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

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 OS152 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 ~/os152
  cd ~/os152

- Execute the following command:

  git clone http://www.cs.bgu.ac.il/~os152/xv6.git

This will create a new folder called xv6 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. This task includes two parts. In the first part you will have to extend the current kernel functionality so as to maintain an exit status of a process and to endow the kernel with an ability to make an implicit system call exit when the process is done. In the second part you will add to the shell (a very simple) job control functionality.

Part 1

In this part you must extend the current xv6 process implementation so as to maintain an exit status. First, you must add a field to the process structure (see proc.h) in order to save an exit status of the terminated process. Next, you have to change all system calls affected by this change (e.g., exit, wait etc.). Finally, you must endow the current implementation of the exec system call with the ability to make an implicit system call exit at the time the process is exiting its main function.

Status, exit and wait:

1.1. Change the exit system call signature to void exit(int status). The exit system call must act as previously defined (i.e., terminate the current process) but it must also store the exit status of the terminated process in the corresponding structure.
   * In order to make the changes in exit system call you must update the following files: user.h, defs.h, sysproc.c, proc.c and all the user space programs that uses exit system call.
   * Note, from now on, all user space programs must supply an exit status when terminated.

1.2. Update the wait system call signature to int wait(int *status). The wait system call must prevent the current process from execution until any of its child processes is terminated (if any exists) and return the terminated child exit status through the status argument.
   * The system call must return the process id of the child that was terminated or -1 if no child exists (or unexpected error occurred).
   * Note that the wait system call can receive NULL as an argument. In this case the child’s exit status must be discarded.

1.3. Add a waitpid system call: int waitpid(int pid, int *status, int options).
This system call must act like wait system call with the following additional properties:
   * The system call must wait for a process (not necessary a child process) with a pid that equals to one provided by the pid argument.
   * The return value must be the process id of the process that was terminated or -1 if this process does not exist or if an unexpected error occurred.
   * You must support the NULL value as status argument (like in the wait system call).
We will support a simplified version of options (i.e., only two available values BLOCKING and NONBLOCKING) where the BLOCKING flag instructs the kernel to prevent the current process from execution until a process with the given pid will terminate. A waitpid call with the NONBLOCKING argument returns immediately, even the process waited for is not finished yet. In this case the return value must be -1.

1.4. Currently, each user program should explicitly perform an exit system call in order to terminate its execution. If no such call is made, the process crashes. In this task you must change the exec system call so that if a user program exits the main function, the exit system call will be implicitly performed, and its return value must be the status argument of the exit system call. The simplest way to perform this task is to inject the exit code to the process memory (in order to know its address) and push its address to the stack so that the ret command will jump to this code.

Part 2

A job is a set of related processes. In our simplified job control, each command the shell gets from the user is consider to be a job (pay attention, every job contains at least one process). You should maintain a list of jobs. Each job consists of the command itself and a list of processes created to execute the command. Each time a process finishes it should be removed from the job it belongs to. A job without processes should be removed from the jobs list.

Job control functionality in shell

In order to support simplified job control you should implement following shell commands:

1.5. jobs – prints a list of pending jobs (all jobs that have at least one process that did not terminate yet). For each job print the command that created it and a list of job processes that are still alive. The format should be as follows:
   - the first line of every job must contain a job index and the job command, for example:
     
     Job 1: read | cat &
   - the following lines describes the processes (name and pid) of this job:
     
     7: read
     9: cat
   - if no jobs currently exist, an appropriate message must appear:
     
     There are no jobs

1.6. fg – the command takes as parameter a job index and brings the job to the foreground, that is, the shell should stop running and let the job's processes run they terminate. If a user does not provide an argument to fg command, the fg selects one arbitrarily.

1.7. Write a user space program kill, that will receive a single argument – pid and will terminate a process with the pid that equals to a given one.
1.8. Write a user space program `read`, which will read user input (character by character), from the standard input, and will print the characters to the standard output. `read` will terminate when the ‘q’ character will be read.

- Tip: you can check your jobs control implementation by running `read` in the background
- Tip: if your job control is not working as desired, you can kill a process using the `kill` command

**Task 2: Performance Measures**

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, ttime, stime, retime and rtime. These will respectively represent the creation time, termination time and the time the process was at one of following states: SLEEPING, READY and RUNNING.

- Tip: These fields retain sufficient information for 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 `wait_stat` which extends the `wait` system call: `int wait_stat(int *wtime, int *rtime, int *iotime)`

Where the three arguments are pointers to integers to which the `wait_stat` function 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).

The `wait_stat` function shall return the `pid` of the terminated child process or -1 upon failure.
**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.

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.

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

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

In the next 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](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=FRR
```

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](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: FIFO Round Robin (SCHEDFLAG=FRR)**

This policy will extend the previous default 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, created or finishes waiting for I/O it is considered to be the last process to arrive and waits for its next turn to run.

**Policy 3: First Come First Served (SCHEDFLAG=FCFS)**

This is a non-preemptive policy. A process that gets CPU runs until it no longer needs the CPU (yield/finish/blocked). This policy is somewhat like FIFO Round Robin with infinite quanta.

**Policy 4: Completely fair scheduling (SCHEDFLAG=CFS)**

As discussed in the practical session, with different priorities handling (no need for group scheduling). For simplicity, supporting 3 priorities (HIGH, MEDIUM and LOW) is enough.

When a new process is created its priority should be MEDIUM. Changing the current process priority should be done by a new system call `int set_priority(int priority)` which takes as input the new process priority.

- Note: other scheduling policies do not use priorities. For simplicity, you are not required to enable / disable the `set_priority` system call according to the scheduling policy. That is, you can assume that this system call always exists regardless to the policy, but it will have no effect on scheduling policies other than CFS.
- We will assume the following decay factors for the different priorities: 1 for HIGH priority, 2 for MEDIUM priority and 3 for LOW priority.
**Task 4: Sanity Test**

In this section you will add 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.

Add a user space program called *sanity*. This program will immediately change its priority to HIGH and then fork 20 child processes. Each child process will update its priority according to its creation index (index modulo 3) and perform a time-consuming computation. This computation must take at least 30 time units (clock ticks) and you must not perform blocking system calls (e.g., `write`, `printf` etc.) during this computation. After the child process finishes its computation, it will exit with exit status equals to its pid. The parent process will wait until all its children exit. For every finished child, the parent process must validate its exit status (by comparing it to child’s pid) and print the waiting time, running time and turnaround time of each child. In addition averages for these measures must be printed.

- 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: Unless you implemented task 1, you have to call the `exit` system call to terminate a process' execution.
- Note on sanity test: before running a test you should estimate what the output should look like. After running the test you should compare the estimated output with the actual output. In case the test does not behave as you assumed, think carefully why it is so. This does not necessarily mean that your implementation is wrong. In any case, you should be able to explain the program's behavior to the grader.
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 noting 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, 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:

- **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](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:

- **Connecting from home to the labs:**
  - Install PuTTY ([http://www.chiark.greenend.org.uk/~sgtatham/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](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
  ```

- **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](http://www.ubuntu.com/desktop/get-ubuntu/windows-installer)

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