OPERATING SYSTEMS – ASSIGNMENT 1

SYSTEM CALLS & SCHEDULING

Responsible TAs: Simion Novikov and Sohel Kanaan

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 getting 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=1733250

or alternatively via VPN:
https://moodle2.bgu.ac.il/moodle/mod/page/view.php?id=1734080

Task 1: Warm up ("HelloXV6")

This part of the assignment is aimed at getting you started. It includes small changes in the xv6 shell. Note that in terms of writing code, the current xv6 implementation is limited: it supports only a subset of the system calls you may use when using Linux and its standard library, and is quite limited.

1.1. Support the PATH environment variable:

When a program is executed, the shell seeks the appropriate binary file in the current working directory and executes it. If the desired file does not exist in the working directory, an error message is printed. Currently all the user programs are located inside the /user folder, therefore in order to run a program you should add a prefix "/user/" to it (for example to run "ls" you should type "/user/ls").

"PATH" is an environment variable which specifies the list of directories where commonly used executables reside. If, upon typing a command, the required file is not found in the current working directory, the shell attempts to execute the file from one of the directories specified by the ***PATH*** variable. An error message is printed only if the required file was not found in the working directory or any of the directories listed in ***PATH***.

Your first task is to add support for the ***PATH*** environment variable. In order to simplify the support for environment variables we require that the value of the ***PATH*** environment variable will reside in file "`/path`". Namely, each time the shell needs to know the value of ***PATH***, it should read the content of the file "`/path`". This means that each change in the content of the file "`/path`" will cause an update of the value of the ***PATH*** variable. The value of the ***PATH*** variable consists of a list of directories where the shell should search for executables. Each directory name listed should be delimited by a colon (':'). For example, if we want to add the *root directory* and the *user directory* to the ***PATH*** variable, we can set the content of the "`/path`" file to:

```
/:/user/:
```

Finally, the shell must be aware of the ***PATH*** environment variable when searching for a program. The first place to seek the binary of the executed command should be the current working directory, and only if the binary does not exist there, the shell must search it in directories defined by ***PATH***. The list of directories can be traversed in any order and must either execute the binary (if it is found in one of the directories) or print an error message in case the program is not found. Note that the user can execute a program by providing to the shell an *absolute path* (i.e., a path which has '/' as its first character) or a *relative path*. Test your implementation by executing a binary which does not reside in your current working directory but is pointed to by ***PATH***. The tests must include commands which use I/O redirection (i.e., file input/output and pipes).

The file "`/path`" can be created in any way, but needs to keep its' contents between shut downs (make qemu), but can reset between reinstalls (make clean qemu).

Task 2: Tracing a program

For debugging programs, it is often useful to record the system calls they make. For this task, you will add a “trace” system call. The argument of the system call is a bit-mask of which system calls to record, and a pid of the process to apply this mask to. The user space signature of the system call is:

`int trace(int mask,int pid)`

For example, `trace(1<< SYS_fork, getpid())`

would activate the tracing of the fork system call for the process whose pid is specified (in this case, the calling process itself) (SYS_fork is the number of the system call from syscall.h).

It will return 0 if the system call is successful, and -1 if not.

The tracing will be active on those system calls specified by the mask: (multiple system calls might be traced at the same time)

Once activated, every system call in the process’s mask will generate a print out of:

process ID,”: syscall”, system call name, system call arguments,-> , return value

Since the system call arguments can vary in number and types, you only need to print out the system call arguments to the FORK, KILL, and SBRK system calls. For the rest, only print out the return value, treating it as an int.

So after the code:

```
mask=(1<< SYS_fork)|( 1<< SYS_kill)| ( 1<< SYS_sbrk) | ( 1<< SYS_write);

trace(mask,2);


we might get the following tracing prints:


2: syscall fork NULL -> 3   (process 2 forked process 3. Since fork doesn’t
have arguments, print NULL)

2: syscall kill 3 -> 0   (process 2 killed process 3 succesfully)

2: syscall kill 5 -> -1   (process 2 tried to kill process 4 unsuccesfully)

2: syscall sbrk 4096 -> 12288   (process 2 acquired 4096 bytes of extra
memory, this will be -1 if sbrk fails, otherwise the pointer to the top of
the memory)
```

```
2: syscall write  -> 0   (process 2 wrote something successfully. Since
write isn't among the 3 system calls we specified to print out the inputs
to, only print out the output)
```

You also need to make sure that the mask is copied from the parent process to the child process.

The system calls are recorded according to the last mask that was used in the trace() system call- that is, the mask overwrites previous masks.

Hints:

1. See how a simple system call works to figure out how to add an additional system call- use a system call that was covered in class.
    a. Keep style rules: the system call function has to be called sys_trace.
2. The mask has to be stored per process (where is per process information stored?).
3. In order to support printing the return value, you should think of where to add the print in the system call sequence.
4. In order to support printing the name of the system call, you should add an array of a certain type.
5. Where in the fork system call sequence is data from the parent copied to the child?
6. How are system call arguments retrieved from user space by the kernel?
7. Different system calls have different argument signatures-in which files can you find the signatures of the system calls?
8. For efficiency purposes, in your implementation do a quick bit-compare to figure out if you need to enter the tracing logic at all before computing the signature you need to print.

Task 3: Measuring the performance of scheduling policies

Before implementing various scheduling policies, we will create the infrastructure that will allow us to examine how they affect performance under different evaluation metrics.

In class, you learned about several quality measures for scheduling policies. In this task you are required to implement some of them and measure your new scheduling policies performance. The first step is to extend the proc struct (see proc.h) by adding the following fields to it:

- ctime – process creation time.
- ttime – process termination time.

- stime – the total time the process spent in the SLEEPING state.
- retime – the total time the process spent in the RUNNABLE state.
- rutime – the total time the process spent in the RUNNING state.
- average_bursttime- approximate estimated burst time (as specified in task 4.3)

These fields retain sufficient information to calculate the turnaround time and the waiting time of each process.

Upon the creation of a new process the kernel will update the process' creation time. The fields (for each process state) should be updated for all processes whenever a clock tick occurs (see trap.c) (<u>you can assume that the process' state is SLEEPING only when the process is currently in the system call sleep</u>). Finally, care should be taken in marking the termination time of the process (note: a process may stay in the 'ZOMBIE' state for an arbitrary length of time. Naturally, this should not affect the process' turnaround time, wait time, etc.).

Since all this information is retained by the kernel, we are left with the task of extracting this information and presenting it to the user. To do so, create a new system called `wait_stat`, which extends the wait system call:

`int wait_stat(int* status, struct perf * performance)`, where the second argument is a pointer to the following structure:

```
struct perf {
    int ctime;
    int ttime;
    int stime;
    int retime;
    int rutime;
    int average_bursttime; //average of bursstimes in 100ths (so average*100)
};
```

The `wait_stat` function will return the pid of the terminated child process or -1 upon failure. It will also fill in the perf structure (created by the user, pointer to it supplied to the kernel) with the data from the kernel.

<u>Task 4: Scheduling Policies</u>

<u>Important:</u> Read the entire task carefully before you start implement. Read the notes at the end of this task.

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 (e.g., existing) scheduling policy. Locate it in the code and try to answer the following questions: which process the policy chooses to run, what happens when a process returns from I/O, what happens when a new process is created and when/how often scheduling takes place.

Second, change the current scheduling code so that swapping out running processes will be done every quantum (measured in clock ticks) instead of every clock tick. Add the following line to param.h and initialize the value to 5.

```
#define QUANTUM
```

In the next part of the assignment you will need to add four different scheduling policies in addition to the existing policy. Add these policies by using the C preprocessing 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 First Come First Serve scheduling:

```
>make qemu SCHEDFLAG=FCFS
```

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

Additional information about makefiles can be found here:
http://www.opussoftware.com/tutorial/TutMakefile.htm

There might be bugs that are created from race conditions due to multiple CPUs. Since we haven't covered synchronization yet, you can reduce the number of CPUs in your VM to 1 if that solves the bugs. Preferably though, make your solution work even when there are multiple CPUs.

4.1 Default Policy (SCHEDFLAG= DEFAULT)

Represents the currently implemented scheduling policy.

4.2 First Come First Serve (SCHEDFLAG= FCFS)

This is a non-preemptive policy. A process that gets the CPU runs until it no longer needs it (yield / finish / blocked). When a process yields the CPU, it is moved to the end of the scheduling queue.

4.3 Shortest Remaining Time (SCHEDFLAG= SRT)

This is a preemptive version of SJF. Each time the scheduler needs to select a new process it will select the process with minimal approximated burst time. In case of a tie, the first process in the array with minimal approximated burst time will be selected.

Let A1 be the approximate estimated burst time in 100ths of a tick for a new process. Initially, A1=QUANTUM*100. Let B be the length of the current burst. If the process runs for B1 clock ticks before blocking or yielding, then the new approximate burst time is A2=αB1+(100-α)A1/100. In general: A(i+1)= αBi+((100-α)Ai)/100. Support the α parameter by adding the following line to param.h and initialize the value to 50.

#define ALPHA <int Value>

Use the `average_bursttime` parameter from Task 3 to store the approximate estimated burst time. The update should occur any time a process is returning (yield, blocked). Calling fork should reset approximate run time to the initial value A1 for the child process.

Completely Fair Schedular with Priority decay (SCHEDFLAG=CFSD):

4.4 Completely Fair Schedular with Priority decay(SCHEDFLAG=CFSD).

A preemptive policy inspired by Linux CFS (this is not actual CFS). Each time the scheduler needs to select a new process it will select the process with the minimum run time ratio:
$\frac{rutime \cdot decay\ factor}{rutime+stime}$.
In case of a tie, the first process in the array with the minimum run time ratio will be selected.

Decay factor is defined using process priority. We will support five priority levels:

1. Test High priority – Decay factor = 1
2. High priority – Decay factor = 3
3. Normal priority – Decay factor = 5
4. Low priority – Decay factor = 7
5. Test Low priority – Decay factor = 25

Priority will be set by a new system call you will add that will change the priority of the current process.

int set_priority(int priority)

input:
• Int priority – The new priority value for the current process (1-5)

Output:
• 0 – New priority was set
• -1 – Illegal priority value

Note: The priority of the initial process is Normal. Calling fork should copy the parent's priority to the child. Calling exec should not change process priority.

Submission Guidelines:

Make sure that your Makefile is properly updated and that your code compiles with no warnings whatsoever. We strongly recommend documenting your code changes with comments – these are often handy when discussing your code with the graders.

XV6 contains tests in the user folder (usertests and forktest): make sure they run the same after your changes.

Due to our constrained resources, assignments are only allowed in pairs. Please note this important point and try to match up with a partner as soon as possible.

Submissions are only allowed through the moodle system. To avoid submitting a large number of xv6 builds you are required to submit a patch (i.e. a file which patches the original xv6 and applies all your changes). You may use the following instructions to guide you through the process:

Back-up your work before proceeding!

Before creating the patch review the change list and make sure it contains all the changes that you applied and noting more.
Execute the command:

```
> make clean
```

Modified files are automatically detected by git but new files must be added explicitly with the '`git add`' command:

```
> git add . -Av; git commit -m "commit message"
```

At this point you may examine the differences (the patch):

```
> git diff origin
```

Once you are ready to create a patch simply make sure the output is redirected to the patch file:

```
> git diff origin > ID1_ID2.patch
```

- Tip: Although grades will only apply your latest patch, the submission system supports multiple uploads. Use this feature often and make sure you upload patches of your current work even if you haven't completed the assignment.

Finally, you should note that the graders are instructed to examine your code on a clean VM as specified in task 0!

We advise you to test your code on the clean VM prior to submission, and in addition after submission to download your assignment, apply the patch, compile it, and make sure everything runs and works.

Tips and getting started

Take a deep breath. You are about to delve into the code of an operating system that already contains thousands of code lines. BE PATIENT. This takes time!