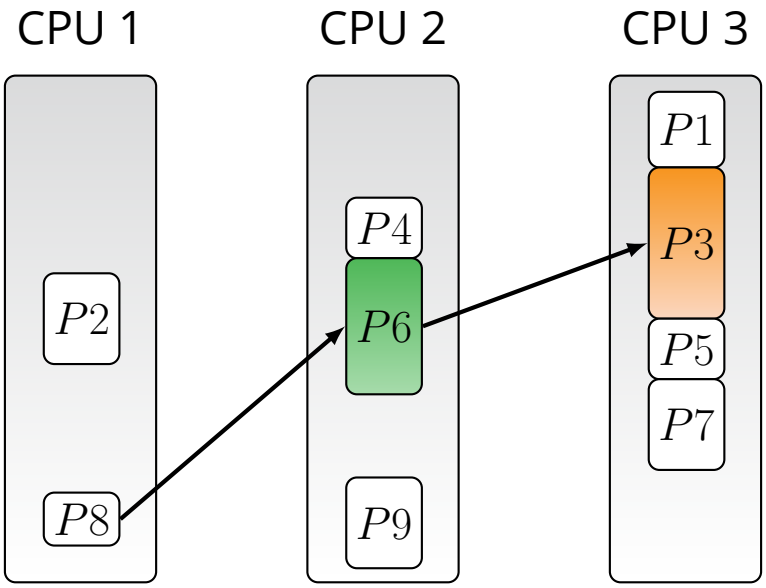
Operating Systems 202.1.3031

Spring 2024 Assignment 1

System Calls and Scheduling

Responsible Teaching Assistants:
Ido Ben-Yair and Eldar Zrihen



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. We recommend to use VS Code with our dev container. See instructions on the course website.
6. Build xv6 by running the following command:

   ```
   $ make
   ```

7. 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 to 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 — Hello World

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.

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.

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

Questions — not for submission

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()`).

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`.

   If the process passes `0` as the string address, the `exit_msg` field is set to the constant string "No exit message".

   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.
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. 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.

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

Questions — not for submission

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?

7 Per-CPU Scheduling Affinity

In the following tasks, we will give the xv6 scheduler the ability to run processes only on specific CPUs. This is useful in systems with multiple CPUs, where we want to control which CPUs may run which processes, for example, to avoid cache thrashing when processes constantly move between cores, or to balance the load between CPUs.

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:

Questions — not for submission

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()`?

7.2 Task 5: CPU Affinity Mask

In order to restrict certain processes to specific CPUs, we will add a new field to the PCB called `int affinity_mask`. This mask will be an integer, where each bit represents a CPU, i.e., bit 0 represents CPU 0, bit 1 represents CPU 1, and so on. If a bit is set to 1, the process is allowed to run on that CPU. Otherwise, the process will not run on that CPU. A mask set to 0 means the process is allowed to run on any CPU.

For example, the following mask will only allow the process to run on CPUs 0 and 2 (in this example, our mask is 16 bits wide):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
—	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

In addition, we will add a new system call:

```
void set_affinity_mask(int);
```

which will allow a process to set its own affinity mask.

Note that we chose to use a signed integer to make it easy to fetch the value of the mask using `argint()` in the system call implementation. Recall that system calls are kernel functions that are called indirectly by user space programs, so passing arguments to them is a bit more complicated than calling a regular function. The OS must copy the arguments from user space to kernel space using a special mechanism, and thus provides us with special functions to fetch these arguments. In the case of xv6 over RISC-V, the arguments are passed in registers, and the kernel must copy them to memory before our code can use them. For our system call, we will use `argint()` which does just that for an integer argument.

To implement the CPU restriction for each process, we will modify the scheduler to take the affinity mask into account when selecting the next process to run. We will also modify various parts of the OS to ensure something sensible happens to the affinity mask when a process is created, exits, or other process lifecycle events occur.

Finally, we will write a test program that sets its own affinity mask and add a print command to the scheduler to show that our changes restrict this process to run only on the CPUs we specified.

Task 5 — Implement CPU affinities

1. Add the `int affinity_mask` field to the PCB.
2. Initialize the field to 0 in `procinit()`, `allocproc()` and when the process is freed in `freeproc()`.
3. When the process tries to fork, we will assume the child process should be restricted to the same CPUs as the parent. Modify `fork()` accordingly.
4. When a process calls `exec()`, we will allow the "new" process to run on any CPU. Modify `exec()` accordingly.
5. Modify the scheduler: fetch the ID of the current CPU with `cpuid()` and check if the process is allowed to run on this CPU (or any CPU if the mask is 0).
6. If the scheduler decides to run a process, print a message to the console with the process ID and the CPU ID.
7. Add the `set_affinity_mask()` system call that accepts an integer mask and sets the affinity mask of the calling process. Use `argint()` to fetch the mask, and see how it is used in other system calls.
8. Finally, write a user space program in `affinity_test.c` that sets its own affinity mask and then runs a loop that prints its own process ID. During the grading sessions you will be asked to change this mask and see how the process behaves.

For submission: See next task.

7.3 Task 6: Load Balancing

We will now simulate a simple load balancing mechanism in xv6 using our new CPU affinity mask mechanism. First, set the affinities for our test program to run only on CPUs 0 and 2. Verify that your operating system is compiled with at least 3 CPUs.

Using the following simple algorithm, we will make sure our test program (or any other process with a non-zero affinity mask) does not run on the same CPU for two time slices in a row. Add a field called `effective_affinity_mask` to the PCB. This field should be initialized properly just like the `affinity_mask` field. Now, when a process yields the CPU, we will update the effective affinity mask to exclude the current CPU by setting the corresponding bit to 0. When the scheduler selects a process to run, it will check the effective mask instead of the original mask to decide if the process is allowed to run on the current CPU. Finally, when the effective mask has no bits set, we will reset it to the `affinity_mask`.

Our test program should now appear to ”jump” between CPUs 0 and 2, as the scheduler will not allow it to run on the same CPU for two subsequent time slices.

Important notes:

- Disable the processor jumping logic for processes with an affinity mask of 0, which can run on any CPU.
- A process can yield the CPU wherever a call to `sched()` is made in the kernel, for example, when the process calls `sleep()`. There are three such places in the xv6 kernel, but for simplicity we will assume we only move the process when it has exhausted its time slice.

Task 6 — Implement load balancing

1. Modify the test program to set its affinity mask to run on CPUs 0 and 2.
2. Add the `int effective_affinity_mask` field to the PCB.
3. Initialize the field to 0 where appropriate and update it when a process yields the CPU or when the affinity mask is updated by the user using `set_affinity_mask()`.
4. Reset the effective mask back to the affinity mask when it has no bits set.
5. Think about what needs to happen during `fork()` and `exec()`. Do something reasonable with the effective mask, but don't worry too much about this.
6. Modify the scheduler to use the effective mask instead of the affinity mask.

For submission: Modified OS files that implement the CPU affinity masks and the test program.

Questions — not for submission

1. Where does the process yield the CPU in the xv6 code?
2. What happens when a process yields the CPU? How does the OS return to the scheduler?
3. What is the difference between yielding via `yield()` vs the other calls to `sched()`?
4. How does the effective mask ensure that a process does not run on the same CPU for two time slices in a row?
5. Without our affinity mechanism, can a process decide which CPU it runs on? Can it use our mechanism to force itself to run on a specific CPU?