INF-2201: OP fundamentals
Project 3 preemptive scheduling

Lise Ekhorn Johansen
UiT id: ljo177@uit.no
Github username: Lise-johansen
GitHub classroom: project-3-Lise-johansen
13.03.2023

1. Introduction

This project aims to transform the non-preemptive kernel into a preemptive kernel, which involves switching the system call mechanism from a simple function call to an interrupt-based one. Additionally, synchronization primitives such as semaphores, barriers, and conditional variables will also be implemented.
Additionally, an unfair philosopher dining problem has been presented, and the goal is to make it fairer.

2. Design

2.1 Interrupt handler

First, enter the critical section, and disable all other interrupts. It then sends an end-of-interrupt signal (EOI) to the PIC to clear the interrupt. Then a context switch is performed for the thread/process to run. Finally, the critical section state is left to enable other interrupts.

2.2 Synchronization primitive mechanic

There are four synchronization mechanics, lock, condition variable, semaphore, and barrier. Each mechanism is designed using structures and is responsible for managing resources or coordinating the execution of multiple threads.

2.2.1 Design of locks

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

2.2.2 Design of condition variables

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.

2.2.3 Design of semaphore

Semaphore is designed as a counter that keeps track of shared resources by counting how many threads are currently accessing or are done with a shared resource.
The semaphore counter will increase when a thread is done using a shared resource, and the counter will decrease when a thread wants to access a shared resource.

2.2.3 Design of barrier

The barrier can be compared to a bouncer waiting for all threads to arrive before unblocking them. If not all threads have arrived, the arriving threads are blocked until all threads have arrived.

3. Implementation

3.1 Interrupt handler

After the EOI, switch the kernel stack, call on the scheduler entry, and switch back to the user stack. And the fake interrupt handler is implemented as a system call.

3.2 Synchronization primitives

All the following implementations follow this pattern: enter critically, do the work, and leave critical.

3.2.1 Semaphore

A semaphore has to support two behaviors. The pseudocode 1 and 2 show the implementation of the two behaviors.
In pseudocode 1 the value of the semaphore is incremented. If the value is positive the threads in the waiting queue are unblocked.
To signal that a thread what to access a shared resource, the value is decremented by one. If the value becomes negative the currently running thread is blocked until something wakes the thread up again (i.e. value becomes positive).

```
void semaphore_up()
{
    value++;

    if (value < 0)
        {
            unblock(waiting);
        }
}
```

Pseudocode 1: Demonstrates the implementation of a counting semaphore, which this initialized through an "init" function that set all starting values. This code represents when a thread is done with the shared resource.

```
void semaphore_down()
{
    value--;

    if(value < 0)
        {
            block(waiting);
        }
    }
```

Pseudocode 2: This code checkers if there are any available shared resources, and if not, blocks the currently running thread and waits for the running threads to release the resources they are using (i.e. done running). When a resource is available, the blocked thread is unblocked and continues execution.

3.2.2 Condition variables

First, unblock the lock and block the thread, and when the thread is unblocked again the lock can be reacquired, which is the implementation of the condition variable. Pseudocode 3 is the implementation of the remaining behaviors needed for condition variables. Both unblock thread(s) from the waiting queue when it is not empty. But one uses a while to loop through and unblock all threads, and the other uses an if statement to unblock only one thread from the waiting queue.

```
void condition_something()
{
    when(waiting queue is not NULL)
    {
        unblock(waiting);
    }
}
```

Pseudocode 3: Collected logic for implementation. All the condition variables function is made through an init function, that initializes a waiting queue to empty.

3.2.3 Barrier

Pseudocode 4 illustrates the implementation of the barrier function. Every time a thread reaches the barrier, it increments a counter by one. When the counter equals the total number of threads required to reach the barrier, all the threads in the waiting queue are unblocked. If the counter is less than the total number of threads required to reach the barrier, the thread is blocked by adding it to the waiting queue. Important to initialize all initial values to empty or zero. Also, reset the counter to zero if the counter equals the required number of threads.

```
void barrier()
{
    n_equal++;

    if(n_equal == n_count)
        {
            n_equal = 0;

            while(waiting is not empty)
                {
                    unblock(waiting);
                }
        }
    else
        {
            block(waiting);
        }
}
```

Pseudocode 4: Illustrate the implementation of a barrier, all initializations happen in an init function.

4. Experiments and Result

4.1 Setup of experiments: fair

Test to see the fairness of the philosopher problem is done by looking at light (representing the philosophers) and see if it has a fair distribution of activity.

4.2 Result: fair

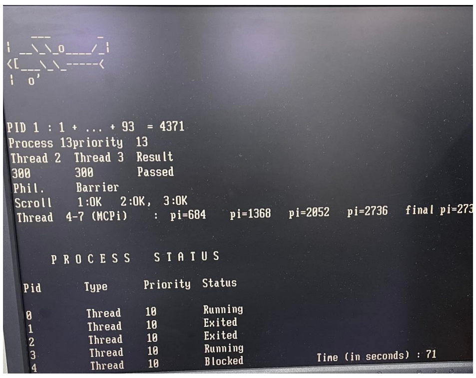
A philosopher's activities look to be evenly distributed in terms of frequency and duration. More on a fair result in the discussion.

4.3 Test for the correctness of the OS

To evaluate correctness, a single test is run by booting the OS image on a PC without an existing OS. Success is confirmed if the output matches the expected output and all tests pass.

4.4 result OS

Picture 1 shows the output when the OS image is run. Where the result matches the expected output.



Picture 1: Shows the results of the OS test. Test for lock (pi), condition variable (thread 1 and 2), and barrier have passed. The philosopher problem has successfully utilized the semaphores.

5. Discussion

5.1 Implementation: Lock

The lock used in this project is from project 2, with one modification. The lock acquire function was implemented without entering and leaving a critical section, making it possible to call it within another function that enters and leaves a critical section. Making the lock acquire function mutual exclusion.

5.1.2 Semaphore

One assumption regarding the implementation of the semaphores is that the threads are a non-busy wait.

5.1.3 Philosifores dining problem

Initialize a lock and a philosopher as condition variables. From the perspective of philosopher 1, a condition is set that only allows philosopher 1 to eat if it is not bigger than the other philosophers. If the condition is not met, the philosopher is put in the waiting queue, releases the lock, and signals to the others that the lock is released. If the condition is met, philosopher 1 gets to eat when, and when it is finished it releases the lock and signals to the others that the lock is released.

5.2 Philosophers' dining problem: fair

Fairness in this project means that every philosopher eats the same amount. Fairness is ensured through the use of a condition variable that places the philosophers in the waiting queue when it has eaten more than the others. Fairness is also ensured through a set eating time. This set eating time guarantees that no philosopher eats for a longer duration than the others.
Starvation is prevented by limiting how much one philosopher can eat. When one philosopher has eaten more than the other, it releases the lock (forks/table) and exists the critical section allowing the other the acquire the lock.
Deadlocks are preventers through the use of a lock, ensuring that only one philosopher can access the critical section and eat at a time.
Can improve the implementation by dividing a number by the time an iteration loop takes, instead of using a magical number for eating time.

5.3 Completion of project

Based on the experiment's results, a working preemptive kernel was made. And an implementation of a philosopher dining problem was made fairer. Can say it is fairer on the foundation of both a theoretical reason, as the use of conditional and locks guarantee that no philosopher can eat more or longer than the others. And an observational reason, as there are no discernible biases towards any of the philosophers when the code is running.

6. Conclusion

A working preemptive kernel was implemented, that passed all tests. Additionally, a fairer implementation of the philosopher dining problem using condition variables was implemented.

Bibliography

[1] educative.com, “What are conditional Variables in OS”, Ahmad. Amazon, Read: 24.02.2023
URIL: What are Conditional Variables in OS? (educative.io)

[2] Scaler.com, “Semaphores in OS”, Mature. Taneesha, Published: 11.02.2022, Read: 24.03.2023
URL: Semaphore in OS (Operating System) | Scaler Topics