CS344 OS Lab
Assignment - 2B

Group 1:
Eshan Trehan - 190123070
Anuraag Mahajan - 190101110
Kshitij Singhal - 190123032
Hardik Suhag - 190101038

TASK 1:

Adding scheduling policies to xv6

The xv6 scheduler uses a straightforward scheduling mechanism that runs each task sequentially. This policy is called round robin.

A round-robin scheduler uses time-sharing to ensure that processes are scheduled equitably, assigning each job a time slot or quantum (its allocation of CPU time) and interrupting the job if it is not completed by that time. When a time slot is assigned to that process, the job is resumed.

The scheduler chooses the first process in the ready queue to execute if the process terminates or changes its state to waiting during its allocated time quantum. In the absence of time-sharing, or if the quanta were enormous in comparison to the work sizes, a process that created large jobs would be preferred. Round-robin scheduling is simple, straightforward, and devoid of famine.

```
#ifdef DEFAULT
  // Round Robin with time quanta = QUANTA (default = 5)
  // 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 && inctickcounter() == QUANTA){
    myproc()->tickcounter = 0;
    yield();
  }
#ifdef FCFS
// do not yield
#ifdef DML
  // 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 && inctickcounter() == QUANTA) {
    myproc()->tickcounter = 0;
    decpriority();
    yield();
  }
#ifdef SML
  // 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 && inctickcounter() == QUANTA) {
    myproc()->tickcounter = 0;
    yield();
  }
#endif
#endif
#endif
#endif
```

To define quanta we changed param.h and trap.c Files and gave it value of 5 such that preemption will be done every quanta(5 clock tick) instead of every clock tick.

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

We modified the Makefile to support SCHEDFLAG (a macro for quick compilation of the appropriate scheduling scheme).

Three more scheduling policies are included in addition with the existing DEFAULT policy.
The list of all the policies are as follows:
Policy 1 : Default Policy (SCHEDFLAG=DEFAULT)
Policy 2 : First come - First Served (SCHEDFLAG=FCFS)
Policy 3 : Multi-level queue scheduling (SCHEDFLAG=SML)
Policy 4 : Dynamic Multi-level queue scheduling (SCHEDFLAG=DML)

For this we changed below file-
Proc.c - Implemented code for every policy
Proc.h- Added variables like running time,sleeping time etc.

```
  uint ctime;
  int stime;
  int retime;
  int rutime;
  int priority;
  int tickcounter;
  char fake[8];
};
```

Modified struct proc in proc.h

Changes in Proc.c

```
    #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.
      cpu->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(cpu->scheduler), p->context);
      switchkvm();

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

    #else
```

Default Policy (SCHEDFLAG=DEFAULT)

The default scheduling policy is the Round Robin Scheduling as discussed above.Run `make qemu SCHEDFLAG=DEFAULT` in the terminal to invoke DEFAULT scheduling.

```
#ifdef FCFS
  struct proc* bezt_proc = &ptable.proc[0];
  int bezt_time = -1;

  for(p = ptable.proc; p<&ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE) continue;
    struct proc* curproc = p;
    if(bezt_time==-1){
      bezt_proc = curproc;
      bezt_time = curproc->ctime;
    }else if(curproc->ctime < bezt_time){
      bezt_proc = curproc;
      bezt_time = curproc->ctime;
    }
  }
  if(bezt_time != -1){
    p = bezt_proc;
    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    cpu->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&(cpu->scheduler), p->context);
    switchkvm();

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

Policy 2 : First come - First Served (SCHEDFLAG=FCFS)

The process that asks the CPU first is given the CPU first in the first-come, first-served (FCFS) scheduling method. A queue is used to implement this policy. When the CPU becomes available, it is given to the process at the front of the queue. After that, the currently executing process is removed from the queue. However, because the FCFS policy does not account for the burst timings of the processes, the average waiting time is often extremely long. It's possible that shorter processes will have to wait for larger processes to complete.

Run `make qemu SCHEDFLAG=FCFS` in the terminal to invoke FCFS scheduling.

```
    #ifdef SML
    struct proc* best_proc = 0;
    int best_time = -1;
    int best_priority = -1;
      for(p = ptable.proc; p<&ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE) continue;
        struct proc* curproc = p;
        if(best_time==-1){
          best_proc = curproc;
          best_priority = curproc -> priority;
          best_time = curproc->ctime;
        }else if(best_priority == curproc->priority){
          if(curproc->ctime < best_time){
            best_proc = curproc;
            best_priority = curproc -> priority;
            best_time = curproc->ctime;
          }
        }else if(curproc->priority > best_priority){
          best_proc = curproc;
          best_priority = curproc -> priority;
          best_time = curproc->ctime;
        }
      }
      if(best_time != -1){
        p = best_proc;
        // Switch to chosen process.  It is the process's job
        // to release ptable.lock and then reacquire it
        // before jumping back to us.
        cpu->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&(cpu->scheduler), p->context);
        switchkvm();
        // Process is done running for now.
        // It should have changed its p->state before coming back.
        cpu->proc = 0;
      }
    #else
```

Policy 3 : Multi-level queue scheduling (SCHEDFLAG=SML)

Here the best process is selected on the basis of two variables - best_priority and best_time. This is a lexicographic comparison where higher priority is given in order 3>2>1. And in case priority results in a tie, the process which was created earlier is given scheduled to enter the cpu.

Run `make qemu SCHEDFLAG=SML` in the terminal to invoke SML scheduling.

```
int set_prio(int priority) {
  if (priority < 1 || priority > 3)
    return -1;
  acquire(&ptable.lock);
  myproc()->priority = priority;
  release(&ptable.lock);
  return 0;
}
```

Added this helper function.
This function is called in the system call sys_set_prio() implemented in proc.c

Policy 4 : Dynamic Multi-level queue scheduling (SCHEDFLAG=DML)

```
#ifdef DML
/*
Finds the next ready process for DML scheduling
*/
struct proc* findreadyprocess(int *index1, int *index2, int *index3, uint *priority) {
  int i;
  struct proc* proc2;
notfound:
  for (i = 0; i < NPROC; i++) {
    switch(*priority) {
      case 1:
        proc2 = &ptable.proc[(*index1 + i) % NPROC];
        if (proc2->state == RUNNABLE && proc2->priority == *priority) {
          *index1 = (*index1 + 1 + i) % NPROC;
          return proc2; // found a runnable process with appropriate priority
        }
      case 2:
        proc2 = &ptable.proc[(*index2 + i) % NPROC];
        if (proc2->state == RUNNABLE && proc2->priority == *priority) {
          *index2 = (*index2 + 1 + i) % NPROC;
          return proc2; // found a runnable process with appropriate priority
        }
      case 3:
        proc2 = &ptable.proc[(*index3 + i) % NPROC];
        if (proc2->state == RUNNABLE && proc2->priority == *priority){
          *index3 = (*index3 + 1 + i) % NPROC;
          return proc2; // found a runnable process with appropriate priority
        }
    }
  }
  if (*priority == 1) {//did not find any process on any of the prorities
    *priority = 3;
    return 0;
  }
  else {
    *priority -= 1; //will try to find a process at a lower priority
    goto notfound;
  }
  return 0;
}
#endif
```

The helper function findreadyprocess() in proc.c finds the next ready process according to the dynamic multilevel scheduling (DML) policy.

```
#ifdef DML
int index1 = 1;
int index2 = 1;
int index3 = 1;
uint priority = 3;
p = findreadyprocess(&index1, &index2, &index3, &priority);
if (p == 0) {
  release(&ptable.lock);
  continue;
}
cpu->proc = p;
switchuvm(p);
p->state = RUNNING;
p->tickcounter = 0;
swtch(&cpu->scheduler, p->context);
switchkvm();
cpu->proc = 0;
#endif
```

Modified scheduler() function to implement the DML scheduling policy. This makes a call to the function findreadyprocess() in proc.c.

```
void decpriority(void) {
  // acquire(&ptable.lock);
  struct proc* proc = myproc();
  proc->priority = proc->priority == 1 ? 1 : proc->priority - 1;
  // release(&ptable.lock);
}
```

Added a decpriority() function in proc.c to facilitate DML scheduling. This function is later called in trap.c to decrease the priority of a process once it runs for a full quanta of time slice.

```
#ifdef DML
  // 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 && inctickcounter() == QUANTA) {
    myproc()->tickcounter = 0;
    decpriority();
    yield();
  }
```

Changed trap.c for DML scheduling. This basically preempts a running process if it's time quanta is over and calls decpriority() [implemented in proc.c] to decrease the priority of the process that just exited the CPU.

```
// Calling the exec system call resets
// the process priority to 2(default priority).
#ifdef DML
proc->priority = 2;
#endif
```

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdi
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;
  np->priority = curproc->priority;
```

Modified fork() function [called by the system call sys_fork()] to contain the line highlighted. It copies the priority of the parent process when a child is created rather than assigning the default value of 2.

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

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->ctime = ticks;
  p->retime = 0;
  p->rutime = 0;
  p->stime = 0;
  p->priority = 2;
  p->fake[0] = '*';
  p->fake[1] = '*';
  p->fake[2] = '*';
  p->fake[3] = '*';
  p->fake[4] = '*';
  p->fake[5] = '*';
  p->fake[6] = '*';
  p->fake[7] = '*';
  release(&ptable.lock);
```

Had to modify allocproc() since this is the function responsible for allocating a new process. The variables ctime, retime, rutime, stime, priority have been initiated with the required values. Note that priority has been initialized with 2, the default value for any new process.

TASK 2: Adding yield system call

To add yield system call we modified following files:

1.syscall.h
2.syscall.c
3.proc.c
4.user.h
5.usys.S

```
#define SYS_yield  25
```

```
extern int sys_yield(void);
```

```
[SYS_yield]   sys_yield
```

Added sys_yield(void) declaration in syscall.c. SYS_yield is the macro for system call number, defined in syscall.h. Lastly, added sys_yield system call inside the array of system calls in sycall.

```
int sys_yield(void){
  yield();
  return(0);
}
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

```
int yield(void);
```

Definition provided in user.h for user programs.

```
SYSCALL(yield)
```

Provides system calls' interface to user programs. Added in usys.S

TASK 3: Adding Sanity and SMLsanity program to test our policies

Created two files sanity.c and SMLsanity.c and updated the makefile

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _drawtest\
    _hardik\
    _sanity\
    _SMLsanity\
```

Policy 1 : **<u>Default Policy (SCHEDFLAG=DEFAULT)</u>**

Expected

It's turnaround depends on quanta and could be high

If value of quanta is low, it compromises turnaround time

For response time thus suitable for user programs.

```
hardik@lvy: /mnt/d/xv6-public
I/O bound, pid: 15, ready: 182, running: 0, sleeping: 100, turnaround: 282
CPU-bound, pid: 22, ready: 253, running: 42, sleeping: 0, turnaround: 295
CPU-S bound, pid: 23, ready: 267, running: 41, sleeping: 0, turnaround: 308
I/O bound, pid: 18, ready: 224, running: 0, sleeping: 100, turnaround: 324
I/O bound, pid: 21, ready: 267, running: 0, sleeping: 100, turnaround: 367
I/O bound, pid: 24, ready: 295, running: 0, sleeping: 100, turnaround: 395


CPU bound:
Average ready time: 128
Average running time: 42
Average sleeping time: 0
Average turnaround time: 170


CPU-S bound:
Average ready time: 138
Average running time: 42
Average sleeping time: 0
Average turnaround time: 180


I/O bound:
Average ready time: 179
Average running time: 0
Average sleeping time: 100
Average turnaround time: 279
```

After running sanity test for default

Policy 2 : First come - First Served (SCHEDFLAG=FCFS)

Expected

It is designed for batch programs. It penalizes short processes and increases waiting time but increases throughput and lower turnaround time.

```
hardik@Ivy: /mnt/d/xv6-public
CPU-S bound, pid: 8, ready: 211, running: 45, sleeping: 0, turnaround: 256
CPU-S bound, pid: 11, ready: 207, running: 51, sleeping: 0, turnaround: 258
CPU-S bound, pid: 14, ready: 215, running: 43, sleeping: 0, turnaround: 258
I/O bound, pid: 6, ready: 161, running: 1, sleeping: 100, turnaround: 262
I/O bound, pid: 9, ready: 164, running: 0, sleeping: 100, turnaround: 264
I/O bound, pid: 12, ready: 161, running: 1, sleeping: 100, turnaround: 262
I/O bound, pid: 15, ready: 160, running: 0, sleeping: 100, turnaround: 260
CPU-S bound, pid: 17, ready: 207, running: 50, sleeping: 0, turnaround: 257
I/O bound, pid: 18, ready: 158, running: 0, sleeping: 100, turnaround: 258


CPU bound:
Average ready time: 86
Average running time: 52
Average sleeping time: 0
Average turnaround time: 138


CPU-S bound:
Average ready time: 208
Average running time: 48
Average sleeping time: 0
Average turnaround time: 256


I/O bound:
Average ready time: 160
Average running time: 0
Average sleeping time: 100
Average turnaround time: 260
```

Policy 3 : Multi-level queue scheduling (SCHEDFLAG=SML)

Expected

Lower level processes face starvation as can be seen by the given test, Priority 1 has the highest turnaround time and ready time.

```
hardik@Ivy: /mnt/d/xv6-public
Priority 1, pid: 13, ready: 847, running: 89, sleeping: 0, turnaround: 936
Priority 1, pid: 16, ready: 851, running: 92, sleeping: 0, turnaround: 943
Priority 1, pid: 19, ready: 924, running: 98, sleeping: 0, turnaround: 1022
Priority 1, pid: 22, ready: 933, running: 96, sleeping: 0, turnaround: 1029
Priority 1, pid: 25, ready: 1010, running: 99, sleeping: 0, turnaround: 1109
Priority 1, pid: 28, ready: 1016, running: 101, sleeping: 0, turnaround: 1117


Priority 1:
Average ready time: 890
Average running time: 93
Average sleeping time: 0
Average turnaround time: 983


Priority 2:
Average ready time: 338
Average running time: 91
Average sleeping time: 0
Average turnaround time: 429


Priority 3:
Average ready time: 343
Average running time: 91
Average sleeping time: 0
Average turnaround time: 434
```

After running SMLsanity test for SML

Policy 4 : Dynamic Multi-level queue scheduling (SCHEDFLAG=DML)

Expected

It allows different processes to move between different queues.

It prevents starvation by moving a process that waits too long for the lower priority queue to the higher priority queue.That way it has better turnaround time then SML and less waiting time.

```
hardik@Ivy: /mnt/d/xv6-public
CPU-S bound, pid: 86, ready: 385, running: 56, sleeping: 0, turnaround: 441
CPU-bound, pid: 88, ready: 409, running: 48, sleeping: 0, turnaround: 457
I/O bound, pid: 84, ready: 385, running: 0, sleeping: 100, turnaround: 485
CPU-S bound, pid: 89, ready: 441, running: 44, sleeping: 0, turnaround: 485
I/O bound, pid: 87, ready: 441, running: 0, sleeping: 100, turnaround: 541
I/O bound, pid: 90, ready: 458, running: 0, sleeping: 100, turnaround: 558


CPU bound:
Average ready time: 202
Average running time: 45
Average sleeping time: 0
Average turnaround time: 247


CPU-S bound:
Average ready time: 222
Average running time: 46
Average sleeping time: 0
Average turnaround time: 268


I/O bound:
Average ready time: 267
Average running time: 0
Average sleeping time: 100
Average turnaround time: 367
```