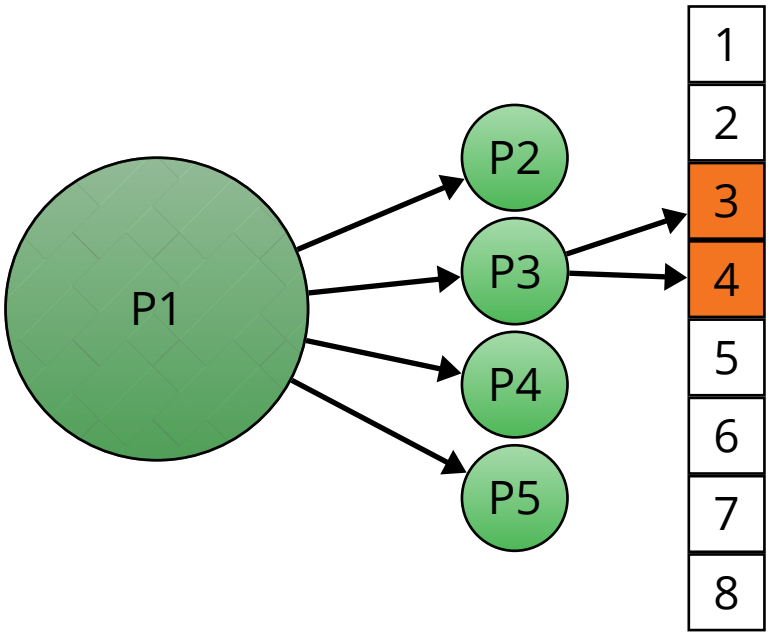
Operating Systems 202.1.3031

Spring 2025 Assignment 1

Processes and Scheduling



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.
- Submit a **single copy** of xv6 containing all of the changes you made for all tasks, in a single `.tar.gz` or `.zip` file. All tasks must be completed in the same copy of xv6 without breaking the operating system or the other tasks.
- 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 and building it inside a Docker container. This will allow you to work in a controlled environment and avoid any issues with your host operating system.

We recommend to use Visual Studio Code with the dev containers extension, which also requires Docker to run the container. See section A for more information on how to set up your work environment.

Follow the instructions below to get started with xv6. Tasks you are required to complete will appear in orange boxes like the one in the next page.

Download and compile xv6

1. Open a shell or terminal. 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
   ```

   You can also use VS Code or any git client to clone the repository if you prefer. You may also download it from GitHub since we do not require the use of git in this class. It is, however, recommended to use git for version control while working on the assignments.

4. Open the project directory in VS Code, which will ask you to reopen the project inside the dev container. It will then build the container and reopen the project inside. This will take a few minutes. At the end, the project should be open with an indication that it is embedded in a dev container.
5. In a VS Code terminal, build and run xv6:

   ```
   make qemu
   ```

6. The emulator will start up and run xv6. Once it boots up, you will see the xv6 shell prompt. You can now interact with the xv6 system. Try running the `ls` command.
7. To exit QEMU, press `ctrl-a` followed by `x`.

For submission: No submission is required for this task.

4 Task 1: Hello World

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

We will be writing a userspace 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 userspace program, you'll need to take a look at an existing userspace 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 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 the project 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 in the xv6 shell:

   ```
   helloworld
   ```

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

5 Task 2: Kernelspace 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 userspace 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 (userspace) 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 code in `syscall.c`.
3. Implement `sys_memsize` in `sysproc.c`.
4. Add the userspace wrapper for the new system call in `usys.pl` and `user.h`.
5. Write the userspace 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 any modified OS files.

Questions — not for submission

1. How does calling a system call differ from calling a regular function? How does it work?
2. How are parameters passed to the system call function? How is the return value returned to userspace?
3. What is the purpose of the `usys.pl` file?
4. What is `struct proc` and where is it defined? Why do we need it? Does a real-world operating system have a similar structure?
5. How much memory does our program use before and after the allocation?
6. What is the difference between the memory size before and after the release?
7. Try to explain the difference before and after release. What could cause this difference? (Advanced: 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 userspace 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 userspace to kernelspace, 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 kernelspace to userspace. You may assume that `wait` is always called with a valid, non-null (0) string pointer.
4. Write a userspace 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 any modified OS files.

Questions — not for submission

1. What happened as soon as we changed the signatures for `exit` and `wait`? Why?
2. Why do we need to add a new field to the PCB for the exit message? Why can't the shell just read the message from the exiting process?
3. When does the shell get the exit message from the child process? Does it happen immediately after the child process exits?
4. What happens if the exit message is *longer* than 32 characters? How do we make sure nothing bad happens?
5. What happens if the exit message is *shorter* than 32 characters? How do we make sure nothing bad happens?
6. How many times is our exit message copied?
7. Where in `sh.c` does the shell receive the exit message? Explain briefly how this code works.
8. What happens if the shell modifies the exit message after it is received?

7 Task 4: Distributing Work by Forking

In this task, we will process a large array of data by distributing the work among multiple processes. To accomplish this, we will implement special versions of the `fork` and `wait` system calls that respectively create and wait for more than one child process.

You are required to implement this task (and all tasks in this assignment) in the same copy of xv6 without breaking the operating system or the other tasks.

Our end goal is to create a userspace program called `bigarray.c`. It will initialize an array of 2^{16} integers with consecutive numbers starting from 0, and calculate the sum of all numbers in the array, i.e., $0+1+2+\ldots+2^{16}-1$. However, since performing this work element by element is time-consuming, we will distribute the work among four child processes, where each process will process a quarter of the array, i.e., $2^{16}/4$ elements. Each child process will calculate the sum of the elements in the sub-array it was assigned and print it. Each sum will be communicated to the parent process using the exit status of the child process. Once all child processes have finished, the parent process will print the sum of the sums.

Since there is no way to synchronize the child processes in xv6, the order of the printed sums may vary, except for the last print, which will happen after all child processes have finished. If your implementation is correct, the sum of the sums should be equal to $\sum_{k=0}^{2^{16}-1} k = 2147450880$.

Our goal requires us to create multiple child processes. It's important that all child processes are created successfully before our parent process continues. We don't want to leave unnecessary child processes in the system, in case our program enters this inconsistent state, as this would be a waste of resources, and the operating system has no way of knowing that these large processes are no longer needed. Therefore, we will implement a new system call called `forkn` that will create multiple child processes and delete them all if the operation fails part-way through. We will also need to establish a mechanism to wait for all child processes to finish, instead of waiting for a single child process to finish. To do this, we will implement a new system call called `waitall`.

Implement the following system calls to accomplish this task:

1. `int forkn(int n, int* pids);`

   This system call will create `n` child processes and return their PIDs via the pointer `pids`. On the parent process, it will return 0 if successful, and -1 if the operation failed. Failure may occur if the number of child processes is less than 1 (including negative numbers) or greater than 16. On each child process, it will return the number of that process in relation to the fork operation, for example, 1, 2, 3, or 4 (note: this is not the PID, and 0 is reserved for indicating that the process is the parent).

   This call assumes that the pointer `pids` points to an array of at least `n` integers. Use the `copyout` function to copy the PIDs to the userspace array.

   The processes allocated by this call **must only be released** to a runnable state when **all** children have been created successfully. Since each individual process creation operation may fail, **all already allocated child processes must be cleaned up and the call should return -1.**

   Do not use or modify the `fork` function in this implementation, and it would not meet the requirements of this task anyway.

2. `int waitall(int* n, int* statuses);`

   This system call will wait for all child processes to finish. It will return the number of child processes that have finished via the pointer `n` and the exit status of each child process via the pointer `statuses`. Upon successful completion, it will return 0. Otherwise, it will return -1.

   If no child processes are found, it will return 0, set `n` to 0, and not modify `statuses`. This call must not return to the calling process if child processes are found that are still operating, i.e., not yet in a `ZOMBIE` state. If an error occurs during the operation, the call will return -1 and not modify `n` or `statuses`.

   The caller must ensure that `statuses` points to an array of at least `NPROC` integers, where the exit status of each child process will be stored. This is because the maximum number of processes in the system is `NPROC` (by default 64, defined in `kernel/param.h`). Like in `forkn`, use the `copyout` function to copy the number of processes and exit statuses to the userspace array.

   This call ignores exit messages from child processes (that we implemented in Task 3) and does not return them to the parent process.

   Do not use or modify the `wait` function in this implementation.

Upon calling `forkn`, the parent will receive the return value of the system call, and the child processes will receive the number of the process in relation to the fork operation. In the parent, print the PIDs of the child processes and check that the call to `forkn` was successful by checking the return value. If the fork operation fails, the parent process should print an error message and exit. For the purposes of this task, we do not care about the exit message returned by the child processes. Return an empty string in the exit message of the child processes and a message indicating that the calculation has completed (or failed) in the parent process.

Also, when calling `waitall`, the parent process will receive the number of child processes that have finished and their exit statuses. Check that the number of children that `waitall` returns is equal to the number of children created by `forkn`.

Another important consideration is that, as learned in class, we must not access the fields of any process without first acquiring the appropriate lock, **not even for reading!** Conversely, holding more than one process lock at a time is problematic for reasons that we have also discussed in class. Therefore, we must be careful when implementing these system calls to avoid deadlocks. It is likely that you will have to lock and unlock each process multiple times during the implementation of these system calls. This does not apply to the current process in the context of a system call (given by `myproc()`), as it is already "protected" by its `RUNNING` status. Locking the current process will almost surely result in a deadlock and a kernel panic or a frozen system.

As a final note, observe that most of this functionality can be implemented using the existing `fork` and `wait` system calls: forking four child processes and waiting for them to finish can be done by calling `fork` four times and `wait` four times. However, there is one thing that can't be done easily with the existing system calls. What is it?

Task 4 — Distributing Work by Forking

1. Implement the `forkn` and `waitall` system calls.
2. Write the userspace program `bigarray.c` that initializes an array of integers and calls the new system calls.
3. Distribute the work among four child processes, where each pro-

cess will work on a quarter of the array. Print the PIDs of the children.

4. Each child process will calculate the sum of the elements in its quarter of the array and print it.
5. The parent process will print the sum of the sums once all child processes have finished.
6. Test your implementation with more and fewer child processes.

For submission: `bigarray.c` and any modified OS files.

Questions — not for submission

1. How does `fork` work? How does it create a new process?
2. How does `wait` work? How does it wait for a child process to finish?
3. How does `wait` decide which child process to wait for? How does it decide whether to sleep or return immediately?
4. What is the purpose of `wait_lock`? Why does it appear in `exit` and `wait`?
5. What is the difference between our new system calls and the existing `fork` and `wait` system calls?
6. What is the main advantage of using `forkn` and `waitall` over using `fork` and `wait` multiple times?
7. How come the child processes have access to the same memory as the parent process? What happens if a child process modifies the memory? Can a real-world operating system optimize this?
8. Why do the child processes need to return their exit status to the parent process?
9. Why does the parent process need to wait for the child processes to finish?
10. What is the overall lifecycle of a process in xv6? How is it expressed in the code? How is it reflected in functions such as `fork`, `exit`, `kill`, `wait` and `sleep`?
11. What do the internal kernel functions `sleep` and `wakeup` do? How are they used in the kernel? How are they related to the process lifecycle?
12. What makes the sum operation suitable for breaking up and distributing among multiple processes? What are the costs to be paid for this?
13. Experiment with different array sizes and number of child processes. What happens if the array size is too small or too large? What happens if the number of child processes is too small or too large? Since we are running inside an emulator on different machines, your results may vary. You might not observe any differences in performance at all.

A Work Environment Setup

To effectively complete assignments in this course, you'll need a a Linux environment. We'll utilize Docker with the Windows Subsystem for Linux (WSL) backend to compile and run code within a development container, facilitated by the Dev Containers extension for Visual Studio Code (VS Code).

While this guide outlines a specific recommended setup, this class does not require you to use any specific tools as long as your submissions follow the submission instructions.

This guide is comprised of two main parts, one of which is a one-time setup and the other is a per-assignment setup.

1. Installing the required tools and extensions:
    - Install WSL 2
    - Install Docker Desktop
    - Install VS Code
    - Install the Dev Containers extension for VS Code
2. Configuring the development environment for each new assignment.

A.1 Installation Steps

Before you begin:

- **Administrative Privileges:** Ensure you have the necessary permissions to install and run Docker, especially if you're using a workplace-provided computer.
- **Enable Virtualization:** (Windows and Linux users only) Verify that virtualization is enabled in your system's BIOS settings. For guidance, refer to Microsoft's instructions on enabling virtualization on Windows.

Windows Subsystem for Linux (WSL) - Windows Users Only

WSL, also known as the Windows Subsystem for Linux, allows you to run a Linux distribution alongside your Windows installation, providing a native Linux experience. We'll use WSL as the backend for Docker to run Linux containers.

1. Open PowerShell or Command Prompt as Administrator:

   Right-click on the Start button and select "Windows PowerShell (Admin)" or "Command Prompt (Admin)".

2. Install WSL by executing the following command:

   ```
   wsl --install
   ```

   This command installs WSL and the default Linux distribution (usually Ubuntu).

3. Restart Your Computer:

   After installation, restart your computer to apply the changes.

Docker Desktop

Docker is an application for building, sharing, and running containerized applications and microservices. In a sense, containers virtualize runtimes and libraries needed for an application.

Installation Steps:

- Windows users:

  Download and install Docker Desktop for Windows.

- Linux users:
  - Ubuntu-based Distributions:
    Follow the installation instructions for Docker Desktop on Ubuntu.
  - Fedora:
    Refer to the Fedora-specific Docker Desktop installation guide.

- macOS users:

  Download and install Docker Desktop for macOS.

  An alternative to Docker Desktop on macOS is colima, a lightweight alternative that sometimes works better. Try it if you encounter issues with Docker Desktop.

Visual Studio Code and Dev Containers

VS Code, developed by Microsoft, is a versatile code editor that supports various programming languages. It has an extension called Dev Containers, allowing a container to be a full-featured development environment. Using the configuration files we provide as part of the assignments, you can easily create new environments for each assignment in this course. Once you have installed everything successfully, you can connect to the development environment by pressing the "Reopen in Container" button in VS Code.

Let's break it down:

1. Install VS Code:

   Download and install Visual Studio Code for your operating system.

2. Install the Dev Containers Extension:
   - Click on the Extensions icon in the sidebar or press ctrl+shift+X (or cmd+shift+X on macOS).
   - In the search bar, type "Dev Containers".
   - Locate the "Dev Containers" extension by Microsoft and click "Install".

3. After installation, you should see an indicator in the bottom-left corner of VS Code, confirming the Dev Containers extension is active.

A.2 Configuring the Development Environment

Set Up the .devcontainer Directory

We have included a `.devcontainer` directory in xv6's root directory that contains the necessary configuration files for the dev container.

To set up a new assignment in this class:

1. Clone or copy our xv6 repository to your local machine, creating a new directory for each assignment.
2. Open the project in VS Code: select your project folder that contains the '.devcontainer' directory.
3. Upon opening your project in VS Code, you should see a prompt to "Reopen in Container" in the bottom-right corner.
4. Click the "Reopen in Container" button.
5. VS Code will reload and begin setting up the development container.
   This process may take a few minutes during the initial setup.
6. If not prompted, press ctrl+shift+P (or cmd+shift+P on macOS) to open the Command Palette. Type "Dev Containers: Reopen in Container" and select it from the list.
7. When finished, VS Code will show a message that it is working on a container.
8. You can click on "Show Log" to view detailed progress and any troubleshoot any potential issues.
9. Once setup is complete, you should see an indication in the status bar confirming that you're connected to the development container.
10. Additionally, opening a new terminal in VS Code should display a prompt indicating that you're operating within the container environment.