CS 344 Assignment 2B Report

Group No 6:

Aditya Trivedi 190101005
Akshat Arun 190101007
Atharva Vijay Varde 190101018
Shashwat Sharma 190123055

Applying changes and executing:

1) Using Patch file:
   - Copy the file patch.txt into “xv6-public” folder
   - `cd xv6_public`
   - `git apply patch.txt`
   - `make clean && make qemu SCHEDFLAG=Z`
     where Z = DEFAULT, FCFS, SML, DML

2) Using modified files:
   - Copy the files in the “modified files” folder and paste/replace into “xv6-public” folder.
   - `make clean && make qemu SCHEDFLAG=Z`
     where Z = DEFAULT, FCFS, SML, DML

Task 1: Scheduling

We define 5 as the quanta in param.h

```
#define QUANTA       5  //The time quantum for Default scheduling algorithm
```

Policy 1: Default
Here we want to implement the default scheduling algorithm. Context switching happens after every 5 ticks.
So we do not change the Default scheduling code a lot and just add an ifdef in the scheduler method.
In trap.c where yield() is called after every tick, we add an extra if condition that yield() the current process if ticks%5==0.

```
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER){
   #ifdef DEFAULT
     uint tempTicks;//declaring a temporary variable to store the current value of ticks
     acquire(&tickslock);
       tempTicks=ticks;
     release(&tickslock);
     if(tempTicks%QUANTA==0){yield();}//we yield only if ticks is divisible by 5
```

Policy 2: FCFS

We need to implement a first come first serve scheduling algorithm. We do this in proc.c in the scheduler() function.
In FCFS that process is picked first whose creation time is the least, so we create a dummy integer variable minCtime which stores the current minimum creation time and initialise it with -1 (If in the end, the value is still -1 then it means that the process table does not has any process which is runnable)
We declare a dummy variable temp initialised to NULL pointer which stores the required process whose creation time was minimum. minCtime is updated accordingly.
In FCFS a process cannot be preempted out just because of the priority of some other process or because of time quantum expiration. Hence in trap.c we remove the yield method inside the ifdef section of FCFS.

In proc.c for FCFS:

```
  #ifdef FCFS
      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);
              int minCtime=-1;//It store the min creation time among all processes
              struct proc *temp = 0;//temporary process variable to store the required process which will be
scheduled now
              for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
                  if(p->state!=RUNNABLE){continue;}
                  if(minCtime==-1){minCtime=p->ctime;temp=p;}
                  else{
                      if((minCtime)>(p->ctime)){minCtime=p->ctime;temp=p;}
                  }
              }
              if(minCtime!=-1){//if minCtime = -1 then no process should be scheduled now
                  c->proc=temp;
                  switchuvm(temp);
                  temp->state=RUNNING;
                  swtch(&(c->scheduler), temp->context);
                  switchkvm();
                  c->proc = 0;
              }
          release(&ptable.lock);
      }
   #else
```

In trap.c for fcfs:

```
#ifdef FCFS
      //No need of yield()
#else
```

Policy 3: SML

We add priority as a parameter in the process struct in proc.h

```
int priority;                //Denotes the priority of the process
```

When the initial shell process is created by userinit() then we define its priority as 2 in the body of userinit() method.

```
      p->priority=2;
```

When a child process is forked from a parent process then we assign the same priority to the child which its parent had.

```
  np->priority = curproc->priority;
```

To implement the SML scheduling algorithm we define 3 index variables for each of the priorities: i1,i2,i3:

- i1 stores the index of the last priority 1 process which was executed
- Similarly, i2 and i3 correspond for priority2 and 3 respectively.

We firstly check from i3 till i3-1 cyclically i.e iterate from i3 till NPROC-1 and then from 0 till i3-1.
Checking is done whether we have a process with priority 3 or not. If we have then we update i3 and start executing the returned process.
If we do not get any priority 3 process in the process table, we check for priority 2 processes and then priority 1 processes.
We also have a flag variable initialised to 1 at the start of every iteration of the infinite for loop. It is changed to 0 when we find a process. This is done so that the scheduler starts executing again when the context is returned back to the scheduler by firstly checking for priority 3 processes and for 2 and so on.

Code in proc.c:

```
#ifdef SML
      int i1=0,i2=0,i3=0;// index variables for the 3 priorities
      struct proc *p;
      struct cpu *c = mycpu();
      c->proc = 0;
      for (;;) {
          int flag=1;//a flag variable so that the scheduler always checks for priority 3
and then for 2 and then 1
          // Enable interrupts on this processor.
          sti();

          // Loop over process table looking for process to run.
          acquire(&ptable.lock);
              for (int i=i3;i<NPROC+i3  && flag==1; i++) {
                  struct proc *temp = &ptable.proc[i%NPROC];//modulo is done to check
cyclically
```

```
                if(temp->state!=RUNNABLE || temp->priority!=3){continue;}
                flag=0;//update the flag as we found a required process
                p=temp;
                i3=(i+1)%NPROC;//updating i3 with its new index value
                c->proc = p;
                switchuvm(p);
                p->state = RUNNING;
                swtch(&(c->scheduler), p->context);
                switchkvm();
                c->proc = 0;
            }
            for (int i=i2;i<NPROC+i2  && flag==1; i++) {
                struct proc *temp = &ptable.proc[i%NPROC];
                if(temp->state!=RUNNABLE || temp->priority!=2){continue;}
                flag=0;
                p=temp;
                i2=(i+1)%NPROC;
                c->proc = p;
                switchuvm(p);
                p->state = RUNNING;
                swtch(&(c->scheduler), p->context);
                switchkvm();
                c->proc = 0;
            }
            for (int i=i1;i<NPROC+i1  && flag==1; i++) {
                struct proc *temp = &ptable.proc[i%NPROC];
                if(temp->state!=RUNNABLE || temp->priority!=1){continue;}
                flag=0;
                p=temp;
                i1=(i+1)%NPROC;
                c->proc = p;
                switchuvm(p);
                p->state = RUNNING;
                swtch(&(c->scheduler), p->context);
                switchkvm();
                c->proc = 0;
            }
        release(&ptable.lock);
    }
#else
```

We modify the following files (just like assignment 1) to add the set_prio system call

- syscall.h
- syscall.c
- user.h
- usys.S

Implementation of set priority function in sysproc.c :

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

```
 if(argint(0, &priority)<0){return 1;};//fetching the arguements
 (myproc()->priority) = priority;//changing the priority of the current
process to the desired priority
 return 0;
}
```

However this system call is only used by SML, hence we wrap it by ifdefs. This is done to ensure that the priority cannot be changed manually in DML

```
#ifdef SML//so that DML cannot access the set prio method and
cannot change priority manually
   int set_prio(int);
#endif
```

Policy 4: DML

Firstly we modify the exec system call to reset the priority of all processes to 2

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

In proc.h we define a new parameter called as shedAt which tells the tick time when a particular process was scheduled.

```
 int stime;                  // Time when SLEEPING
 int retime;                 // Time when RUNNABLE
 int rutime;                 // Time when RUNNING
 int priority;               //Denotes the priority of the process
 uint shedAt;
};
```

We modify trap.c to implement DML
We reduce the priority (but don't change it if it's already 1) whenever a process runs for a complete quanta, so we check whether the current ticks (the current running process's scheduled time) is a multiple of 5 and also greater than 0.
We yield if a quanta has been completed.
proc.c

```
#ifdef DML
     uint tempTicks;
     acquire(&tickslock);
       tempTicks=ticks;
     release(&tickslock);
```

```
    if((tempTicks-(myproc())->shedAt)>0 &&
(tempTicks-(myproc())->shedAt)%QUANTA==0){
      // Reduce Priority, but do not make it below 1
      int newPriority = ((myproc())->priority)-1;
      if(newPriority!=0){
        (myproc())->priority = newPriority;
      }
    }
    if(tempTicks%QUANTA==0){
      yield();
    }
  #endif
```

Implementation is similar to SML, except now we also update the shedAt parameter of the scheduled process. So we add this line inside userinit() method when it initialises the first process :

```
  p->shedAt=ticks;
```

We also add this line in the 3 for-loops when we found the process to be scheduled.

Makefile modifications :

To allow compilation using SCHEDFLAG, we take the value of the variable SCHEDFLAG from the command, and use the -D option of gcc to define a Macro of the same name in the compilation.

```
CFLAGS += -D $(SCHEDFLAG)
```

To handle default case :

```
ifndef SCHEDFLAG
SCHEDFLAG := DEFAULT
endif
```

Task 2: Yield

We modify the following files (just like assignment 1)

- syscall.h
- syscall.c
- user.h
- usys.S

The yield system call is implemented as follows in sysproc.c

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

Task 3: Sanity Test

Policy 1: Default

Output: Quanta of 5 ticks

```
$ sanity 1
pid:6 CPU w:0 ru:11 io:0
pid:4 S-CPU w:0 ru:14 io:0
pid:5 IO w:0 ru:0 io:100
Group:CPU: slp:0, rdy:0, trnarnd:11
Group:S-CPU: slp:0, rdy:0, trnarnd:14
Group:IO: slp:100, rdy:0, trnarnd:100
```

Detailed explanation :

We have two processors in xv6, let they be c0 and c1. Upon execution, the parent process creates 3 children processes:

- S(Short job CPU type process)
- C(CPU type process without yields)
- IO(Process which calls sleep after each iteration)

in increasing order of their creation time respectively.

When the timer begins c0 starts processing and picks the first process which is runnable and exists in the process table, here it is S.
Similarly, c1 picks up IO to process. But because IO has a sleep call after every iteration, as soon as it starts executing, sleep is called.
This one iteration happens very fast, so IO was not able to run for a complete tick and now c1 runs C.
When the first dummy loop of 10^6 iterations ends for S, yield() is called and c0 chooses the next runnable instruction available inside the process table, which will be IO.

Again IO goes back to sleep very fast without running for at least a complete tick and again S is chosen by c0.
This continues and hence we conclude that the running time of IO will be 0 as it never runs for a complete tick and because 100 sleeps are called for IO hence its IO time will be 100 ticks.
The waiting time will also be zero because sleep(1) is called. This means sleep for 1 tick and then go to the ready queue. As soon as it goes to the ready queue and waits, one dummy cycle gets finished.
Sometimes, it might happen that the waiting time becomes 1 or 2 ticks because IO was in the ready queue waiting to get executed at the end of a tick cycle.
C and S do not have any sleep instructions, so they will never go to the waiting state. So their sleep time should be zero. Their wait time should also be zero, they rarely wait in the ready queue because IO is rarely executed. Both S and C are running the majority of the time.
During the interval when S and C wait in the queue, the tick cycle will end and then their wait time will be increased (however this case is very rare).
As n=1, the average values and individual values are identical. Also as
turnaround time = completion time - creation time
Hence turn around time will be equal to (run time + sleep time + wait time).

Output: Sanity 2

```
$ sanity 2
pid:9 CPU w:0 ru:15 io:0
pid:12 CPU w:2 ru:13 io:0
pid:13 S-CPU w:16 ru:12 io:0
pid:10 S-CPU w:14 ru:15 io:0
pid:8 IO w:10 ru:0 io:100
pid:11 IO w:10 ru:0 io:100
Group:CPU: slp:0, rdy:1, trnarnd:15
Group:S-CPU: slp:0, rdy:15, trnarnd:28
Group:IO: slp:100, rdy:10, trnarnd:110
```

Detailed explanation :
Now we call sanity with n=2. We have 6 instructions: IO1, C1, S1, IO2, C2, S2.
Similar to sanity 1, C1 and C2 don't wait for a complete tick cycle inside the ready queue because S and IO type processes take less than one tick to complete one dummy iteration.

Hence wait time for C1 and C2 is almost 0 and as they do not have any IO operations neither is sleep called explicitly for C1 and C2, hence their IO time is 0.
IO for S1 and S2 is 0 in similar fashion.
We also have average sleep ready and turnaround time which this time differs from individual times due to n being 2.

Policy 2: FCFS

Output : Sanity 1

```
$ sanity 1
pid:4 S-CPU w:0 ru:12 io:0
pid:6 CPU w:0 ru:11 io:0
pid:5 IO w:11 ru:0 io:100
Group:CPU: slp:0, rdy:0, trnarnd:11
Group:S-CPU: slp:0, rdy:0, trnarnd:12
Group:IO: slp:100, rdy:11, trnarnd:111
```

Detailed Explanation :
Here, the execution takes place as follows: P4 and P5 start executing.
However, P5 being an IO process calls sleep(1), allowing P6 to begin execution. Since the policy is non-preemptive, one CPU will keep running it.
For the other CPU, we observe that since P4 was created earlier than P5, P4 will always be rescheduled whenever it yields. Hence, after waking up from the first sleep() call, P5 remains in the ready queue until either of the processes complete.
Thus, we expect P4 and P6 to have near-zero wait-time (some might exist due to the testing process also running in the beginning). P5 shall have waiting time equal to the running time of whichever process ends quickly, and its IO-time shall be 100.

Output: Sanity 2

```
$ sanity 2
pid:4 S-CPU w:0 ru:21 io:0
pid:6 CPU w:1 ru:21 io:0
pid:7 S-CPU w:21 ru:21 io:0
pid:9 CPU w:21 ru:21 io:0
pid:5 IO w:21 ru:0 io:100
pid:8 IO w:41 ru:0 io:100
Group:CPU: slp:0, rdy:11, trnarnd:32
Group:S-CPU: slp:0, rdy:10, trnarnd:31
Group:IO: slp:100, rdy:31, trnarnd:131
```

Detailed Explanation :
Firstly, processes P4 and P5 shall start to run. As P5 is IO, it immediately goes to sleep, as a result, P6 starts running on that CPU. Since the policy is non-preemptive, P6 shall run until completion on that CPU.
For the other CPU, we observe that all other processes have a creation time higher than that of P4. Hence whenever P4 yields, it is scheduled right back again in accordance with FCFS policy.

We thus expect P4 and P6 to complete with almost zero waiting time (some +-1 due to the testing process also being executed in the beginning).
The waiting time for P5 IO: Since one CPU is occupied with P6, the other always prefers P4, after the first sleep call, the P5 remains in the ready state and thus counted as such. Thus, we expect P5 and other processes to have their waiting time at this stage be equal to the running time of P4 and P6. (which it is).

Next, we consider the remaining processes. Sorted in ascending order of creation time, they are as follows: IO, SCPU, IO, CPU.
A similar line of reasoning follows. P4 sleeps, which allows P7 to take the CPU.
The same happens with the other IO process and p9 takes the other CPU.
An important difference here is since P4 has the lowest creation time, whenever P7 yields, P4 will be executed and it shall set its state to sleep accordingly.
The last IO process, P8, will have a similar condition as P4 earlier, it will need to wait for the other processes to end (as P9 occupies one CPU, and the other processes have lower creation time). Hence, another round of the complete run-time will be added to its waiting time.

Policy 3: SML

Output : SMLsanity test

```
int main(void){
    set_prio(3);
    int n = 5;
    for(int i=0; i<3*n; ++i){
        if(fork() == 0){
            int pid = getpid();
            set_prio(pid % 3 + 1);
            for(int k = 0; k < 100; ++k)
                for (int j = 0; j < 1000000; ++j)
                    __asm__ volatile("" : "+g"(j) : :);
            exit();
        }
    }
    for(int i=0; i<3*n; ++i){
        printChildStats();
    }
    exit();
}
```

Detailed Explanation:
The test process forks 3*n children, where the children have their priority equal to their (process_id % 3) + 1.
All children run a dummy loop of 100 * 10^6 iterations and then terminate.

Finally, the testing process prints the process id, priority and the termination time of all the children.

```
$ SMLsanity
id:5 P:3 @ 659
id:8 P:3 @ 661
id:11 P:3 @ 662
id:14 P:3 @ 661
id:17 P:3 @ 665
id:4 P:2 @ 704
id:10 P:2 @ 704
id:13 P:2 @ 705
id:16 P:2 @ 706
id:7 P:2 @ 705
id:12 P:1 @ 738
id:15 P:1 @ 740
id:18 P:1 @ 744
id:9 P:1 @ 746
id:6 P:1 @ 747
```

The output is shown on the left, where id is the process id, P denotes the priority and the value to the right of the @ is the termination time.

We expect all processes of higher priority to terminate before the lower ones, and this matches with the observed values.

Policy 4: DML

Output: Sanity 1

```
$ sanity 1
pid:6 CPU w:0 ru:12 io:0
pid:4 S-CPU w:0 ru:13 io:0
pid:5 IO w:0 ru:0 io:100
Group:CPU: slp:0, rdy:0, trnarnd:12
Group:S-CPU: slp:0, rdy:0, trnarnd:13
Group:IO: slp:100, rdy:0, trnarnd:100
```

Detailed Explanation:
Here, the main insight is that the priority of CPU type process will decease to 1 after their initial run, while because S-CPU processes yield CPU, they never achieve a complete quanta of running, meaning their priority remains 2.
IO type processes will have their priority as 3 whenever they return from sleep.

In this case, P4 and P5 execute first. P5 goes to sleep, giving the CPU to P6. Now whenever the CPU is yielded by P4 or by P6 through quanta preemption, IO will always be executed due to its higher priority. Hence, IO will have no waiting time in this example, as is observed.

Output: Sanity 2

```
$ sanity 2
pid:10 S-CPU w:7 ru:16 io:0
pid:13 S-CPU w:6 ru:16 io:0
pid:9 CPU w:16 ru:17 io:0
pid:12 CPU w:16 ru:16 io:0
pid:8 IO w:12 ru:0 io:100
pid:11 IO w:12 ru:0 io:100
Group:CPU: slp:0, rdy:16, trnarnd:32
Group:S-CPU: slp:0, rdy:6, trnarnd:22
Group:IO: slp:100, rdy:12, trnarnd:112
```

Detailed Explanation:
Using the main insight of sanity 1, we develop the analysis further. The average turnaround time of SCPU processes will be lower than that of CPU processes due to higher priority and the fact that it does not decrease.

IO processes: The IO time will be 100 due to the 100 sleep(1) calls, and running time will be 0 as observed in the previous policies as well. The non-zero waiting time of IO processes despite higher priority is because, firstly, the initial priority of these processes is 2, and secondly, Whenever an IO process goes to sleep thereby giving the CPU to another process, it will need to wait for the full quanta before quanta-based pre-emption happens.

CPU-SPU: Since CPU processes ultimately get a lower priority, the 2 CPU processes lose preference with the two SCPU processes, which results in them waiting for the SCPU processes to end. We expect their waiting time(of the CPU processes) to be equal to running time of the SCPU processes. Additionally, the waiting time of CPU processes will be greater than that of SCPU.

The observed values match with the theoretical analysis in all sanity tests.