Faculty of Engineering and Natural Sciences

Department of Information Technologies



**IT 204: Operating Systems**

**Project 3: Concurrency**

**Task 3.2: Parenting A Child**

Student: Mirza Sijamić 23004409

Professor: Dr. Elma Avdic

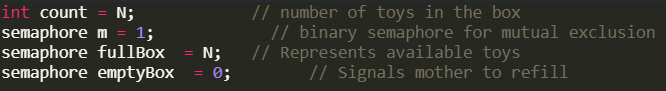
Teaching Assistants:

MSc. Nedim Bandzovic, MSc. Mirza Krupic, BSc. Benjamin Peljto

# Introduction

In this project, I had to use concurrency to synchronize the actions of multiple child threads and a single mother thread that were sharing a toybox with a limited number of toys. The goal was to ensure that the threads interacted according to two strict rules: first, to prevent a child from taking a toy if the box was empty, forcing them to wait for a refill. Second, I had to make sure the mother would only refill the toybox after it had become completely empty, preventing her from acting prematurely. This coordination was essential to prevent race conditions and manage the shared resource correctly.

# Pseudocode



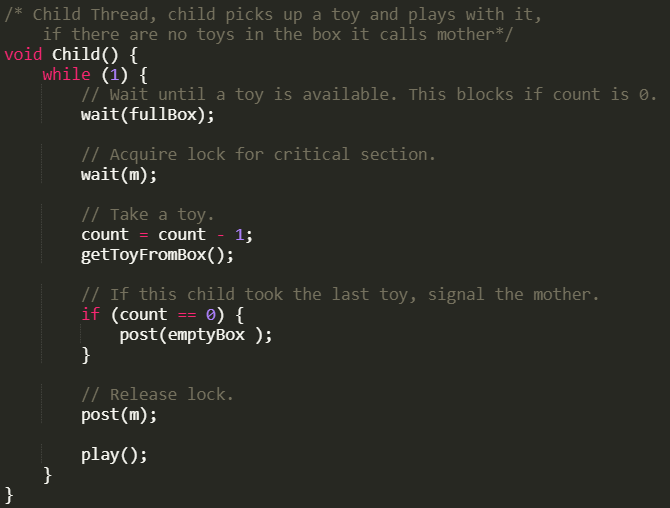
These are the declared variables in the project.

Variable count represents the number of toys that are currently inside the box.

I also declared 3 semaphores to fix this concurrency problem. Semaphore m is a binary semaphore that’s used for mutual exclusion. Any time a thread wants to read or change count, it must first acquire this lock. This prevents two children from trying to take the same toy at the exact same time.

Semaphore fullBox is a counting semaphore that represents how many toys are available for the child to pick up. A child must "acquire" one of these resources before it can take a toy.

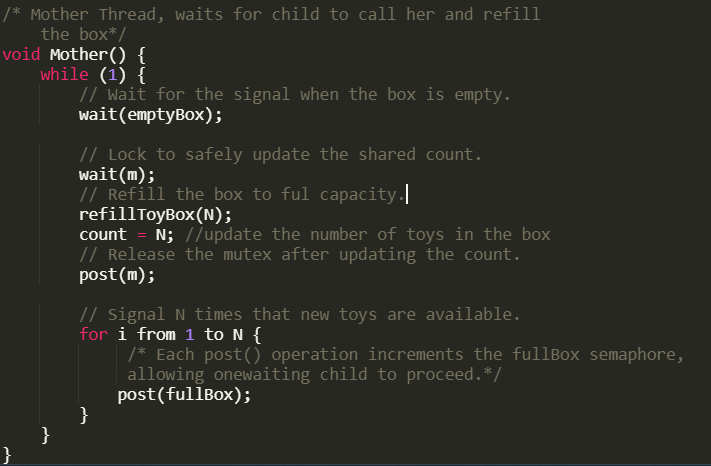
Third semaphore is emptyBox. It signals to mother that the box is empty and that she needs to refill it. The mother will wait on this semaphore until its value is greater than 0. It is also initialised to 0 which will cause the mother to wait immediately.



Child’s job is to pick up available toy from the toy box and play with it. It will first wait for fullBox semaphore to see if there are available toys. If the count is 0 this semaphore will block further action.

After that, it will lock the semaphore m (mutex) so that we can update the number of available toys in the box. If we didn’t lock it, one thread could accidentally cause a race condition and update the count prematurely.

After updating the count and getting a toy from the box, it will check if it took the last toy. If that’s the case, child will signal to mother that the box is empty using emptyBox semaphore. It will then release the mutex lock and start playing with it’s toy.



Mother thread has a job to refill the toy box if the child signals that it’s empty. It will wait until the signal comes that the box is empty. When that signal comes, we protect the shared variable count by locking it inside mutex and then she will refill the box to it’s full capacity.

After the box is refilled, we release the mutex and signal N times that new toys are available. Each post action will increment the fullBox semaphore so it allows one waiting child to pick up it’s toy.

# Synchronization Strategy

My synchronization strategy revolves around the classic Producer-Consumer problem. I used three distinct semaphores to manage the coordination between the Mother (the producer) and the Children (the consumers). The fullBox counting semaphore is the primary mechanism for the children, it represents the number of available toys. A child must successfully wait on fullBox semaphore before proceeding. This ensures a child only takes a toy when the box is not empty, as the semaphore's internal count directly corresponds to the available resources. The emptyBox semaphore acts as a signal for the mother. It is initialised to 0, causing the mother to wait immediately. Only the specific child who takes the last toy will post to this semaphore, waking the mother at the exact moment the box becomes fully empty.

Mutual exclusion for the shared count variable is guaranteed by the binary semaphore m, which acts as a mutex. Any thread, whether it's a child decrementing the count or the mother resetting it to N, must first acquire this lock. This prevents race conditions, such as two children attempting to take the last toy simultaneously. This design also handles the edge case of multiple children arriving at an empty box, the fullBox semaphore ensures they are queued up in a waiting line, and when the mother refills the box, her N post operations will correctly wake up exactly N of those waiting children, ensuring fair and accurate resource distribution without any further intervention.

# Test scenarios

**1. Scenario where the box becomes empty with a waiting child**

The toybox has a capacity of N=3. There are 4 children who want to play. The box starts full, so there are 3 toys in the box.

The first three children will each successfully call wait(fullBox), acquire the mutex, take a toy from the box and decrement the count. When Child 3 takes the last toy it sees that count is now 0 and calls post(emptyBox) to signal the Mother. Child 4 now arrives and calls wait(fullBox). Because the semaphore's internal count is 0, Child 4 will be blocked and put into a waiting state until the box becomes full again.

The Mother thread will execute, refill the box and call post(fullBox) three times. This will increment the semaphore's count and unblocking Child 4 immediately. Child 4 will then proceed to acquire the mutex and take a toy. This test demonstrates that the signaling between child and mother works correctly and that children are properly and waiting when no toys are available.

**2. Scenario where multiple children are waiting for a reffil**

The toybox has a capacity of N=5 but lets say the box starts empty (count = 0), and there are 7 children already waiting. This means 7 child threads are currently blocked on the wait(fullBox) call. The emptyBox semaphore has just been signaled, and the Mother is about to run.

The Mother thread executes. She refills the box with 5 toys and then enters the for loop, calling post(fullBox) five times. Each call to post() increments the semaphore's counter and unblocks one child from the semaphore's waiting queue.

Exactly five of the seven waiting children will be unblocked and will proceed, one by one, to take a toy. The remaining two children will stay blocked, waiting for the next refill cycle. This test confirms that the counting semaphore correctly manages a queue of waiting threads and distributes the exact number of available resources.

**3. Scenario where two children race for the last toy**

The toybox capacity is N=1. There are two children, Child A and Child B, ready to run. The box starts full with one toy.

Both Child A and Child B attempt to run at the same time. Both will try to call wait(fullBox). Since the semaphore's count is 1, only one child (let's say Child A) will succeed and proceed. The semaphore's count now becomes 0. Child B will be immediately blocked on the wait(fullBox) call. Child A will then acquire the mutex m, decrement count to 0, and see that it took the last toy. It will then post(emptyBox) to wake the mother and finally release the mutex m.

Only Child A successfully gets the toy and goes on to play. Child B remains blocked until the mother has finished refilling the box and posts to fullBox again. The count variable will never drop below zero, and the mother is signaled exactly once. This demonstrates that the mutex provides perfect mutual exclusion around the critical section, even under high contention for a single resource. Race condition is prevented by mutex (m).

# Conclusion

For this project, I used semaphores to solve the "Parenting A Child" problem, making sure the Mother and Child threads could work together without issues. I provided a pseuodocode that stops race conditions and makes sure the rules for using the shared toybox are always followed. I used the fullBox semaphore to keep track of available toys, and the emptyBox semaphore to act as an signal for the mother to know when to refill. The main lock (m) was very important because it protected the count variable by eliminating possibility of a race condition and making sure it was always accurate.

Working on this solution gave me a much better idea of how semaphores are used to fix these kinds of problems. This project was great practice for learning how to manage shared items between different threads, which is a key part of making sure programs run smoothly without crashing.