OPERATING SYSTEMS – ASSIGNMENT 1

XV6, PROCESSES, SYSTEM CALLS AND SCHEDULING

Introduction

Throughout this course we will be using a simple, UNIX like teaching operating system called **xv6**. 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 all CS lab computers).

- **Tip**: 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 there: http://pdos.csail.mit.edu/6.828/2014/xv6.html
- **Tip**: xv6 has a very useful guide. It will greatly assist you throughout the course assignments: http://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf
- **Tip**: you may also find the following useful: http://pdos.csail.mit.edu/6.828/2014/xv6/xv6-rev8.pdf
- **Tip**: it's recommended to read all the assignment before starting to work on it.

In this assignment we will start exploring xv6 and extend it to support various features including new scheduling policies.

Task 0: Running xv6

Begin by downloading our revision of xv6, from the MIT xv6 git repository:

- Open a shell, and traverse to a directory in your computer where you want to store the sources for the OS course. For example, in Linux:

```
> mkdir ~/os162
> cd ~/os162
```

- Execute the following command:

```
> git clone git://github.com/mit-pdos/xv6-public.git
```

This will create a new folder called xv6 which will contain all project files.

- Build xv6 by calling make:

```
> make
```

- Run xv6 on top of QEMU by calling:

```
> make clean qemu
```

go ahead and play with that, see what it can do.

Task 1: warm up - Improving the console

Modern consoles support many features that help users work more efficiently. In the first part of the assignment you will implement two basic features that will hopefully allow you a more efficient workflow when testing your code.
in the first part you will extend the kernel to support Caret navigation (text cursor navigation).
In the second part you will add partial support of "Shell History Ring".

1.1: Caret navigation

In the current xv6 console, pressing the keyboard keys ← or → will result in their appropriate ascii signs to appear on the console. When working inside your linux terminal those keys will cause the caret to move to the appropriate location (left or right) allowing the user to edit what he wrote more efficiently.

In the first part of your assignment you will need to implement caret navigation. Notice:

- End line must result in moving to the next line no matter where the caret is and it's ascii value entered at the end of the buffer.
- When editing from the middle of a buffer you must shift the text accordingly.

1.2: Shell History Ring

History of past shell commands allows terminal users to evaluate multiple requests very fast without writing the entire command. In this part of the assignment you will have to implement the history feature and the ability to easily update the console to fit the needed history. In modern operating systems the history is implemented in the shell, To allow for a simple implementation you will implement history in kernel. Your implementation should support a maximum of 16 commands. To do so you can add: #define MAX_HISTORY 16 to your code.

Once history is implemented we need a way to access the history. You will implement two mechanisms to do so:

1) The ↑ / ↓ keys will need to retrieve the next / last item in the history respectively. The item retrieved should now appear in the console.
2) Add a history system call:
   `int history(char * buffer, int historyId)`

   Input:

   `char * buffer` - a pointer to a buffer that will hold the history command, Assume max buffer size 128.

   `historyId` - The history line requested, values `0` to `15`

   Output:

   0 - History copied to the buffer properly
   -1 - No history for the given id
   -2 - historyId illegal

Once this is implemented add a "`history`" command to the shell user program (see *sh.c*) so that it upon writing the command a full list of the history should be printed to screen like in common.

- A good place to start for both 1.1 and 1.2 is the ***console.c*** fie.
- Notice that this features will only work on the QEMU console and not on the terminal. Running QEMU in ***nox*** mode for ssh is not advised while testing part 1.

Task 2: Statistics

In Task 3 you will implement various scheduling policies. However, before that, we will implement an infrastructure that will allow us to examine how these policies affect performance under different evaluation metrics.

The first step is to extend the proc struct (see ***proc.h***). Extend the proc struct by adding the following fields to it: `ctime`, `stime`, `retime` and `rutime`. These will respectively represent the creation time and the time the process was at one of following states: `SLEEPING`, `READY (RUNNABLE)` and `RUNNING`.

- **Tip**: These fields retain sufficient information to calculate the turnaround time and 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 (you can assume that the process' state is SLEEPING only when the process is waiting for I/O). 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 question of extracting this information and presenting it to the user. To do so, create a new system call `wait2` which extends the wait system call:

```
int wait2(int *retime, int *rutime, int *stime)
```

Input:

`int * retime / rutime / stime` - pointer to an integer in which `wait2` will assign:
The aggregated number of clock ticks during which the process **waited** (was able to run but did not get CPU) The aggregated number of clock ticks during which the process was **running**
The aggregated number of clock ticks during which the process was **waiting for I/O** (was not able to run).
Output:

`pid` of the terminated child process - if successful
`-1` - upon failure

Task 3: 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 and 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 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 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 follow 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 it's 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 4: add yield system call

add the system call yield, this system call will yield execution to another process:
`int yield()`
input:
Non
output:
`0` - if successful
`-1` - otherwise

Task 4: 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.

4.1: general sanity test:

Add a program called *sanity* this program get 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 (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 to was waiting to CPU (for each process type group)
- Turnaround time - average time for a job to complete (for each process type group)

4.2. priority schedule test:

Add a another program called *SMLsanity* to test policy 3
this should be a small test who prove that the scheduling order is prioritized as expected.

- ➢ **hint**: `fork` lots (let say 20) CPU-bound processes, give each process different priority then print they termination time.

analyze (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 consists your prediction.

- be ready to explain the graders how you calculate the statistics and how they reflect the correctness of you scheduling policies

➢ **Tip**: 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 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.

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 submission 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 nothing more. 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 **lab computers only**! We advise you to test your code on lab computers prior to submission, and in addition after submission to download your assignment, create a clean xv6 folder (by using the git clone command), apply the patch, compile it, and make sure everything runs and works. The following command will be used by the testers to apply the patch:

```
> patch -p1 < ID1_ID2.patch
```

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!

Two common pitfalls that you should be aware of:

- Quota – as you may know, your CS share is limited. Before beginning your work we recommend cleaning your home folders and running xv6 with no modifications. If you still encounter problems you can try to work on freespace (another file server). Note that unlike your home folders, freespace data is not backed up – remember to back up your work as often as possible.
- IDE auto-changes – we are aware that many of you prefer to work under different IDEs. Note that unless properly configured these often insert code lines or files which may cause problems in later stages. Although we do not limit you, our advice is to use the powerful vi editor or GNU Emacs. If you want an X application you can try running vim or gedit.

Debugging

You can try to debug xv6's kernel with gdb (gdb/ddd is even more convenient). You can read more about this here: http://zoo.cs.yale.edu/classes/cs422/2011/lec/l2-hw

Working from home

The CS lab computers should already contain both **git** and **qemu**. Due to the large number of students taking this course we will only be able to support technical problems that occur on lab computers. Having said that, students who wish to work on their personal computers may do so in several ways:

- on windows: Connecting from home to the labs:
  - Install PuTTY (http://www.chiark.greenend.org.uk/~sgtatham/putty/).
  - Connect to the host: lvs.cs.bgu.ac.il, using SSH as the connection type.
  - Use the ssh command to connect to a computer running Linux (see http://www.cs.bgu.ac.il/facilities/labs.html to find such a computer).
  - Run QEMU using: `make qemu-nox`.
  - Tip: since xv6 may cause problems when shutting down you may want to consider using the screen command:

    screen make qemu-nox
- on Linux (recommended): install QEMU on your own PC.
  - Microsoft windows users can easily install a dual boot (Windows-Linux) host with wubi: http://www.ubuntu.com/desktop/get-ubuntu/windowsinstaller

Again, we will not support problems occurring on students' personal computers.

Have fun!