

ASSIGNMENT 2(B)

Group 2:-

- 1) Ritish Bansal 190101076
- 2) Suryansh Singh 190101089
- 3) Anant Shankhdhar 190101011
- 4) Mayank Chandak 190101052

Task1: Scheduling

Locating the current policy

The default policy for scheduling in xv6 is **Round Robin Scheduling**. We can see it from the **proc.c** file

```
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

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

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        #ifdef DEFAULT

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

            // Switch to chosen process. It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            c->proc = p;
            switchvm(p);
            p->state = RUNNING;

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

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

For the default scheduling policy, a process runs for 1 clock tick, followed by execution of the first process which is currently in the runnable state

- 1) The process selected by the policy is the first process in the ready queue, i.e the first process which is in RUNNABLE state
- 2) Whenever a process returns from I/O it waits for the scheduler to schedule it .
- 3) Whenever a new process using the fork() system call, the policy first changes its state from embryo to RUNNABLE, and then it waits for the scheduler to be scheduled.
- 4) Scheduling happens whenever we perform the sleep() or wakeup() system call or call the yield function. Also, whenever an IRQTIMER interrupt occurs, the yield function is called which causes scheduling

Enabling preemption to occur after every quanta size:-

We first define QUANTA in the **param.h** file

```
#define QUANTA 5
```

In the file **proc.h** we add a new parameter known as **scheduletime** for a process that gives the time it was scheduled

```
int scheduletime;
```

In the file **traps.c** we need to add the case when the time for next scheduling exceeds the ticks. For this case we will need to perform the **yield()** system call.

```
// 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)
{
    #ifdef DEFAULT
    //cprintf("THIS IS DEFAULT\n");
    if(myproc()->scheduletime + QUANTA < ticks)
    {
        yield();
    }
}
```

Adding SCHEDFLAG:-

To add the SCHEDFLAG option while performing make, we edit the **Makefile**

```
ifndef SCHEDFLAG
SCHEDFLAG := DEFAULT
endif

CC = $(TOOLPREFIX)gcc
AS = $(TOOLPREFIX)gas
LD = $(TOOLPREFIX)ld
OBJCOPY = $(TOOLPREFIX)objcopy
OBJDUMP = $(TOOLPREFIX)objdump
CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer
CFLAGS += $(shell $(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1 && echo -fno-stack-protector -D $(SCHEDFLAG) )
ASFLAGS = -m32 -gdwarf-2 -Wa,-divide
# FreeBSD ld wants ``elf_i386_fbsd''
LDFLAGS += -m $(shell $(LD) -V | grep elf_i386 2>/dev/null | head -n 1)
```

Scheduling Algorithms:-

1)Default (Round Robin) Scheduling:-

The changes regarding introducing QUANTA have been made above

2)First come first serve(FCFS) Scheduling:-

To implement FCFS we edit the scheduler function and add the case for FCFS

```

#elif FCFS
//cprintf("I AM FCFS");
struct proc *p;
struct proc *minP = 0;
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == RUNNABLE){
        if(minP != 0){
            if(p->ctime < minP->ctime){
                minP = p;
                //cprintf("pid_min: %d | ctime_min: %d\n", minP->pid, minP->ctime);
            }
        }
        else{
            minP=p;
            //cprintf("pid_min: %d | ctime_min: %d\n", minP->pid, minP->ctime);
        }
    }
}

if(minP!=0){
    p=minP;
    c->proc=p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    switchkvm();
    c->proc=0;
    //cprintf("pid: %d\n", p->pid);
}
}

```

To implement FCFS, we iterate through all the processes and find the process which has the least creation time. That particular process is the one which is run .

3)Static Multilevel Queue(SML) scheduling:-

To implement the SML we edited the following files:-

1)proc.h:-

1. Added the priority parameter to the proc struct which defines the priority of the process

```
int priority;
```

2. Defined the function struct proc* prioritySearch(int *index1, int *index2, int *index3, int *priority); which will be used to select the next process to be run after switching

```
struct proc* prioritySearch(int *index1, int *index2, int *index3, int *priority);
```

2)proc.c:-

1. Initialised 2 priority for a process in allocproc()

```

p->stime = 0;
p->retime = 0;
p->rtime = 0;
p->priority = 2;

```

2. Inside the **fork()** function whenever a child is created , we set its priority to the priority of its parent and its state is changed from embryo to runnable

```
np->priority=np->parent->priority;
np->state = RUNNABLE;
```

3. Now we need to implement the function `struct proc* prioritySearch(int *index1, int *index2, int *index3, int *priority)`; which we had defined to pick the next process. In the function we start with priority 3 and search for a process with that priority by iterating through the ptable . If no such process is found then we decrease the priority by 1 and search for a process with that priority. If we are not able to find any process with priority 1, 2 or 3 we return 0. Three different indices are used for searching in order to prevent duplicate processes from being selected.

```
struct proc* prioritySearch(int *index1, int *index2, int *index3, int *priority)
{
    struct proc* temp;

    for(;*priority>0;*priority=*priority-1)
    {
        for(int i=0;i<NPROC;i++)
        {
            temp=&ptable.proc[( *index1+i)%NPROC];
            if(temp->state == RUNNABLE && temp->priority == *priority)
            {
                *index1=( *index1+1+i) % NPROC;
                return temp;
            }
        }
    }

    return 0;
}
```

4. Now in the **void scheduler(void)** we need to add the case for scheduling for SML . We first perform the prioritySearch function to get the process and then

perform switching operation in order to start running the next process.

```
#elif SML
struct proc *p;
struct proc *highestP = 0;
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == RUNNABLE){
        if(highestP != 0){
            if(highestP->priority < p->priority){
                highestP = p;
                //cprintf("pid_min: %d | ctime_min: %d\n", minP->pid, minP->ctime);
            }
        }
        else{
            highestP=p;
            //cprintf("pid_min: %d | ctime_min: %d\n", minP->pid, minP->ctime);
        }
    }
}

if(highestP!=0){
    p=highestP;
    c->proc=p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    switchkvm();
    c->proc=0;
}
```

In SML the priorities can be set by the user . To perform that we need to add a new system call known as **set_prio(int priority)** . We edit the following files to implement it

1)proc.c :-

1. We add the function **int set_prio(int priority)** . If the priority is invlaid (i.e < 1 or > 3) we return 1 else we set the priority

```
int set_prio(int priority)
{
    if(priority < 1 || priority > 3)
    {
        return 1;
    }

    myproc()->priority=priority;
    return 0;
}
```

2)sysproc.c:-

1. We add the system call function **int sys_set_prio(void)**

```
int sys_set_prio(void)
{
    int priority;

    if(argint(0, &priority) < 0)
        return 1;

    return set_prio(priority);
}
```

3)syscall.h:-

1. defined SYS_set_prio as register 25

```
#define SYS_set_prio 25
```

4)syscall.c:-

1. defined the extern function and added set_prio to the array of system calls

```
extern int sys_set_prio(void);
```

```
[SYS_set_prio] sys_set_prio,
```

5)user.h:-

1. defined the **int set_prio(int priority)** function

```
int set_prio(int priority);
```

6)usys.S:-

1. added the set_prio syscall

```
35 SYSCALL(set_prio)
```

4)Dynamic Multi-level queue scheduling (DML)

In order to implement DML we edited the following files:-

1)proc.c:-

1. We add the case for DML in the scheduler. The scheduling will be similar to SML with the extra functionality that the priority will change dynamically. For scheduling will iterate through the ptable and find the next process to run using the

prioritySearch function followed by switching .

```
#elif DML
struct proc *p;
int priority=3;
p=prioritySearch(&index1, &index2, &index3, &priority);
if(p!=0)
{
c->proc=p;
switchvm(p);
p->state = RUNNING;
p->scheduletime=ticks;
swtch(&(c->scheduler), p->context);
switchkvm();
c->proc=0;
}

#endif
```

2. Whenever a process returns from I/O we need to set its priority to the highest value i.e 3. This will be done by editing the **wakeup1(void *chan)** function.

```
static void
wakeup1(void *chan)
{
    struct proc *p;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;
            #ifdef DML
                p->priority=3;
            #endif
}
```

2)exec.c:-

1. Whenever exec() is called we want the priority of the process to be changed to 2. To implement this we add the following line

```
#ifdef DML
curproc->priority = 2;
#endif
```

3)traps.c:-

1. If the process runs a full quanta then we need to decrease its priority. In order to perform this we check if the scheduletime+quanta is greater than ticks then we decrease the priority if possible.


```

//priority time to sleep
#elif DML
if(myproc()->scheduletime + QUANTA <= ticks)
{
    yield();
    if(myproc()->priority>1) myproc()->priority--;
}
#endif

```

Task 2 -Adding the yield system call

In order to add the yield system call we edited the following files:-

1)sysproc.c:-

1. Added the system call function **int sys_yield(void)**

```

int sys_yield(void)
{
    yield();
    return 0;
}

```

2)syscall.h:-

1. Added the definition for SYS_yield

```

#define SYS_yield 26

```

3)syscall.c:-

1. Added the extern function and added SYS_yield to array of system calls.

```

extern int sys_yield(void);

```

```

[SYS_yield] sys_yield,

```

4)user.h:-

1. Added the yield function to the system calls

```

int yield(void);

```

5)usys.S:-

1. Added SYSCALL(yield)

```

SYSCALL(yield)

```

Task 3.1- General Sanity Test

To implement the general sanity test we implemented the following file:-

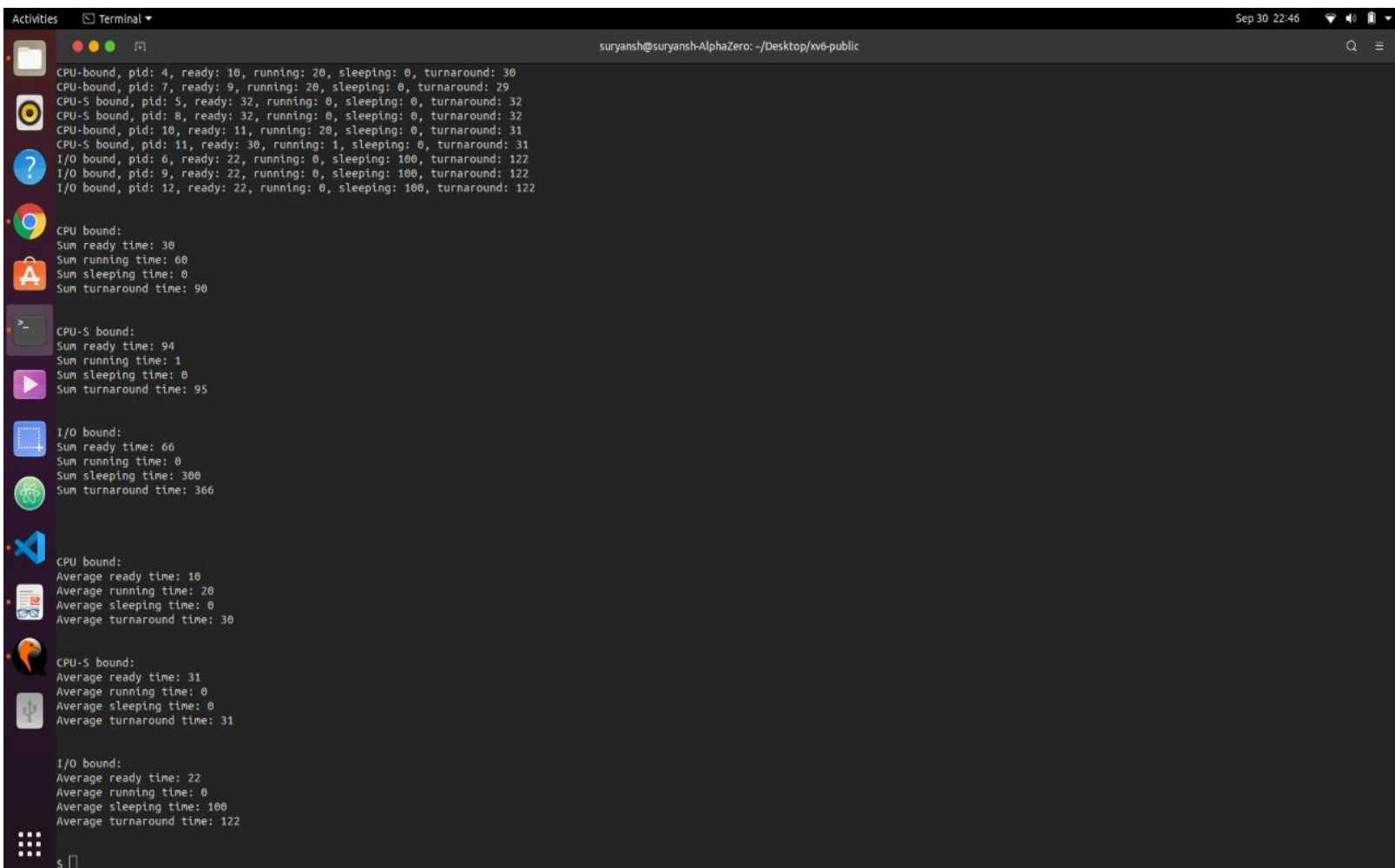
1)sanity.c:- The file contains the a c program for the sanity test which is available in the code.

To add the sanity user program we edited the **Makefile**. To do so we added the sanity command in the **UPROGS** as **_sanity/**

Results:-

The results for different types of scheduling were as follows:-

1)Default Scheduling:-



A terminal window titled 'Terminal' with the user 'suryansh@suryansh-AlphaZero' and the path '~/Desktop/xv6-public'. The terminal displays the output of a scheduling simulation. It lists individual process statistics for various process types (CPU-bound, CPU-S bound, I/O bound) and then provides summary statistics (Sum ready, running, sleeping, turnaround times) for each type. The process types and their respective statistics are as follows:

Process Type	Process ID	Ready	Running	Sleeping	Turnaround
CPU-bound	4	10	20	0	30
	7	9	20	0	29
CPU-S bound	5	32	0	0	32
	8	32	0	0	32
CPU-bound	10	11	20	0	31
	11	30	1	0	31
I/O bound	6	22	0	100	122
	9	22	0	100	122
	12	22	0	100	122

Process Type	Sum ready time	Sum running time	Sum sleeping time	Sum turnaround time
CPU bound:	30	60	0	90
CPU-S bound:	94	1	0	95
I/O bound:	66	0	300	366

Process Type	Average ready time	Average running time	Average sleeping time	Average turnaround time
CPU bound:	10	20	0	30
CPU-S bound:	31	0	0	31
I/O bound:	22	0	100	122

2)FCFS Scheduling:-

```
Activities Terminal ▾ Sep 30 22:47 suryansh@suryansh-AlphaZero: ~/Desktop/xv6-public

CPU-S bound, pid: 5, ready: 0, running: 0, sleeping: 0, turnaround: 0
CPU-S bound, pid: 8, ready: 4, running: 0, sleeping: 0, turnaround: 4
CPU-bound, pid: 7, ready: 0, running: 29, sleeping: 0, turnaround: 29
CPU-S bound, pid: 11, ready: 29, running: 0, sleeping: 0, turnaround: 29
CPU-bound, pid: 10, ready: 4, running: 31, sleeping: 0, turnaround: 35
CPU-bound, pid: 13, ready: 29, running: 22, sleeping: 0, turnaround: 51
I/O bound, pid: 6, ready: 30, running: 0, sleeping: 100, turnaround: 130
I/O bound, pid: 9, ready: 31, running: 0, sleeping: 100, turnaround: 131
I/O bound, pid: 12, ready: 33, running: 0, sleeping: 100, turnaround: 133

CPU bound:
Sum ready time: 33
Sum running time: 82
Sum sleeping time: 0
Sum turnaround time: 115

CPU-S bound:
Sum ready time: 33
Sum running time: 0
Sum sleeping time: 0
Sum turnaround time: 33

I/O bound:
Sum ready time: 94
Sum running time: 0
Sum sleeping time: 300
Sum turnaround time: 394

CPU bound:
Average ready time: 11
Average running time: 27
Average sleeping time: 0
Average turnaround time: 38

CPU-S bound:
Average ready time: 11
Average running time: 0
Average sleeping time: 0
Average turnaround time: 11

I/O bound:
Average ready time: 31
Average running time: 0
Average sleeping time: 100
Average turnaround time: 131

$
```

3)SML Scheduling:-

```
Activities Terminal ▾ Sep 30 22:48 suryansh@suryansh-AlphaZero: ~/Desktop/xv6-public

CPU-S bound, pid: 5, ready: 0, running: 0, sleeping: 0, turnaround: 0
CPU-S bound, pid: 8, ready: 3, running: 0, sleeping: 0, turnaround: 3
CPU-bound, pid: 7, ready: 0, running: 23, sleeping: 0, turnaround: 23
CPU-S bound, pid: 11, ready: 23, running: 0, sleeping: 0, turnaround: 23
CPU-bound, pid: 10, ready: 3, running: 22, sleeping: 0, turnaround: 25
CPU-bound, pid: 13, ready: 23, running: 23, sleeping: 0, turnaround: 46
I/O bound, pid: 6, ready: 22, running: 0, sleeping: 100, turnaround: 122
I/O bound, pid: 9, ready: 24, running: 0, sleeping: 100, turnaround: 124
I/O bound, pid: 12, ready: 25, running: 0, sleeping: 100, turnaround: 125

CPU bound:
Sum ready time: 26
Sum running time: 68
Sum sleeping time: 0
Sum turnaround time: 94

CPU-S bound:
Sum ready time: 26
Sum running time: 0
Sum sleeping time: 0
Sum turnaround time: 26

I/O bound:
Sum ready time: 71
Sum running time: 0
Sum sleeping time: 300
Sum turnaround time: 371

CPU bound:
Average ready time: 8
Average running time: 22
Average sleeping time: 0
Average turnaround time: 30

CPU-S bound:
Average ready time: 8
Average running time: 0
Average sleeping time: 0
Average turnaround time: 8

I/O bound:
Average ready time: 23
Average running time: 0
Average sleeping time: 100
Average turnaround time: 123

$
```

4)DML Scheduling-

```
Activities Terminal ▾ Sep 30 22:49 suryansh@suryansh-AlphaZero: ~/Desktop/xv6-public

CPU-S bound, pid: 5, ready: 1, running: 0, sleeping: 0, turnaround: 1
CPU-S bound, pid: 8, ready: 10, running: 0, sleeping: 0, turnaround: 10
CPU-bound, pid: 10, ready: 3, running: 29, sleeping: 0, turnaround: 32
CPU-S bound, pid: 11, ready: 9, running: 0, sleeping: 0, turnaround: 9
CPU-bound, pid: 13, ready: 8, running: 28, sleeping: 0, turnaround: 36
CPU-bound, pid: 7, ready: 26, running: 27, sleeping: 0, turnaround: 53
I/O bound, pid: 6, ready: 23, running: 0, sleeping: 100, turnaround: 123
I/O bound, pid: 9, ready: 23, running: 0, sleeping: 100, turnaround: 123
I/O bound, pid: 12, ready: 24, running: 0, sleeping: 100, turnaround: 124

CPU bound:
Sum ready time: 37
Sum running time: 84
Sum sleeping time: 0
Sum turnaround time: 121

CPU-S bound:
Sum ready time: 20
Sum running time: 0
Sum sleeping time: 0
Sum turnaround time: 20

I/O bound:
Sum ready time: 70
Sum running time: 0
Sum sleeping time: 300
Sum turnaround time: 370

CPU bound:
Average ready time: 12
Average running time: 28
Average sleeping time: 0
Average turnaround time: 40

CPU-S bound:
Average ready time: 6
Average running time: 0
Average sleeping time: 0
Average turnaround time: 6

I/O bound:
Average ready time: 23
Average running time: 0
Average sleeping time: 100
Average turnaround time: 123

$ suryansh@suryansh-AlphaZero:~/Desktop/xv6-public$
```

Task 3.2 - SML Sanity Test:-

To implement the SML sanity test we implemented the following file:-

1)SMLsanity.c:- The file contains the a c program for the sanity test which is available in the code.

To add the sanity user program we edited the **Makefile**. To do so we added the sanity command in the **UPROGS** as **_SMLsanity/**

Results:-

The results for different types of scheduling were as follows:-

SML Scheduling:-

```
Activities Terminal
suryansh@suryansh-AlphaZero: ~/Desktop/xv6-public

pid: 32 | ctime: 640
pid: 33 | ctime: 640
Priority 1, pid: 4, ready: 0, running: 3, sleeping: 0, turnaround: 3
Priority 2, pid: 5, ready: 3, running: 2, sleeping: 0, turnaround: 5
Priority 3, pid: 6, ready: 4, running: 3, sleeping: 0, turnaround: 7
Priority 2, pid: 8, ready: 7, running: 2, sleeping: 0, turnaround: 9
Priority 3, pid: 9, ready: 9, running: 3, sleeping: 0, turnaround: 12
Priority 2, pid: 11, ready: 10, running: 3, sleeping: 0, turnaround: 13
Priority 3, pid: 12, ready: 11, running: 2, sleeping: 0, turnaround: 13
Priority 2, pid: 14, ready: 13, running: 3, sleeping: 0, turnaround: 16
Priority 3, pid: 15, ready: 14, running: 3, sleeping: 0, turnaround: 17
Priority 2, pid: 17, ready: 15, running: 3, sleeping: 0, turnaround: 18
Priority 3, pid: 18, ready: 16, running: 3, sleeping: 0, turnaround: 19
Priority 2, pid: 20, ready: 18, running: 2, sleeping: 0, turnaround: 20
Priority 3, pid: 21, ready: 19, running: 2, sleeping: 0, turnaround: 21
Priority 2, pid: 23, ready: 20, running: 2, sleeping: 0, turnaround: 22
Priority 3, pid: 24, ready: 21, running: 3, sleeping: 0, turnaround: 24
Priority 2, pid: 26, ready: 22, running: 3, sleeping: 0, turnaround: 25
Priority 3, pid: 27, ready: 23, running: 3, sleeping: 0, turnaround: 26
Priority 2, pid: 29, ready: 24, running: 3, sleeping: 0, turnaround: 27
Priority 3, pid: 30, ready: 25, running: 3, sleeping: 0, turnaround: 28
Priority 2, pid: 32, ready: 27, running: 3, sleeping: 0, turnaround: 30
Priority 3, pid: 33, ready: 29, running: 2, sleeping: 0, turnaround: 31
Priority 1, pid: 7, ready: 39, running: 2, sleeping: 0, turnaround: 41
Priority 1, pid: 10, ready: 39, running: 2, sleeping: 0, turnaround: 41
Priority 1, pid: 13, ready: 40, running: 3, sleeping: 0, turnaround: 43
Priority 1, pid: 16, ready: 40, running: 2, sleeping: 0, turnaround: 42
Priority 1, pid: 19, ready: 41, running: 3, sleeping: 0, turnaround: 44
Priority 1, pid: 22, ready: 41, running: 3, sleeping: 0, turnaround: 44
Priority 1, pid: 25, ready: 43, running: 2, sleeping: 0, turnaround: 45
Priority 1, pid: 28, ready: 42, running: 3, sleeping: 0, turnaround: 45
Priority 1, pid: 31, ready: 44, running: 2, sleeping: 0, turnaround: 46

Priority 1:
Average ready time: 12
Average running time: 0
Average sleeping time: 0
Average turnaround time: 12

Priority 2:
Average ready time: 5
Average running time: 0
Average sleeping time: 0
Average turnaround time: 5

Priority 3:
Average ready time: 5
Average running time: 0
Average sleeping time: 0
Average turnaround time: 5

$
```

Here we can see that initially priority 1 process started but when priority 2 process has come all the priority 1 processes are preempted and similarly when process of priority 3 comes it preempt all other processes. And when this process is completed again priority 2 process runs and this thing go on until all priority 3 processes are completed and then priority 2 processes are completed and at last priority 1 processes are completed. This order also depends on run time of process and compiling time of process and also how we change priority of process by set_prio system call.

To apply the patch command is - **git apply --reject --whitespace=fix patch.txt**