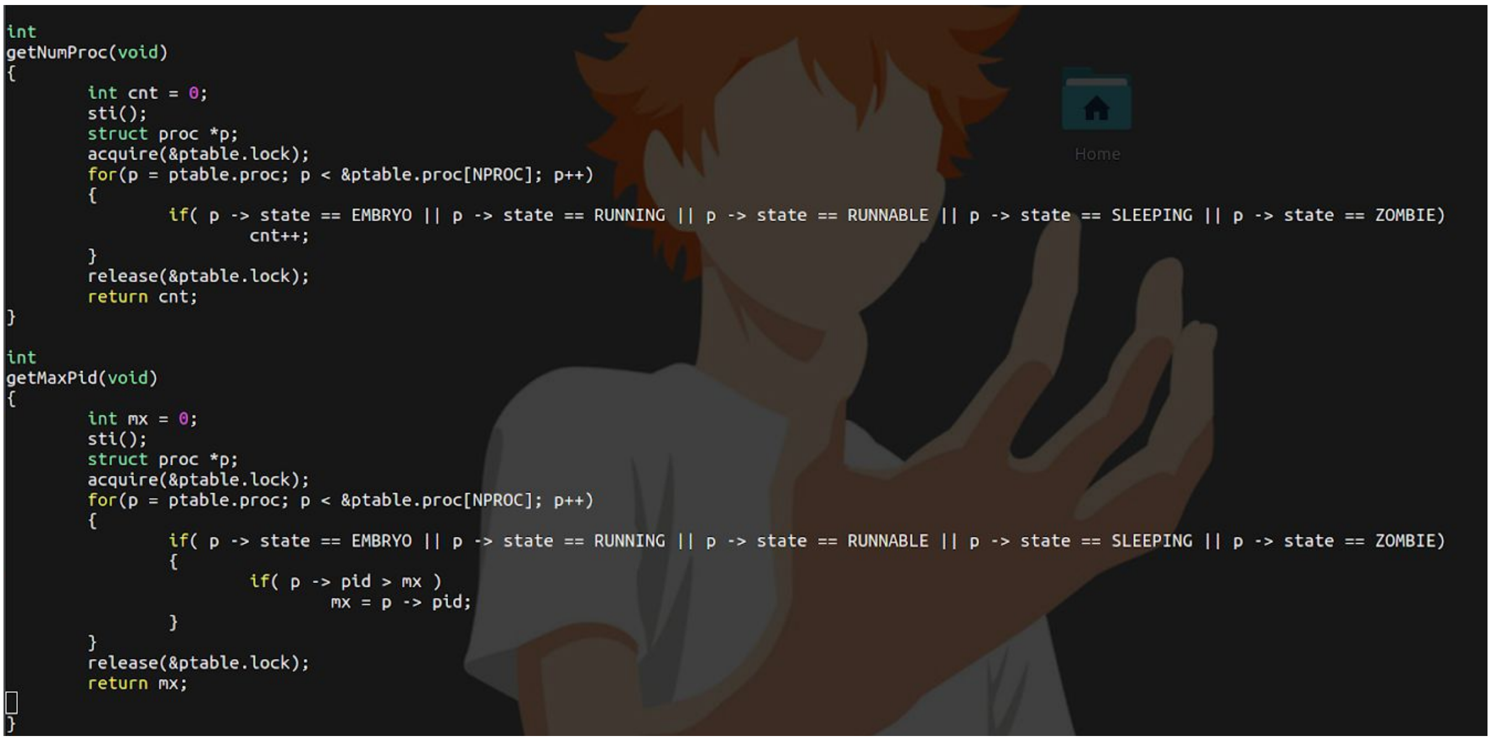
proc.h

The proc structure is modified to include fields numberContextSwitches and burstTime to hold extra info about our process.



```
// 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 numberContextSwitches;    // Number of times Context Switched
  int burstTime;                // Burst time (1-20); Higher the value, longer the process runs
};
```

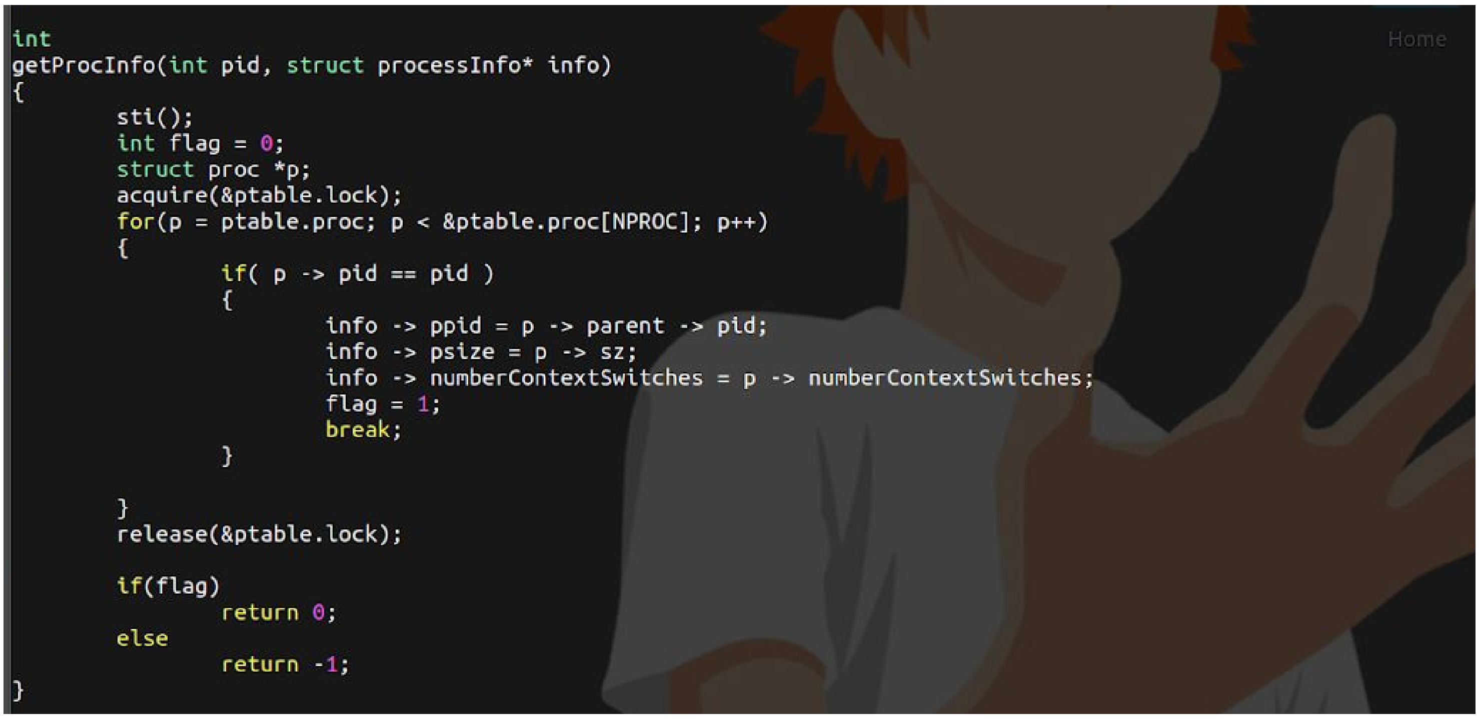
proc.c



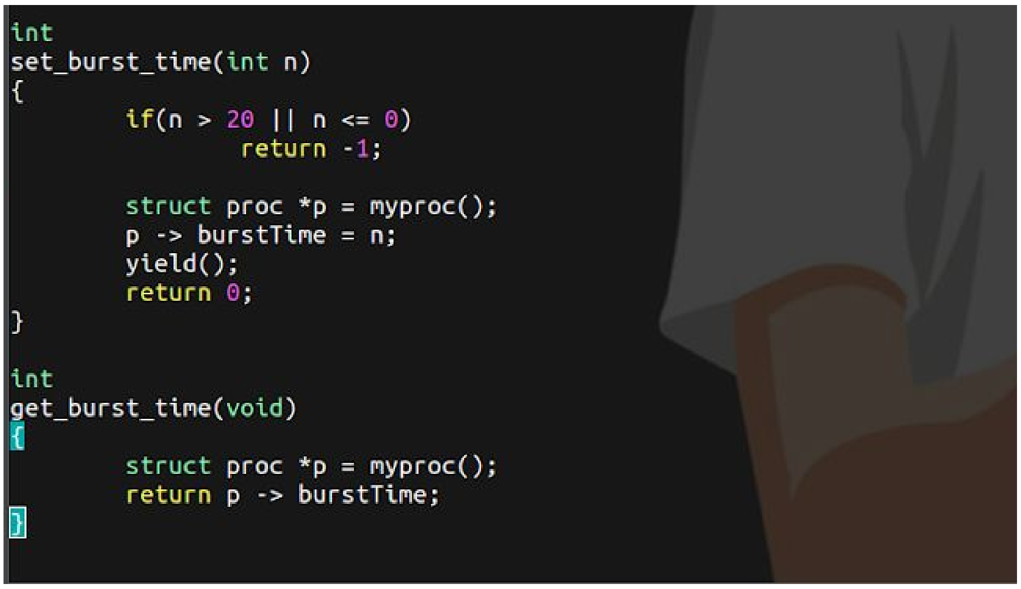
```
int
getNumProc(void)
{
        int cnt = 0;
        sti();
        struct proc *p;
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        {
                if( p -> state == EMBRYO || p -> state == RUNNING || p -> state == RUNNABLE || p -> state == SLEEPING || p -> state == ZOMBIE)
                        cnt++;
        }
        release(&ptable.lock);
        return cnt;
}

int
getMaxPid(void)
{
        int mx = 0;
        sti();
        struct proc *p;
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        {
                if( p -> state == EMBRYO || p -> state == RUNNING || p -> state == RUNNABLE || p -> state == SLEEPING || p -> state == ZOMBIE)
                {
                        if( p -> pid > mx )
                                mx = p -> pid;
                }
        }
        release(&ptable.lock);
        return mx;

}
```

1) The sti() function is called to enable interrupts/traps in the syscall. Then the ptable lock is acquired. The counter cnt is initialized to zero. We run a for loop that iterates the array of processes and for every process that is not "UNUSED", we increment cnt. Finally, the counter cnt is returned after releasing the ptable lock.

In getMaxPid(), a similar technique is used where the maximum pid of processes that is not "UNUSED" is maintained in mx. The ptable lock is acquired before use and released after. Finally, mx is returned.


```
int
getProcInfo(int pid, struct processInfo* info)
{
	sti();
	int flag = 0;
	struct proc *p;
	acquire(&ptable.lock);
	for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
	{
		if( p -> pid == pid )
		{
			info -> ppid = p -> parent -> pid;
			info -> psize = p -> sz;
			info -> numberContextSwitches = p -> numberContextSwitches;
			flag = 1;
			break;
		}

	}
	release(&ptable.lock);

	if(flag)
		return 0;
	else
		return -1;
}
```

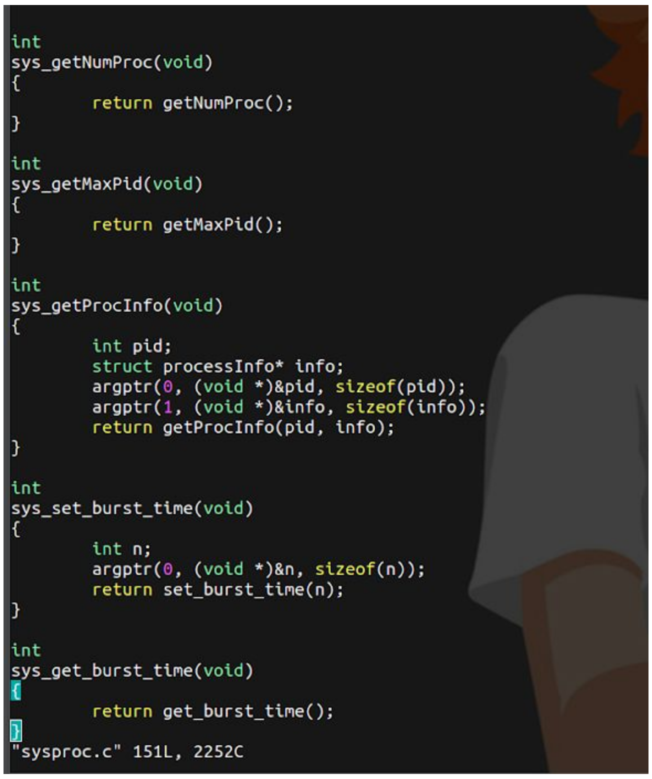
2) We run a for loop to loop over all the processes and find the process with the given pid (This is taken as an argument). When the process is found, all relevant information is copied into ‘info’ which is a user-defined struct processInfo.
Also, it is worth noting that proc.h was modified to include ‘numberContextSwitches’ as an attribute of proc. Further Allocproc function was modified in proc.c to initialize numberContextSwitches to 0 . Also in scheduler, this variable is increased whenever there is a switch from the process to scheduler(identified by calls to switchkvm()). It is worth noting here that even if a process is getting scheduled for 10 consecutive quantums, these are taken as 10 context switches.


```
int
set_burst_time(int n)
{
	if(n > 20 || n <= 0)
		return -1;

	struct proc *p = myproc();
	p -> burstTime = n;
	yield();
	return 0;
}

int
get_burst_time(void)
{
	struct proc *p = myproc();
	return p -> burstTime;
}
```

3) ‘burstTime’ is added as an attribute of proc in proc.h. set_burst_time(int n) first checks if n lies in [1,20]. If not, error (-1) is returned. Otherwise the current process is

acquired in p using myproc() and the burst time of p is set to n.
(Here yield() is added so that a child process relinquishes the cpu and goes to the ready queue after changing its burst times , hence all other child processes can change their burst times too and afterwards scheduling can be carried out according to those burst times.)

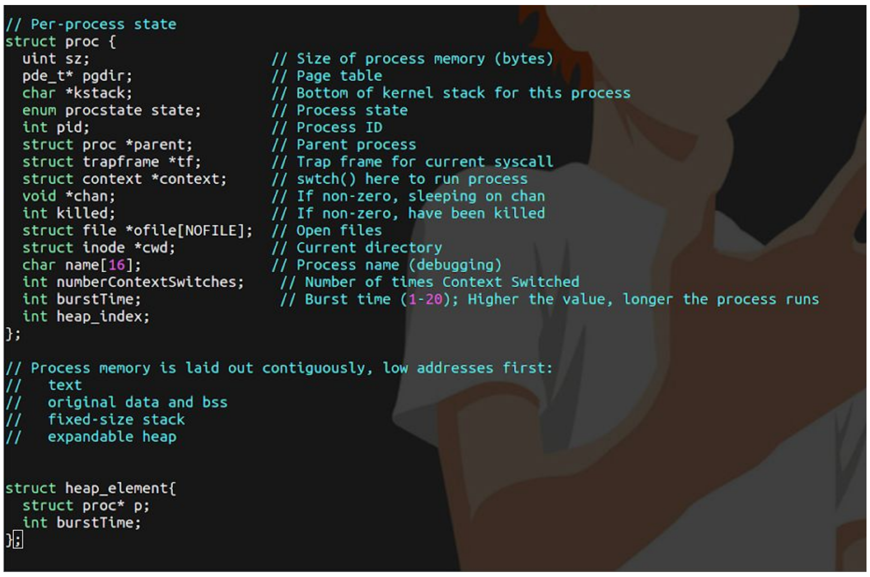
sysproc.c
We need to pick the arguments provided by the user from its stack using argptr, since the process related to system call must have a void argument. Here we call the actual functions defined in proc.c.



```
int
sys_getNumProc(void)
{
        return getNumProc();
}

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

int
sys_getProcInfo(void)
{
        int pid;
        struct processInfo* info;
        argptr(0, (void *)&pid, sizeof(pid));
        argptr(1, (void *)&info, sizeof(info));
        return getProcInfo(pid, info);
}

int
sys_set_burst_time(void)
{
        int n;
        argptr(0, (void *)&n, sizeof(n));
        return set_burst_time(n);
}

int
sys_get_burst_time(void)
{
        return get_burst_time();
}
"sysproc.c" 151L, 2252C
```

Ans. B - Files modified for this question: defs.h, proc.h, proc.c, param.h, exec.h, trap.c

proc.h
A new field heap_index is added to the proc structure to indicate the index of the process in the heap. A new structure(heap_element) is defined which contains a pointer to the process and also its burstTime. Then a priority queue of heap_elements is constructed. This priority queue contains heap_elements corresponding to all RUNNABLE processes.



```
// 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 numberContextSwitches;    // Number of times Context Switched
  int burstTime;                // Burst time (1-20); Higher the value, longer the process runs
  int heap_index;
};

// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap


struct heap_element{
  struct proc* p;
  int burstTime;
};
```

proc.c

The data structure:

We have used a priority queue to find the process with the shortest burst time. The priority queue supports mainly the following functions.

1. heap_extractMin() - This function returns a pointer corresponding to the process with the minimum burst time. It also deletes this process from the priority queue.
2. heap_delete(struct proc* p) - This function deletes the process pointed to by p from the heap.
3. heap_insert(struct proc *p) - This function inserts the process pointed to by p into the heap.
4. There are a few other helper functions like heap_decreaseKey, heap_swapElements etc which are used to implement the above three functions.

Some of the other modifications made are as follows:

- In allocproc, the default values of heap_index is set to -1, burstTime is set to 5 and the number of Context Switches is set to 0.
- In the scheduler, just the extract_min is called to get the process with the minimum burstTime.
- Whenever the set_burst_time function is used to change the burstTime of a particular process, the heap_element corresponding to that process is also updated.
- The heap_elements are inserted into the heap whenever a process becomes RUNNABLE from any other state or changes from RUNNABLE to others (for example , in the functions userinit, fork, yield, wakeup1, kill etc.)

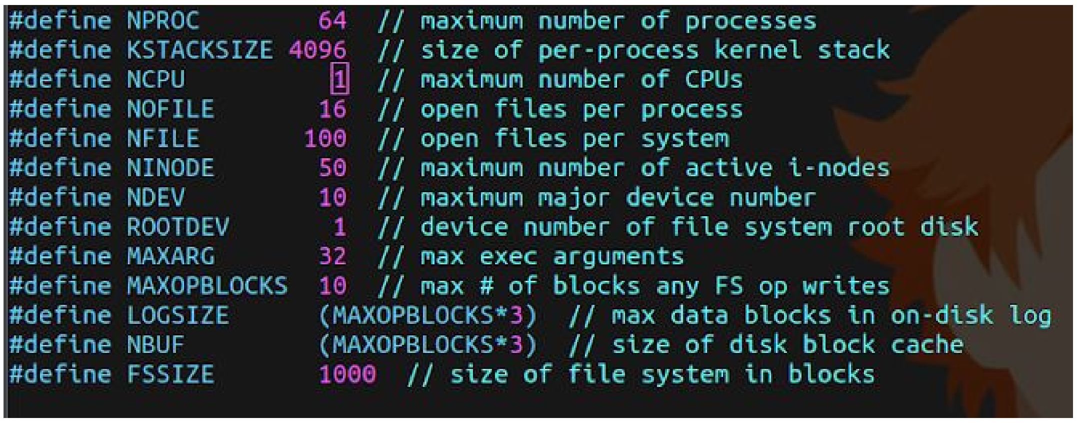
- A lock was acquired on the ptable whenever changes to the process table were made by some of the heap functions.

Time complexity:

- The heap_extractMin, heap_delete and heap_insert all run in O(log n) time where n is the number of processes in the heap.

Param.h

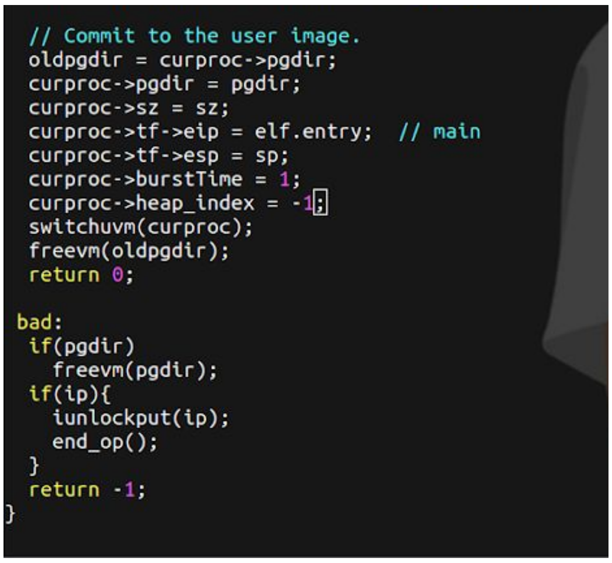
As the design is for a single processor system the NCPU param is changed to 1.



```
#define NPROC        64  // maximum number of processes
#define KSTACKSIZE 4096  // size of per-process kernel stack
#define NCPU          1  // maximum number of CPUs
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
#define NINODE       50  // maximum number of active i-nodes
#define NDEV         10  // maximum major device number
#define ROOTDEV       1  // device number of file system root disk
#define MAXARG       32  // max exec arguments
#define MAXOPBLOCKS  10  // max # of blocks any FS op writes
#define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
#define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
#define FSSIZE       1000  // size of file system in blocks
```

exec.h

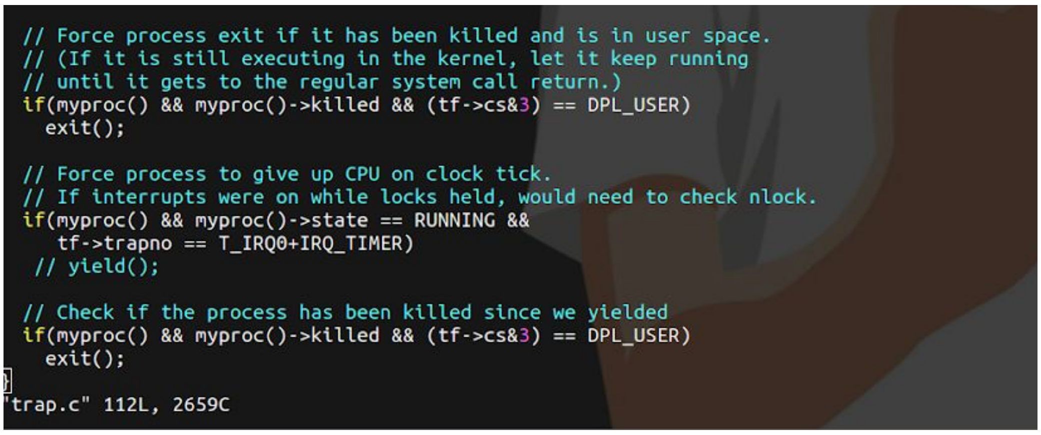
The system processes are called using exec , hence their burstTime is set to 1 and heap Index is set to -1, here so that they can be scheduled before the user processes.



```
  // Commit to the user image.
  oldpgdir = curproc->pgdir;
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
  curproc->burstTime = 1;
  curproc->heap_index = -1;
  switchuvm(curproc);
  freevm(oldpgdir);
  return 0;

 bad:
  if(pgdir)
    freevm(pgdir);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

trap.c

Here yield is turned off so that process is not interrupted while running like round robin after finish of a quanta , and instead the selected process leaves the cpu only either if it is finished or goes to the waiting state.



```
// Force process exit if it has been killed and is in user space.
// (If it is still executing in the kernel, let it keep running
// until it gets to the regular system call return.)
if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
  exit();

// 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();

// Check if the process has been killed since we yielded
if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
  exit();
}
"trap.c" 112L, 2659C
```

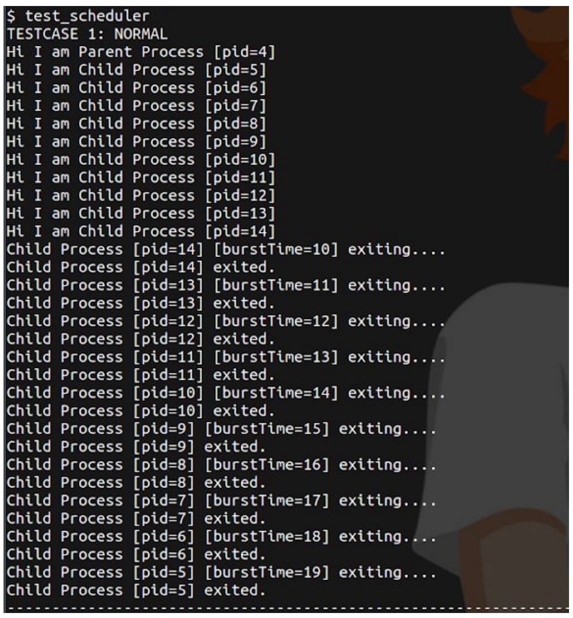
Test_scheduler.c

Testcase1(It consists of only CPU bound processes)

The difference between outputs of SJF and Round robin can be clearly seen here. In SJF , the processes are scheduled in the increasing order of their burst times(process with pid 14 has the least burst time 10 hence it is executed first and so on).In the output this can be seen in the exiting order of the child processes.

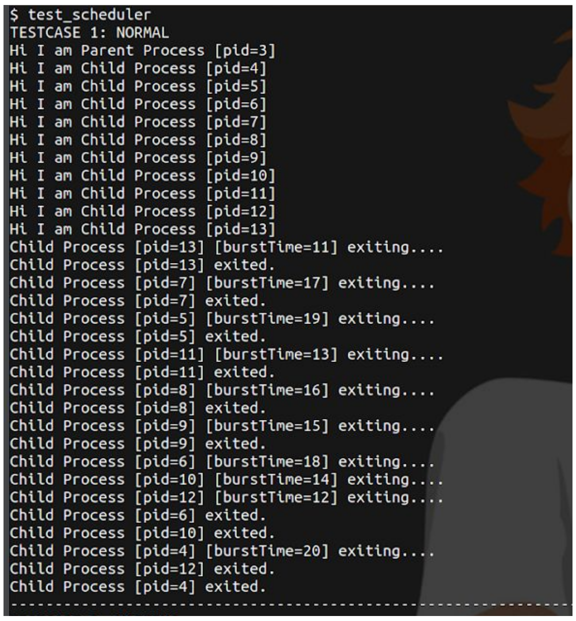
In case of round robin scheduling , process with pid 4(which is created first) is scheduled first (despite of the fact that it has highest burst time as processes here are selected in fcfs order) for one quanta , then process 5 is executed for one quanta and so on till process 14 and the cycle starts again.(The scheduler was made to print the running processes and it was checked that this is indeed the case).

Although in the output of round robin, processes are not exiting in that same order . This is as even if a process is scheduled before it is not guaranteed to finish before, because the cpu speed is not constant throughout. Hence the exiting order is not FCFS as it should be for processes of the same size in a round robin scheduling.



```
$ test_scheduler
TESTCASE 1: NORMAL
Hi I am Parent Process [pid=4]
Hi I am Child Process [pid=5]
Hi I am Child Process [pid=6]
Hi I am Child Process [pid=7]
Hi I am Child Process [pid=8]
Hi I am Child Process [pid=9]
Hi I am Child Process [pid=10]
Hi I am Child Process [pid=11]
Hi I am Child Process [pid=12]
Hi I am Child Process [pid=13]
Hi I am Child Process [pid=14]
Child Process [pid=14] [burstTime=10] exiting....
Child Process [pid=14] exited.
Child Process [pid=13] [burstTime=11] exiting....
Child Process [pid=13] exited.
Child Process [pid=12] [burstTime=12] exiting....
Child Process [pid=12] exited.
Child Process [pid=11] [burstTime=13] exiting....
Child Process [pid=11] exited.
Child Process [pid=10] [burstTime=14] exiting....
Child Process [pid=10] exited.
Child Process [pid=9] [burstTime=15] exiting....
Child Process [pid=9] exited.
Child Process [pid=8] [burstTime=16] exiting....
Child Process [pid=8] exited.
Child Process [pid=7] [burstTime=17] exiting....
Child Process [pid=7] exited.
Child Process [pid=6] [burstTime=18] exiting....
Child Process [pid=6] exited.
Child Process [pid=5] [burstTime=19] exiting....
Child Process [pid=5] exited.
-------------------------------------------------------------
```

SJF Scheduling



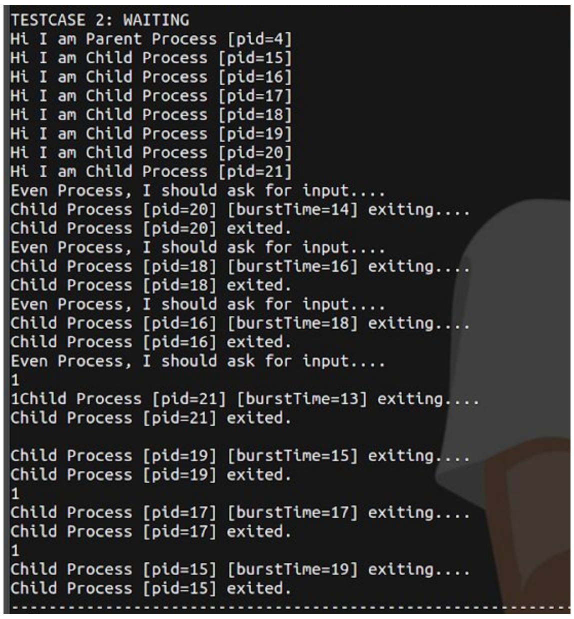
```
$ test_scheduler
TESTCASE 1: NORMAL
Hi I am Parent Process [pid=3]
Hi I am Child Process [pid=4]
Hi I am Child Process [pid=5]
Hi I am Child Process [pid=6]
Hi I am Child Process [pid=7]
Hi I am Child Process [pid=8]
Hi I am Child Process [pid=9]
Hi I am Child Process [pid=10]
Hi I am Child Process [pid=11]
Hi I am Child Process [pid=12]
Hi I am Child Process [pid=13]
Child Process [pid=13] [burstTime=11] exiting....
Child Process [pid=13] exited.
Child Process [pid=7] [burstTime=17] exiting....
Child Process [pid=7] exited.
Child Process [pid=5] [burstTime=19] exiting....
Child Process [pid=5] exited.
Child Process [pid=11] [burstTime=13] exiting....
Child Process [pid=11] exited.
Child Process [pid=8] [burstTime=16] exiting....
Child Process [pid=8] exited.
Child Process [pid=9] [burstTime=15] exiting....
Child Process [pid=9] exited.
Child Process [pid=6] [burstTime=18] exiting....
Child Process [pid=10] [burstTime=14] exiting....
Child Process [pid=12] [burstTime=12] exiting....
Child Process [pid=6] exited.
Child Process [pid=10] exited.
Child Process [pid=4] [burstTime=20] exiting....
Child Process [pid=12] exited.
Child Process [pid=4] exited.
-------------------------------------------------------------
```

Round Robin Scheduling

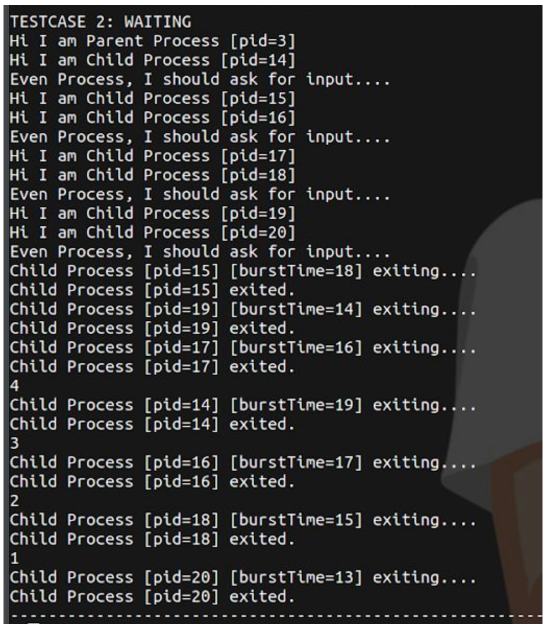
Testcase2(It has a mix of I/O and CPU bound processes)
The processes with even burst times are CPU bound processes and don't require input while the odd ones do.

Hence in case of shortest job first scheduling , the even burstTime processes are executed first in the increasing order of their burst times(14,16,18 ->burstTime of processes in order of execution) as these are the only runnable processes , while the odd ones are in the waiting state . When inputs are given , the odd burstTime processes change their state from waiting to runnable and then execute . Here too they are selected in the order of their increasing burst time (13,15,17,19 -> burstTime of processes in order of execution).

In case of round robin too, the even burstTime processes are executed first as they are the runnable ones. But they are picked in the fcfs order instead (pid 15 is picked first as it was created first , then pid 17 , then pid 19 for one quant each. This was checked by printing the running processes in the scheduler). Although here too the exiting order need not be the same because of the cpu speed reasoning given in the previous test case.
When inputs are given , then also they are picked in increasing order of pids (pid 14 is picked first , then pid 16,18,20).



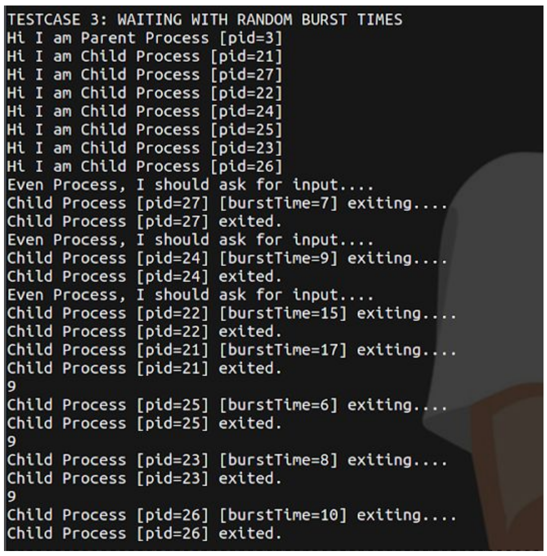
```
TESTCASE 2: WAITING
Hi I am Parent Process [pid=4]
Hi I am Child Process [pid=15]
Hi I am Child Process [pid=16]
Hi I am Child Process [pid=17]
Hi I am Child Process [pid=18]
Hi I am Child Process [pid=19]
Hi I am Child Process [pid=20]
Hi I am Child Process [pid=21]
Even Process, I should ask for input....
Child Process [pid=20] [burstTime=14] exiting....
Child Process [pid=20] exited.
Even Process, I should ask for input....
Child Process [pid=18] [burstTime=16] exiting....
Child Process [pid=18] exited.
Even Process, I should ask for input....
Child Process [pid=16] [burstTime=18] exiting....
Child Process [pid=16] exited.
Even Process, I should ask for input....
1
1Child Process [pid=21] [burstTime=13] exiting....
Child Process [pid=21] exited.

Child Process [pid=19] [burstTime=15] exiting....
Child Process [pid=19] exited.
1
Child Process [pid=17] [burstTime=17] exiting....
Child Process [pid=17] exited.
1
Child Process [pid=15] [burstTime=19] exiting....
Child Process [pid=15] exited.
----------------------------------------------------
```



```
TESTCASE 2: WAITING
Hi I am Parent Process [pid=3]
Hi I am Child Process [pid=14]
Even Process, I should ask for input....
Hi I am Child Process [pid=15]
Hi I am Child Process [pid=16]
Even Process, I should ask for input....
Hi I am Child Process [pid=17]
Hi I am Child Process [pid=18]
Even Process, I should ask for input....
Hi I am Child Process [pid=19]
Hi I am Child Process [pid=20]
Even Process, I should ask for input....
Child Process [pid=15] [burstTime=18] exiting....
Child Process [pid=15] exited.
Child Process [pid=19] [burstTime=14] exiting....
Child Process [pid=19] exited.
Child Process [pid=17] [burstTime=16] exiting....
Child Process [pid=17] exited.
4
Child Process [pid=14] [burstTime=19] exiting....
Child Process [pid=14] exited.
3
Child Process [pid=16] [burstTime=17] exiting....
Child Process [pid=16] exited.
2
Child Process [pid=18] [burstTime=15] exiting....
Child Process [pid=18] exited.
1
Child Process [pid=20] [burstTime=13] exiting....
Child Process [pid=20] exited.
----------------------------------------------------
```

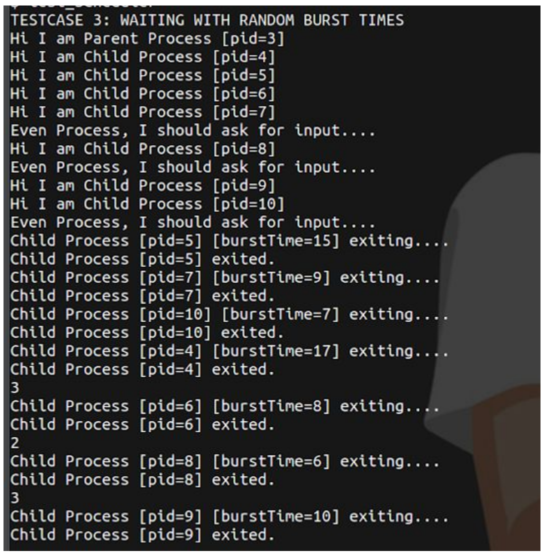
SJF Scheduling **Round Robin Scheduling**

Testcase3(A Mix of CPU bound processes and I/O bound processes , but the burst times are given at random)



```
TESTCASE 3: WAITING WITH RANDOM BURST TIMES
Hi I am Parent Process [pid=3]
Hi I am Child Process [pid=21]
Hi I am Child Process [pid=27]
Hi I am Child Process [pid=22]
Hi I am Child Process [pid=24]
Hi I am Child Process [pid=25]
Hi I am Child Process [pid=23]
Hi I am Child Process [pid=26]
Even Process, I should ask for input....
Child Process [pid=27] [burstTime=7] exiting....
Child Process [pid=27] exited.
Even Process, I should ask for input....
Child Process [pid=24] [burstTime=9] exiting....
Child Process [pid=24] exited.
Even Process, I should ask for input....
Child Process [pid=22] [burstTime=15] exiting....
Child Process [pid=22] exited.
Child Process [pid=21] [burstTime=17] exiting....
Child Process [pid=21] exited.
9
Child Process [pid=25] [burstTime=6] exiting....
Child Process [pid=25] exited.
9
Child Process [pid=23] [burstTime=8] exiting....
Child Process [pid=23] exited.
9
Child Process [pid=26] [burstTime=10] exiting....
Child Process [pid=26] exited.
```

SJF Scheduling



```
TESTCASE 3: WAITING WITH RANDOM BURST TIMES
Hi I am Parent Process [pid=3]
Hi I am Child Process [pid=4]
Hi I am Child Process [pid=5]
Hi I am Child Process [pid=6]
Hi I am Child Process [pid=7]
Even Process, I should ask for input....
Hi I am Child Process [pid=8]
Even Process, I should ask for input....
Hi I am Child Process [pid=9]
Hi I am Child Process [pid=10]
Even Process, I should ask for input....
Child Process [pid=5] [burstTime=15] exiting....
Child Process [pid=5] exited.
Child Process [pid=7] [burstTime=9] exiting....
Child Process [pid=7] exited.
Child Process [pid=10] [burstTime=7] exiting....
Child Process [pid=10] exited.
Child Process [pid=4] [burstTime=17] exiting....
Child Process [pid=4] exited.
3
Child Process [pid=6] [burstTime=8] exiting....
Child Process [pid=6] exited.
2
Child Process [pid=8] [burstTime=6] exiting....
Child Process [pid=8] exited.
3
Child Process [pid=9] [burstTime=10] exiting....
Child Process [pid=9] exited.
```

Round Robin Scheduling

Burst times are taken from the set {7,9,15,17,6,8,10} , here cpu bound processes have got 7,9,15,17 while i/o bound processes have got 6,8,10 .
In SJF , the runnable cpu bound processes are executed first, in the increasing order of their burst times {pid 27:burst Time 7 -> pid 24: burst Time 9 , pid 22:burstTime 15,pid 21:burstTime 17} . For I/O bound processes as and when the input comes , the processes are picked up in increasing burst time order and are given the inputs.{ pid 25: burst Time 6, pid 23: burst Time 8 ,pid 26: burst Time 10}.
In round robin , the runnable cpu bound processes are scheduled in fcfs order(pid 4 -> pid 5 -> pid 7 -> pid 10) for one quanta each and then the cycle is repeated again(although , here too the exit order is different).
For the I/O bound process the same thing is followed , when input comes they are selected in fcfs(pid 6-> pid 8-> pid 9) order to get the inputs.

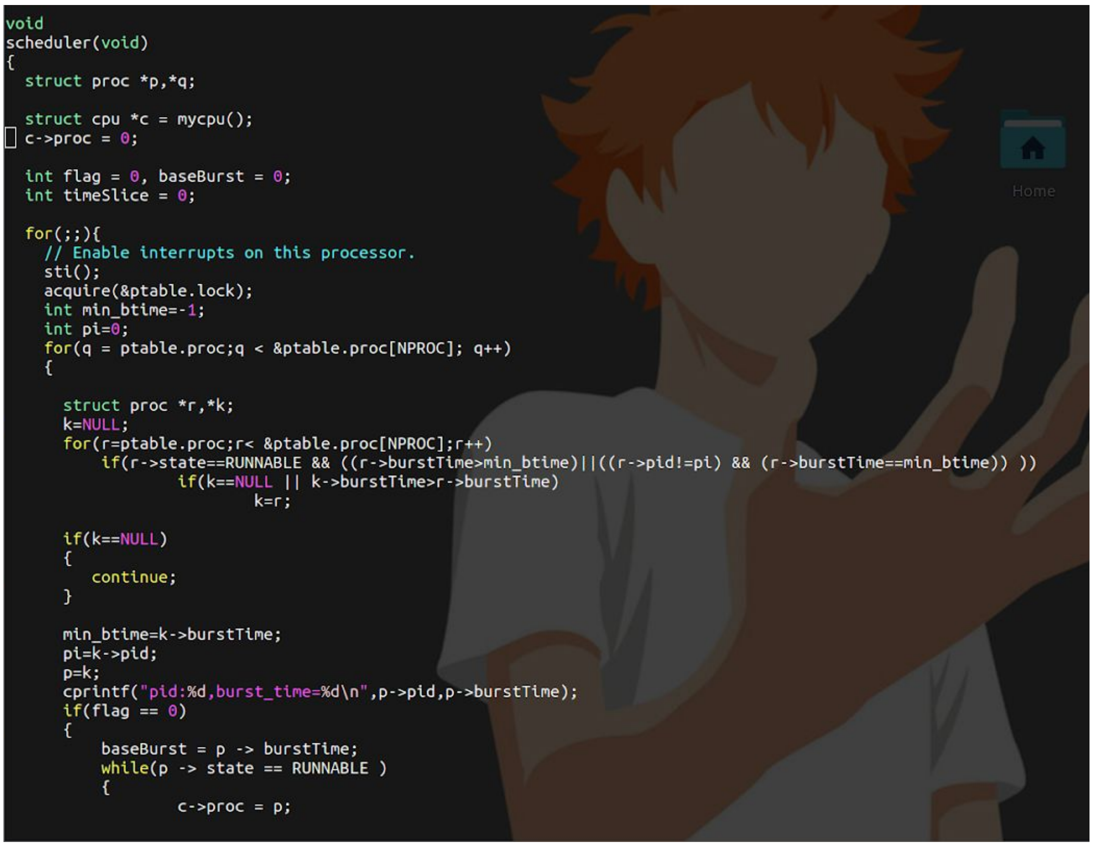
BONUS: Hybrid Scheduling

All other files are kept same as SJF except the scheduler in proc.c and the test_scheduler program(the only change in test_scheduler is that burst time limit is changed from 20 to 200 , as implementation of SJF requires updation of burst times , and the outputs can be clearly seen with the increased limit as problem of burst time reducing to 0 does not occur).

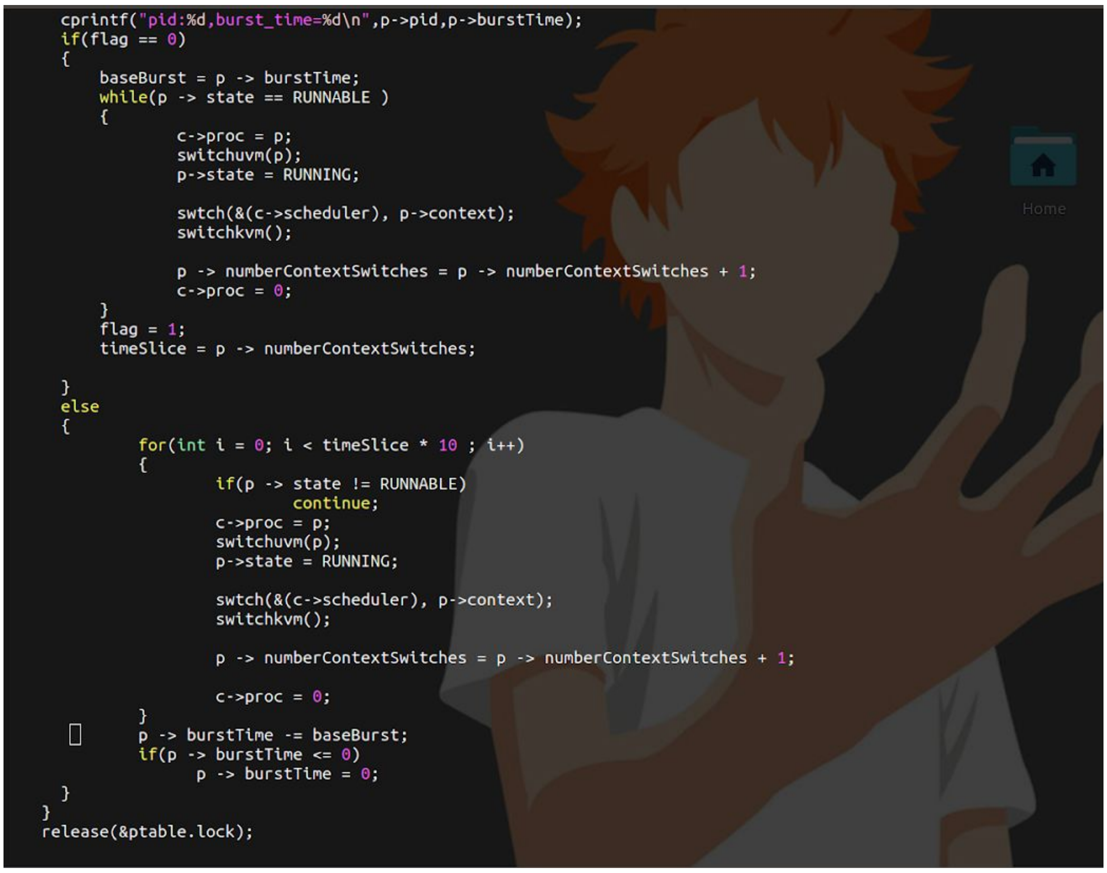
Scheduler

In this scheduler , two proc variables p and q are declared at the top . p is the process which will be scheduled in the given iteration , q is used to iterate once across the ptable. In every iteration , we use variable r to go through all the runnable processes once , and select the one which comes next to the previous scheduled process in the increasing order of burst times. This process is stored in the k proc variable which is later made equal to p.

We decided to define the slice of time given to one process as 10 quanta . The flag value defines whether it's the first process that is being executed(0 in that case) or not(1 in that case). The first process is used to set the variables baseBurst and timeslice(which come out to be 5 and 1). baseBurst is the burst time of the primary process and this is the amount that is reduced from a process's burst Time every time it finishes its given time. Also timeslice is just 1 quanta as first process goes to sleep after single context switch initially . Hence 10* timeslice(10 quanta) is the time given to a single process.

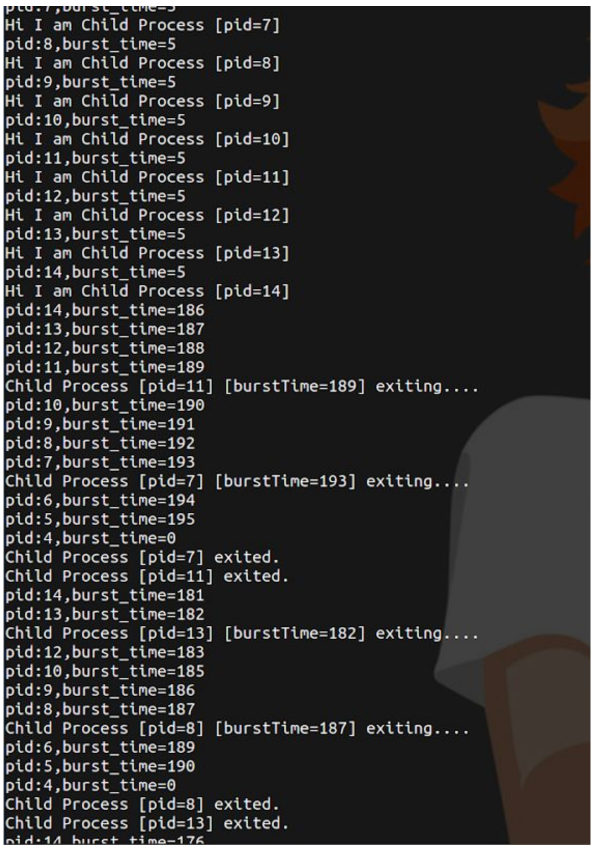


```c
void
scheduler(void)
{
  struct proc *p,*q;

  struct cpu *c = mycpu();
  c->proc = 0;

  int flag = 0, baseBurst = 0;
  int timeSlice = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();
    acquire(&ptable.lock);
    int min_btime=-1;
    int pi=0;
    for(q = ptable.proc;q < &ptable.proc[NPROC]; q++)
    {

      struct proc *r,*k;
      k=NULL;
      for(r=ptable.proc;r< &ptable.proc[NPROC];r++)
          if(r->state==RUNNABLE && ((r->burstTime>min_btime)||((r->pid!=pi) && (r->burstTime==min_btime)) ))
                  if(k==NULL || k->burstTime>r->burstTime)
                          k=r;

      if(k==NULL)
      {
         continue;
      }

      min_btime=k->burstTime;
      pi=k->pid;
      p=k;
      cprintf("pid:%d,burst_time=%d\n",p->pid,p->burstTime);
      if(flag == 0)
      {
          baseBurst = p -> burstTime;
          while(p -> state == RUNNABLE )
          {
                  c->proc = p;
```



```
  cprintf("pid:%d,burst_time=%d\n",p->pid,p->burstTime);
  if(flag == 0)
  {
      baseBurst = p -> burstTime;
      while(p -> state == RUNNABLE )
      {
              c->proc = p;
              switchuvm(p);
              p->state = RUNNING;

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

              p -> numberContextSwitches = p -> numberContextSwitches + 1;
              c->proc = 0;
      }
      flag = 1;
      timeSlice = p -> numberContextSwitches;

  }
  else
  {
          for(int i = 0; i < timeSlice * 10 ; i++)
          {
                  if(p -> state != RUNNABLE)
                          continue;
                  c->proc = p;
                  switchuvm(p);
                  p->state = RUNNING;

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

                  p -> numberContextSwitches = p -> numberContextSwitches + 1;

                  c->proc = 0;
          }
          p -> burstTime -= baseBurst;
          if(p -> burstTime <= 0)
                p -> burstTime = 0;
  }
}
release(&ptable.lock);
```

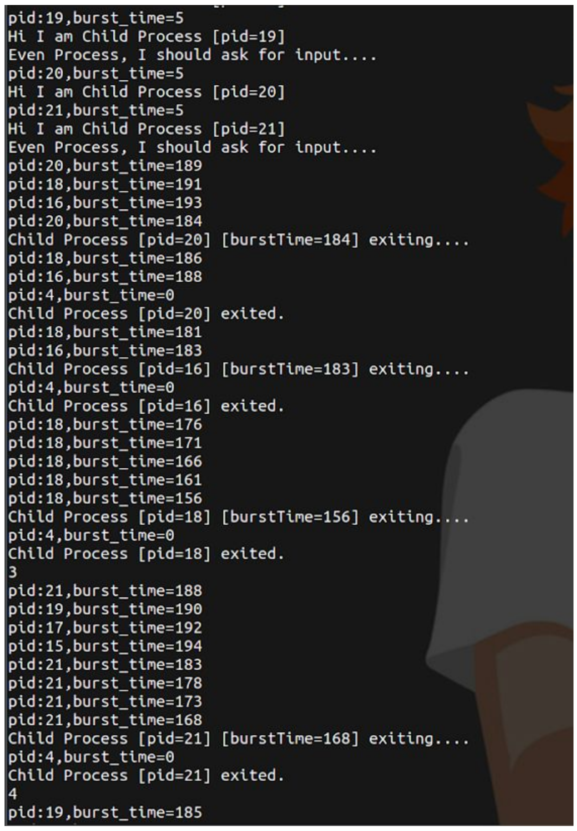
TestCase1(It consists of CPU bound processes)



```
pid:7,burst_time=5
Hi I am Child Process [pid=7]
pid:8,burst_time=5
Hi I am Child Process [pid=8]
pid:9,burst_time=5
Hi I am Child Process [pid=9]
pid:10,burst_time=5
Hi I am Child Process [pid=10]
pid:11,burst_time=5
Hi I am Child Process [pid=11]
pid:12,burst_time=5
Hi I am Child Process [pid=12]
pid:13,burst_time=5
Hi I am Child Process [pid=13]
pid:14,burst_time=5
Hi I am Child Process [pid=14]
pid:14,burst_time=186
pid:13,burst_time=187
pid:12,burst_time=188
pid:11,burst_time=189
Child Process [pid=11] [burstTime=189] exiting....
pid:10,burst_time=190
pid:9,burst_time=191
pid:8,burst_time=192
pid:7,burst_time=193
Child Process [pid=7] [burstTime=193] exiting....
pid:6,burst_time=194
pid:5,burst_time=195
pid:4,burst_time=0
Child Process [pid=7] exited.
Child Process [pid=11] exited.
pid:14,burst_time=181
pid:13,burst_time=182
Child Process [pid=13] [burstTime=182] exiting....
pid:12,burst_time=183
pid:10,burst_time=185
pid:9,burst_time=186
pid:8,burst_time=187
Child Process [pid=8] [burstTime=187] exiting....
pid:6,burst_time=189
pid:5,burst_time=190
pid:4,burst_time=0
Child Process [pid=8] exited.
Child Process [pid=13] exited.
pid:14,burst_time=176
```

Here processes from pid 4 to pid 14 are child processes. In every iteration they are picked by the scheduler in increasing order of their burst times and then their burst times are reduced by 5. For example in the first iteration after setting burst times , pid 14 with least burst time of 186 is picked, then pid 13 with burst time of 187 and so on till pid 5 with burst time 195. In the next iteration burst time of pid 14 process is reduced to 181(186-5) , and the processes are scheduled again in increasing order of their burst times which are now reduced by 5 (except pid 11 which has already exited).
So here what is happening is the scheduler is picking the processes by burst time order , giving them 10 quanta of time , reduces their burst time by 5 and moves onto the next process continuing in a cycle. In between the processes which get completed get exited.

Testcase2(It consists of both I/O bound and CPU bound processes)



```
pid:19,burst_time=5
Hi I am Child Process [pid=19]
Even Process, I should ask for input....
pid:20,burst_time=5
Hi I am Child Process [pid=20]
pid:21,burst_time=5
Hi I am Child Process [pid=21]
Even Process, I should ask for input....
pid:20,burst_time=189
pid:18,burst_time=191
pid:16,burst_time=193
pid:20,burst_time=184
Child Process [pid=20] [burstTime=184] exiting....
pid:18,burst_time=186
pid:16,burst_time=188
pid:4,burst_time=0
Child Process [pid=20] exited.
pid:18,burst_time=181
pid:16,burst_time=183
Child Process [pid=16] [burstTime=183] exiting....
pid:4,burst_time=0
Child Process [pid=16] exited.
pid:18,burst_time=176
pid:18,burst_time=171
pid:18,burst_time=166
pid:18,burst_time=161
pid:18,burst_time=156
Child Process [pid=18] [burstTime=156] exiting....
pid:4,burst_time=0
Child Process [pid=18] exited.
3
pid:21,burst_time=188
pid:19,burst_time=190
pid:17,burst_time=192
pid:15,burst_time=194
pid:21,burst_time=183
pid:21,burst_time=178
pid:21,burst_time=173
pid:21,burst_time=168
Child Process [pid=21] [burstTime=168] exiting....
pid:4,burst_time=0
Child Process [pid=21] exited.
4
pid:19,burst_time=185
```

Here the processes with pid 20,16,18 are the CPU bound processes while processes with pid 15,17,19,21 are the I/O bound ones. The CPU bound processes are the ones in runnable state after setting of burst times hence they are executed first and that too in the increasing order of their burst times. In the first iteration process with pid 20 , which is having least burst time of 189 is executed first (it is given time of 10 quanta , its burst time is decreased by 5 and it is preempted).. It is followed by pid 18 having burst time 191 and then pid 16 having burst time 193 . In the next iteration burst times of all of them are reduced by 5 (184,186,188 as reflected in the screenshots).
For I/O bound processes, when the first input is given all the processes waiting for input (21,19,17,15 pid) become runnable as wakeup call is issued on input channel by the OS.

Now pid 21 is picked first as its burst time is least and it takes the console input. Then after pid 21 is preempted and 19,17,15 are run , they find no input and hence again go to the waiting stage . Now after the first cycle ,as pid 21 is the only runnable process left,it is scheduled again and again until it exits on its own . Same happens with the pid 19 process when next input is given.