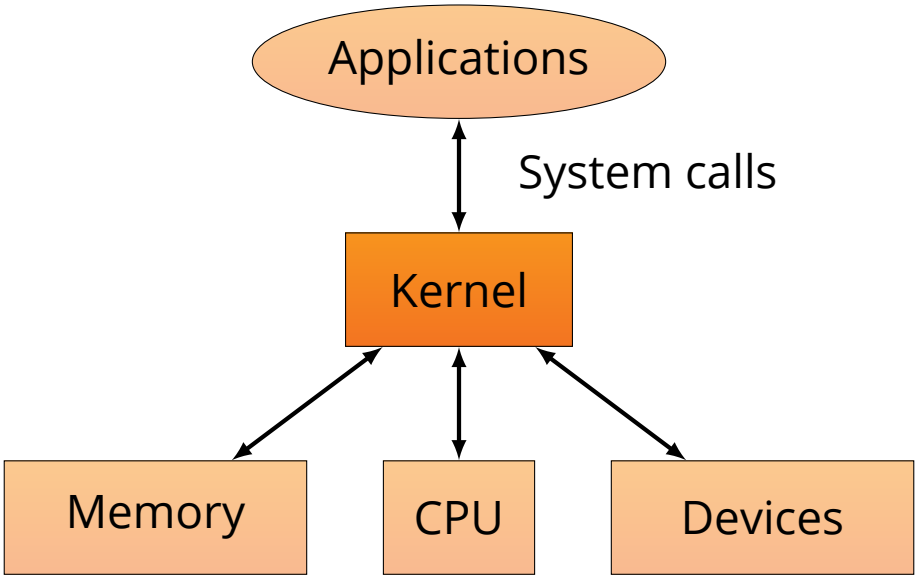
Operating Systems 202.1.3031

Spring 2022/2023 Assignment 1

System Calls and Scheduling

Responsible Teaching Assistants:
Ido Ben-Yair and Hadar Cochavi Gorelik



Ben-Gurion University
of the Negev

Contents

1 Introduction

Welcome to the world of operating systems!

Throughout this class, we will be using a simple, UNIX-like teaching operating system called **xv6**. Specifically, we use the RISC-V version of xv6 called **xv6-riscv**. 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 is developed as part of MIT's 6.828 Operating Systems Engineering class. You can find a lot of useful information and getting started tips here:
https://pdos.csail.mit.edu/6.828/2022/overview.html
See also the xv6 book, which can also serve as a reference for our class:
https://pdos.csail.mit.edu/6.828/2022/xv6/book-riscv-rev3.pdf

In the following sections we will perform a few tasks to get you familiar with the xv6 OS and operating system development in general.

On a more personal note, we are very excited to be teaching this class. We realize there is a lot to learn, but we are here to help.

An operating system, even a simple one like xv6, is a complex piece of software. Such low-level code is often challenging at first. This stuff takes time, but it can be fun and rewarding! We hope this class will serve to bring together a lot of what you've learned in your degree program here at BGU. **Take a deep breath and be patient with yourself.** When things don't work, keep calm and start debugging!

Good luck and have fun!

2 Submission Instructions

- Make sure your code compiles without errors and warnings and runs properly!
- We recommend that you comment your code and explain your choices, if needed. This would also be helpful for the discussion with the graders.
- You should submit your code as a single `.tar.gz` or `.zip` file.
- We advise you to work with git and commit your changes regularly. This will make it easier for you to track your progress, collaborate and maintain a working version of the code to compare to when things go wrong.
- Theoretical questions in this assignment are **not for submission**, but you are advised to understand the answers to these questions for the grading sessions.
- Submission is allowed **only in pairs** and **only via Moodle**. Email submissions will not be accepted.
- Before submitting, run the following command in your xv6 directory:

  ```
  $ make clean
  ```

  This will remove all compiled files as well as the `obj` directory.
- Help with the assignment and git will be given during office hours. Please email us in advance.

3 Task 0: Compiling and Running xv6

Start by downloading the xv6 source code from the class GitHub.

Tasks will appear in this assigment in orange boxes like this one:

Download and compile xv6

1. Open a shell or terminal in your virtual machine. We will use the **bash** shell in this document.
2. Point your shell to the directory where you want to download the xv6 source code. For example:

   ```
   $ cd ~/projects/os
   ```
3. Run the following command in your terminal:

   ```
   $ git clone https://github.com/BGU-CS-OS/xv6-riscv.git
   ```
4. You can then move into the source directory to inspect the source code or open it in your favorite editor:

   ```
   $ cd xv6-riscv
   ```
5. Build xv6 by running the following command:

   ```
   $ make
   ```
6. Make xv6 run in QEMU by running the following command:

   ```
   $ make qemu
   ```

Note: Every terminal or shell command in this document is preceded by a dollar sign ($). This is a convention used to distinguish between commands and their output. You should not type the dollar sign when running the commands. Also, your terminal prompt may be different than the one shown here.

4 Task 1: Warm up – Hello World

Let's get started by writing a simple program that prints `"Hello World"` to the screen! We will need to write user space programs to debug and test our kernel code.

We will be writing a user space-only program, meaning that it does not run in the kernel. However, note that our program is intended to run on xv6 inside the QEMU emulator, not on our host operating system (e.g., Linux, macOS, Windows, etc). It's important to understand that the host system can't run or debug our program directly!

In order to build a new user space program, you'll need take a look at an existing user space program. For example, examine `echo.c` — see how it works, and how it gets built by the xv6 makefile.
Hint: find `echo.c` and `"_echo"` in Makefile.

Then, complete and submit the following task:

Task 1 – write the `Hello World` program

The following steps will guide you through the process of writing a `Hello World` program:

1. Create a new file called `helloworld.c` in the `user` directory of xv6.
2. Add `helloworld.c` to the `Makefile` in the top-level directory.
3. Write C code that prints `"Hello World xv6"` to the screen.
4. Compile your program by running the following command:

   ```
   $ make qemu
   ```

   This essentially builds the entire xv6 system, including your program. If your program isn't built, check the `Makefile` to make sure you added it correctly.
5. Run your program by running the following command:

   ```
   $ ./helloworld
   ```

For submission: `helloworld.c` and modified `Makefile`.

5 Task 2: Kernel Space and 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 and a user space program to test it.

To accomplish this, you will need to modify the xv6 kernel code and add a new system call, called `memsize()`. The (user space) signature of this new system call should be:

```
int memsize(void);
```

This system call should return the size of the running process' memory in bytes. Look at the list of system calls, and find one that performs a similar function, getting data from the current running process. The size of a given process can be obtained from the PCB.

Then, complete and submit the following task:

Task 2 – Implement memsize system call

1. Add a new system call to the list of system calls in `syscall.h`.
2. Add the appropriate additions in `syscall.c`.
3. Implement `sys_memsize()` in `sysproc.c`.
4. Add the user space wrapper for the new system call in `usys.pl` and `user.h`.
5. Write the user space test program `memsize_test.c`:
   (a) Print how many bytes of memory the running process is using by calling `memsize()`.
   (b) Allocate 20k more bytes of memory by calling `malloc()`.
   (c) Print how many bytes of memory the running process is using after the allocation.
   (d) Free the allocated array.
   (e) Print how many bytes of memory the running process is using after the release.

Questions:

1. How much memory does the process use before and after the allocation?
2. What is the difference between the memory size before and after the release?
3. Try to explain the difference before and after release. What could cause this difference? (Hint: look at the implementation of `malloc()` and `free()`).

For submission: `memsize_test.c`, and modified OS files.

6 Task 3: Goodbye World

In most operating 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, or if the parent process is the shell, the status is returned to the shell (bash, for example, makes the exit status code available as the special variable `$?`).

The goal of this task is to add an "exit message" to `exit()`. This exit message will be saved in the PCB and can be retrieved by the parent process using the `wait()` system call, which we will modify as well. Then, we will write a user space program called `goodbye.c` that immediately calls `exit()` with the message "Goodbye World xv6". Finally, to show that we can retrieve the exit message, we will modify the shell to print the exit message when a child process of the shell terminates.

Task 3 – Implement exit message

1. Add a new field to the PCB called `exit_msg` of type `char[32]`.
2. Modify the `exit()` system call to receive an additional argument of type `char*` and save it in `exit_msg`.

   Guidance: use `argstr()` to copy the string from user space to kernel space, rather than attempt to copy it directly. We will understand why this is important later on in the class when we talk about *virtual memory*. Look for uses of `argstr()` in the xv6 code to see how it is used. You may assume that `exit()` is always called with a valid, non-null (0) string pointer.
3. Modify the `wait()` system call to accept an additional pointer, which will be used to copy the exit message of the child process to the caller of `wait()`.

   Guidance: use `copyout()` to copy the string from kernel space to user space, for the same reason as above. You may assume that `wait()` is always called with a valid, non-null (0) string pointer.
4. Write a user space program called `goodbye.c` that immediately calls `exit()` with the message "Goodbye World xv6".
5. Modify the shell to print the exit message when a child process of the shell terminates.

Questions:

1. What happened as soon as we changed the signatures for `exit()` and `wait()`? Why?
2. What happens if the exit message is *longer* than 32 characters? How do we make sure nothing bad happens?
3. What happens if the exit message is *shorter* than 32 characters? How do we make sure nothing bad happens?
4. How many times is our exit message copied?
5. Where in `sh.c` does the shell receive the exit message? Explain briefly how this code works.
6. What happens if the shell modifies the exit message after it is received?

For submission: `goodbye.c`, modified `sh.c`, and modified OS files.

7 Scheduling Policies

7.1 Task 4: Understand the existing scheduling policy

Scheduling processes is a basic and important service of any operating system. The process scheduler aims to satisfy several conflicting objectives: fast process response time, good throughput for jobs, avoidance of process starvation, reconciliation of the needs of low-priority and high-priority processes, and many more. The set of rules used to determine when and how to select a new process to run is called a *scheduling policy*.

First, we need to understand the current (i.e., existing) scheduling policy in xv6:

Task 4 – Understand the current scheduling policy

Questions:

1. Find the scheduling policy in the xv6 code. Where is it implemented?
2. How does the policy choose which process to run?
3. What happens when a new process is created and when/how often does scheduling take place?
4. What happens when a process calls a system call, for instance `sleep()`?

For submission: Nothing.

In the following tasks, we will replace the current scheduling policy by testing two new policies and measuring the impact of these policies on the performance of the system.

7.2 Task 5: Priority Scheduling

This scheduling policy is based on accumulating values in a manner that 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 value. To support this mechanism, you should add a field named `accumulator` to the PCB (defined in `proc.h`) that will store this accumulated value. We advise you to define this field as type `long long`, because its value can grow to be very large. You should implement a new system call:

```
void set_ps_priority(int);
```

which can be used by a process to change its own 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, but remains runnable), add the process' priority to its accumulator field. Each time a new process is created, or a process shifts from blocked to runnable, set its accumulator to the minimum value of the accumulators of all the runnable/running processes. If it is the only runnable process in the system, set its accumulator to 0. This gives high priority to new processes or to processes returning from I/O. In case of a tie between two or more processes 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, so the scheduler will favor them.

Task 5 – Implement priority scheduling

1. Add the `long long accumulator` and `int ps_priority` fields to the PCB.
2. Update these fields when the appropriate events occur (see `trap.c`, `proc.c`).
3. Implement the `set_ps_priority()` system call.
4. Modify the scheduler and other places in the OS as needed.

For submission: Modified OS files that implement priority scheduling and the `set_ps_priority()` system call.

7.3 Task 6: Completely Fair Scheduler with Priority Decay

This preemptive scheduling policy is a simplified version of the Completely Fair Scheduler (CFS) algorithm available in Linux. Each time the scheduler needs to select a new process it will select the process with the smallest *virtual runtime* (or *vruntime*) given by:

$$\text{vruntime} = \text{decay factor} \times \frac{\text{run time}}{\text{run time} + \text{sleep time} + \text{runnable time}}$$

In case of a tie, the first process in the array that tied will be selected.

The decay factor[1] is defined using the concept of a process priority. Three priority levels should be supported:

1. **High priority:** decay factor = 75.
2. **Normal priority:** decay factor = 100.
3. **Low priority:** decay factor = 125.

The default priority for a new process is normal. Calling `fork()` will copy the priority of the parent process to the child process.

Priority will be set by a new system call:

```
int set_cfs_priority(int priority);
```

which takes a priority value (0, 1, or 2) and returns 0 on success and -1 on failure.

[1] **Note:** since our programming environment does not support floating point operations, the decay factor is defined using integers and you should implement vruntime using integer arithmetic only.

Task 6 - Implement CFS with Priority Decay

1. Implement the CFS with Priority Decay scheduling policy as described above: add the `cfs_priority`, `rtime`, `stime`, and `retime` fields to the PCB, corresponding to the run time, sleep time, and runnable time of the process, respectively.
2. Make sure to update the values of these fields on every clock tick (see `trap.c`) and in other relevant places.
3. Modify the scheduler.
4. Implement the `set_cfs_priority()` system call.
5. Implement a `get_cfs_stats()` system call that takes a process ID and returns the values of the `cfs_priority`, `rtime`, `stime`, and `retime` fields for that process to user space.
6. Write a test program `cfs.c` that:
   (a) Forks 3 processes with a low, normal and high priority respectively.
   (b) Each child should perform a simple loop of 1,000,000 iterations, sleeping for 1 second every 100,000 iterations.
   (c) Each child process should then print its PID, CFS priority, run time, sleep time, and runnable time.

Questions:

1. Is run time and sleep time of a process the same thing? If not, what is the difference between them?
2. Is the run time and runnable time of a process the same thing? If not, what is the difference between them?
3. Run the test a few times. Did you notice any difference in the run times, sleep times, and runnable times of the three child processes? Explain your results.
4. Do the three priority levels actually matter for which process gets more CPU time? Does it make sense to give a higher priority to a process that we want to run more often?
5. Does this policy make sense for a real-life operating system? Explain your answer.

For submission: `cfs.c` and modified OS files that implement CFS with Priority Decay and the `set_cfs_priority()` and `get_cfs_stats()` system calls.

7.4 Task 7: Dynamically changing scheduling policies

In a real-life operating system, the user or administrator may want to change the scheduling policy at runtime. In order to be able to select a desired scheduling policy, we will implement a system call:

```
int set_policy(int policy);
```

which takes a policy value 0, 1, or 2, for the default xv6 policy, priority scheduling, and CFS with priority decay, respectively, and returns 0 on success and -1 on failure.

A global variable that keeps the scheduling type should be declared at the end of `defs.h`:

```
int sched_policy;
```

Task 7 - Implement dynamically changing scheduling policies

1. Implement the `set_policy()` system call.
2. Write a test 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 `set_policy()` system call with the given argument and reports success or error.

For submission: `policy.c` and modified OS files that implement the `set_policy()` system call.