

OS-344 Assignment-2B

Instructions

- This assignment has to be done in a group.
 - Group members should ensure that one member submits the completed assignment within the deadline.
 - Each group should submit a report, with relevant screenshots/ necessary data and findings related to the assignments.
 - We expect a sincere and fair effort from your side. All submissions will be checked for plagiarism through a software and plagiarised submissions will be penalised heavily, irrespective of the source and copy.
 - There will be a viva associated with assignment. Attendance of all group members is mandatory.
 - Assignment code, report and viva all will be considered for grading.
 - Early start is recommended, any extension to complete the assignment will not be given.
-

The goal of this lab is to understand process management and scheduling in xv6.

Before you begin

- Download, install, and run the original xv6 OS code.
- For this lab, you will need to understand and modify following files: `proc.c`, `proc.h`, `syscall.c`, `syscall.h`, `sysproc.c`, `user.h`, `usys.S` and others.

Below are some details on these files.

- ✕ `user.h` contains the system call definitions in xv6. You will need to add code here for your new system calls.
- ✕ `usys.S` contains a list of system calls exported by the kernel.
- ✕ `syscall.h` contains a mapping from system call name to system call number. You must add to these mappings for your new system calls.
- ✕ `syscall.c` contains helper functions to parse system call arguments, and pointers to the actual system call implementations.
- ✕ `sysproc.c` contains the implementations of process related system calls. You will add your system call code here.
- ✕ `proc.h` contains the struct `proc` structure. You may need to make changes to this structure to track any extra information about a process.
- ✕ `proc.c` contains the function scheduler which performs scheduling and context switching between processes.

Task1: Scheduling

Scheduling is a basic and important facility of any operating system. The scheduler must satisfy several conflicting objectives: fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low-priority and high-priority processes, and so on.

The set of rules used to determine when and how to select a new process to run is called a scheduling policy. You first need to understand the current scheduling policy. Locate it in the code and try to answer the following questions:

- Which process does the policy select for running?
- What happens when a process returns from I/O?
- What happens when a new process is created?
- When/how often does the scheduling take place?

First, change the current scheduling code so that process preemption will be done every quanta size (measured in clock ticks) instead of every clock tick. Add this line to **param.h** and initialize the value of QUANTA to 5.

```
#define QUANTA <Number>
```

In the next part of the assignment, you will add three different scheduling policies in addition to the existing policy. Add these policies by using the C pre-processing abilities.

➤ **Tip:** You should read about `#ifdef` macros. These can be set during compilation by gcc (see <http://gcc.gnu.org/onlinedocs/cpp/Ifdef.html>)

Modify the **Makefile** to support **SCHEDFLAG** – a macro for quick compilation of the appropriate scheduling scheme. Thus the following line will invoke the xv6 build with the default scheduling:

```
> make qemu SCHEDFLAG=DEFAULT
```

The default value for SCHEDFLAG should be DEFAULT (in the Makefile).

➤ **Tip:** you can (and should!) read more about the make utility here:

<http://www.opussoftware.com/tutorial/TutMakefile.htm>

Policy 1: Default Policy (SCHEDFLAG=DEFAULT)

Represents the scheduling policy currently implemented at xv6 (with the only difference being the newly defined QUANTA).

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

Represents a non-preemptive policy that selects the process with the lowest creation time. The process runs until it no longer needs CPU time (IO / yield / block).

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

Represents a preemptive policy that includes a three priority queues. The initial process should be initiated at priority '2' and the priority should be copied upon fork. In this scheduling policy the scheduler will select a process from a lower queue only if no process is ready to run at a higher queue. Moving between priority queues is only available via a system call. Priority 3 is the highest priority. The following system call will change the priority queue of the caller process:

int set_prio (int priority)

input:

priority- A number between 1- 3 for the new process priority


output:

0 - if successful

1 - otherwise

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

Represents a preemptive policy similar to Policy 3. The difference is that the process cannot manually change its priority. This are the dynamic priority rules:

-  Calling the **exec** system call resets the process priority to 2(default priority).
- Returning from SLEEPING mode (in our case IO) increases the priority of the process to the highest priority.
- Yielding the CPU manually keeps the priority the same.
- Running the full quanta will result in a decrease of priority by 1.

Task 2:

Add yield system call add the system call yield, this system call will yield execution to another process:

int yield()

input:

None

output:

0 - if successful

1- otherwise

Task 3: Sanity Test

In this section you will add two applications that test the impact of each scheduling policy. Similarly, to several built-in user space programs in xv6 (e.g., ls, grep, echo, etc.), you can add your own user space programs to xv6.

3.1 General sanity test:

Add a program called sanity this program gets a number (n) as argument, then it will fork(3*n) processes and wait till all of them finish, for each child process that end print its statistics.

Each of the 3n processes is of one of the three types:

- process with (pid **mod** 3 = 0) - CPU-bound process (CPU):
 - run 100 times dummy loop of 1000000 iterations
- process with (pid **mod** 3 = 1) - short tasks-based CPU-bound process (S-CPU):
 - run 100 times dummy loop of 1000000 iterations
 - after each dummy loop (out of the 100) yield the CPU
- process with (pid **mod** 3 = 2) - I/O bound process (IO):
 - to simulate the I/O waiting we will make dummy sleeps.
makes 100 times: sleep(1)

Printing the statistics:

For each terminated process print in new line:

- The process id and his type (CPU / S-CPU / IO)
- His wait time, run time and I/O time

Then after all 3n processes terminates print the following:

- Sleep time - average time that a job was sleeping (for each process type group)
- Ready time - average time that a job was waiting to CPU (for each process type group)

- Turnaround time - average time for a job to complete (for each process type group)

3.2 Priority schedule test:

Add another program called *SMLsanity* to test policy 3

- This should be a small test which prove that the scheduling order is prioritized as expected.

➤ **Hint:** *fork* lots of CPU-bound processes (let say 20), give each process different priority then print their termination time.

Analyse (for both parts):

Run the test for all the different policies, before checking the results try to predict the results for each policy, then check if the statistics results consist your prediction.

- Provide valid reasoning in the report for the results generated using each type of scheduling policy.
- Also be ready to explain how you calculate the statistics and how they reflect the correctness of you scheduling policies during evaluation.

➤ **Tip1:** to add a user space program, first write its code (e.g., *sanity.c*). Next update the **makefile** so that the new program is added to the file system image. The simplest way to achieve this is by modifying the lines right after "UPROGS=\".

➤ **Tip:** You have to call the *exit* system call to terminate a process' execution.

Submission instructions

- Place all the files including Makefile that you modified for each part into separate subfolders named as Task[1 or 2] and put all of them into a zip file, with the name being your group number (say, G10.zip).
- report.pdf should contain a detailed description of all of your implementation including screenshots of code and output.
- Create a patch of your modified files.
- Further, you must describe your test cases in some detail, and the observations you made from them.

--End of Assignment-2B--