INF-2201: OP fundamentals

Project 4 Inter-Process Communication and Process Management

Lise Ekhorn Johansen
UiT id: ljo177@uit.no
Github username: Lise-johansen
GitHub classroom: project-4-Lise-johansen

03.04.2023

1. Introduction

This report describes the implementation and design of a message-passing mechanism and process management for a simple preemptive scheduler OS. Additionally, the implementation of mutexes and condition variables are discussed.

2. Technical background

Use **spinlock** to achieve mutual exclusion without disabling interrupts. The thread that does not hold the lock spins waits in a loop, and checks if a lock is available. When a lock is available the thread stops spinning and takes the lock. [1]

A circular buffer is a data structure at a fixed size connected end-to-end. [2]

Massage passing is for process communication mechanism and synchronization of their action. [3] A typical example of these mechanics is **mailboxes**, where one or more tasks use a shared memory location to transfer data to and from each other. [4]

Process management refers to the tasks performed by the OS to manage the execution of processes. One of these tasks is the allocation and management of memory. [5]

3. Design

3.1 Synchronization primitive mechanic

There are two synchronization mechanics, mutexes and condition variables. Each mechanism is designed using structures and is responsible for the synchronization of behavior and management of the shared resources.

3.1.1 Mutexs & Conditional variable

Only one thread holds the lock at a time, while other threads are blocked from accessing.

3.1.2 Conditional variable

A waiting queue is associated with the conditional variable and threads are blocked and unblocked in the queue based on the state of the condition.

3.2 Process Message

Process messaging uses a circular buffer expressed as an indirect mailbox working as a FIFO queue. It is used to enable communication between processes by facilitating data transfer in and out of the buffer between processes.

3.3 Keyboard

The keyboard is the producer, placing input characters into a mailbox buffer. The process serves as the consumer and reads the character from the mailbox. When the mailbox is full, the producer discards the character to avoid data corruption, and if it is empty, the consumer can not read data from the buffer.

3.4 Process management

Processes have their own address space and are locked from accessing kernel memory. The OS uses a simple file system where the bootblock resides in sector zero. And another sector holds the process directory, which contains one entry per process.

3. Implementation

3.1 Use of Lockspin with Synchronization Primitive Mechanic

Spinlocks ensure that operations are atomic. A spinlock is initiated together with the primitives' mechanism, and when releasing a spinlock be careful not to do it twice since the block function itself releases it.

3.1.1 Mutex

When a thread tries to acquire an unavailable lock, the thread is blocked and placed in the waiting queue. When the lock is available again, the first thread in the queue is unblocked and acquires the lock.

The lock is released when the thread is done, if there are threads in the queue, the first thread is unblocked and can acquire the lock. Otherwise, the lock status is set to unblocked.

3.1.2 Condition variables

A waiting queue is created for the threads that are blocked by the conduction. When a certain event or condition occurs, either all or just one (the first) thread is unblocked and can reacquire (failed the first time ergo waiting queue) the lock, enabling the thread to continue its execution.

3.2 Transfer of data

Data transfer via a temporary array with a static size. A pointer with data type as the source of the data is set to point at the first element of the temporary array. Transfer one byte/character, a pointer is used instead of an array. Then a loop is used to iterate through the data.

3.3 Message Passing

A mailbox is an array of mailbox structures, with a head and tail pointer. A mailbox has a lock and two condition variables one for space and data. It supports two operations: (1) Fetching messages from the buffer and storing them in a struct. Check if the buffer is empty and wait for more data. If not empty, iterates through the message and copies it from the buffer to the struck, updates the tail, count, and signals to the producer that there is more space for data.
(2) Inserting messages into a mailbox, first check for space, if not space waits for more. Iterate through the massage and copy it into the buffer. Update head, count, and signal to the consumer that there are data available.

3.4 Keyboard

Consumer reads ASCII characters from the keyboard queue via mailbox buffer. Fetch the message from the buffer, check if the result is valid (bigger than zero), and transfer the ASCII character into a variable that points to a place in memory. Producer, insert the ASCII character in the mailbox. Create a message with the size of one char and transfers the character (data) into the message body. If there is space for the message in the buffer, insert the message into the buffer. If not discard the message and return fail.

3.5 Process Management

Initialize a pointer to the buffer and an array (temp) with the size of one sector. Read the bootblock (sector 0) into the temp and find the offset to the process directory, and save it in a variable. This variable is used to jump to the start of the process directory. Can now read the process directory into the temp, and from there transfer the data in temp to the pointer pointing at the buffer.
To load the process from the disk, allocate enough memory for one process (base address). Read the data from a location on the disk the base address. Then create a process using the base address and size of one process plus stack size.

3.6 Extra credits: Kill Command

If the pid matches the one entered into the terminal, mark the process as killed and return success. Otherwise, check if the pid corresponds to an existing process pid in the buffer. If it does not match, return failure and exit the loop to move to the next job.

4. Experiments and Result

4.1 Test for the correctness of the OS

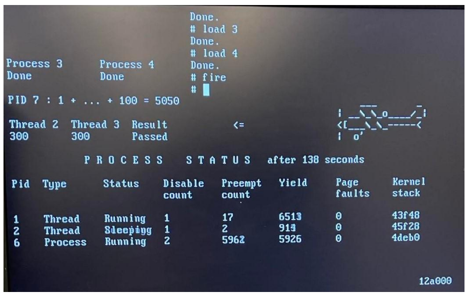
Correctness is evaluated by running the OS image on a PC without an existing OS. If the OS runs and all test is passed.

4.2 Extra credit

The extra credit is shown in picture 2. Can see that the kill command stops the killed process and removes it from the process status information board. More on this in the discussion.

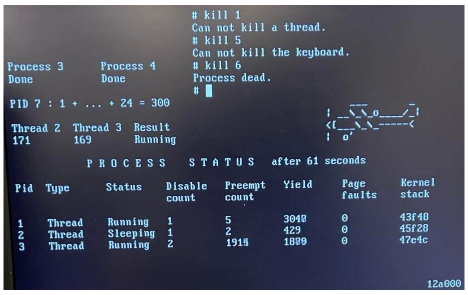
4.3 Result of OS

Picture 1 shows the result of the OS image, which mache the expected outcome. Picture 2 shows the kill command.



```
                                      Done.
                                      # load 3
                                      Done.
                                      # load 4
Process 3        Process 4            Done.
Done             Done                 # fire
                                      #
PID 7 : 1 + ... + 100 = 5050

Thread 2   Thread 3   Result              <=
300        300        Passed

            P R O C E S S    S T A T U S   after 138 seconds

Pid  Type      Status     Disable   Preempt   Yield    Page     Kernel
                          count     count              faults   stack

1    Thread    Running    1         17        6513     0        43f48
2    Thread    Sleeping   1         2         913      0        45f28
6    Process   Running    2         5962      5926     0        4deb0

                                                                12a000
```

Picture 1: Thread 1 and 2 get result passed. Processes 2, 3,4, and the plane was loaded from memory and is showed one screen successfully.



```
                                      # kill 1
                                      Can not kill a thread.
                                      # kill 5
                                      Can not kill the keyboard.
Process 3        Process 4            # kill 6
Done             Done                 Process dead.
                                      #
PID 7 : 1 + ... + 24 = 300

Thread 2   Thread 3   Result
171        169        Running

            P R O C E S S    S T A T U S   after 61 seconds

Pid  Type      Status     Disable   Preempt   Yield    Page     Kernel
                          count     count              faults   stack

1    Thread    Running    1         5         3048     0        43f48
2    Thread    Sleeping   1         2         429      0        45f28
3    Thread    Running    2         1915      1889     0        47e4c

                                                                12a000
```

Picture 2: Displays the restrictions: The message "Can not kill a thread" is displayed when attempting to kill the keyboard or a thread pid. And responses from the kill function when the kill command is used. "Pid is not real" is shown for non-existing pids or killed processes. And "Process dead" is output for successfully killed processes.

5. Discussion

5.1 Mailbox

The current implementation does not consider a scenario where a 4-byte message is inserted into the end of the buffer without enough space for the full message. If the implementation needs to consider it, simply add a check if there is enough space for 4 bytes to the existing solution.
At most, three bytes may not be used. Another solution is to split the message into 1-byte chunks and insert them, use a pointer that points at the start of the buffer, and then insert the rest of the message.

5.2 Extra credit: kill command

Picture 2 shows the kill command removing a process from the running queue. This indicates that it is killed. Kill commands implemented are delayed kill, and are done in 5 steps. (1) Make a new killed status. (2) Make a new case handler in the scheduler, it is her the process is removed from the currently running queue if the process is not holding locks/condition variables. (3) A variable is incremented and decremented depending on whether the process uses any lock/condition variable. (4) Function is for killing a process is made. (5) How the shell handles the kill command is dictated.
Improvement in implementation is to not hard code what an invalid pid is and restrict behavior (such as killing threads, shell, or keyboard) in the shell. And a virtual improvement is the clean screen of the texts of the killed process.

5.3 Completion of project

Based on the experiment's results, processes were successfully loaded from memory, and keyboard input was correctly inserted into and read from the mailbox buffer. A working OS that met all requirements was implemented.
Patch for processes 3 and 4 and a spinlock were provided on the 27. Mars. Deemed unnecessary since OS worked as expected (without the patch), and also since the patch came so late it has not been extensively tested and could therefore negatively impact the OS system. The decision was made to use the original pre-code for the exam.

6. Conclusion

An OS was implemented where a message-passing mechanism was used to enable an interactive input feature, and where all the processes no longer load at boot time, but instead are loaded when they are called at run time.

Bibliography

[1] (wiki.osdev.com), OSDev Wiki, "Spinlock", Last modified: 12.01.2023, Read: 15.03.2023.
URL: Spinlock - OSDev Wiki

[2] (wikipedia.com), Wikipedia, "Circular buffer", Las modified: 15.03.20232, Read: 17.03.2023
URL: Circular buffer - Wikipedia

[3] (INF-2201 Operating System Fundamentals), Phuong Ha, " Message Passing", Lecture: 16.03.2023, Read: 16.03.2023

[4] (uio.no), "RTOS: Mailbox", Read: 16.03.2023
URL: rtos_messages_mailbox.pdf (uio.no)

[5] (codestudio.com), Mishar. Sagar, "Process Management in Operating System(OS)", Last modified: 21.03.2023, Read: 19.03.2023
URL: Process Management in Operating System(OS) - Coding Ninjas

[6] Andrew S. Tanenbaum, Modern Operating Systems, 4th Ed., Pearson Educational Limited, 2015. ISBN 10: 1-292-06142-1. Chapters: 1-6, 8, 9, 12. [MOS]