

**OPERATING SYSTEM**

Process Synchronization

**GROUP MEMBERS**

Syed Shaheer Hassan (02-134201-068)

Kamal Kumar (02-134201-058)

Muhammad Hammad Ahmad (02-134201-063)

**BS (CS) – 5: SECTION: A**

**Course Instructor: Sir Shahid**

**Lab Engineer: Miss Fatima**

Department of Computer Science Bahria University, Karachi Campus

Table Of Contents

Introduction

Scope Of Project

Algorithms

* **Petereson’s Algorithm**
* **Dining Philosopher Problem Algorithm**
* **Producer Consumer Problem Algorithm**

**Source code**

Outputs

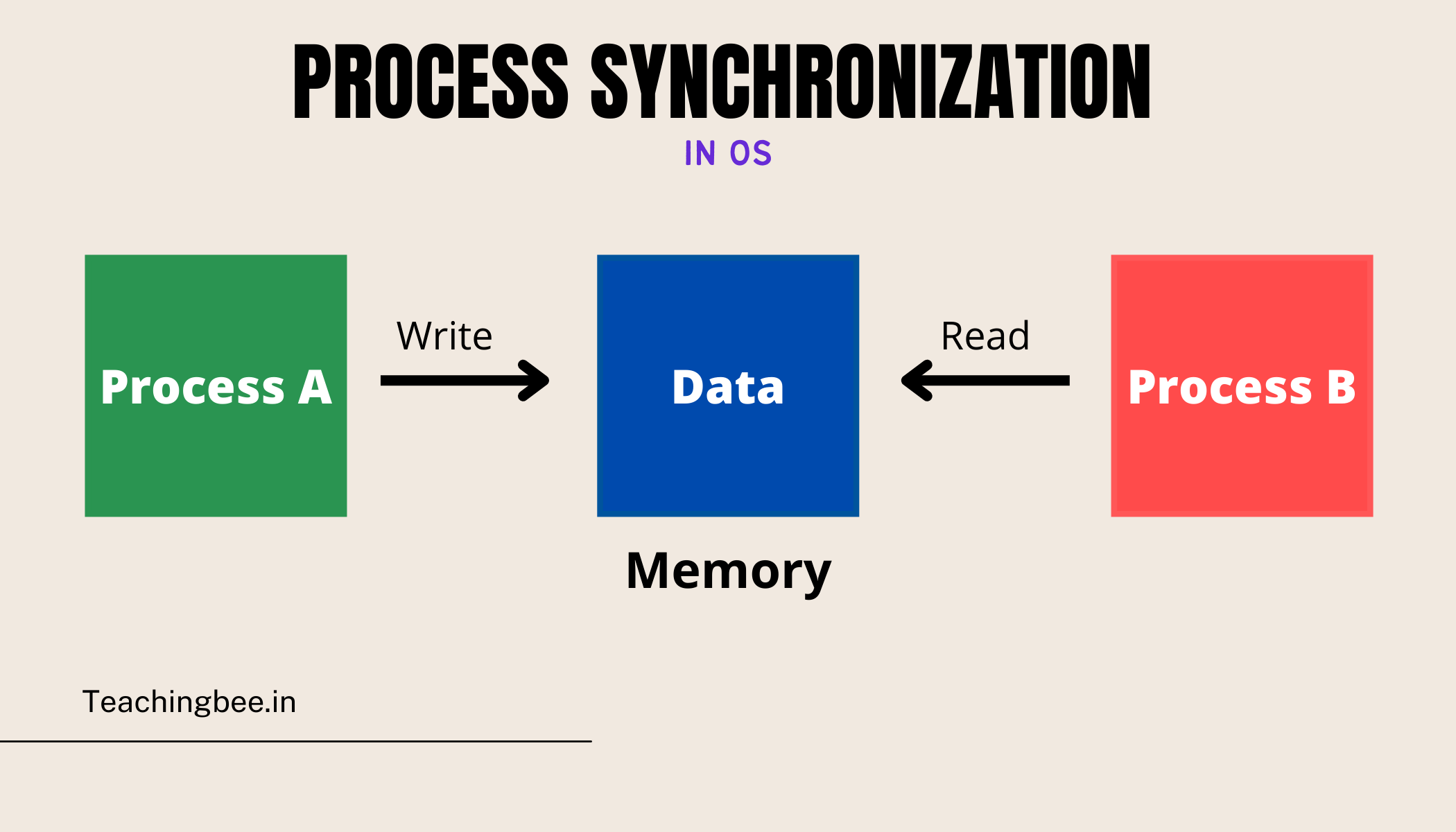
Reference Links

# Introduction:

**Process Synchronization** is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources.

It is specially needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or data at the same time.

Processes have scheduled to ensure that concurrent access to shared data does not create inconsistencies. Data inconsistency can result in what is called a race condition. A race condition occurs when two or more operations are executed at the same time, not scheduled in the proper sequence, and not exited in the critical section correctly.



# Scope Of Project:

The main purpose of Process synchronization is the sharing of resources without interference using mutual exclusion. The other purpose is the coordination of the process interactions in an operating system. Semaphores and monitors are the most powerful and most commonly used mechanisms to solve synchronization problems.

And our main object is to not slow down the process execution but to make the process synchronized. We’ve used different methods which will determine the priority of the processes and execute them depending on those priorities.

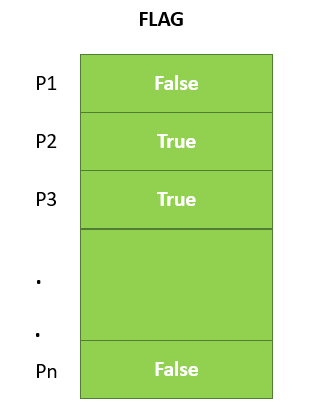
# Algorithms:

The three algorithms we sed in our project are:

* **Petereson Algorithm**
* **Dining Philosopher Problem Algorithm**
* **Producer Consumer Problem Algorithm**

**Peterson Algorithm:**

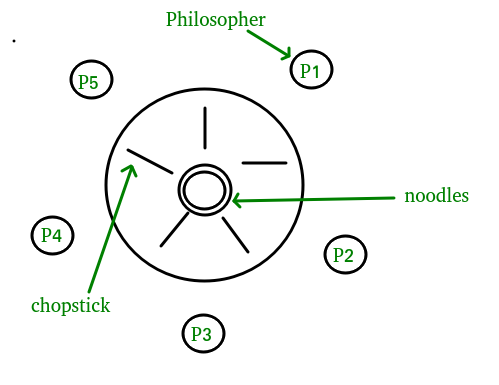
Petereson’s Algorithm is used to synchronize two processes. It uses two variables, a bool array flag of size 2 and an int variable turn to accomplish it. In the solution the flags are initially false. When a process wants to execute it’s critical section, it sets it’s flag to true and turn as the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process performs busy waiting until the other process has finished it’s own critical section.  
After this the current process enters it’s critical section and adds or removes a random number from the shared buffer. After completing the critical section, it sets it’s own flag to false, indication it does not wish to execute anymore.



**Dining Philosopher Problem Algorithm:**

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

There are three states of the philosopher: THINKING, HUNGRY, and EATING. Here there are two semaphores: Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher.

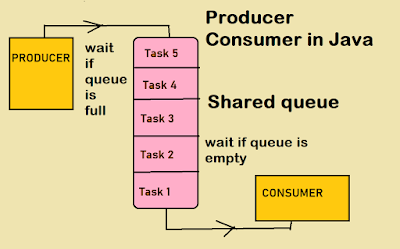


**Producer Consumer Problem Algorithm**

The **Producer-Consumer problem** is a classic synchronization problem in operating systems.

The problem is defined as follows: there is a fixed-size buffer and a Producer process, and a Consumer process.

The **Producer** process creates an item and adds it to the shared buffer. The **Consumer** process takes items out of the shared buffer and “consumes” them.



The solution to the Producer-Consumer problem involves three *semaphore* variables.

**Semaphore Full**: Tracks the space filled by the Producer process. It is initialized with a value of 00 as the buffer will have 00 filled spaces at the beginning

**Semaphore Empty:** Tracks the empty space in the buffer. It is initially set to **buffer\_size** as the whole buffer is empty at the beginning.

**Semaphore mutex**: Used for mutual exclusion so that only one process can access the shared buffer at a time.

# SourceCode:

import time

import random

import threading

**#Producer consumer code start**

class Semaphore():

def \_\_init\_\_(self, initial=0):

self.lock = threading.Condition(threading.Lock())

self.value = initial

def up(self):

with self.lock:

self.value += 1

self.lock.notify()

def down(self):

with self.lock:

while self.value == 0:

self.lock.wait()

self.value -= 1

class Memory():

def \_\_init\_\_(self):

self.buffer\_mutex = Semaphore(1)

self.fill\_count = Semaphore(0)

self.empty\_count = Semaphore(100)

self.memory\_buffer = []

class Consumer(threading.Thread):

def \_\_init\_\_(self, memory, producer=None):

super().\_\_init\_\_()

self.memory = memory

self.producer = producer

self.sleep()

def wake(self):

self.awake = True

self.asleep = False

def sleep(self):

self.awake = False

self.asleep = True

def run(self):

while True:

self.consume()

def consume(self):

self.memory.fill\_count.down()

self.memory.buffer\_mutex.down()

item = self.memory.memory\_buffer.pop()

print("Consumer consumes the item {}".format(item))

self.memory.buffer\_mutex.up()

self.memory.empty\_count.up()

class Producer(threading.Thread):

def \_\_init\_\_(self, memory, consumer=None):

super().\_\_init\_\_()

self.memory = memory

self.consumer = consumer

self.wake()

def wake(self):

self.awake = True

self.asleep = False

def sleep(self):

self.awake = False

self.asleep = True

def run(self):

while True:

self.produce()

def produce(self):

it = random.randint(1, 10)

print("Producer produces {}".format(it))

self.memory.empty\_count.down()

self.memory.buffer\_mutex.down()

self.memory.memory\_buffer.append(it)

print("Producer put {} in buffer memory".format(it))

self.memory.buffer\_mutex.up()

self.memory.fill\_count.up()

def mainPC():

memory = Memory()

consumer = Consumer(memory)

producer = Producer(memory)

consumer.producer = producer

producer.consumer = consumer

consumer.daemon = True

producer.daemon = True

consumer.start()

producer.start()

while True:

if consumer.asleep and producer.asleep:

print("Deadlock ocurred")

break

#Producer Consumer Code End

**#Petersons code start**

class MemoryPet():

def \_\_init\_\_(self, processes):

self.quantity = processes

self.flag = [False] \* processes

self.turn = 0

def calculate\_next\_process(self, process):

return (process + 1) % self.quantity

def enter\_region(self, thread):

other = self.calculate\_next\_process(thread)

self.flag[thread] = True

turn = thread

while self.turn == thread and self.flag[other]:

time.sleep(random.randint(1, 3))

def leave\_region(self, thread):

self.flag[thread] = False

class ProcessPet(threading.Thread):

def \_\_init\_\_(self, id, memory):

super().\_\_init\_\_()

self.id = id

self.memory = memory

self.incriticarea = False

def run(self):

while True:

while self.memory.turn != self.id:

time.sleep(random.randint(1, 3))

self.memory.enter\_region(self.id)

self.incriticarea = True

print("Process {} is in the critic area".format(self.id))

time.sleep(random.randint(1, 5))

self.memory.leave\_region(self.id)

self.incriticarea = False

print("Process {} left the critic area".format(self.id))

self.memory.turn = self.memory.calculate\_next\_process(self.id)

time.sleep(1)

def mainPet():

n = int(input("Type the quantity of processes: "))

memory = MemoryPet(n)

processes = []

for i in range(n):

p = ProcessPet(i, memory)

p.daemon = True

print("Create Process {}".format(i))

p.start()

processes.append(p)

for p in processes:

p.join()

**#Dining Philosopher code start**

hashis = [False] \* 5

philosofers = ["A", "B", "C", "D", "E"]

class Philosofer(threading.Thread):

def \_\_init\_\_(self, id):

"""Initiates the philosofer' properties """

super().\_\_init\_\_()

self.id = philosofers[id]

self.left = id

self.right = (id + 1) % len(hashis)

self.n\_hashis = 0

self.waiting = False

self.eating = False

self.thinking = True

def run(self):

"""The cerne of a philosofer: think and eat """

while True:

self.think()

self.eat()

def think(self):

""" Think a certain amount of time """

self.thinking = True

print("Philosofer {} is thinking...".format(self.id))

time.sleep(random.randint(1, 5))

def eat(self):

""" Eat the pasta when get the hashis """

self.take\_hashi(self.left)

self.take\_hashi(self.right)

self.thinking = False

self.eating = True

print("Philosofer {} is eating...".format(self.id))

time.sleep(random.randint(1, 10))

self.eating = False

self.return\_hashi(self.left)

self.return\_hashi(self.right)

print("Philosofer {} over your meal".format(self.id))

def take\_hashi(self, i):

""" Try to get the hashi nearby """

if self.n\_hashis >= 0 or self.n\_hashis < 2:

self.waiting = True

while hashis[i]:

print("Philosofer {} waiting the hashi {}".format(self.id, i))

time.sleep(random.randint(1, 5))

self.waiting = False

self.n\_hashis += 1

hashis[i] = True

print("Philosofer {} took the hashi {}".format(self.id, i))

def return\_hashi(self, i):

""" Return the hashi to the table """

hashis[i] = False

self.n\_hashis -= 1

print("Philosofer {} returned the hashi {}".format(self.id, i))

def mainDinner():

print("Philosofer's dinner is starting")

filosofos = []

for i in range(5):

print("Philosofer {} arrive".format(philosofers[i]))

a = Philosofer(i)

a.start()

filosofos.append(a)

while True:

# If all hashis are taken by the philosofers and all the philosofers are waiting other hashi

if all(hashis) and all(i.waiting for i in filosofos):

print("Deadlock ocurred")

break

print("Dining Philosophers is over")

#Dinners code end

def menu():

print("Process Synchronization\n")

# using the while loop to print menu list

while True:

print("MENU")

print("1. Producer Consumer Algorithm ")

print("2. Peterson Algorithm")

print("3. Dining Philosophers")

print("4. Exit")

users\_choice = int(input("\nEnter your Choice: "))

if users\_choice == 1:

print("\n=============Producer Consumer Algorithm=============\n")

mainPC()

elif users\_choice == 2:

print("\n=============Peterson Algorithm=============\n")

mainPet()

elif users\_choice == 3:

print("\n=============Dining Philosophers =============\n")

mainDinner()

elif users\_choice == 4:

raise SystemExit

if \_\_name\_\_ == "\_\_main\_\_":

menu()

# Outputs:

# 

# 

# 

# Reference Links:

# <https://www.geeksforgeeks.org/petersons-algorithm-in-process-synchronization/>

# <https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/>

# <https://www.educative.io/answers/what-is-the-producer-consumer-problem>