# Enhancing xv6 OS


## Sriram Devata (2019113007)


## Introduction

Run xv6 with the command:

```
make qemu
```

The default scheduling mechanism is Round Robin. To use any of the other scheduling mechanisms, pass the compile-time flag `SCHEDULER` with the command:

```
make qemu SCHEDULER={RR,FCFS,PBS,MLFQ}
```

This version of xv6 has the following additions on top of the current version of xv6:

- Scheduling mechanisms - implemented FCFS, PBS, and MLFQ
- `strace`: a syscall to keep track of particular syscalls that a process makes
- Added more information to `procdump`
- `waitx`: a syscall similar to `wait`, but also gives the time spent running and waiting for a child process
- `time`: a user program to calculate the runtime and wait time of a command
- `schedulertest`: a user program to schedule a combination of IO-bound and CPU-bound programs to test the schedulers

## Specification 1: Syscall tracing with `strace`

A user program `strace mask command [args]` uses the syscall `strace` to print the syscall information whenever the set of syscalls encoded into *mask* are called by the *command*.

An attribute `mask` is stored in each `struct proc`.

```
struct proc {
    ...
    int mask;
    ...
}
```

When the syscall `strace` is called, it sets the `mask` attribute of the `myproc()`, i.e. the current process.

```
uint64
sys_strace(void)
{
    ...
    struct proc *p = myproc();
    ...
    p->mask = mask;

    return 0;
}
```

Since `strace` enables the printing of syscall for a process and all of its children, a child process formed by `fork()` should get the mask of its parent.

```
int
fork(void)
{
    ...
    // copy the strace mask to the child
    np->mask = p->mask;
    ...
}
```

When a syscall is handled in `syscall.c`, if the process has a non-zero mask, the information of the syscall is printed when the syscalls encoded in the process's mask are called.

```
void
syscall(void)
{
    ...
    if (p->mask & (int)1<<num) {
        printf(...);
    }
    ...
}
```

## Specification 2: Scheduling Mechanisms

The default scheduler in xv6 is Round Robin. This version of xv6 has other non-default schedulers that can be set at compile time with the flag `SCHEDULER`.

### FCFS - First Come First Served

The scheduler takes the *RUNNABLE* process that came first, and runs it till the process finishes.

Struct proc stores the time of creation in `ctime`, and it is set to 0 in `allocproc()`.

```
struct proc {
    ...
    uint ctime;
    ...
}
```

The `scheduler()` finds the *RUNNABLE* process with the lowest `ctime`, and runs that process.

```
      for(p = &proc[0]; p < &proc[NPROC]; p++) {
        if(p->state == RUNNABLE) {
          if(first_proc == NULL) {
            first_proc = p;
          }
          else if (first_proc->ctime > p->ctime) {
            first_proc = p;
          }
        }
      }

      if (first_proc != NULL && first_proc->state == RUNNABLE) {
          ...
      }
```

Once chosen, a process should not *yield* until it finishes running. In `trap.c`, the process yields if the macro `FCFS` is not defined.

```
void
usertrap(void)
{
    #if !defined FCFS
    ...
        yield();
    ...
    #endif
}
```

```
void
kerneltrap(void)
{
    #if !defined FCFS
    ...
```

```
        yield();
    ...
    #endif
}
```

### PBS - Priority Based Scheduler

The scheduler chooses the process with the highest priority.

The process now also has the following information:

```
struct proc {
    ...
    uint num_scheduled;
    uint s_priority;
    int nice;
    uint stime;
    uint rtime_ls;
    ...
}
```

`stime` and `rtime_ls` are the times the process spent sleeping and the time spent running from the last time it was scheduled. `num_scheduled` is the total number of times that the process has been scheduled.

Each process has a default static priority of 60. The syscall `setpriority` can change the static priority of a process. The dynamic priority of the process is calculated from the static priority and the nice value while scheduling. As soon as a process is scheduled, its nice values are updated in `update_process`.

Since this is a non-preemptive PBS, the process does not yield in `usertrap` and `kerneltrap`.

```
void
usertrap(void)
{
    #if !defined PBS
    ...
        yield();
    ...
    #endif
}
```

```
void
kerneltrap(void)
{
    #if !defined PBS
```

```
    ...
        yield();
    ...
    #endif
}
```

If multiple processes have the same priority, the tie is broken with the number of times they were scheduled, and the creation time of the process.

### MLFQ - Multi Level Feedback Queue

There are 5 priority queues where the timer tick quotas for each are 1, 2, 4, 8, 16. `mlfq_q` has the priority queues, and `procs_q` stores the number of processes in that queue. Each process also stores the queue it is in, in `struct proc->q_num`.

```
struct proc *mlfq_q[5][NPROC];
uint max_q_ticks[5] = {1,2,4,8,16};
uint procs_q[5] = {0,0,0,0,0};
```

The functions `remove_p_from_q` and `add_p_to_q` remove and add processes to the priority queues.

```
int remove_p_from_q(struct proc *p){
    ...
}

int add_p_to_q(struct proc *p, int q_no){
    ...
}
```

At the start of each scheduling round, the processes whose wait times are above `AGE_LIMIT` are moved to the higher priority queue.

```
    for(int q_no = 1; q_no < 5; q_no++){
        for(int p_no = 0; p_no < procs_q[q_no]; p_no++){
            if (mlfq_q[q_no][p_no]->age_t > AGE_LIMIT){
                ...
            }
        }
    }
```

The processes are allowed to run according to the quota of time limit they have based on the priority queue they are in. The processes yield in `usertrap` and `kerneltrap` only if they have exhausted their limit.

```
void
usertrap(void)
{
  #ifdef MLFQ
  if (which_dev == 2 && p->rtime_ls >= max_q_ticks[p->q_num]) {
    ...
    yield();
  }
  #endif
}
```

```
void
kerneltrap()
{
  #ifdef MLFQ
  if (which_dev == 2 && p != 0 && p->state == RUNNING
     && p->rtime_ls >= max_q_ticks[p->q_num]) {
    ...
    yield();
  }
  #endif
}
```

The round robin in the lowest priority queue is implemented implicitly since whenever the process in any priority queue exhausts its ticks limit, it is removed from the present queue and added to the next lower priority queue. If it is already in the lowest priority queue, it is added to the end of the lowest priority queue.

When a process gives up control of the CPU on its own, it leaves the queuing network and then reenters at the tail end of the same queue. If a process knows the quota of ticks in each priority queue, it can exploit this by using upto 99% of its allocated time and then give up the CPU. This makes sure that the process stays in the same high priority queue.

### Testing the schedulers

The user program `schedulertest` will run fork multiple child processes and prints when the processes are done, along with the average run time and average wait time.

### Analysis

With the userprogram `schedulertest` (the output might be weird if there are multiple CPUs), the default test results for the different scheduling mechanisms for 1 CPU are:

| Scheduler | Average Run Time | Average Wait Time |
|---|---|---|
| RR | 42 | 269 |

| Scheduler | Average Run Time | Average Wait Time |
|---|---|---|
| FCFS | 85 | 382 |
| PBS | 44 | 312 |
| MLFQ | 39 | 332 |

## Specification 3: Adding details to `procdump`

`procdump` in `proc.c` has the relevant information printed for the different schedulers.

### MLFQ Scheduling Analysis

After printing the queue of all processes in `update_time`, the output was stored by `script` and plotted using a python script.