INF-2201
Exam, project 4
April 2022
Inter-Process Communication and Process Management

Name: Magnus Dahl-Hansen
Github Username: MagnusDH
Email: mda105@uit.no

Introduction

This report describes the implementation and uses of: locks, spinlocks, condition variables and mailboxes which processes can use to communicate with each other. The report also describes how processes are loaded from a storage device so that a user can load and run processes on demand.

Technical background

Before reading this report the reader should be familiar with spinlocks, mailboxes and how programs are loaded and executed.

A spinlock is a synchronization mechanism that causes a thread/process to run in a loop if the current spinlock can not be acquired. The process/thread will run in a loop until the spinlock is released by the other thread/process using it. This mechanism is primarily used in operating system kernels because they are most efficient when used in short periods of time. While a thread/process is waiting for a spinlock it will waste computer cycles since it remains active, but not performing any useful tasks.

In computer science, a mailbox is a buffer where a certain number of messages are stored and allows processes to communicate with each other. A mailbox have two functions, namely to send and receive messages. If a process tries to send a message to a mailbox that is full, it is suspended until there is room for the message in the mailbox.

In this implementation of process management a program or process is loaded from a storage device via three steps. At first, enough space is allocated for the program along with a user stack for the program. Then the program is read from the provided storage/memory location into the allocated space and finally a process control block is created for the program/process and it is inserted into the running queue and executed.

Design&Implementation

For the processes and threads to run simultaneously the synchronization primitives locks and condition variables have been implemented. The locks in this implementation are equipped with a status variable that is initially set to “unlocked”, a waiting queue for blocked thread/processes and a spinlock which is also initially set to “unlocked”.

Before a thread/process can acquire a lock it must first acquire the spinlock associated with the lock. This is to ensure that no race conditions occurs. If a thread/process have successfully acquired the spinlock then it can proceed to either be blocked if the lock is “locked” or it can proceed to acquire the lock, set its status to “locked” and release the spinlock. When a process/thread wants to release a lock it must go through the same steps. It must first acquire the spinlock connected to the lock, then proceed to release all possibly blocked threads/processes, unlock the lock and finally release the spinlock.

The condition variables of this implementation are also equipped with a spinlock along with a waiting queue for blocked threads/processes. If a condition is not met and a process/thread has been told to wait, then the spinlock must first be acquired. Then the current process/thread must release any locks it is using and block itself until the condition has been met. When the process/thread blocks itself, it will proceed to release the spinlock it was using. When the condition has been met it will eventually try to acquire the lock it was using again. This condition variable has two possible ways to release threads/processes. Both “signal” and “broadcast” functions must acquire a spinlock before they can release any threads/processes. The main difference is that “signal” will release only the first thread/process in the queue while “broadcast” will release all waiting threads/processes if they exist in the queue.

The implementation of the mailboxes support 6 functions: initialize, open, close, status, send and receive. The mailboxes are initialized with 7 variables: one to count how many processes that have opened a specific mailbox, a lock and two condition variables to identify if a process has to wait for the mailbox to get more space or for it to receive more data. The count variable is used to keep track of the number of messages in the mailbox while the tail and head keep track of where the first message starts and where the first available byte in the mailbox is. All variables are initialized to zero and the lock and spinlocks to “unlocked”.

To use a mailbox it must first be opened. This is done by incrementing the “used” variable and return an identification number for this mailbox. The mailbox is also closed in a similar fashion simply by decrementing the “used” variable. If the mailbox goes unused it will also reset all the variables to what they initially were.
The status function will store how many messages the given mailbox holds, along with how much space is still available in two provided integer variables.

To insert a message in a mailbox a lock must first be acquired so that only one process at a time can insert messages to it. If the mailbox does not have enough space for the message however, the process trying to insert the message must wait until there is. Provided there is space in the mailbox, the size of the message is first converted into ASCII characters and inserted at the first available byte in the mailbox buffer. This is done by using the provided “integer to ASCII” function. After the size of the message is inserted then the message itself is inserted. The “head” variable that points to the first available byte in the mailbox is incremented with as many spaces the size and the message uses and is reset to zero if it exceeds the size of the mailbox buffer, since it is a rounded buffer. Finally the count variable is incremented since there is one more message in the mailbox, any processes waiting for a message to arrive is released and the lock is released, allowing other processes to insert messages.

For a process to fetch a message from a mailbox it needs to be the only one doing so by acquiring a lock. Once the lock is acquired the process will wait if there are no messages in the mailbox. However, if there is a message to be received from the mailbox then the size of the message is converted from ASCII to integer and placed withing the message. After this the message itself is copied from the mailbox to the process and the tail variable is incremented to the start of the next message in the mailbox. Finally the count variable is decremented to show that there is one less message in the mailbox, any processes waiting for the mailbox to get more space is released and the lock is unlocked.

The implementation of the keyboard support is fairly simple. When a key is pressed and the keyboard controller has stored the scan code in a buffer and generated the necessary hardware interrupts the “put character” function is called. This function will allocate enough space for the pressed character and insert it to a given mailbox provided there is space in the mailbox. If there is no space in the mailbox then the character is discarded and the user must try to press a key again. When the character which is put in the mailbox should be received/read it is necessary to enter a critical section. This is because we can not allow the information being received to be lost due to interrupts. Once inside the critical section, the message from the mailbox is stored inside an allocated memory space, the provided integer pointer is set to this space and finally the critical section is exited.

Loading processes from a storage device involves reading the process directory from the image file. The process directory is located after the bootblock and kernel image (if present) in the image file and is copied into a character array using the provided “scsi_read” function. To load a process from the disk and run it some space have to be allocated at first. The allocated memory contains enough space for the process along with a stack of 8 kilo bytes. The process code is then copied into the allocated memory space by locating it from the process directory with a given location and size and using the “scsi_read” function. Finally a process control block is created for this process and placed inside the ready queue.

Results & Discussion

The implemented code is able to load all processes and run them simultaneously along with the threads.
Some changes has been made to the original code. Line number 104 in process3.c caused an error while running the Bochs emulator. The error caused process 3 to stop counting at 999 and process 4 to stop counting at 994 when they should both count to 1000 and print “done”. This error was reported on a discord server and has been used in this implementation.
The simulator has also been modified to run slower by changing the “cpu_mhz” variable in time.c, line 106 from 1000 to 100. This slows down the clock rate at which the processes run and makes the simulation more readable.

At one point during the mailbox implementation, a small but noteworthy error occurred. This error was due to that the condition broadcast and signal functions did not use spinlocks. This small error caused inconsistencies in the message sizes that were sent between process 3 and 4. The reason for the inconsistencies was that the processes had the opportunity to release other processes either waiting for a message to arrive in the mailbox or for the mailbox to get more space. Telling a process that there is space in the mailbox or let it fetch a message when there is no message will cause it to overwrite or/and mess with the contents of the mailbox causing inconsistencies.

This implementation has been run and tested on a HP laptop with the following specs: AMD Ryzen 5 2500U processor, Linux operating system, 8GB RAM, using the Bochs emulator.

Conclusion

This report described the design and implementation of a mailbox as a message passing mechanism along with how programs are loaded from a memory location and executed, in a simple operating system. For the mailboxes to work correctly the synchronization primitives locks and condition variables were implemented which also made use of the provided spinlock primitives. The implemented code is able to load and run all processes&threads simultaneously without any errors.