Fusion Lotus Restaurant Simulation

Caroline Varner
CS 3502 Section 03
NETID: 000960251

March 1, 2025

1 Introduction

The Lotus Fusion Restaurant Simulation is a multi-threaded application that mimics a restaurant kitchen handling multiple customer orders simultaneously. My project aims to demonstrate a restaurant updating system that utilizes efficient thread management, synchronization mechanisms, and inter-process communication (IPC) techniques. The primary objective when building this project is to simulate a realistic kitchen environment where multiple chefs process orders concurrently while also ensuring the maintenance of proper synchronization and deadlock avoidance.

To achieve these goals and more, project A is written in C and utilizes ConcurrentQueues, mutexes, and monitor-based synchronizations to manage multiple chef threads safely. Project B implemented named pipes for IPC to enable chefs to log order completions in a shared logging system. This approach is shown by handling 100 customer orders with 15 chef threads while also ensuring efficient parallel processing. Additionally, race conditions and deadlocks are prevented and documented as the code simulation is running.

2 Implementation Details

2.1 Project A Threading Solution

```
try
{
    semaphore.Wait(); // Limit concurrent chefs to 5 at a time

    // Introduce a deadlock scenario: Try acquiring multiple locks
    lock1Taken = Monitor.TryEnter(lockObject, TimeSpan.FromSeconds(2));
    lock2Taken = Monitor.TryEnter(loggingLock, TimeSpan.FromSeconds(2));

    if (lock1Taken && lock2Taken)
    {
        if (ordersQueue.TryDequeue(out string order))
        {
            Console.WriteLine("...");
            Console.WriteLine($"\n   IN THE KITCHEN:    {chefName} is preparing {order}...\n");
```

```
15              Thread.Sleep(new Random().Next(1000, 3000)); //
    Simulate cooking time
16              Console.WriteLine($"                        {chefName}
     has completed {order}!\n");
17              Console.WriteLine("...");
18
19              LogOrderCompletion(order, chefName);
20              Interlocked.Increment(ref completedOrders);
21          }
22      }
23      ...
24  }
25  finally
26  {
27      // Release locks properly
28      if (lock1Taken) Monitor.Exit(lockObject);
29      if (lock2Taken) Monitor.Exit(loggingLock);
30      ...
31  }
```

Listing 1: Thread Synchronization Example

The simulation creates 15 threads representing the different chefs running the kitchen and 100 customer orders in a shared ConcurrentQueue of Strings (ConcurrentQueue¡string¿ ordersQueue). The Task Parallel Library (TPL) in C [1] was implemented to efficiently manage the multiple chef tasks while also ensuring thread synchronization and safety. For example, the Task.WhenAll() function is employed to make certain that all chef tasks have been executed by the time the program prints the final results. The synchronization mechanisms used include mutexes, interlocked operations, and deadlock handling. As shown in Listing 1, the mutexes (Monitor.TryEnter) used help prevent concurrent access conflicts whenever the chefs are ”preparing” their orders. Additionally, a semaphore (SemaphoreSlim(5)) was created to allow only five chefs to prepare orders concurrently to prevent excessive thread contention.

The interlocked operations [2](completedOrders, totalThreadsRun) used aided in atomic updates to shared counters (Listing 1, line 14). By acquiring lockObject, order processing is handled by only one chef at a time modifying shared order data. A separate lock, loggingLock, handles logging operations by ensuring that multiple chefs do not write logs simultaneously, preventing mixed or corrupt log entries. The use of Monitor.TryEnter() with a timeout mechanism, enables the program to prevent deadlocks, because chefs that fail to secure a lock within 2 seconds will attempt again, instead of waiting endlessly. This approach allows

for smooth parallel execution without compromising on data integrity and consistency and preventing thread starvation. The combined use of TPL and mutex locks offers a efficient model for a small scale multi-threaded restaurant simulation with chefs "preparing orders" concurrently while avoiding clashes in shared data processing.

2.2 Project B Inter-Process Communication (IPC) Solution

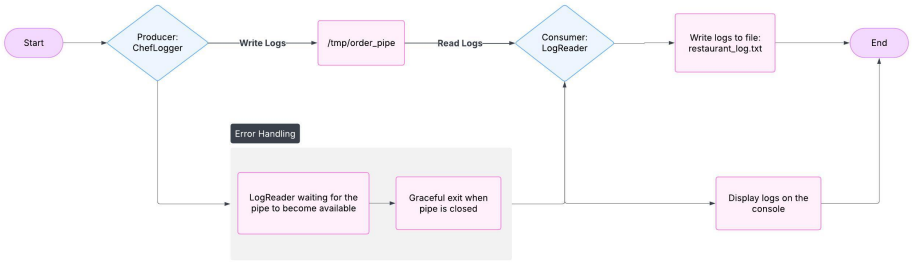

Figure 1: IPC Solution Flow Diagram

This project demonstrates Inter-Process Communication (IPC) through a producer-consumer pattern model, using C and named pipes to aid in real-time data exchange between ChefLogger and LogReader. As shown in Figure 1, ChefLogger acts as a producer that generates log messages about current food orders and writing them into a named pipe ($/tmp/order_pipe$). LogReader process serves as a consumer that reads the messages and displays them to the console, showing smooth communication between two applications. Additionally, ChefLogger waits for a connections before writing log messages so that no data is lost and once connected, it continuously reads from the pipe and prints each order. With the program being asynchronous in nature, the logs are processed in real-time and without delays. To secure robust error handling, the system accounts for numerous different failure senarios.

Including, LogReader starting before ChefLogger, with the named pipe waiting to become available to prevent crashes, and various empty read conditions in case ChefLogger unexpectedly terminates and LogReader needs to exit safely. These mechanisms and more ensure that there is smooth and error-free execution. Separate processes being able to communicate through named pipes is an essential element in multi-process systems, inter-service communication, and logging mechanisms in distributed applications. Therefore, by implementing a struc-

tured producer-consumer (Chef produces orders, log reads completed orders) this project highlights an efficient approach to navigating data exchange between independent processes.

Lastly, challenges that where encountered during this project mainly included utilizing basic functions in Linux (like dotnet run etc..) that I had never tried before that I needed to learn from Microsoft [3]. Once I was able to utilize these commands and successfully navigate Linux I wanted to make sure that I successfully understood how piping worked in Linus and how to create a basic piping project that would fit with my simulation. After consulting the logic presented on [4], I was able to understand how piping worked on a more in depth scale and was able to perform my piping operations smoothly.

3 Environment Setup and Tool Usage

My Project A multi-threading implementation code was developed using Visual Studio 2022 with the C console app. I decided to create my project using C because I plan to go into Web Development in my future career and thought that this project would give me more exposure to working with the language. In addition, I wanted my simulation to run similarly to a restaurant website that updates its customers as orders are prepared on a smaller scale. So I thought C would be the most appropriate as it gives developers the tools to write responsive applications, and I could utilize the Task Parallel Library (TLP). I have employed Visual Studio 2022 in previous projects and therefore did not need to download or modify the IDE at all for this project, however, I had never used Linux before now so I needed to install it for Project B.

Setting up the development environment for Project B involved installing and configuring Windows Subsystem for Linux (WSL) on a Windows 11 (version 24H2) laptop to allow cross-platform compatibility. I ran into some challenges when configuring WSL on my laptop including error 0x80370114 and 0x80370102 in Ubuntu. I addressed the first error 0x80370114 by consulting Microsoft forums [5] and a YouTube video [6] that walked me through accessing BIOS/UEFI settings on Windows 11 to turn on my virtualization and fix the issue. I then used another Microsoft forum [7] to find a solution for my 0x80370102 error that required me to go into my Windows features settings and turn on my Virtual Machine Platform feature. Even though it took some digging to find a resolution to these road blocks, once I fixed these two technical issues, I was able to finally access Ubuntu and create an account to successfully complete my project.

4 Challenges and Solutions

Several challenges were encountered during the project, including managing dependencies and ensuring compatibility with WSL. Unlike regular file-based communication, pipes require precise synchronization between processes, leading to one of my problems. Initially, my ChefLogger program would attempt to write data before the LogReader had a chance to establish a connection causing some writing failures. In order to fix this, I debugged by introducing a waiting mechanism that ensures that LogReader signaled its readiness before data transmission is allowed to begin.

In addition, error handling was also quite different because of how intricate it was. If the LogReader process terminated unexpectedly, the ChefLogger program would attempt to write to a closed pipe which lead to some system-level exception. To resolve this I implemented some try-catch blocks and checked for pipe availability before writing which helped alleviate these issues. Through trial and error, I gained a valuable experience in debugging IPC mechanisms as they where something I had never worked with before. I learned how to handle process synchronization, manage Linux-based development environments, and while the set-up process was challenging, it also helped me strengthen my ability to problem solve and trouble-shoot system-level programming issues.

5 Results and Outcomes

While I really enjoyed working on this project and feel like I learned a lot from my experience, I do feel like there are some areas I can improve on in my future projects. These areas being, the limitation with the named pipe implementation relying on a predefined producer-consumer structure that makes it less flexible for dynamic process connections. In the future, my iterations could explore message queues or shared memory to make a more scalable IPC. I also feel like error handling could be more enhanced by implementing a retry mechanism for failed transmissions instead of logging errors. Lastly, I could also further look into profiling tools that could help identify additional bottlenecks or better optimize resources. Even though I did not implement some of these ideas into my project I plan to learn more about how to utilize them and grow my understanding as I am working with IPCs more and I am proud of the work that I was able to accomplish because I got to explore concepts I had never tried before.

5.1 Project A Threading Solution

Run Cycle	50 Orders	100 Orders	1,000 Orders
1	99.69 seconds	194.10 seconds	2012.71 seconds
2	97.03 seconds	209.48 seconds	2032.89 seconds
Average:	98.36 seconds	201.79 seconds	2022.8 seconds

Figure 2: Threading Solution Stress Testing

The Lotus Fusion Restaurant Simulation was rigorously tested to ensure proper concurrency, synchronization, deadlock resolution, and performance under stress testing. The program was successful in demonstrating Concurrent Testing of 15 chef threads processing 100 orders because under testing the program run 25 times, I found that all the chefs worked simultaneously without interference. This was shown by the consol output showing multiple chefs preparing food at the same time:

Chef 1 is preparing Order 1: Sushi Platter...

Chef 2 is preparing Order 2: Kung Pao Chicken...

Chef 3 is preparing Order 3: Pad Thai...

Thus confirming that the ordersQueue was accessed safely and no two chefs processed the same order simultaneously. For Synchronization Validation testing, the synchronization mechanisms (Monitor and SemaphoreSlim) where effective in preventing race conditions because even with intentional delays introduced by the Process Orders Method, the program made sure that only one chef accessed the shared resources of ordersQeueu and logging at a time. No duplicate orders or corrupted data where found, and the consol only printed proper synchronization evidence:

Chef 4 has completed Order 4: Ramen Bowl!

Chef 5 has completed Order 5: Bibimbap!

Chef...

The program successfully showed Deadlock Detection and Resolution testing by printing the encountered potential deadlocks and alerting the user when those deadlocks have been resolved:

- Chef 6 encountered a potential deadlock and will retry.

+ Chef 6 successfully acquired and released locks.

Lastly, as shown in Figure 2, Under various loads of stress including minor (50 orders), medium (100 orders), and major (1,000 orders), the system stayed stable and still completed all orders without crashing and while maintaining a consistent execution time. The total execution time increased about 902.43 percent between completing 100 orders and 1,000 but the final output stayed consistent:

Number of Orders Completed: 1000
Number of Chefs: 15
Total Threads Run: 15
Total Execution Time: 2012.72 seconds

5.2 Project B Inter-Process Communication (IPC) Solution

To perform testing on my IPC solution and validate its inter-process communication, I conducted numerous tests focused on its data integrity, error handling ability, and performance benchmarking. For example, I ensured that data passed through the named pipe and remained intact by transmitting structured JSON-formatted order logs between the ChefLogger and LogReader. Additionally, by comparing the sent and received data, I was able to confirm that the information was correctly transmitted without being corrupted. I also test different data formats (text, numerical values etc.) to verify consistent behavior.

To cover error handling, I simulated what would happen if pipes broke and unexpected terminations happened to see how the system would respond. I found that if the LogReader process closed prematurely, the ChefLogger would catch the exception and handle it with ease to prevent crashes. Furthermore, placing malformed data into the pipe allowed me to verify that the LogReader could detect and discard invalid entries while also logging valid entries. When performing regular testing, I found that while small files had minimal delays, large files caused slight performance degeneration because of the system-level buffering. To solve this issue, I optimized data batching to improve efficiency and reduce some of the overhead that was happening.

6 Reflection and Learning

Working on both Project A - Multi-Threading and Project B - Inter-Process Communication, I gained deep insight into operating system fundamentals, concurrency, and process synchronization. The IPC project helped me understand

how to get independent processes to comminate efficiently. I learned how to install and work with Linus and how to navigate named pipes as a mechanism for passing data between two processes, as well as how important it is to deal with error handling with both of the independent programs. Debugging pipe-related issues helped emphasize how much I need to properly synchronize and handle exceptions to avoid race conditions and broken communication links.

The Threading project was very important is deepening my understanding of concurrent programming as initially I underestimated how complex it would be to manage shared resources across multiple threads. The project required the use of mutexes, semaphores and deadlock handling which reinforced my understanding of proper synchronization techniques. One of my favorite parts of this project was the concept of deliberately creating and resolving deadlocks. I had never done this before but by introducing dependencies between locked resources, I got to see how a deadlock scenario worked and how to resolve it (lock operating and timeouts). This type of hands on experience made a lot of the operating system concepts we have been learning in class more tangible and understandable. Overall, this project bridged a lot of the gaps I had in my understanding of operating system concepts and real world implementations. I feel like I have a stronger ability to write robust, well-synchronized, and efficient programs. I also feel more comfortable now working with low-level OS features which I can take into future programming projects to grow my knowledge.

References

[1] IEvangelist, "Task-based asynchronous programming - .net," Microsoft Learn, 2024, (accessed Feb. 22, 2025). [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming

[2] Dotnet-Bot, "Interlocked class (system.threading)," (System.Threading) — Microsoft Learn, 2023, (accessed Feb. 19, 2025). [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.threading.interlocked?view=net-9.0

[3] Tdykstra, "Dotnet run command - .net cli," command - .NET CLI — Microsoft Learn, 2023, (accessed Feb. 24, 2025). [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-run

[4] C. et al., “Piping in unix or linux,” GeeksforGeeks, 2023, (accessed Feb. 18, 2025). [Online]. Available: https://www.geeksforgeeks.org/piping-in-unix-or-linux/

[5] 13657895, “Resolving error: 0x80370114 the operation could not be started because a required feature is not installed.” Microsoft QA, 2023, (accessed Feb. 20, 2025). [Online]. Available: https://learn.microsoft.com/en-us/answers/questions/1187339/resolving-error-0x80370114-the-operation-could-not

[6] W. Report, “How to fix wslregisterdistribution error 0x80370114 on windows 11,” YouTube, 2022, (accessed Feb. 20, 2025). [Online]. Available: https://www.youtube.com/watch?v=E59qdiaFqws

[7] B. Mordecai., “Wslregisterdistribution failed with error: 0x80370102,” Microsoft QA, 2022, (accessed Feb. 20, 2025). [Online]. Available: https://answers.microsoft.com/en-us/windows/forum/all/wslregisterdistribution-failed-with-error/5fdd8aa6-175e-4a14-8fe6-07673133fd15

[1–7]