OPERATING SYSTEMS – ASSIGNMENT 1

SYSTEM CALLS & SCHEDULING

Responsible TAs: Simion Novikov and Erez Alfasi

Introduction

Throughout this course, we will be using a simple, UNIX like teaching operating system called xv6: https://pdos.csail.mit.edu/6.828/2018/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 all 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/6.828/2018/overview.html
- xv6 has a very useful guide. It will greatly assist you throughout the course assignments: https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf
- You may also find the following useful: https://pdos.csail.mit.edu/6.828/2018/xv6/xv6-rev11.pdf

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

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

Task 0: Running xv6

Begin by downloading our revision of xv6, from our course Assignment 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 ~/os202
  ➔ cd ~/os202
- Execute the following command:
  ➔ `git clone` https://github.com/mit-pdos/xv6-public.git
  ➔ cd xv6-public/
  Build xv6 by calling make:
  ➔ make
- Run xv6 on top of QEMU by calling:
  ➔ make qemu

Task 1: Warm up- User Space

The goal of this part of the assignment is to get you started. The objective is to write a userspace-only program. You will need to write such programs for debugging/ testing assignments and tasks.

Create the file "helloworld.c"

When a program is executed, the shell seeks the appropriate binary file in the current working directory and executes it by calling the "exec" system call. In order to do that, you'll need to understand how a user-space program is working. You can take a look on an existing user-space only program (for example: echo.c) – see how its working, and how it's being compiled within xv6 makefile (**Hint**: seek for 'echo.c' and '_echo' in the Makefile).

Write a simple c program – helloworld.c, and add it to the Makefile.
The program should simply print "Hello World Xv6".
For testing, execute the program from the shell:

```
$: helloworld
Hello World XV6
```

Task 2: Warm up- Kernel Space & System Calls:

The goal of this part of the assignment is to get you started with system calls. The objective is to implement a simple system call that outputs the size of the running process' memory in bytes.

The system call signature should be:
int memsize(void)

Hint: Look at the list of system calls, and find one that is very similar in its logic of getting data from the current running process. The size of a given process can be obtained from the PCB.

Write a user-space program that tests this system call, much like in task 1: It should

1. Print how much memory the process is using.
2. Allocate more memory by using malloc (array containing 2 KB).
3. Print how much memory the process using now after the allocation.
4. Free the array that was allocated in 2.
5. Print how much memory the process is using.

```
$: memsizetest
The process is using: 2048B
Allocating more memory
The process is using: 4096B
Freeing memory
The process is using: 4096B
```

Notice that ‘free()’ doesn’t reduce the memory consumption of a process, since it doesn’t necessarily release the memory back to the operating system. Most often, it just puts back that memory area in a list of free blocks. These free blocks could be reused for the next calls to ‘malloc()’.

Task 3: The wait and exit systems calls

In most 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. 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.. The following task will modify xv6 in order to support the common behavior.

In this part you are required to extend the current kernel functionality so as to maintain an exit status of a process. You must add a field to the process control block PCB (see proc.h – the proc structure) in order to store the exit status of the terminated process. Then, you have to change all the calls for 'exit()' and 'wait()' affected by this change or otherwise – the kernel won't compile (You changed both signatures – then you must change the way its being used across files).

3.1. 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 upon success and 1 upon error. (There will be a compilation error on every place that still calling exit(void) since you've changed its signature).

3.2. Updating the wait system call:

Update the *wait* system call signature to `int wait(int *status)`. The `wait` system call must block the execution of the calling process until any of its child processes terminate (if any exist), and return the terminated child's exit status through the `status` argument.

- The system call must return the ***process id*** of the terminated child or ***-1*** if no child exists (or if an unexpected error occurred).
- Note that the `wait` system call can receive `null` (define `null` in types.h) as an argument. In this case, the child's exit status must be discarded.
- Note that like in task 2.1 (exit system call), you must change all the previously existing user space programs so that each call to wait will use the new signature.

Note: when you add/change a system call, you must update both kernel sources and user-space program sources.

Task 4: Scheduling Policies

Important: Read the entire task carefully before you start implementing it. Read the notes at the end of this task.

Scheduling is a basic and important service of any operating system. The scheduler aims to 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 (i.e. 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.

In this assignment, you are required to replace the current scheduling policy of Xv6 by testing each of the following policies and measuring the impact of these policies on the performance of the system.

4.1. Default - the existing scheduler:

Use the default scheduling policy already defined in Xv6, which is round robin.

4.2. Priority Scheduling:

This scheduling policy is based on accumulating values in a manner we explain soon. This accumulation takes process priorities into consideration. Whenever the scheduler needs to select the next process to execute, it will choose the process with the lowest accumulated number. To support this mechanism, you should add a field to the PCB (Process Control Block) data structure (defined in the proc.h file) that will store this accumulated value, named `accumulator`. You should implement a new system call: `int set_ps_priority(int)`, which can be used by a process to change its priority. The priority of a new process is 5, the lowest priority is 10 and the highest priority is 1 (thus, lower values represent higher priorities). Each time a process finishes its time quantum (that is, each time it exhausts it and remains runnable), the system should add the process' priority to its `accumulator` field in the PCB. We advise you to define this field as type long long (yes, **long long**, this is not a mistake), because its value can grow to be very large.
Each time a new process is created or a process shifts from the blocked state to the runnable state, the system should set the value of its `accumulator` field to the minimum value of the `accumulator` fields of all the runnable / running processes. If it is the only runnable process in the system, the system should set its accumulator value to 0. This gives high priority to new processes or to processes that are returning from I/O.
Whenever the scheduler needs to select a new process to run, it selects a process with the lowest `accumulator` value. In case of tie between 2 processes (or more) with the same `accumulator` value, run the first in order within the array.

Note that the `accumulator` fields of processes with high priorities (small values) grow more slowly, hence the scheduler will favor such processes.

Advise: add the following fields to the PCB: ps_priority and accumulator.

4.3. Completely Fair Schedular with Priority decay:

A preemptive policy inspired by Linux CFS (but different from it).
Define the following:
rtime: time process was in running state.
stime: time process was in sleeping state.
retime: time process was in ready/runnable state.
Each time the scheduler needs to select a new process it will select the process with the smallest run time ratio: $\frac{\text{rtime} \times \text{decay factor}}{\text{rtime+wtime}}$. Note that "wtime" as referenced here is the sum of retime and stime. In case of a tie, the first process in the array that tied will be selected. The decay factor is defined using process priority. Three priority levels should be supported:

1. High priority – Decay factor = 0.75
2. Normal priority – Decay factor = 1
3. Low priority – Decay factor = 1.25

Priority will be set by a new system call that you will add that will change the priority of the current process. `int set_cfs_priority(int priority)`. Its input is the new priority value for the current process (1-3). Its Output is one of the following values:

• 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.
Advise: add the following fields to the PCB: decay factor (cfs_priority), rtime,stime, retime.

4.4. Dynamically changing the scheduling policy:

In order to be able to select a desired scheduling policy, you are required to implement a system call: `int policy(int)`. This system call receives a policy identifier (0 – for Default, 1 – for Priority Scheduling and 2 – for CFS) as an argument and changes the currently used policy.

A global variable that keeps the scheduling type should be saved at the end of defs.h:
int sched_type;

Implement a user-space program called “policy.c” that receives a single argument (via argc, argv), which is the code of the new policy, and performs a call to the *`policy`* system call. It will print a success message if the policy change is successful, otherwise, print an error message.

```
$: policy 0
Policy has been successfully changed to Default Policy
$: policy 1
Policy has been successfully changed to Priority Policy
$: policy 2
Policy has been successfully changed to CFS Policy
$: policy 3
Error replacing policy, no such a policy number (3)
```

4.5.Ouput Processes information

In class, you have learnt about several quality measures for scheduling policies. In this task, you are required to measure your new scheduling policies performance according to these measures. The first step is to extend the proc struct (see proc.h) by adding the following fields to it:

- ps_priority – from task (4.2).
- stime – the total time the process spent in the SLEEPING state.
- retime – the total time the process spent in the READY state.
- rtime – the total time the process spent in the RUNNING state.

Note - The statistics gathered for each scheduling policy should continue to be gathered even when other policies are active, so that when we switch policies, the information remains correct.

These fields (for each process) should be updated for all processes whenever a clock tick occurs (see trap.c, where tick occurs).

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, implement a new system call: `int proc_info(struct perf * performance)`, where the argument is a pointer to the following structure:

```
struct perf {
    int ps_priority;
    int stime;
    int retime;
    int rtime;
};
```

For testing, create a user program "sanity.c" that shows how the priority level and the scheduling policy affect process waiting times.

This program should fork 3 processes and wait until all of them terminate. Each child process should perform a simple loop of 1,000,000 iterations as follows:

```
..
int i = 1000000;
int dummy = 0;
while(i--)
    dummy+=i;
..
```

Each child process should then print its statistics (using system call `proc_info()`) before it exits. The first child will be set as low priority, the second child will be medium priority, and the last child will be set as high priority (using both the *`set_cfs_priority()`* and the *`set_ps_priority()`* functions).

The testing will be repeated for all 3 scheduling policies. We will measure the statistics in the following manner:

```
~$: policy 0
Policy has been successfully changed to Default Policy
~$: sanity
PID    PS_PRIORITY        STIME       RETIME      RTIME
900    10                 100     …
901    …
902    …
~$: policy 1
Policy has been successfully changed to Priority Policy
~$: sanity
…
```

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.

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 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!

Enjoy !!!