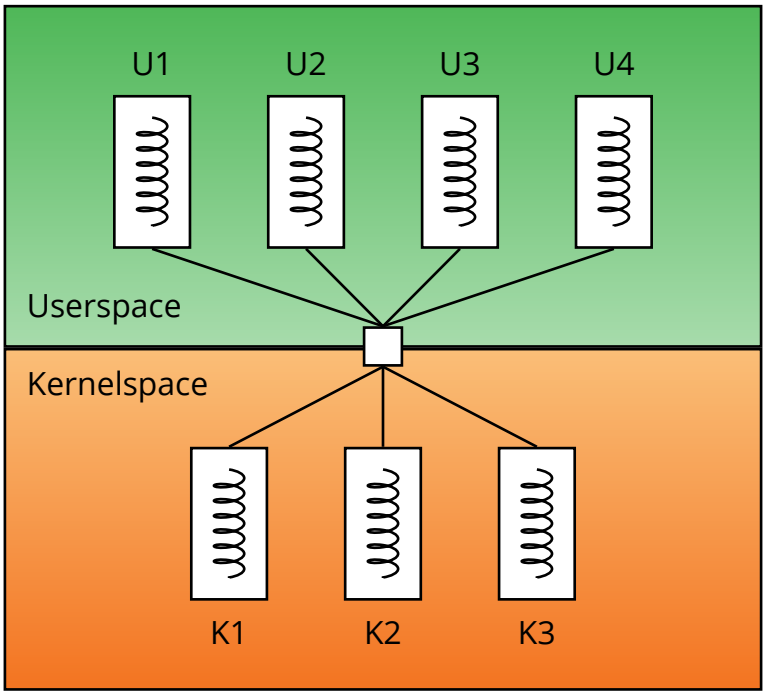
Operating Systems
202.1.3031

Spring 2024 Assignment 2

Synchronization and Interprocess Communication

Responsible Teaching Assistants:
Pan Eyal and Alex Lazarovich



Ben-Gurion University
of the Negev

Contents

1 Introduction

Welcome back to another exciting chapter in the world of operating systems! We hope that by now you are more familiar with xv6 and ready to delve deeper into the OS. In this assignment, you are going to implement a cooperative multi-process coroutine system in the xv6 kernel. By developing this new system, you will learn about process synchronization as well as the sleep and wakeup mechanisms provided by the xv6 kernel.

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 solution as a **single copy of xv6 with all tasks included and ready to compile with `make qemu`**.
- Before submitting, run the following command in your xv6 directory:

  ```
  $ make clean
  ```

  This will remove all compiled files as well as the `obj` directory.
- Submit as a single `.tar.gz` or `.zip` file. `.rar` files will not be accepted.
- 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.
- Help with the assignment will be given during office hours. Please email us in advance.

3 Task 1: Kernel Channels

In this task, you will implement a new synchronization system in xv6 called a *channel*. This is not to be confused with the unfortunately named `chan` field in the `proc` struct, which is a pointer to the object that the process is currently waiting on. Our new channels will be used to send information between processes. Processes will sleep whenever no data is available to be retrieved, or conversely, when no space is available to store new data. Similar mechanisms exist in many programming environments, such as Go's channels or Java's `BlockingQueue`. Think of a channel as a queue coupled with a synchronization mechanism. For simplicity, in this assignment we will only implement channels with space for one data item, i.e., queues of length one `int`.

How do we implement such a mechanism? Of course, since processes can send data to one another using the channel, this cannot be a userspace-only system, as processes are isolated from each other. Therefore, the kernel must be aware of, and facilitate, the interprocess communication.

First, let's describe the API for the channel mechanism. We will need several system calls — to create a channel, to put data in and get data from the channel, and to delete a channel we no longer need:

1. `int channel_create(void);`

   Creates a new channel and returns a new *channel descriptor* number, which is an index into the global channel table in the kernel, much like a file descriptor. In case the channel cannot be created, returns -1.

2. `int channel_put(int cd, int data);`

   Takes a channel descriptor and attempts to put `data` in the channel, if possible. Otherwise sleeps until the data can be inserted. In either case, when (and if) the function returns, it is guaranteed that `data` has been inserted into the channel and no uncollected data item has been overwritten by this call. Returns 0 on success and -1 on error (e.g., the channel descriptor is invalid).

3. `int channel_take(int cd, int* data);`

   Takes a channel descriptor and attempts to take a data item from the channel, if possible. Otherwise sleeps until data becomes available. In

either case, must place the data in the provided pointer and guarantee that no other process has received the same data. Returns 0 on success and -1 on error (e.g., the channel descriptor or data pointer is invalid).

Note: The `data` pointer resides in the user's address space, which cannot be accessed directly by the kernel. Use the `copyout()` function to copy data from the kernel to the user space. We will discuss this later on in the course. The corresponding call to copy data from the user space to the kernel is `copyin()`, but it does not need to be used in this assignment.

4. `int channel_destroy(int cd);`

   Deletes the channel with the given descriptor regardless of whether it is empty or not. Once this function returns, the channel descriptor is invalid and cannot be used. Returns 0 on success and -1 on error (e.g., the channel descriptor is invalid). All calls to `channel_put()` and `channel_take()` on this channel descriptor should return -1 after this call completes, **even if the calls have been initiated before the call to `channel_destroy()`**. Processes that are sleeping on the channel should be woken up with a -1 return value.

Consider the following example usage of the channel system calls:

```
int cd = channel_create();
if (cd < 0) {
    printf("Failed to create channel\n");
    exit(1);
}
if (fork() == 0) {
    if (channel_put(cd, 42) < 0) {
        printf("Failed to put data in channel\n");
        exit(1);
    }
    channel_put(cd, 43); // Sleeps until cleared
    // Handle error
    channel_destroy(cd);
    // Handle error
} else {
    int data;
    if (channel_take(cd, &data) < 0) { // 42
        printf("Failed to take data from channel\n");
```

```
        exit(1);
    }

    data = channel_take(cd, &data); // 43
    // Handle error

    data = channel_take(cd, &data); // Sleep until child destroys channel
    // Handle error
}
```

The data structure for the channel on the kernel side needs to do a few things. First, it must have a way to avoid concurrent access by multiple CPUs, which might lead to data races. Use a spinlock to protect the data structure, much like the lock in `struct proc`. Design a data structure that can store a data item and keep track of whether a data item is available in the channel or not. Finally, we need a way to keep track of which process created the channel so it can be deleted when the process exits or is killed. A more sophisticated system would probably want to do something more robust, such as keeping track of the references to the channel to avoid deleting it when it is still in use by other processes, but we will not be doing that. Design the channel struct to meet these requirements (hint: take inspiration from the `proc` struct).

Next, create a global array of channels, decide on how many to support (for this assignment, a few will be enough), and write code to initialize the array when the kernel starts. See the initialization of the `proc` array for an example.

Finally, implement the four system calls while maintaining the guarantees required by the API. Use the `sleep()` and `wakeup()` functions provided by the kernel to do this. Note, however, that we can't use the `sleep()` *system call* for this. Also, delete each channel when the process that created it exits or is killed.

Tips

- This task hinges on your understanding of the sleep and wakeup mechanisms in xv6. Make sure you understand how they work in the usual case where `sleep()` is called by a process. Be sure to understand what `chan` is, how it is used in the kernel, what happens to the lock passed to `sleep()` and why. Why is a lock even needed in a call to `sleep()`?
- In the implementation of `wakeup()`, the kernel wakes up *all* processes sleeping on the provided object. Make sure this doesn't pose a problem when multiple processes wait on the channel to read or write, or be ready to explain why this isn't a problem if you think it isn't.

Task 1 — implement kernel channels

1. Add the channel struct to the xv6 kernel, create a global array of channels and initialize it.
2. Implement `channel_create()`, `channel_put()`, `channel_take()` and `channel_destroy()`.
3. Delete each channel when the process that created it exits or is killed (see the `exit()` and `kill()` functions).
4. Test your implementation with the example code provided above.

For submission: Changes to the xv6 kernel to implement the channel mechanism.
Bonus points (hard): Implement reference tracking for channels (+1 point to **final grade**).

Questions — not for submission

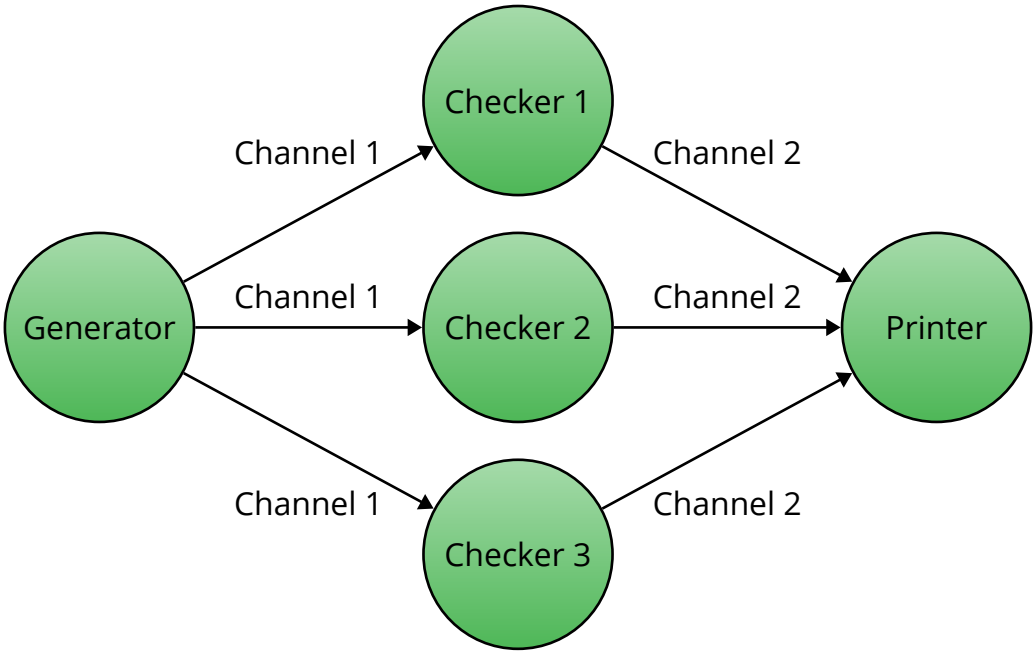
1. Why do we need a lock in the channel struct?
2. Why do we need to keep track of the state of the channel? Can't we just sleep if the lock is held?
3. Is the fact that all processes sleeping on a channel are woken up when data is available, a problem? Will our system still work if only one process is woken up?
4. How do we make sure that the data in the channel is not overwritten by another process?
5. How do we make sure that the data in the channel is not read by multiple processes?
6. Why does a call to `sleep()` require a lock to be passed to it?
7. Why does `sleep()` release the lock passed to it? Why does it reacquire the lock when it wakes up?
8. How is the system call `sleep()` different from the function `sleep()`? How does the kernel know when the specified time has passed to wake up the process?
9. How do we make sure that calls on a channel that has been destroyed return -1 even if they were initiated before the call to `channel_destroy()` completes?

4 Task 2: Prime Numbers

Using the channel mechanism you implemented in the previous task, write a program that finds and prints prime numbers. The program will consist of several processes: one to generate numbers, a few to check which numbers are prime, and one to print them. You will need two channels: one for the generator to send numbers to the checkers, and one for the checkers to send results to the printer. Given that the only way to coordinate which channels to use is by forking all the processes from the same parent, create the channels in the parent process and fork multiple times to create the other processes.

The generator will generate numbers starting from 2 and send them to the checkers via the first channel. Checkers will receive numbers from the generator, check if they are prime, and send the prime numbers to the printer via the second channel. The printer will receive results from the checkers and print them with an appropriate message. Use the simplest algorithm to check for primality. Accept the number of checkers (i.e., the degree of parallelism in the system) as a command-line argument, with a default value of 3. If your system works, only prime numbers should be printed.

The system is shown below for an instance with three checkers:



Finally, shut down the system gracefully: when the printer detects that 100 prime numbers have been found, it should signal the program to terminate by deleting the second channel and exiting. The checker processes should detect the deletion of the channel and exit as well, destroy the first channel, and exit. Then, the generator process should detect the deletion of the first channel and wait for all child processes (checkers and printer) to exit. The generator will then prompt the user to start the system again, or exit. If the user chooses to start the system again, the generator will create new channels and fork the processes again. Every step of the shutdown process should print the PID, role of the process (e.g. "Printer", "Checker 1") and an appropriate message to show that the shutdown is properly sequenced.

Task 2 — implement the prime number finder

1. Implement the prime number finder in a user program, `primes.c`.
2. Accept the number of checkers as a command-line argument.
3. Create the channels in the parent process and fork the generator, checkers, and printer.
4. Implement the generation, checking, and printing logic.
5. Gracefully shut down the system when 100 prime numbers have been found.

For submission: `primes.c` with the appropriate changes to `Makefile` to compile it.

Questions — not for submission

1. Why do we need two channels? Isn't one enough?
2. What happens if one of the processes is slower than the others? For example, checking large numbers for primality takes longer than generating or printing them.
3. Try to insert a delay in the checker or printer process to simulate a slow process. You can also insert print statements to see the order of execution. Does the system still work as expected? Does the generator process block when the checker is slow?
4. This is a common problem in systems with multiple processes what run at different speeds. How can we improve the system so that it is harder to stall?
5. What guarantees that the printer process will not print the same prime number multiple times?
6. What guarantees that prime numbers will be printed in order? Do we have this guarantee at all?
7. The concept of halting production or consumption of data when the other side is not ready is sometimes called *flow control* or *backpressure*. How does it come into play in this system?
8. How come the printer decides when to shut down the system? Why not the generator or the checkers?
9. What happens if the printer process is killed before the system is shut down? How can we handle this situation?
10. What if a checker or generator process is killed before the system is shut down? How can we handle this situation?
11. One tradeoff in building a multi-process system is the complexity of managing the processes and fault tolerance. What are the advantages and disadvantages of using multiple processes instead of threads?