OPERATING SYSTEMS – ASSIGNMENT 1

SYSTEM CALLS, SCHEDULING AND SIGNALS

In this assignment we will extend xv6 to support an implicit call to exit, various scheduling policies implementations, and simple signal support.

You can download xv6 sources for the current work by executing the following command:

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

Task 1: Warm up ("HelloXV6")

This part of the assignment is aimed at getting you started. In most of the operation systems the termination of a process is performed by calling an exit system call. The exit system call receives a single argument called "status", which can be collected by a parent process using the wait system call. If a process ends without an explicit call to exit, an implicit call to exit is performed with the status obtained from the return value of the main function. This is not the case in xv6 – the exit system call does not receive a status and the wait system call does not return it. In addition, no implicit call to exit is performed. The following task will modify xv6 in order to support this common behavior.

In this part you are required 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. First, you must add a field to the process control block PCB (*see proc.h – the proc structure*) in order to save an exit status of the terminated process. Next, you have to change all system calls affected by this change (*i.e., exit and wait*). 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 without explicitly calling the exit system call.

1.1. Updating the exit system call:

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 proc structure.

- In order to make the changes in *the exit* system call you must update the following files: *user.h, defs.h, sysproc.c, proc.c* and all the user space programs that use *the exit* system call.
- Note, you must change all the previously existing user space programs so that each call to exit will be called with a status equal to 0 (otherwise they will fail to compile).

1.2. Updating the wait system call:

Update the `wait` system call signature to `int wait(int *status)`. The `wait` system call must block the current process 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.
- Note that like in task 1.1 (exit system call) you must change all the previously existing user space programs so that each call to wait will be called with a status equal to 0 (NULL) (otherwise they will fail to compile).

*Pay attention, when you add/change a system call, you must update both **kernel** sources and **user space programs** sources.*

1.3. Implicit call to exit:

In the current implementation of xv6 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 (`see exec.c`) so that if a user program exits the main function, the `exit` system call will be implicitly performed, and the return value of the main function must be the status argument for the implicit `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 of the main function will jump to this code.

In order to better understand how process termination works (and eventually change it) implement a user space program that does not perform an explicit call to exit upon its termination. If such a user space program is executed, it will "crash" with the following error message:

```
pid 3 test: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff—kill proc
```

The reason for such a message comes from the fact that the process (CPU) tried to perform a restricted operation – jumps to the return address of the main function, which is invalid and therefore failed. As a result the kernel "kills" this process. Your task is to find the code (`in exec.c`) that initializes the process's user space stack. First, inject a code that calls exit directly into the user space stack. Next, the main function arguments must be placed on the stack (as in the unmodified version) and finally, change the return address of the main function to point to the injected code that calls exit.

Task 2: 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 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 the scheduling takes place.

In this task, you will modify the scheduling policy to Lottery Scheduling with several different sub-policies and measure the impact of such policies on the performance of the system.

*Important note: Read the following sections **completely** before you start implementing this task.*

2.1. Changing the scheduling policy:

You are required to implement Lottery Scheduling. Lottery Scheduling is a probabilistic scheduling algorithm for processes in an operating system. Processes are each assigned some number of lottery tickets, and the scheduler draws a random ticket to select the next process. The distribution of tickets doesn't have to be uniform; granting a process more tickets provides it a relative higher chance of selection. This technique can be used with different sub-policies of tickets distribution to approximate other scheduling algorithms. In the next part of the assignment you will add three different sub-policies.

Implementation of the Lottery Scheduling will require the following steps:

- Implement a Pseudorandom number generator for the use of Lottery Scheduler.
- Update the proc structure
    - Add a single field called "ntickets" that will hold the amount of tickets allocated to the process – we will use this field in order to decide which tickets belong to which process.
    - We will assume that the processes are ordered by their pid and the numbers of the tickets owned by each process p_i are between $\sum_{j<i} ntickets_j$ and $\sum_{j\leq i} ntickets_j - 1$, where $ntickets_j$ represent the value of the "ntickets" field of the process with pid_j.
    - Example: assume two process in the system with process ids pid1 and pid2 (pid1 < pid2) and ticket allocation nt1 and nt2 accordingly. In such a case we assume that tickets 0 to nt1-1 belong to the process with pid1 and tickets nt1 to nt1+nt2-1 belong to the process with pid2.

- Change the code of the scheduler to generate a single random number (between 0 and the total number of the allocated tickets), which will represent a ticket number. The scheduler then will chose the process owning that ticket for execution.
- There are many different ways to determine how to distribute tickets between processes, You are required to implement all the policies given below and add a system call: `int schedp(int sched_policy_id)`, which will be used to change the sub-policy used – and will re-distribute the tickets accordingly.
- Write a user space program called `policy` which accepts a single argument – the policy identifier. The program must update the scheduling policy accordingly.

Policy 1: Uniform time distribution

In this policy you should distribute the tickets in such a way, so to achieve a uniform time allocation to the processes (assuming that your implementation of the random number generator achieves a uniform distribution of the return values).

Policy 2: Priority scheduling

This scheduling policy will take the process priority into consideration while deciding the number of tickets to allocate. For example, given two processes p1 and p2 having priorities 1 and 2 accordingly, process p2 will receive approximately twice the run-time received by p1. You should implement a new system call: `void priority(int);` which can be used by a process to change its priority. The priority of a new processes is 10.

Policy 3: Dynamic tickets allocation

This policy will dynamically reallocate the tickets in response to the process behavior. A newly created process will get 20 tickets. Each time a process performs a blocking system call, it will receive additional 10 tickets (up to maximum of 100 tickets) and each time a process ends the quanta without performing a blocking system call, the amount of the tickets owned be the process will be reduced by 1 (to the minimum of 1).

2.2. Measuring the performance of sub-policies:

In class, you learned about different 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 time the process spent on the *SLEEPING* state
- retime – the time the process spent on the *READY* state
- rutime – the time the process spent on the *RUNNING* state

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 (*see trap.c*) (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 task 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* status, stcuct perf *)*,where the second argument is a pointer to the following structure:

```
struct perf {

	int ctime;

	int ttime;

	int stime;

	int retime;

	int rutime;

};
```

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

2.3. Sanity test:

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

Add a user space program called `sanity`. This program will immediately fork 30 child processes. There will be 10 processes of each of the following kinds:

- CPU only: the processes will perform a cpu-only time-consuming computation (no blocking system calls are allowed). This computation must take at least 30 ticks (you can use the uptime system call to check if a tick passed).
- Blocking only: the processes will perform 30 sequential calls to the sleep system call (each of a single tick).
- Mixed: the processes will perform 5 sequential iterations of the following steps - cpu-only computation for 5 ticks followed by system call sleep of a single tick.

The parent process will wait until all its children exit. For every finished child, the parent process must print the waiting time, running time and turnaround time of each child. In addition averages for these measures must be printed.

Tips:

- 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=\". Check other built-in user space programs in xv6 (`e.g., ls, grep, echo, etc.`) for examples.
- 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.

Task 3: The Signals framework

As you have seen in class, signals are a simple form of inter process communication (currently not implemented in xv6). In this part of the assignment, you will add a framework that will enable the passing of signals from one process to another. This implementation will cover the basic features needed for a signals framework, and despite its resemblance to the Linux framework it is far from being complete.

3.1. Updating the process data structure:

Our basic implementation will be composed of a single bitset per process that will hold the pending signals, and an array of signal handlers (pointers to handler functions) together with several system calls that will allow the process to register signal handlers, send signals, etc.

The first step towards meeting this goal is to extend the *'proc' struct* located at *proc.h*. The struct should contain a field called *pending* that represents all currently unhandled (pending) signals. For example, when this field's value is 0x3 (the hex representation of the binary word 00…0011) then the process will eventually receive two signals whose identifiers are 1 and 0 (pending is a bitset). For simplicity, you may assume that your implementation will never have to support more than *32* signals (you can add appropriate definition: *#define NUMSIG 32*)

- Note that this representation means that multiple instances of a signal with the same type (having the same identifier) do not stack (are treated as a single signal).

Each signal must also be associated with some action. To support this, add an array of *NUMSIG* entries where every entry is a pointer to a function.

- Every signal is assumed to have a default handler. This handler must only print the following message: "A signal %num% was accepted by process %pid%", where %num% is the current signal identifier and %pid% is the pid of the process.

3.2. Registering signal handlers:

A process wishing to register a custom handler for a specific signal will use the following system call which you should add to xv6*:*

```
sighandler_t signal(int signum, sighandler_t handler)
```

This system call will register a new handler for a given signal number (signum). If failed, -1 is returned. Otherwise, the previous value of the signal handler is returned. The type *sighandler_t* should be defined as:

```
typedef void (*sighandler_t)(int);
```

3.3. Sending a signal:

So far, we described how a process should register to new signals. Next we will add the ability to send a signal to a process. Add the following system call:

```
int sigsend(int pid, int signum);
```

Although "kill" is the standard name for the system call in charge of sending a signal, it is already used in xv6 for terminating processes. Given a process id (pid) and a signal number (signum), the sigsend system call will send the desired signal to the process pid. Upon successful completion, 0 will be returned. A -1 value will be returned in case of a failure (think about the cases where sigsend can fail).

3.4. Getting the process to handle the signal:

Finally, you are required to implement a mechanism which will make sure that a process receiving a signal actually executes the relevant signal handler. In your extension of xv6, a signal should be handled (if at all) whenever control for the receiving process is passed from the kernel space to the user space. That is, before restoring the process context (*see trapasm.S*), the kernel first checks for the pending signals of that process. In the case where a signal handling is required, the kernel must first save the current process state, create a stack frame for the appropriate signal handler execution and change the processes control block so that it will execute the signal handler. This way, when the process returns from kernel space to user space, the code for the signal handler is executed.

In order to correctly return from signal handler execution you must implement a new system call: `int sigreturn(void)` and execute it on return from signal handler. The `sigreturn` system call must restore the process state (that was saved before executing a signal handler) to return to the normal process execution.

- Note that signal handler must receive a single argument – the identifier of the signal.
- Note: `sigreturn` system call should **never** be called directly.
  - The call for `sigreturn` system call must be implicit (see exit enhancement in Task 1)
- You should not allow recursive signal handling, i.e., to deliver a signal (execute signal handler) while performing another signal handler.
- Create a user space program that will test your signal framework implementation and submit it with your assignment. Note that your test must be extensive enough to check all the requirements defined above.

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

Working from home

The CS lab computers should already contain both git and qemu. 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/).
  - 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
```

- Install Linux and QEMU on your own PC.
- Again, we will not support problems occurring on students' personal computers.

Enjoy !!!