INF-2201-1 23V: Operating system fundamentals

Assignment 4

Amund Strøm

Email: amund.strom@gmail.com

GitHub user: ast283

March 29, 2023

1 Introduction

This report describes the design and implementation of a message passing system, more specifically a mailbox mechanism, and the implementation of process management. The message passing system will be used to communicate between processes and give support for keyboard interrupts.

2 Technical Details

These are some of the concepts and problems that is discussed in this report, and it is important to have an common understanding of these.

- Spinlocks is a tool used for minimize disabling interrupts in the code. This is important because by disabling interrupts the operating system may miss out on important hardware interrupts such as the timer or keyboard interrupts. A spinlock is used whenever a process or thread wants to do an atomic operation such as utilizing the synchronization primitives lock and condition variables. A spinlock is utilized with two operations, acquire and release which works atomically and self-explanatory.
- Mailbox is a mechanism that allows processes to communicate with each other by sending (produce) and receiving (consume) messages in a shared memory space. A mailbox can communicate with two or more processes, but it is the callee's responsible to send and receive messages in the correct order, since a mailbox simply operates in the First In First Out (FIFO) method. A message can either be a fixed size (always the same size) or a variable size (changes depending on the size of the content).
- Bounded buffer problem or also know as the producer-consumer problem occurs when multiple consumers and producers share a memory space. Producers will write data to the memory space and consumers will read data from the memory space. This problem is solved in the implementation section.
- Process management allows the operating system to load processes dynamically on demand, rather than statically allocating every process at the startup. To achieve this a file system must be established. In the case of this project the file system is of limited size and flat. It is of limited size, because the root directory will only occupy a single sector, therefore only a limited number of processes can be stored. It is flat due to the absence of a directory structure, meaning all files resides in the same root directory.
- Root directory contains the length and offset of programs that exists on the disk. It works like an header for every program and is used for loading in the requested program into the operating system.

3 Design

This project required four concepts to be implemented, that is synchronization primitives with the use of spinlock, a mailbox mechanism, support for keyboard, and process management.

3.1 Synchronization primitives

The synchronization primitives used in this project was locks and condition variables, which works the same as previous projects. But these primitives must not disable interrupts (described in the technical background). Therefore whenever one of these operations is about to be executed, the operation must first acquire a spinlock, and release it when the operation is complete.

3.2 Mailbox mechanism

The mailbox in this project has a fixed size meaning that once it is allocated in the memory, it can no longer change the size of the buffer which contains the messages. It also supports messages with variable lengths, meaning a message can be any sizes less than the size of the actual buffer in the mailbox. The buffer is circular meaning the next index after the last, is at the beginning.

This is an illustration of how the mailbox buffer looks before and after a message is written into the buffer. The size of the message is 5 bytes, while in the buffer it is 6 bytes.

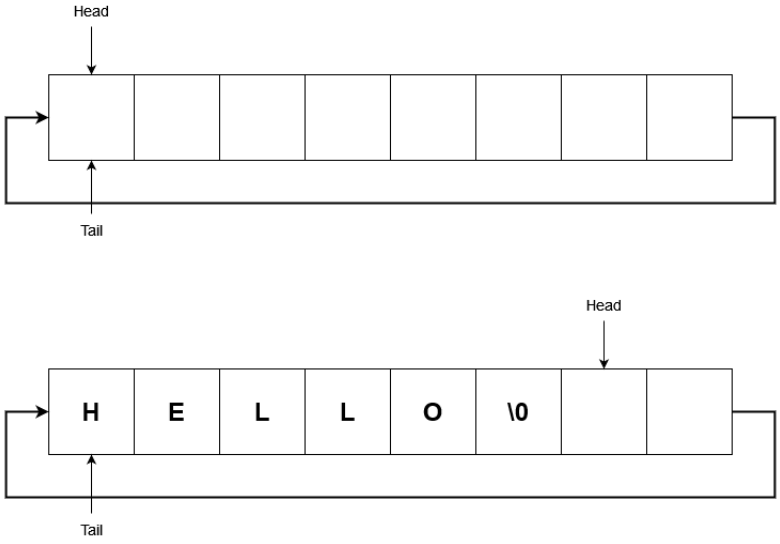

Figure 1: Buffer in mailbox

This mailbox supports 6 operations:

- mbox_init initializes the mailbox. This must be done for multiple mailboxes, since there are multiple mailboxes for different proposes.
- mbox_open uses a key to know which specific mailbox has been opened, this same key must be used to access the same mailbox
- mbox_close uses a key to know which specific mailbox has been closed.
- mbox_stat uses a key to know the number of messages and available space in bytes for that specific mailbox. This operation must be done atomically, therefore it must acquire and release locks accordingly.
- mbox_send uses a key to write a message to that specific mailbox. This is accomplished by iterating through each byte in the message and copying it into a circular buffer in the mailbox. After the message is copied into the buffer, a null-terminator is inserted at the end of the message in the buffer. The reasoning for this is to know the length of the message. A head "pointer" must also be update to point

to the next available space to write into the mailbox (the index right after the null-terminator). mbox_send operation must also be done atomically and must acquire and release locks accordingly. To solve the bounded buffer problem this operation must check if the mailbox buffer is full, if it is, it must wait until there is enough available space for the message that it is writing into the buffer. In addition it must signal the consumer (receiver) that there is a message in the buffer, if it is waiting.

- mbox_recv uses a key to read a message at that specific mailbox. This is accomplished by iterating through each byte in the circular buffer in the mailbox and copying it into the message, until a null-terminator is encountered. Afterwards update the size of the message and the tail "pointer". The size of the message does not include the null-terminator. The tail is updated so that it points to the start of the next message (the index right after the null-terminator). mbox_recv operation must also be done atomically and must acquire and release locks accordingly. To solve the bounded buffer problem this operation must check if the mailbox is empty, if it is, it must wait until there is a message in the buffer it is reading from. in addition it must signal the producer (sender) that there is new available space in the buffer, if it is waiting.

3.3 Keyboard support

To give support for keyboard inputs an interrupt handler must exist that is able to handle multiple types of interrupts (timer and keyboard). When handling keyboard interrupts it simply reads the key pressed, checking if special treatment must take place (shift, control, or key release), and places it in the appropriate mailbox. This is done in the pro-code.

The remaining work was to implement 3 operations:

- keyboard_init opens the mailbox used for passing keyboard inputs.
- putchar writes a keyboard input into the mailbox. Before writing it into the mailbox, it must check if there is enough available space. If it is not enough space, simply discard the input since putting this operation on wait will cause errors because then there is no one to handle the keyboard inputs. But most importantly, it is not possible to put this operation on wait because it is not a process/thread.
- getchar reads a message from the mailbox, and updates a pointer to the value it read from the mailbox.

3.4 Process management

As mentioned in the technical background, process management requires a file system where a root directory exists that contains the lengths and offset of every process. Therefore, whenever a process is requested to be loaded into the operating system, the first task is to read the process directory at the location that the requested processes resides. This results in knowing the size (length) and offset of the process. Thereafter allocated memory for the process and write it into the allocated memory.

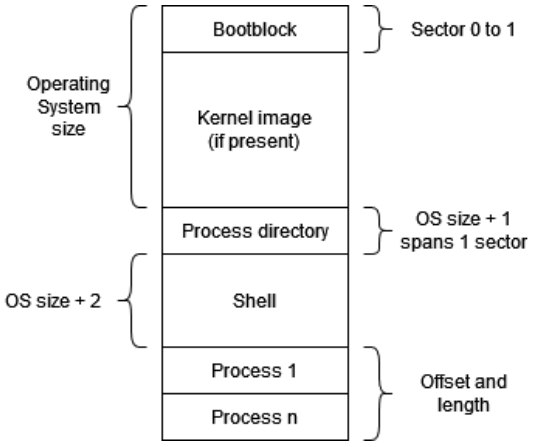

Figure 2: The established file system. Not to scale. A block and sector spans same amount of bytes.

loading a process is mainly achieved with 2 operations:

- readdir reads the process directory, and places it into a user provided buffer, this operation is divided into 4 steps. First acquire the bootblock which resides on block 0 to 1 on the disk. Secondly acquire the size of the operating system, which occupy byte 2 to 4 in the bootblock. Thirdly using the knowledge of the established file system, read a single block at sector "size of OS" + 1 sector, this is where the process directory resides. Finally copy the read data into the user provided buffer.
- loadproc loads the requested process into the operating system. To accomplish this, first calculate the amount of memory this process will use, then allocate the memory. Afterwards read the offset and size of the process on the disk, and copy it into the allocated memory. Finally create the process by initializing values and a program control block (PCB).

3.5 Extra credits

Implementing an "ps" command required creating a new system call. This command shows the process ID and statues (represented in integers) of every running process. The system call iterates through the ready queue while ignoring threads, and adding information about the processes to a user provided buffer.

4 Discussion

There is not much to discuss about results since the only result in this project is either if things work or not.

Extensive testing on the mailbox have been done to make sure that the circular buffer utilizes every index. This includes not skip the last index when circling around the buffer, and when inserting the null-terminator that it does not overwrite the last byte of the message or skip an index. Another solution to this would be to insert the size of the message at the start of the message in the buffer. But an advantage the former solution has is that the null-terminator will only occupy 1 byte, compared to using "x" amount of bytes to deduce how large the message is. For example lets say a message is of size 300 bytes, the number 300 must be represented with the use of 2 bytes. A disadvantage would be that if the message contains a null-terminator, things would break, but that is a highly unlikely scenario.

The established file system in this project is still very primitive, therefore the implementation to load in processes would most likely not work on other systems. This does not mean that the work done in this project is redundant, since you would still use the same techniques to accomplish the same goal in a more complex system. A more complex system would simply contain more information.

5 Conclusion

This report described a high-level view of how synchronization primitives with the use of spinlock, an mailbox mechanism, support for keyboard, and process management have been implemented. Some advantages and disadvantages with the implemented mailbox mechanism and process management have also been discussed.

6 Sources

References

[1] Tanenbaum, Andrew S. Modern Operating Systems, 4th edition. Pearson, 2015. Chapters 2.3.8, 3.3-3.4.

[2] Neso Academy. Message Passing Systems (Part 2). Youtube Sep 21, 2018, from `https://www.youtube.com/watch?v=S3mS8MR7bUY`