INF 2201
Assignment 3, Preemptive Scheduler
March 2022

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

Introduction
This report describes how barriers, condition variables, locks, semaphores and a interrupt timer are implemented to support a preemptive scheduler. The report also describes a fair solution to the dining philosophers problem using condition variables.

Technical background
Before reading this report the reader should be familiar with critical sections, the dining philosophers problem and the synchronization primitives: barriers, semaphores and condition variables.

A critical section is a section of code that multiple threads or processes have access to and are able to modify. In such sections concurrent programming problems can occur which is where synchronization primitives come in handy.

One synchronization technique are barriers. A barrier is a point where a group of threads or processes must stop and are not allowed to continue until all other threads/processes have reached this point. Once the threads/processes have left the barrier they are realigned and will be in the same stage of execution.

Another synchronization method are condition variables. A condition variable allows you to take one thread or process of the scheduling queue and stop its execution if a specified condition is not met. When the condition is met the thread/process will be put back into the scheduling queue. This synchronization method is useful when you don't want a thread to continue until another thread has reached a specific point of execution. A condition variable has three functions: wait, signal and broadcast. The wait function will block the execution of a thread until a condition is met, the signal function will unblock one thread/process and continue its execution while the broadcast function will unblock all threads/processes that are blocked by the wait function.

The last synchronization mechanism are semaphores. Semaphores are implemented in such a way that they can allow one or multiple threads/processed to have access to a resource simultaneously. How many threads/processes have access depends on the value of the semaphore. If the semaphore value is zero then threads/processes that try to acquire it becomes blocked and remains blocked until they are released by the semaphore. A semaphore has two functions: down and up. The "down" function will decrement the value of the semaphore if it's not zero and will block a thread/process if so. The "up" function will increment the semaphore value if there are no threads/processes waiting for it and will unblock a waiting thread/process if it exist.

The dining philosophers problem is an illustration of a synchronization issue that may occur in concurrent programming and is created by Edsger Dijkstra. The dining philosophers problem states that five philosophers sits around a round table with one fork in between each philosopher where each philosopher can either be in a thinking or eating state. When a philosopher is hungry and wants to eat he will need to acquire the two forks adjacent to him, but if he cannot do this he cannot eat and will be in a thinking state. In this illustration the philosophers represents processes or threads that want to do some work and the forks represents their resource for executing the work.

Design&Implementation

In order for the threads and processes to be scheduled in a preemptive way, they need to be interrupted and scheduled by a timer interrupt. This is done by a hardware interrupt that occurs every 10 milliseconds. In order for the interrupt to execute correctly the critical section function is used so that the interrupt work can be carried out safely. Enter_critical and Leave_critical are helper functions that will enable or disable interrupts. By using these functions one can be sure that a critical section of a thread or process is not interrupted by the interrupt timer. When the interrupt occurs the context of a thread/process is saved and a “end of interrupt” signal is issued before the context of the thread is restored, the scheduler called and the critical section is exited.

Most of the threads that are executed make use of locks and functions related to barriers, condition variables and semaphores. The work performed in these functions make heavy use of critical sections except when these methods are initialized. At the start of a function the critical section is entered and when the work has been done the critical section is exited.

The locks in this implementation are equipped with a status variable indicating if it is locked or not and a queue where blocked threads trying to acquire the lock can be placed. If the status of the lock is “locked” when a thread tries to acquire it, the thread will be blocked and placed inside the locks queue. If the lock is “unlocked” however, it will be set to “locked” and the current thread can continue its execution. When the lock is released by a thread its status will be set to “unlocked” and all threads that are blocked will be placed back into the ready queue.

The condition variable structure is equipped with a queue where threads can be placed if a special condition is not met, this queue is initialized to be empty. If a condition has not been met and the thread has to wait, it will release any locks that it has acquired before it will be blocked by the condition variable. If the condition has been met and another thread makes the signal function, this thread will be released, placed back into the ready queue, try to acquire its lock again and continue its execution. If a thread calls the broadcast function a loop will assure that all waiting threads in the condition variable’s queue is released

The semaphores contains a variable that counts how many threads have been allowed to execute by the semaphore and a queue for blocking threads. In a case where a thread will try to take a semaphore and there is space in the semaphore, the semaphore counter will be decremented. If this counter is zero then the current thread will be blocked. When the semaphore “up” function is called

a single thread may be released if there are threads waiting in the queue, if the waiting queue is empty then the semaphore counter is incremented indicating that there is space in the semaphore.

The barriers are implemented with a variable to keep track of how many threads it allows to block, a counter for how many threads that are currently blocked and a blocked queue. When the barrier wait function is called, the counter that counts the number of blocked threads is incremented. Then, two if-check are made: if there is room in the barrier then the current thread will be blocked, but if the maximum number of threads have been reached then all the blocked threads will be released and put back into the ready queue. At the end of the function the counter for the number of threads is set to zero indicating that the barrier is now empty again.

A fair solution to the dining philosophers problem would be that every philosopher eats the same amount of food. This solution is implemented using a condition variable and three locks that represent the forks. By using these methods one can prevent a philosopher from eating if he does not acquire two forks and can furthermore be placed in a waiting queue. Each of the philosopher functions are implemented in the same way except that the first philosopher “num” will initialize the condition variable and the lock. The other two philosophers will therefore not be able to eat until this is done.

When a philosopher starts executing he will check if his neighboring philosophers “is_eating” variable is set to 1, indicating that they are currently eating. If none of the neighboring philosophers are eating he will proceed to check how many times they have eaten. If the case is that they have eaten more or equal than the current philosopher, then he is allowed to take up two forks/locks and start eating. The current philosopher’s “is_eating” variable is then set to 1 and a counter for how many times he has eaten is incremented by 1. Should the case be that one of his neighboring philosophers are eating or one or both have eaten less than himself, he will be blocked by the condition variable.
When a philosopher has eaten for a random amount of time he will release both forks, set “is_eating” to 0 and call the broadcast function releasing all other philosophers that may have been blocked by one of the forks/locks he was using. Now the other philosophers have an opportunity to eat. Each philosopher will also print a statement to the Bochs emulator showing that they eat the same amount.

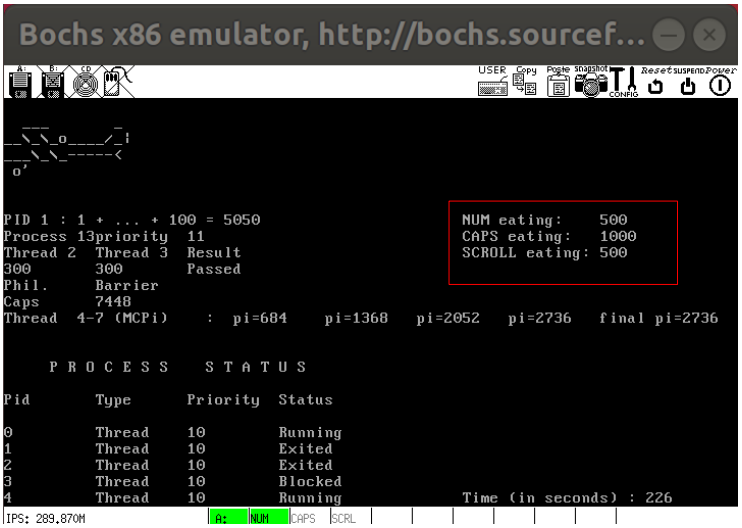
Results & Discussion

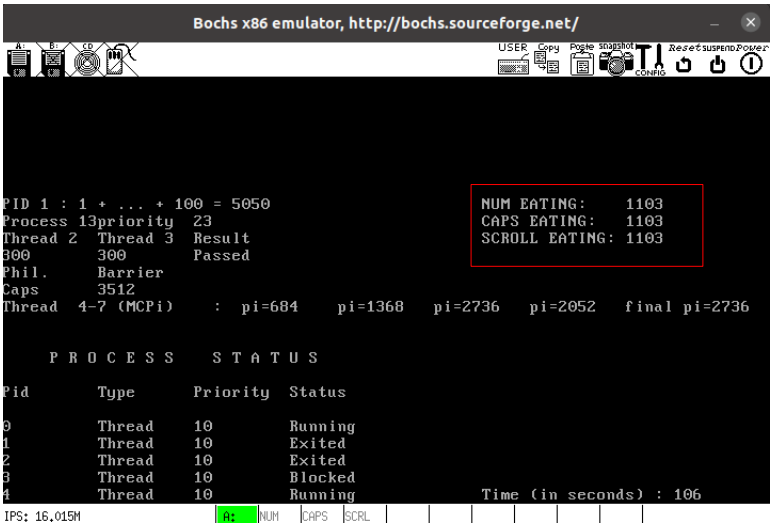
The implemented code is able to run all processes and threads simultaneously in a preemtive way. However due to a possible error in the fair implementation of the dining philosophers an error occurs. The message: “*internal keyboard buffer full (imm)”* will appear after 35 – 40 seconds of running the Bochs emulator and is caused by the “*update_keyboard_LED*” function running in philosophers.c. The short term solution to this problem has been resolved by commenting out these functions in the fair implementation, however the reason why this happens is still unresolved.

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, and using the Bochs emulator.

The reason why the first implementation of the dining philosophers were not fair is because that the “caps” philosopher is scheduled twice as much as the other two philosophers and therefore is eating twice as much. This is caused by the order of how “num” is picking up his forks and the use of semaphores. Philosopher “num” always starts to pick up his left fork instead of his right, such as the other philosophers are doing. This is causing an interruption in the rhythm of how thing are flowing. Forcing “num” to pick up his right fork before he picks up his left fork in the unfair implementation will cause “caps” to eat the same amount as the other two.
Another problem is that the implementation uses semaphores. By using semaphores a philosopher may be blocked while he holds one fork, which causes bad efficiency and interrupts the flow of the program, because the fork is not available even though it is not used. By using condition variables one can make sure that a philosopher will only pick up two forks if both are available.

The pictures below compares how much each philosopher eats from the pre-implemented verison versus the fair implementation:





Conclusion

This report described the design and implementation of a preemptive scheduler. For the preemptive scheduler to work correctly the synchronization primitives: barriers, condition variables and semaphores were implemented along with locks. A fair solution to the dining philosophers problem has also been described and implemented. The implemented code is able to run all processes and threads without the use of the keyboard LED functions.