UNIVERSITI MALAYA

WIA2004 Operating Systems

Lab 9: Synchronization

Group：F5U

Occ: 10

Lecturer: Dr. Fazidah Othman

GROUP MEMBERS	MATRICS NUMBER
ZHANG ZHIYANG	S2193685
MEILIN LI	S2174975
HUMYRA TASMIA	S2176677
HUSNA NADIAH BINTI ROSTHAM	22060027
HUSNA BINTI IHSANUDDIN	U2102305

<u>Table of Content</u>

1.0 Introduction

Synchronization refers to the coordination of multiple tasks or threads in multi-tasking, concurrent programming, or distributed systems to ensure data consistency and system stability. It is a method of controlling access to shared resources to prevent issues such as data races, deadlocks, or other concurrency problems.

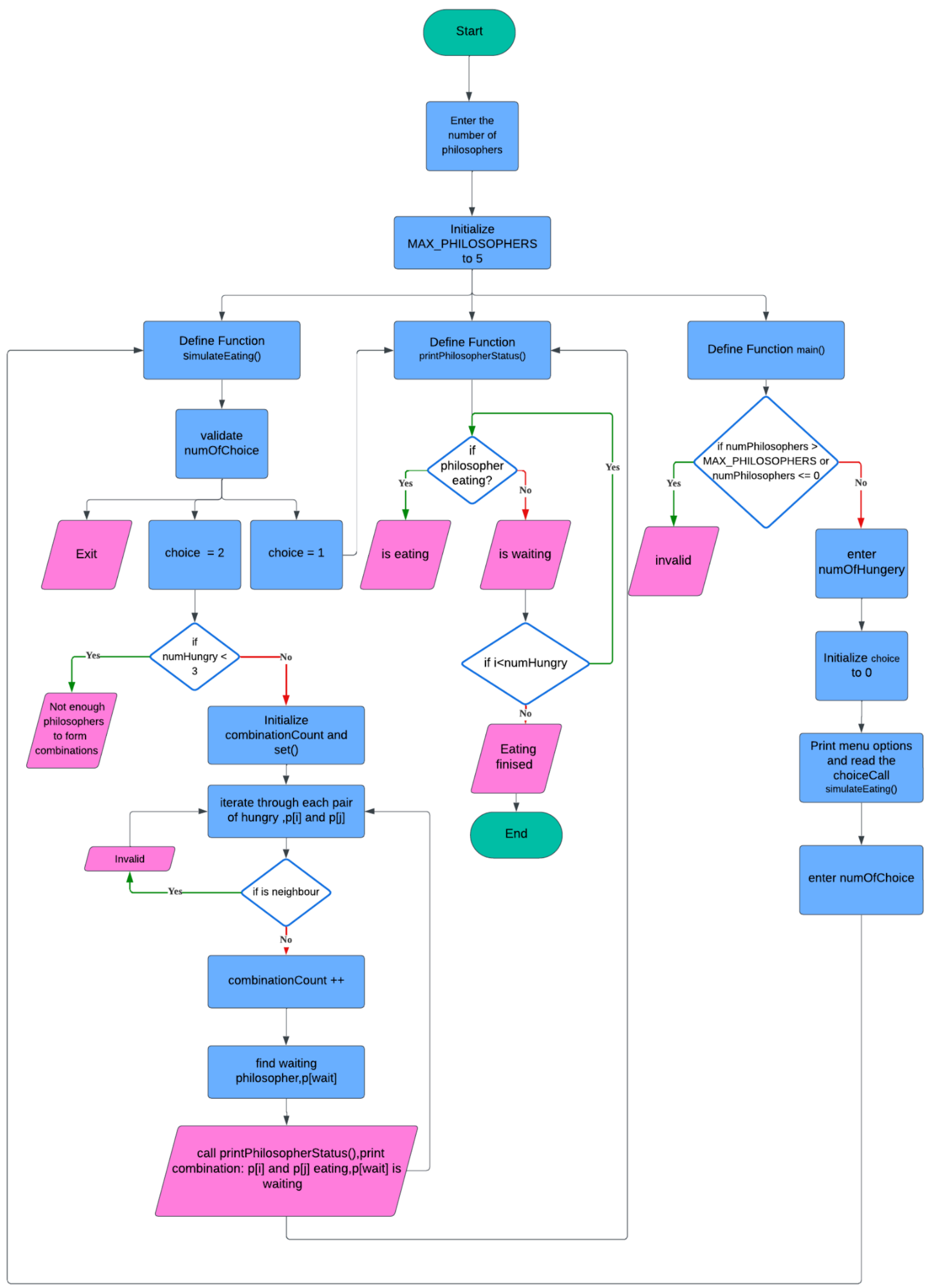
In a multi-threaded environment, multiple threads might access or modify shared data simultaneously. Without proper control, this can lead to data inconsistencies or system crashes. To address these issues, programmers use various synchronization mechanisms such as mutexes (mutual exclusion locks), semaphores, and condition variables. These mechanisms ensure that only one thread can access critical code sections or resources at a time, maintaining data consistency and ensuring the correct operation of the system.

In distributed systems, synchronization challenges are more complex because they involve multiple independent computing nodes. Common synchronization techniques include distributed locks, distributed transactions, and consensus protocols (such as Paxos or Raft). These techniques coordinate the operations of different nodes to ensure the overall consistency and reliability of the system.

In summary, synchronization is a crucial technology in concurrent programming and distributed systems. Effective synchronization mechanisms ensure the stability and data consistency of systems operating in multi-tasking environments.

2.0 Methodology

Flowchart



Pseudocode

```
1. MAX_PHILOSOPHERS = 5

2. Function printPhilosopherStatus(philosopher, eating):
  2.1. Print the status of a philosopher (eating or waiting)

3. Function simulateEating(numHungry, philosophers, choice, totalPhilosophers):
  3.1. If choice == 1:
     3.1.1. For each hungry philosopher:
        3.1.1.1. Print the philosopher is eating
        3.1.1.2. For each other hungry philosopher:
           3.1.1.2.1. Print the philosopher is waiting
     3.1.2. For each hungry philosopher:
        3.1.2.1. Print the philosopher is waiting
     3.1.3. Print "Eating finished"
  3.2. Else if choice == 2:
     3.2.1. If not enough philosophers to form combinations:
        3.2.1.1. Print "Not enough philosophers to form combinations"
     3.2.2. Else:
        3.2.2.1. Initialize combination count and set of paired philosophers
        3.2.2.2. For each pair of philosophers:
           3.2.2.2.1. If the philosophers are not neighbors:
              3.2.2.2.1.1. Increment combination count
              3.2.2.2.1.2. Add philosophers to paired set
              3.2.2.2.1.3. Print the combination with eating and waiting philosophers
        3.2.3. If no valid combinations:
           3.2.3.1. Print "No valid combinations for two philosophers eating at the same time"
        3.2.4. Else:
           3.2.4.1. For each philosopher not in paired set:
              3.2.4.1.1. Print the philosopher is eating
           3.2.4.2. Print "Eating finished"
  3.3. Else if choice != 3:
     3.3.1. Print "Invalid choice. Please try again"

4. Function main():
  4.1. Read the number of philosophers
  4.2. Validate the number of philosophers
  4.3. Read the number of hungry philosophers
  4.4. Validate the number of hungry philosophers
  4.5. Read the positions of the hungry philosophers
  4.6. Validate the positions
  4.7. Initialize choice to 0
  4.8. While choice is not 3:
     4.8.1. Print menu options
     4.8.2. Read the choice
```

```
    4.8.3. Call simulateEating with the choice
  4.9. Print "Exiting..."

5. If the script is executed directly:
  5.1. Call main()
```

3.0 Implementation

3.1 Coding

Lab9.py > printPhilosopherStatus

```python
MAX_PHILOSOPHERS = 5

def printPhilosopherStatus(philosopher, eating):
    #Print the status of a philosopher, whether they are eating or waiting.
    print(f"P{philosopher} ", end="")
    if eating:
        print("is eating.")
    else:
        print("is waiting.")

def simulateEating(numHungry, philosophers, choice, totalPhilosophers):
    #Simulate the philosophers' eating process based on the user's choice.
    if choice == 1:
        # One philosopher eats at a time
        for i in range(numHungry):
            printPhilosopherStatus(philosophers[i], True)
            for j in range(numHungry):
                if i != j:
                    printPhilosopherStatus(philosophers[j], False)
        for i in range(numHungry):
            printPhilosopherStatus(philosophers[i], False)
        print("Eating finished.")
    elif choice == 2:
        if numHungry < 3:
            print("Not enough philosophers to form combinations.")
        else:
            combinationCount = 0  # Initialize combination count

            # Iterate through each pair of hungry philosophers
            for i in range(numHungry - 1):
                for j in range(i + 1, numHungry):
                    phil1 = philosophers[i]
                    phil2 = philosophers[j]

                    # Check if philosophers are neighbors (cannot eat together if they are neighbors)
                    if (abs(phil1 - phil2) == 1 or abs(phil1 - phil2) == totalPhilosophers - 1):
                        continue  # Skip to the next pair if they are neighbors
```

```python
                    combinationCount += 1

                    # Find philosophers who are waiting
                    waiting_philosophers = [philosophers[k] for k in range(numHungry) if k != i and k != j]
                    waiting_status = " and ".join(f"P{p}" for p in waiting_philosophers) + " is waiting."

                    # Print the eating combination and the status of waiting philosophers
                    print(f"Combination {combinationCount}: P{phil1} and P{phil2} are eating. {waiting_status}")

            # If no valid combinations were found
            if combinationCount == 0:
                print("No valid combinations for two philosophers eating at the same time.")

    elif choice != 3:
        print("Invalid choice. Please try again.")

def main():
    #Main function to handle user input and start the simulation.
    numPhilosophers = int(input(f"Enter the number of philosophers (max {MAX_PHILOSOPHERS}): "))
    if numPhilosophers > MAX_PHILOSOPHERS or numPhilosophers <= 0:
        print(f"Invalid number of philosophers. Please enter a number between 1 and {MAX_PHILOSOPHERS}.")
        return

    numHungry = int(input("How many are hungry: "))
    if numHungry > numPhilosophers or numHungry <= 0:
        print(f"Invalid number of hungry philosophers. Please enter a number between 1 and {numPhilosophers}.")
        return

    philosophers = []
    for i in range(numHungry):
        philosopher = int(input(f"Enter philosopher {i + 1}'s position: "))
        if philosopher < 1 or philosopher > numPhilosophers:
            print(f"Invalid position for philosopher {i + 1}. Please enter a position between 1 and {numPhilosophers}.")
            return
        philosophers.append(philosopher)

    choice = 0
    while choice != 3:
        print("\nChoose one of the following:")
        print("1. One can eat at a time.")
        print("2. Two can eat at a time.")
        print("3. Exit.")
        choice = int(input())

        simulateEating(numHungry, philosophers, choice, numPhilosophers)

    print("Exiting...")

if __name__ == "__main__":
    main()
```

3.2 Sample output

```
PS C:\Users\hp\Desktop\OS> & C:/Users/hp/AppDat
Enter the number of philosophers (max 5): 5
How many are hungry: 3
Enter philosopher 1's position: 1
Enter philosopher 2's position: 2
Enter philosopher 3's position: 4

Choose one of the following:
1. One can eat at a time.
2. Two can eat at a time.
3. Exit.
1
P1 is eating.
P2 is waiting.
P4 is waiting.
P2 is eating.
P1 is waiting.
P4 is waiting.
P4 is eating.
P1 is waiting.
P2 is waiting.
P1 is waiting.
P2 is waiting.
P4 is waiting.
Eating finished.
```

```
Choose one of the following:
1. One can eat at a time.
2. Two can eat at a time.
3. Exit.
2
Combination 1: P1 and P4 are eating. P2 is waiting.
Combination 2: P2 and P4 are eating. P1 is waiting.

Choose one of the following:
1. One can eat at a time.
2. Two can eat at a time.
3. Exit.
3
Exiting...
PS C:\Users\hp\Desktop\OS>
```

4.0 Conclusion

In conclusion, we explored the Dining Philosophers Problem to understand synchronization and concurrency in computing. Through the code simulation, participants experimented with different strategies for managing resource sharing among processes, such as allowing only one or two philosophers to eat at a time while preventing deadlocks. This practical application helped highlight the complexities of resource allocation and the importance of careful planning in systems where multiple processes access shared resources. This lab offers a practical and clear example of concurrent programming and how to avoid deadlocks, which are situations where different parts of a program block each other, stopping the entire program. However, a drawback is that this simple model doesn't fully show the more complicated behaviors and challenges found in real-life systems, where many processes might be running at the same time and interacting in more complex ways.

5.0 References

1. Herlihy, M., & Shavit, N. (2012). The Art of Multiprocessor Programming (Revised Reprint). Morgan Kaufmann. (Chapter 3, pp. 45-67).
   The Art of Multiprocessor Programming, Revised Reprint (acm.org)

2. Yadav, B.(2023). Process Synchronization in OS. Scaler. Retrieved May 29, 2024, from https://www.scaler.com/topics/operating-system/process-synchronization-in-os/

3. Jain, S. (2023). *Introduction of Process Synchronization*. GeeksforGeeks. Retrieved May 29, 2024, from https://www.geeksforgeeks.org/introduction-of-process-synchronization/