How Modern Machines Can Solve Synchronization Problems

Tim Ryan, Greg MacGown, Josh Rosenthal

Abstract

In this study, we have examined some classic synchronization problems (Namely the Dining Philosophers problem, the Bounded Buffer problem, the Sleeping Barber problem, and the Cigarette Smokers problem) and researched some ways in which said problems can be solved. In our studies, we have found that a majority of the problems are solvable using semaphores; however, there are usually other variations of solutions that use mutex locks in addition. We then explain why these solutions are viable and necessary.

1. Overview

One of the largest issues in computing is the issue of synchronization: A lack of correct synchronization can cause massive issues in a program, and it is extremely hard to consistently get the same errors as the timing of most synchronization issues is hard to predict. This is due to the fact that an error can happen at a random interval based on how the resources are getting allocated. Most modern operating systems solve this issue by using semaphores or mutexes (for example, the Windows developer kit allows usage of semaphore objects, mutex objects, and more).[1] Our goal in this research paper is to find the solutions to a few famous (or infamous) synchronization problems, explore these solutions, and explain both why a solution that requires semaphores or mutex locks is necessary as well as demonstrate both how and why the solution given works.

2. Semaphores

Windows treats semaphores as objects that have many parameters, the main one being a non-negative number that represents the count of the semaphore. Semaphores, similarly to how we learned about them in class, have two main states, being signaled, when the value of the semaphore is greater than zero, and non signaled, when the value is exactly zero.

Important methods that Windows uses in regards to semaphores are CreateSemaphore, CreateSemaphoreEx (a special call which adds an access mask to the semaphore), and OpenSemaphore, a call that allows processes that did not create

the semaphore to access it anyways. ReleaseSemaphore is also used to increase the count of the semaphore, assuming it is within the bounds of zero and the maximum after the call. Usually, Windows will create and release a semaphore in the same call, setting the value of the semaphore to maximum value and decrementing it as the resource that the semaphore is protecting is used.

When a semaphore is done being used, calling CloseHandle will close it. Closing the handle does not affect the semaphore's count. Therefore, calling ReleaseSemaphore before closing the handle or before the process terminates is to the benefit of the programmer to ensure consistency.[2]

3. Mutexes

Mutex signals are very similar to semaphores in multiple ways. Their functionality is nearly identical to semaphores, limiting access to certain protected areas of code. The main difference, however, is that mutexes have a maximum count of 1. They will only allow one process into the critical section at a time, similarly signaling when the mutex is occupied and not signaling when it has a count of zero. Many of the method names used with semaphores are similarly used, such as CreateMutex, OpenMutex, and ReleaseMutex. One notable difference, however, is that if a thread terminates without releasing its ownership of a mutex object, the mutex object is considered to be abandoned, rather than needing to be closed with a CloseHandle call and crashing the processes. Mutexes can only be released by the functions that they are owned by, while semaphores can be signaled and modified by any process in the system. This makes mutexes more secure, as not being able to be interrupted by outside processes is most certainly something beneficial and necessary to a programmer in certain situations. If this secure functionality is not required, a binary semaphore would be simpler to implement, as it would function very similar to a mutex without the unreleasable aspect.[3]

4. Dining Philosophers

The Dining Philosophers problem is most likely the most well-known synchronization problem. The premise behind the problem is that there are a number of philosophers (said number being greater than one) which represent processes. The philosophers are sitting in a circle around a big bowl of rice, and between each philosopher is a chopstick which represents a critical resource that the processes need to share. The philosophers will spend their time thinking when they are not hungry (representing an idle state) and eating when they are (representing a critical state). In order to eat, a philosopher must pick up a pair of chopsticks, but can only pick up one chopstick at a time. They will not pick up a chopstick if they notice that the other that they have access to is missing, but also will not let go of a chopstick unless they are no longer hungry. This may seem like it could be functional, but upon closer inspection we run into a problem: If every philosopher picks up a single chopstick at the same time,

then none of them have a second chopstick to pick up. Since each philosopher is still hungry, they will not let go of the chopstick they have until they are able to get a second chopstick. Since each philosopher will continuously be waiting for the philosopher next to them to put down their chopstick so they can eat, the program this case represents will result in deadlock and starvation. However, there are multiple ways that one can solve this synchronization problem, the most notable of which we will go over.

Resource Hierarchy: The first solution that we'll go over is the Resource Hierarchy solution. The main premise behind this solution is that you can assign each chopstick (or resource) a number, and have the philosophers pick up with the chopstick with the lowest number first. This is able to solve the problem due to the fact that the philosophers pick up their chopsticks in an asymmetrical manner: If the first philosopher picks up the chopstick on his left first, then that means that each other philosopher will pick up the chopstick on their right (and vise versa) due to the fact that the higher numbered chopstick will be on the opposite side for the philosopher in between the highest-labeled and lowest-labeled chopsticks. This means that if each philosopher were to attempt to grab their chopsticks at the same time, the first two philosophers would both attempt to get the lowest-labeled chopstick. After some sort of tiebreaker resolves the situation, one of the philosophers would notice that the chopstick he was going to grab is gone, and will therefore wait for that one to come back before attempting to grab the other.[4]

Waiter Solution: The Waiter solution, also known as the Arbiter Solution, suggests that instead of leaving the chopsticks between the philosophers that they are instead all left with a waiter (representing another program that manages the resources that the chopsticks represent). If a philosopher wants a chopstick, instead of being able to take it directly they must ask the waiter to have one, and once the philosopher is done eating they will give their chopsticks back to the waiter. The waiter will then give them the chopstick they requested, but if there's only one chopstick left and no philosophers holding enough chopsticks to finish eating the waiter may also deny the philosopher their request. This means that the waiter will only give a chopstick to a philosopher as long as the request would not put the system into a deadlock state.[5]

Chandy/Misra Solution: The Chandy/Misra solution is the most controversial solution to the Dining Philosophers problem, as it changes the setup of the problem to work. The first change made is that each chopstick is always being held by a philosopher at all times. The philosophers are now also assigned an ID number, and each chopstick is initialized to be in the possession of the philosopher with the lower ID of the two that it's being shared by. Furthermore, each chopstick now has a boolean stating whether they're clean or not, with each chopstick being initialized to a dirty state.

When a philosopher is hungry, they will request the necessary chopsticks from the philosophers that they're sharing with, and upon receiving the two that they need they will eat. Once they're done eating, the chopsticks they're holding will become dirty. The philosopher will then continue holding onto the chopsticks until they either become hungry again (in which case they'll eat with the chopsticks they have) or they get a request from another philosopher for their chopstick. When a philosopher gets a request for their chopstick, if the chopstick that they're holding is clean they refuse to give it away until they're finished eating. Otherwise, they will clean off the chopstick and give it to the philosopher that was requesting it. This is able to prevent deadlock due to the fact that a philosopher will only refuse to relinquish a chopstick if they're hungry or eating, and it is impossible for each philosopher to be holding exactly one clean chopstick. This approach is controversial however, as the dining philosopher's problem is usually given the stipulation that the philosophers are unable to directly communicate with each other, which this solution does not account for.[5, 6]

Mutex Locks: The mutex lock solution for the Dining Philosophers problem is relatively straightforward. By declaring and initializing a mutex, we can set each philosopher to lock the mutex before they pick up any chopsticks, and release the mutex once they're finished picking up both of their chopsticks. While this only allows one philosopher to pick up chopsticks at a time, it ensures that deadlock cannot occur.[6]

Semaphores: The easiest way to use semaphores to solve this problem would be make a single semaphore and implement it as if it was a mutex. One would assume that there would be a more efficient way to use semaphores to solve the problem, but unfortunately there is no clear solution to do so.[6, 7]

5. Bounded Buffer

Another well-known semaphore problem is the bounded buffer problem, also known as the producer-consumer problem. The idea behind the bounded buffer problem is that there is a producer that produces a certain thing (whether that be a file, an integer, or whatever else) and puts it in a space to be picked up. Then a consumer, which needs the thing the producer made, intakes the item and does something with it. This may seem pretty clear-cut, but unfortunately there are two issues in regards to the implementation: The spot that the producer drops off to and the producer picks up from is designed to only hold one item, meaning that if the producer creates an item before the consumer gets the chance to pick up the item created before it, the producer is unable to do anything with the first item and doesn't know where to put it, leading to errors. Similarly, if the consumer goes to pick up an item before the producer is able to produce something, it'll cause the consumer to attempt to pick up a nonexistent object,

which obviously could cause some issues. That being said, these issues can be solved with the usage of mutexes and semaphores in a relatively hassle-free manner.

Semaphores: In this example, we can take advantage of the fact that semaphores can be implemented to only allow a single process to go at once, since the two processes in question cannot be allowed to "double-dip" and go twice in a row. In order to create a solution with semaphores, two semaphores are required. To start, the two semaphores, referred to as "full" and "empty", should be initialized to 0, and 1 respectively. Once the producer wants to start producing, the producer should wait until empty is signalled to make sure that the buffer the item being stored in is empty. Then, the producer can produce the item in question and put it in the buffer, signal full, and start waiting for empty to be signalled. Similarly, the consumer starts by waiting for full to be signalled, then is able to consume the item in the buffer and signal that the buffer is empty. The consumer should then start over, waiting for the buffer to be full again.[8]

6. Cigarette Smokers

The idea of the cigarette smokers problem is that there's three cigarette smokers sitting at a table, along with a non-smoking agent. The agent has an infinite supply of matches, rolling paper, and tobacco, and each of the smokers has an infinite supply of one of these items. The agent will, on occasion, randomly pick two ingredients needed to make a cigarette and place them on the table, and the smoker who doesn't have these ingredients should pick them up and make a cigarette out of them to smoke. However, two problems can come out of this: Firstly, if the smokers aren't correctly guided by the agent, then there wouldn't be anything stopping two smokers each grabbing one ingredient the agent supplies that they need at the same time, which would cause them to have to continue to wait for new ingredients that would never come. Furthermore, the table only has enough room for two ingredients at a time, meaning that if the agent attempts to give the smokers new materials before the old ones are cleared off he wouldn't have anywhere to put them.

Semaphores: To solve this problem with semaphores, four semaphores are needed: One to represent each smoker (we'll call them "match", "paper", and "tobacco" based on the material that the smoker has), and one additional semaphore, "agent", to let the agent know when he's able to set new materials onto the table. These semaphores should be initialized to have each smoker's semaphore be 0 and agent be 1. For the agent, he will start off waiting for the agent semaphore. Then, the agent will randomly pick a number between 1 and 3. Depending on what he picks, he will either place matches and rolling paper on the table and signal tobacco, place matches and tobacco on the table and signal paper, or place tobacco and rolling paper on the table and signal match. After this, he can restart. For each smoker, they will wait for their

respective semaphore to be signalled by the agent. After this, they will pick up the items on the table and signal the agent that he can place new items on the table. The smoker can then roll and smoke his cigarette at his leisure, then restart his process.[9]

7. Sleeping Barber

The basic idea behind the sleeping barber is that there is a barbershop with three waiting seats in it run by a barber. If the barber has no customers in the shop, he decides that he should take a much deserved rest and sleeps in a chair stationed in the corner in his shop. If a customer comes in and sees that the barber is sleeping, they'll go over to him and wake him up to get a haircut. If another customer comes in while the barber is giving a haircut, they'll sit in one of the waiting chairs if there's a vacant one, or will leave if there's nowhere for them to sit. Once the barber finishes giving his current customer a haircut, he'll then check if there's anyone waiting in the waiting seats, and if there is he'll give them a haircut while he's up. Otherwise, the barber will go back to sleep. There are a myriad of issues that can arise due to this problem, some being that multiple customers can try to sit in the same seat or check on the barber at the same time, the barber not seeing customers due to them not being seated when he checks, or the customer trying to wake the barber despite him already being awake due to the fact he's not currently cutting hair.

Semaphores: While the task of fixing all of the synchronization issues in this problem is a daunting task, it's not impossible. To start, we'll need four semaphores, a boolean, and an int: The semaphores will be "barber", "customer", and "isSitting", the boolean will be "sleeping" and the int will be "emptyChairs". During initialization, barber should be set to 1, customer should be 0, isSitting should be 1, sleeping should be true, and emptyChairs should be 3. The pseudocode would be as follows:

Barber:

```
while(true){
        //sleep
        wait(customer);
        wait(isSitting);        //makes sure all customers are seated before checking
        signal(barber);
        signal(isSitting);
        //cut the customer's hair
}
```

Customer:

```
while(true)
{
        //check for open chairs
        wait(isSitting);                //makes sure two customers can't take the same chair
```

```
        if(emptyChairs != 0)
        {
                emptyChairs --;
                signal(isSitting);
                signal(customer);
                //waits for barber
                wait(barber);
                emptyChairs ++;
                //gets hair cut
        }
        else
        {
                signal(isSitting);
        }

}
```

As you can see, even though there were a myriad of problems at the start, we were able to fix them all using only a few semaphores.[10]

8. Practical Applications of Synchronization

Windows OS is closed-source, so the following examples will be based off of community code/projects that happen to use Windows synchronization tools. In particular, we will look at the use of critical sections, mutexes, and semaphores, used in a way to allow code to execute in the correct order. The following code calculates prime numbers in a given range. It uses the critical section object to synchronize threads in the same process, ensuring that both threads don't compete for the same counter variable.

```
        volatile int counter = 0;
        CRITICAL_SECTION critical;

        int isPrime(int n)
        {
                for(int i = 2; i < (int)(sqrt((float)n) + 1.0) ; i++) {
                        if (n % i == 0) return 0;
                }
                return 1;
        }
```

```
unsigned int __stdcall mythread(void*)
{
	char* s;
	while (counter < 25) {
		EnterCriticalSection(&critical;);
		int number = counter++;
		LeaveCriticalSection(&critical;);
		s = "No";
		if(isPrime(number)) s = "Yes";
		printf("Thread %d value = %d is prime = %s\n",
			GetCurrentThreadId(), number, s);
	}
	return 0;
}

int main(int argc, char* argv[])
{
	HANDLE myhandleA, myhandleB;

	InitializeCriticalSection(&critical;);

	myhandleA = (HANDLE)_beginthreadex(0, 0, &mythread;, (void*)0, 0, 0);
	myhandleB = (HANDLE)_beginthreadex(0, 0, &mythread;, (void*)0, 0, 0);

	WaitForSingleObject(myhandleA, INFINITE);
	WaitForSingleObject(myhandleB, INFINITE);

	CloseHandle(myhandleA);
	CloseHandle(myhandleB);

	getchar();

	DeleteCriticalSection(&critical;);

	return 0;
}
```

The EnterCriticalSection() method attempts to enter the critical section. If no thread is currently using it, then the requesting thread successfully enters and continues executing its code. Otherwise, the thread will sleep until the other

thread in the critical section calls LeaveCriticalSection(), which signals the next thread to enter. This can be taxing because the code in the critical section is very short and executed quickly, which can cause wasted time because the thread that sleeps has to spend time waking up.

An alternative to this is the TryEnterCriticalSection() method, which signals whether the critical section is in use or not. It can be paired with a while loop like such to cut back on wasted time. This will cause the thread to spin until it's ready to enter.

```
while(!TryEnterCriticalSection(&critical;)){}
        int number = counter++;
        LeaveCriticalSection(&critical;);
```

Both of these methods can be combined by setting the spin count with the SetCriticalSectionSpinCount() method, or using the InitializeCriticalSectionAndSpinCount() to set the spin count on initialization. The thread will spin for a specified amount of attempts before it sleeps.

The next example is a mutex, which can be created with the CreateMutex() method. When finished, it will be freed with the CloseHandle() method, freeing the kernel resources used by the mutex.

```
volatile int counter = 0;
HANDLE mutex;

int isPrime(int n)
{
        for(int i = 2; i < (int)(sqrt((float)n) + 1.0) ; i++) {
                if (n % i == 0) return 0;
        }
        return 1;
}

unsigned int __stdcall mythread(void*)
{
        char* s;
        while (counter < 25) {
                WaitForSingleObject(mutex, INFINITE);
                int number = counter++;
```

```
                ReleaseMutex(mutex);
                s = "No";
                if(isPrime(number)) s = "Yes";
                printf("Thread %d value = %d is prime = %s\n",
                        GetCurrentThreadId(), number, s);
        }
        return 0;
}

int main(int argc, char* argv[])
{
        HANDLE myhandleA, myhandleB;

        mutex = CreateMutex(0, 0, 0);

        myhandleA = (HANDLE)_beginthreadex(0, 0, &mythread;, (void*)0, 0, 0);
        myhandleB = (HANDLE)_beginthreadex(0, 0, &mythread;, (void*)1, 0, 0);

        WaitForSingleObject(myhandleA, INFINITE);
        WaitForSingleObject(myhandleB, INFINITE);

        CloseHandle(myhandleA);
        CloseHandle(myhandleB);

        getchar();

        CloseHandle(mutex);

        return 0;
}
```

The WaitForSingleObject() method will wait for the mutex to be signaled, or until the specified timeout time is reached. In this case, the timeout is infinite, so it will always wait for a signal. This is essentially the same as a critical section coding wise. In terms of resources, mutexes use kernel resources while critical sections do not.

Lastly, semaphores can be used with the CreateSemaphore() method, or the OpenSemaphore() method if it's already created. Semaphores are also kernel objects, so CloseHandle() will be used to free the resources. The

semaphore is decremented through the WaitForSingleObject() method, and incrememented through the ReleaseSemaphore() method. Do note that it can't be incremented above the maximum specified value.

The following code uses a semaphore to increment a count variable on two threads, ensuring that the count variable is correct each time a thread accesses it.

```
HANDLE semaphore;
int count = 0;

void addCount(int increment)
{
	WaitForSingleObject(semaphore, INFINITE);
	count += increment;
	ReleaseSemaphore(semaphore, 1, 0);
}

unsigned int __stdcall mythread(void*)
{
	for ( int i = 0; i < 50; i++) {
		addCount(2);
	}
	return 0;
}

int main(int argc, char* argv[])
{
	HANDLE myhandleA, myhandleB;

	semaphore = CreateSemaphore(0, 1, 1, 0);

	myhandleA = (HANDLE)_beginthreadex(0, 0, &mythread;, (void*)0, 0, 0);
	myhandleB = (HANDLE)_beginthreadex(0, 0, &mythread;, (void*)0, 0, 0);

	WaitForSingleObject(myhandleA, INFINITE);
	WaitForSingleObject(myhandleB, INFINITE);

	CloseHandle(myhandleA);
	CloseHandle(myhandleB);
```

```
        printf("count = %d\n", count);

        getchar();

        CloseHandle(semaphore);

        return 0;
}
```

9. Conclusion

Our conclusion is that while semaphores are much more flexible than mutexes in their usability with mutexes mainly being used only to help code readability, the idea of using synchronization techniques as a whole is vital to being able to keep a system running smoothly.[11]

10. Sources

1) Schofield, McLean. “Synchronization Objects - Win32 Apps.” *Win32 Apps | Microsoft Docs*, 31 May 2018, https://docs.microsoft.com/en-us/windows/win32/sync/synchronization-objects
2) Multiple Authors. “Semaphore Objects - Win32 Apps.” *Win32 Apps | Microsoft Docs*, 31 May 2018, https://docs.microsoft.com/en-us/windows/win32/sync/semaphore-objects
3) Multiple Authors. “Mutex Objects - Win32 Apps.” *Win32 Apps | Microsoft Docs*, 31 May 2018, https://docs.microsoft.com/en-us/windows/win32/sync/mutex-objects
4) “Entity Generator.” *MATLAB & Simulink*, https://www.mathworks.com/help/simevents/ug/dining-philosophers-problem.html
5) rab@stolaf.edu. “Solutions to the Dining Philosopher Problem.” *Solutions to the Dining Philosopher Problem (CS 300 (PDC))*, 5 Oct. 2018, https://www.stolaf.edu/people/rab/pdc/text/dpsolns.html
6) Shih, Jenny. *The Dining Philosophers Problem*, 2 May 2019, https://jennycodes.me/posts/operating-system-the-dining-philosophers-problem
7) Biswas, Subham. “Dining Philosopher Problem Using Semaphores.” *GeeksforGeeks*, 16 Aug. 2019, https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/

8) Biswas, Subham. “Dining Philosopher Problem Using Semaphores.” *GeeksforGeeks*, 16 Aug. 2019, https://www.geeksforgeeks.org/producer-consumer-problem-using-semaphores-set-1/
9) “Cigarette Smoker's Problem.” *412--Cigarette Smoker's Problem*, http://www.cs.umd.edu/~hollings/cs412/s96/synch/smokers.html
10) kunaljoshi1. “Sleeping Barber Problem in Process Synchronization.” *GeeksforGeeks*, 14 Aug. 2019, https://www.geeksforgeeks.org/sleeping-barber-problem-in-process-synchronization/
11) Venki. “Mutex vs Semaphore.” *GeeksforGeeks*, 10 July 2018, https://www.geeksforgeeks.org/mutex-vs-semaphore/