INF-2201: Computer architecture and organization

Exam 2 / Assignment 4

Tarek Lein

UiT id: Tle044@uit.no

GitHub users: TrakeLean

April 2, 2023

1 Introduction

In this exam we were tasked with implementing functions for message passing, we use this to allow processes to talk to each other. An example of this would be every time you press a key on the keyboard it goes through the message passing functions and end up on the screen. We also had to implement process management, this part was a little tricky which I'll gladly discus later in the report. We have to implement the process management part to dynamically start up and/or end processes while the operating system is running.

2 Technical Background

2.1 Mailbox

The mailbox concept was a huge part of this project, the mailbox concept is that when you want to send data from/to different processes they're going to need a way of communicating, one process puts mail into the mailbox and another process fetches and reads the mail from that mailbox, hence the name "mailbox" the mailbox consists of a buffer with a finite size, in our case it was set to 1024bytes, the sending and retrieving of mails are first in first out (FIFO) and we're allowed to send varying sizes of mails. This type of mailbox buffer implementation would be called "Bounded buffer" where we're also allowing multiple producers and consumers access the mailbox if they have the key.

2.2 Keyboard

For the keyboard we're simply interrupting the operating system, handling the interrupt and sending scancode of the key pressed to a function that takes into consideration if we held shift, alt or any of those types of keys. Our job was to take the character sent to us after all that and put it in a mailbox.

2.3 Process management

The file system for processes management told us that everything was stored in a flat address space, where if we knew the size of bootblock, kernel, process directory we would be able to traverse the data and read/write what ever we wanted from/to exactly where we wanted it to be.

3 Design

There was a few design choices that had to be made when creating the mailbox, the first thing to take into consideration was how to handle the buffer, how do we move the head and tail. We chose to move the head when writing data into the buffer, so if we write four bytes we would have to move the head four bytes too, this is because we need to know where we can start writing data the next time we decide the write. This logic is also used when reading data but for the tail, we have to move the tail up till the next data segment. This means that we write where the head is and read from where the tail is, next we have to know how much to write and read.

For writing it's simple, we have the data contents and know the size, but how will we know the size when we're reading? This is where the second design choice of the mailbox was made. We decided that the four first bytes of the data is reserved for the size of size of the "message" we want to read. This decision was based off of packets and elf files, because when packets are sent around the internet it has a "header" that tells the receiver what to read, the same goes for elf files, they have headers that tells the receiver information about the data beyond. so if we send in "mail" to the mailbox, we would put "8" into the first four bytes. The read would then get the first four bytes and know that the data is eight bytes and it has already read 4 of those so the data it wants is the next four bytes.

But what happens if lets say the head reaches 15 and the tail is 13, that would mean that we have 2 bytes in the buffer and 13 free bytes for data to be written to, but we cant write about 15 since that would exit our buffer and charter into "unknown" memory space. To fix this we just check if there actually is space for the data we're about to write and if there is space we just take the head's value modulo the max buffer size, this would wrap the head back around to the start of the buffer. The same logic goes for the tail's value. By doing this we end up with a circular buffer that never hold garbage values and always has control how what is within it. you could picture is as a cat chasing a mouse, where the cat is the tail and the mouse is the head.

We also used condition variables in our mailbox to let them sort of communicate with each other. The producer would lock it self if the buffer was full, which meant that the consumer would have to consume and make more room in the buffer, when this was done it signaled to the producer and said "Okey I've consumed, check if you have enough space".

The consumer would lock it self if there wasn't anything to consume in the buffer, and the producer would then put something in the mailbox and signal the consumer that there was something to consume.

Overall, our implementation of the mailbox using a circular buffer and a size header for data provides an efficient and reliable solution to the producer-consumer problem.

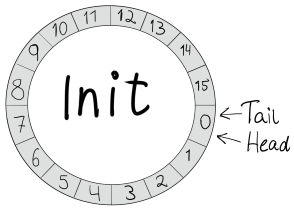


Figure 1: Initialize mailbox

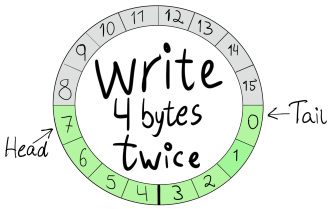


Figure 2: Write to mailbox

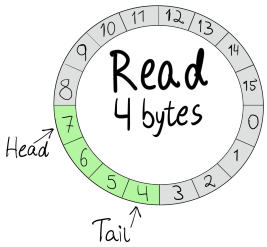
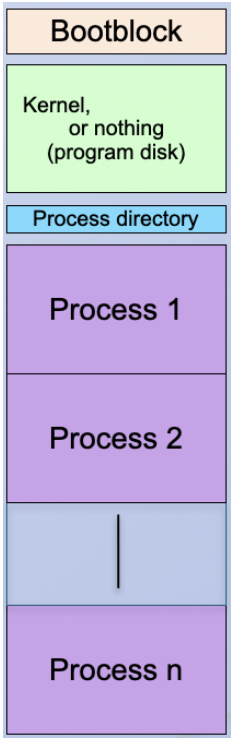


Figure 3: Read from mailbox

The keyboard implementation was simple, we receive a character and build up a message where the size is one and the body is the character, then it's sent into the mailbox for our consumer to fetch it, read the body and store it into the given pointer. There is one thing to keep in mind though which is when the consumer is interrupted while receiving a message, this can happen when we spam the keyboard which interrupts the receiving and we're never able to get back. This means that we have to treat the receive as a critical section, so we enter critical before and leave after. This fixes the interrupt bug and will not let us interrupt while receiving.



Implementing a way for reading and loading processes from memory was not a easy feat. We had to navigate through memory and find the processes we were looking for. The reading part of this was done by finding out what the operating systems size was, that size lies within the bootblock. After finding the size we were able to navigate our way into the process directory, which is a simple files system given to us. Inside the process directory we can find the location and sizes of the processes.

That helps us when we want to load up a process, because we can just go to the location and read the size of the process. But before that we have to allocate some memory for all the data to be stored, then we're able to load and start the chosen process.

4 Implementation

This is where you go into detail about your specific implementation. Questions you should answer here are things such as "How does your implementation match your design?" and "Are there any bugs, and do you have any ideas about what may be causing them?". What sort of difficulties did you experience when working, and how did you overcome them? If you found a clever solution to the problem, this is also the place to write about that.

We were given a simple file system that laid all the processes into a flat memory space. These are the steps that was taken for reaching that file system, first was to find out the size of our operating system, we do this by reading the bootblock, now we have the whole bootblock, and within this we're able to find the operating system size. To find the size we have to go four bits in and read the next two bits we find there, this is because in our bootblock.s file, there's a variable called "os_size" that gets stored in these bits which is originally set to 0, but we've told our createimage to write the size of the operating system there after it being calculated. Here is the first 32 bits when we hex dump our image, where the bootblock is written first.

"00000000: eb04 9600 0000 b800 708e d0bc feff b8c0 ....p......."

so we skip four bits "eb04" and read the next two bits "96", 0x96 in decimal is 150 which means the operating system size is 150 sectors, and by looking at the picture above, we know that the process directory is after the kernel image. So we then read sector 151 and what we end up with is the simple files system, were it tells us the location and size of each process.

When we want to load a process we have to first give it space in memory, this space is calculated by multiplying the sector size with how many sectors there are, and then on top of that we have to add the stack size. This gives us enough memory space for the process data and stack. Then we allocate this space to a pointer and read the process into that. We're able to do this because we now know the location of where we want to get the process data from and its size. Lastly we simply use a given function that sort of builds up the process within a pcb struct and makes it ready to be ran.

4.1 Extra Credits

For the kill command we send in the PID given my the user, and iterate through the running queue till we hit the PID that the user wants stopped, if the code finds out that i has reached the PID it started on, it exits and tells us that the PID does not exist. But if the PID is found, we just set its status to "EXITED" and the scheduler will do the rest. For kill all we just iterate through all PIDS over 5.

The logic for PS was to find out when we have the PID that's before the first process running. We start by searching for that starting PID, when that is found we store the values for all the PIDS above 5. There was a problem where the PIDS wouldn't show in descending order because of how the queue system works, we fixed this by having a loop within a while statement that start by printing the first active PID and then incremented the active PID by one, until everything was printed in order.
Another problem was that the shell sometimes threw it self into the queue. This was a problem because if we wanted to read PIDS 7 - 8 - 9 - 10 and it got in between 8 and 9, we would exit the while loop because the while checks if the current PID is above 5, so it would exit and not store 9 and 10.

Suspend, resume and kill are basically the same. That's why we use the same logic as kill, but instead of setting the status to "EXITED" we set it to "SUSPENDED" for suspend and "RUNNING" for resume.

5 Discussion

At first we made the mailbox not care about sending the message size, which worked. The way we did it was by kind of null terminating every message sent, we added a "\0" behind every message. Then when the consumer wanted to read the message, it just read until it hit the null termination.

Even though this worked we got worried about what could happen if the message being produced contained a null termination, would it then prematurely stop reading and then the next consumer would read the rest and what it was supposed to read. Another problem was that we could never know if the buffer had room for the message, since we didn't know its size.

It also kind of ruined the mailbox concept where we send the message with a header the proper way and read the exact amount, we wouldn't have learned half of what we know now about this concept if we didn't do it the right way.
We got stuck for a long time at a part in the "mbox_recv" function where we were unable to read the correct message body. We started by printing out the size in send and recv and noticed that they for some reason wasn't the same, it was always one more in receive. After a lot of testing we finally found out that the size had to be cast to unsigned char and then into an int, this ensured us the right size value.

6 Conclusion

All in all we're satisfied with our code, there are no known bugs since we managed to debug them all. The code is understandable and everything is commented, and we had fun while developing the functions and solutions that was required for this exam.

7 Sources

References

[1] Lectures

[2] Precept PowerPoint

[3] Pre-code read-me

[4] https://www.asciitable.com/