Operating Systems

Assignment 1: system calls and scheduling

Responsible TA: Nitsan Soffair, soffair@post.bgu.ac.il
Grader: Neyema Awaskar, neyema@post.bgu.ac.il

1 Introduction

Throughout this course we will be using a simple, UNIX like teaching operating system called xv6: https://pdos.csail.mit.edu/6.828/2020/xv6.html.

The xv6 OS is simple enough to cover and understand within a few weeks yet it still contains the important concepts and organizational structure of UNIX. To run it, you will have to compile the source files and use the QEMU processor emulator (installed on cicero2 CS lab computers).

xv6 was (and still is) developed as part of MIT's 6.828 Operating Systems Engineering course.

You can find a lot of useful information and get started tips here:
https://pdos.csail.mit.edu/6.828/2020/overview.html

xv6 has a very useful guide. It will greatly assist you throughout the course assignments: https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf

You may also find the following useful:
https://pdos.csail.mit.edu/6.828/2018/xv6/xv6-rev11.pdf

Task 0: Running xv6
Follow the instructions available on the course website to run xv6 on a virtual machine:
https://moodle2.bgu.ac.il/moodle/mod/page/view.php?id=2099266

or alternatively on your machine:
https://pdos.csail.mit.edu/6.828/2019/tools.html

2 System calls

Implement the following system calls.

1. *pause_system*: pause all user processes for the number of **seconds** specified by the second's integer parameter.
   signature of proc.c function to add*: int pause_system(int seconds).*
   Each of the CPUs will not run any process until the time is UP, and the state of currently running processes will change to **RUNNABLE**, until the time is up. When the timer is finished normal execution should be resumed.

2. *kill_system*: terminate all user processes, use the *kill* system call.
   signature of proc.c function to add*: int kill_system(void).*
   Do not kill the **init process** or the **shell process**.

Add core functions at proc.c, add wrapper functions at sysproc.c, add signatures at syscall.c, defs.h, add system call numbers at syscall.h.

To kill the system you should turn on the killed flag of all processes except for init: struct proc *initproc in proc.c, and the shell process.
The shell and init processes are using the same PIDs in each run, examine the process table after system init (which can be done either by printing or by GDB) to see what PIDs these two processes use.
Make sure the above system calls that you implement do not affect these two processes.

Examples:

pause_system prints t*wo variables in each loop iteration and calls pause_system in the middle iteration.*

kill_system does the same but calls *kill_system* instead.

See this code for an example.

3 Scheduling policies

In the next part of the assignment, you will need to add two new scheduling policies in addition to the existing policy. Add these policies by using the C preprocessing abilities.

Please read this #IFDEF macros example, and this short guide.

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 Approximate SJF scheduling (defined shortly).

$ make qemu SCHEDFLAG=SJF

Add this to the makefile:

ifndef SCHEDFLAG
SCHEDFLAG := DEFAULT
endif

and in the CFLAGS section
add
-D $(SCHEDFLAG)

Make sure you do not override previous flags!

Now it can be used with the IFDEF macro.

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

Additional information about makefiles can be found here.

There might be bugs that are created from race conditions due to multiple CPUs. In order to avoid said bugs, reduce the number of CPUs in your QEMU VM to 1. This can be done via the Makefile, or by changing NCPU to 1.

Implement the following scheduling policies

1. *Approximate SJF*.
   Select the process with the minimum ***mean_ticks*** score.
   add ***mean_ticks***, ***last_ticks*** params in proc.h and initialize both to 0. initialize ***rate = 5*** as a global variable.
   Measure the number of ticks in the last CPU burst and set the ***last_ticks*** variable accordingly, update ***mean_ticks*** by the formula:

   mean_ticks = ((10 - rate) * mean_ticks + last_ticks * (rate)) / 10.

   *Note: there are no floats in xv6, this is why the above formula is used instead of 0.5*mean_ticks + 0.5*last_ticks*

   The next process to run is the one with a minimum value of *mean_ticks*.
   Each process runs until it either exits or blocks (its state is changed to the SLEEPING state).

   You may read more about this scheduling algorithm here.

2. *First-Come-First-Served (FCFS)*.
   Select the process that is in the runnable state for the longest time.
   Add to proc data structure in proc.h the field ***last_runnable_time.*** Whenever a process changes to RUNNABLE state, write to this field the value of the global ticks variable (see trap.c).

   The scheduler will always select the process with the **lowest *last_runnable_time***, and let it run until it either exits or blocks (its state is changed to the SLEEPING state).

4 Performance measurements

Statistics

Single process behavior:

1. *sleeping_time*: Total time spent in the SLEEPING state.

2. *runnable_time*: Total time spent in the RUNNABLE state.

3. *running_time*: Total time spent in the RUNNING state.

On system init, initialize the following global variables s*leeping_processes_mean*, *running_processes_mean*, *running_time_mean* to 0, you can place them at proc.c.
When a process exits the system (via the syscall exit), you should update these values accordingly.

For example, if the current value of *running_processes_mean* is 100, there have been 10 processes so far and a process exits with *running_time* value of 50, you will update it using the following formula : ((100*10)+50)/11.

Total system performance:

1. *program_time*
   On system init, initialize a global variable *program_time* to 0, you can place it at proc.c.
   This value will hold the sum of all running time of all processes (again, excluding init and shell). You should update it on each process exit, as described previously.

2. *cpu_utilization*
   On system init, initialize the global variables *start_time,* and *cpu_utilization*.
   During *procinit* set *start_time* to the value of *ticks*.
   On each update to *program_time* (when a process exists), you should also update the *cpu_utilization variable*, using the following calculation:
   cpu_utilization = *program_time* / (*ticks* - *start_time)*.

Exporting the values:

In order to export the aforementioned global variables, you should write a new system call *print_stats(void)* which will print the values to the shell. Make sure that your printing function is easy to process, you will use these later on!

Testing Environments:

For evaluating your assignment, you will use **two different user programs, each of them with its own unique characteristics.**

env-large user program with a few large tasks repeatedly makes *calculations* with large numbers.

env-freq with many small tasks, repeatedly make *calculations* with small numbers.

Use the following base code for these two user programs: code (it already contains the code for the calculations).
In each of these user programs, the final line before exit should be the system-call *print_stats*.

For each of the policies described in section 3, and the default (round-robin) xv6 policy, run 10 experiments (note that you need to restart qemu between experiments).
Report the ***mean*** and ***standard error*** for each policy, each environment, and each metric.

You can read about the **standard error** here.

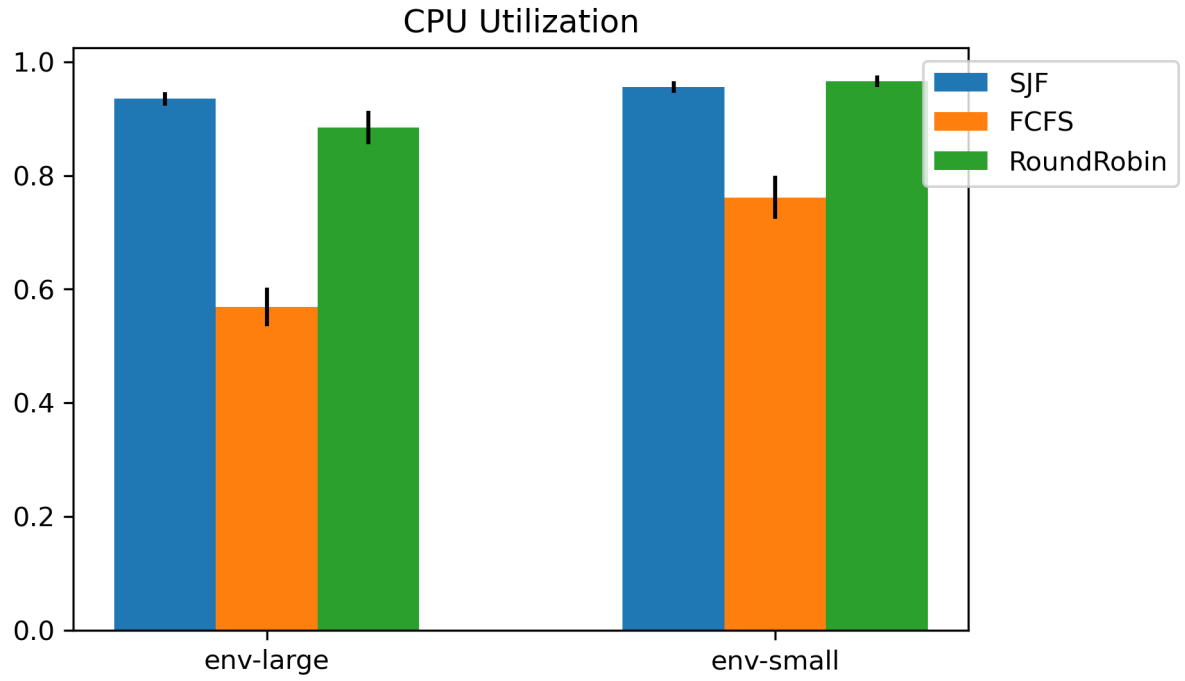
Save your results in the file *results.txt*

Visualization:

For visualizing the results, you will create 5 different graphs, we recommend using *python* and *matplotlib* (see this guide).

Each graph will be for one of the metrics and will contain both the **mean** and the **standard error** (as an error bar), the abscissa will hold the different methods and environments.

Example Graph (with pseudo experiment results):



Conclusions:

Analyze your results and write sharp conclusions for each criteria-winner.
example: **SJF** has a clear *cpu_utilisation* advantage over xv6 on **env-large**.

5 Grading

You will be graded according to the quality of the code, theoretical questions, the analysis, and the conclusions which followed it.

Exceptional projects will have a 10 points bonus to the grade.

Submission

A Zip file containing your **code** (as a patch file), and a **pdf file** containing your graphs, and analysis.

Patch create commands sequence
$ make clean
$ git add –Av; git commit –m ”commit message”
$ git diff origin
$ git diff origin > ID1_ID2.patch

Test your assignment by cloning a clean copy of xv6 and applying the patch:
$ patch –p1 < ID1_ID2.patch.