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: http://pdos.csail.mit.edu/6.828/2010/xv6-book/index.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 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/6.828/2010/overview.html*
- *Tip: xv6 has a very useful guide. It will greatly assist you throughout the course assignments: http://pdos.csail.mit.edu/6.828/2011/xv6/book-rev6.pdf*
- *Tip: you may also find the following useful: http://pdos.csail.mit.edu/6.828/2011/xv6/xv6-rev6.pdf*

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

Task 0: Running xv6

Begin by downloading our revision of xv6, from the OS132 ***svn*** 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 ~/os132
cd ~/os132
```

- Execute the following command (in a single line):

```
svn checkout http://bgu-os-132-xv6.googlecode.com/svn/trunk assignment1
```

  This will create a new folder called assignment1 which will contain all project files.
- Build xv6 by calling: `make`
- Run xv6 on top of QEMU by calling: `make clean qemu`

Task 1: warm up ("HelloXV6")

This part of the assignment is aimed at getting you started. It includes two extensions to the xv6 shell. These extensions will require simple changes to the Makefile and to the shell file.

Note that in terms of writing code, the current xv6 implementation is limited: it does not support system calls you may use when writing on Linux and its standard library is quite limited.

Extension 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. For example:

```
<bad command>: Command not found.
```

A '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. An error message is printed only if the required file was not found in the working directory or any of the directories listed in the PATH.

Your first task is to support a PATH environment variable. You will do this with the following command in the shell:

```
export PATH <name of path1>:<name of path2>:…:<name of pathN>:
```

where each directory name listed is delimited by a colon (':').
For example, if we wanted to add the root directory to the PATH variable we would write in the shell:

```
export PATH /:
```

the Path variable will include the root directory

```
export PATH /:/etai/:/etai/gal/:
```

the Path variable will include the root director, and the folder named etai, and the folder named gal (which is located in the folder etai)

Every time you call the command export PATH, the new paths will replace the existing PATH.

Tip: you can reuse the current shell code in sh.c (e.g. parsing a command).

Extension 2: support right and left arrows (← →)

The ability to edit the command typed is a basic service provided by most shells. By pressing the right or left arrows ('←', '→') the user can move between the chars typed and decide where to delete a char or add a new char (insert mode).

In this task you will emulate this behavior. You will need to support the movement of the cursor right and left, by pressing the '←' and '→' keys respectively, so that the line typed in the shell will change where the cursor currently is.

Tip: a good place to start working on this task is console.c

Task 2: Performance Measures

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

The first step is to extend the *'proc'* struct located at proc.h (line 61). Extend the *'proc'* struct by adding the following fields to it: ***ctime***, ***etime*** and ***rtime***. These will respectively represent the creation time, end time and total running time of the process.

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 run time should be incremented by updating the current process whenever a clock tick occurs. Finally, care should be taken in marking the end 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, extend the wait system call so that it will support the following system call:

```
int wait2(int *wtime, int *rtime)
```

Where the two arguments are pointers to integers to which the wait2 function will assign (1) the aggregated number of clock ticks during which the process waited, (2) the aggregated number of clock ticks during which the process was running.

The wait2 function shall return the pid of the child process caught or -1 upon failure.

Tip: Adding a system call requires some delicate work and proper registration. Be sure to add changes to syscall.c, syscall.h, usys.S and sysproc.c

We've added an example for using and testing your wait2 system call in the additional files (in the assignments page). This program is for your own use and debugging, do not submit it.

Task 3: Scheduling Policies

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.*

In this part of the assignment you will 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 Round Robin scheduling:

```
make qemu SCHEDFLAG=RR
```

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

Tip: you can (and should!) read more about the make utility here: http://www.opussoftware.com/tutorial/TutMakefile.htm

Policy 1: Round Robin (SCHEDFLAG=RR)

Locate the current scheduling code. Notice that the scheduler goes over all processes in the current process table and attempts to run the first RUNNABLE one. The process is swapped out on every clock tick (this code is handled by the kernel whenever a timer trap occurs). Fix this scheduling policy so that the resulting scheduling policy will be round robin scheduling with a quanta size (measured in clock ticks).

```
#define QUANTA <int value>
```

Add this line to param.h and initialize the value of QUANTA to 5.

You can test your code for this policy before moving on by implementing and running task 4.1 now.

Tip: What are the expected results? You should think about the answer before the frontal check.

Policy 2: FIFO Round Robin (SCHEDFLAG=FRR)

This policy will extend the previous round robin policy. In addition to the previously implemented round robin policy, in this policy the decision of which process will run next will be on a First In First Out basis. When a process finishes running for QUANTA time (or executes yield) it is considered to be the last process to arrive and wait for its next turn to run.

Policy 3: Guaranteed (Fair-share) Scheduling (SCHEDFLAG=GRT)

This policy constantly calculates, per process, the ratio of allocated CPU time and its total time in the system (time past since the process was created), and schedules the process with the lowest ratio to run next.

To implement this policy the decision of which process should run next will be made according to the ratio:

$$\frac{rtime}{current\_time - ctime}$$

Where the process with the lowest ratio value is scheduled to run next, or chosen arbitrarily in case of a tie.

Notice that since time is measured in discrete units, the expression (current_time-ctime) can be zero. Make sure you handle these delicate situations.

You can test your code for this policy before moving on by implementing task 4.2 now.

Tip: What are the expected results? You should think about the answer before the frontal check.

Policy 4: Multi Level Queue Scheduling (SCHEDFLAG=3Q)

This policy will maintain three queues.

- The first queue will hold the high priority processes. The processes in this queue will be scheduled according to the guaranteed scheduling policy.
- The second queue will hold medium priority processes. The processes in this queue will be scheduled according to the FIFO round robin scheduling policy.
- The third queue will hold low priority processes. The processes in this queue will be scheduled according to the round robin scheduling policy.

A process with a higher priority will be preferred and run before a process with a lower priority. This means that no low or medium priority process will be allowed to run while there are still high priority runnable processes (or running), and no low priority process will be allowed to run while there are still medium priority runnable processes.

To define the priority mechanism for the processes you will have to change the '*proc*' struct, so that it will also support different priorities.

How is the priority of processes determined? We will let the application programmers take care of priorities. That is, when a process is created it will be marked as a high priority process by default. However, programmers will be able to reduce process' priority by calling the *'nice'* system call from within the process' code. Each call to nice() reduces the priority of the current process by one level - from High to Medium, from Medium to Low. Make sure you add the *'nice'* system call to xv6 (currently not supported):

```
int nice()
```

which returns 0 upon success and -1 otherwise.

- *Tip: we only support three levels of priorities at this point – be sure to handle multiple invocations of nice.*

Task 4: Sanity Test

In this section you will add applications which test the impact of each scheduling policy.

Similar 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.

Task 4.1: Test 1 – Round Robin Sanity Test

Add a program called RRsanity. This program will fork 10 child processes. Each child process will print 1000 times (each time in a new line): "child <pid> prints for the <i> time", where <pid> is that child's pid, and <i> is the iteration number. After the child process prints 1000 times, it will exit. The parent process will wait until all of its children exit, and only after they all exit, it print the waiting time, running time and turnaround time of each child.

Task 4.2: Test 2 – Guaranteed (Fair-share) Sanity Test

Add a program called Gsanity. This program will print "Father pid is <pid>", and then sleep for 10 seconds. Then it will fork once, and both child and parent will print 50 times (each time in a new line): "process <pid> is printing for the <i> time", and then exit.

Test 4.3: Sanity Test

Add a program called sanity. This program will fork 30 child processes. Each child process will have a unique id (from 0 to 29) denoted by cid. All processes whose cid%3=0 will immediately activate the nice system call once, reducing their priority to medium priority. All processes whose cid%3=1 will immediately activate the nice system call twice, reducing their priority to low priority. All processes whose cid%3=2 will remain with high priority.

The code of the child processes will print its cid 500 times and will then call exit. The parent process will wait until its children exit, and then print the average waiting time, running time and turnaround time of all its children. It will then print the waiting time, running time and turnaround time of all children in each priority group (i.e the statistics for each group), followed by the waiting time, running time and turnaround time of each child process.

- *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

Assignment due date: 9.4.2013

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 noting more. Modified files are automatically detected by svn but new files **must be added** explicitly with the 'svn add' command:

```
> svn add <filename>
```

In case you need to revert to a previous version:

```
> svn revert <filename>
```

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

```
> svn diff
```

Alternatively, if you have a diff utility such as kompare:

```
> svn diff | kompare -o -
```

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

```
> svn diff > 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, 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!

Two common pitfalls that you should be aware of:

1. 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.
2. 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.

Another useful tip is to invoke grep often to quickly navigate through the code:

```
grep -rni <search argument> *.c
```

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 svn 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:

1. Connecting from home to the labs:
   a. Install PuTTY (http://www.chiark.greenend.org.uk/~sgtatham/putty/).
   b. Connect to the host: lvs.cs.bgu.ac.il, using SSH as the connection type.
   c. 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).
   d. 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`

2. Install Linux and 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/windows-installer

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

Enjoy !!!