UNIVERSITY OF ESWATINI

CSC411 – MINI PROJET

MDUDUZI COMFORT – 202004888

MSIMISI MATSE – 202003895

THE PRODUCER CONSUMER PROBLEM

The Producer-Consumer Problem is a classic synchronization problem in computer science and concurrent programming. The problem involves two types of processes, producers, and consumers, which share a common, fixed-size buffer or queue.

The objective of the Producer-Consumer Problem is to ensure that producers can safely put items into the buffer (produce) and consumers can safely remove items from the buffer (consume) without running into issues like data inconsistency, race conditions, or deadlock.

The main challenges of the Producer-Consumer Problem are:

1. Synchronization: Producers and consumers must be synchronized to avoid conflicts when accessing the shared buffer.
2. Buffer management: The buffer has a fixed size, so the producer must wait when the buffer is full, and the consumer must wait when the buffer is empty.

The typical solution involves using synchronization mechanisms like semaphores, mutexes (locks), or condition variables to coordinate the activities of producers and consumers. These mechanisms ensure that the buffer is accessed safely, and that the producers and consumers are properly blocked or awakened when necessary.

The Producer-Consumer Problem is a fundamental concept in parallel computing and can be found in various real-world scenarios, such as task scheduling, data processing, and multi-threaded applications.

Implementing the Producer-Consumer Problem with the described requirements in the Mini Project document involves several components and considerations, including generating random student information, working with XML files, implementing concurrency and synchronization using semaphores, and calculating average marks and pass/fail status. Below is a Python implementation to demonstrate the basic structure of the solution:

[**PYTHON**]

```
import random
import xml.etree.ElementTree as ET
import os
import threading
import time

# Class representing student information
class ITStudent:
    def __init__(self, name, student_id, programme, courses, marks):
        self.name = name
        self.student_id = student_id
        self.programme = programme
        self.courses = courses
        self.marks = marks

# Buffer class to implement shared buffer
class Buffer:
    def __init__(self, max_size=10):
        self.max_size = max_size
        self.queue = []
        self.semaphore_producer = threading.Semaphore(max_size)
        self.semaphore_consumer = threading.Semaphore(0)
        self.mutex = threading.Lock()

    def add_to_buffer(self, item):
        self.semaphore_producer.acquire()
        self.mutex.acquire()
        self.queue.append(item)
        self.mutex.release()
        self.semaphore_consumer.release()

    def remove_from_buffer(self):
        self.semaphore_consumer.acquire()
        self.mutex.acquire()
        item = self.queue.pop(0)
        self.mutex.release()
        self.semaphore_producer.release()
        return item

# Function to generate random student information
def generate_student_info():
    name = "Student" + str(random.randint(1, 100))
    student_id = str(random.randint(10000000, 99999999))
    programme = "Programme" + str(random.randint(1, 5))
    num_courses = random.randint(3, 6)
    courses = [f"Course{str(i)}" for i in range(1, num_courses + 1)]
    marks = [random.randint(40, 100) for _ in range(num_courses)]
    return ITStudent(name, student_id, programme, courses, marks)

# Function to write student information to an XML file
def write_to_xml(student, file_path):
    root = ET.Element("Student")
    ET.SubElement(root, "Name").text = student.name
    ET.SubElement(root, "StudentID").text = student.student_id
    ET.SubElement(root, "Programme").text = student.programme
    courses_elem = ET.SubElement(root, "Courses")
    for course, mark in zip(student.courses, student.marks):
        course_elem = ET.SubElement(courses_elem, "Course")
        ET.SubElement(course_elem, "CourseName").text = course
        ET.SubElement(course_elem, "Mark").text = str(mark)

    tree = ET.ElementTree(root)
    tree.write(file_path)

# Function to read student information from an XML file
def read_from_xml(file_path):
    tree = ET.parse(file_path)
    root = tree.getroot()
    name = root.find("Name").text
    student_id = root.find("StudentID").text
    programme = root.find("Programme").text

    courses = []
    marks = []
    for course_elem in root.find("Courses"):
        courses.append(course_elem.find("CourseName").text)
        marks.append(int(course_elem.find("Mark").text))

    return ITStudent(name, student_id, programme, courses, marks)

# Function to calculate average mark and pass/fail status
def calculate_average(student):
    total_marks = sum(student.marks)
    average = total_marks / len(student.marks)
    return average, "Pass" if average >= 50 else "Fail"

# Producer function to generate student information and write to XML files
def producer(buffer):
    for i in range(1, 11):
        student_info = generate_student_info()
        file_path = f"student{i}.xml"
        write_to_xml(student_info, file_path)
        print(f"Produced: {file_path}")
        buffer.add_to_buffer(i)
        time.sleep(random.randint(1, 3))

# Consumer function to read student information from XML files and calculate results
def consumer(buffer):
    while True:
        file_num = buffer.remove_from_buffer()
        file_path = f"student{file_num}.xml"
        student_info = read_from_xml(file_path)
        os.remove(file_path)
        print(f"Consumed: {file_path}")
        average, pass_fail = calculate_average(student_info)
        print(f"Student Name: {student_info.name}")
        print(f"Student ID: {student_info.student_id}")
        print(f"Programme: {student_info.programme}")
        print("Courses and Marks:")
        for course, mark in zip(student_info.courses, student_info.marks):
            print(f"  {course}: {mark}")
        print(f"Average Mark: {average:.2f}")
        print(f"Pass/Fail: {pass_fail}")
        print("\n")
        time.sleep(random.randint(1, 3))

if __name__ == "__main__":
    buffer = Buffer(max_size=10)

    # Create producer and consumer threads
    producer_thread = threading.Thread(target=producer, args=(buffer,))
    consumer_thread = threading.Thread(target=consumer, args=(buffer,))

    # Start the threads
    producer_thread.start()
    consumer_thread.start()

    # Wait for both threads to finish
    producer_thread.join()
    consumer_thread.join()
```

This implementation demonstrates the Producer-Consumer Problem, generating random student information, writing to and reading from XML files, and using semaphores for synchronization.

Python implementation:

```
[Python]
import random
import xml.etree.ElementTree as ET
import os
import threading
import time
```

```
# Rest of the code from the previous


if __name__ == "__main__":
    buffer = Buffer(max_size=10)


    # Create producer and consumer threads
    producer_thread = threading.Thread(target=producer, args=(buffer,))
    consumer_thread = threading.Thread(target=consumer, args=(buffer,))


    # Start the threads
    producer_thread.start()
    consumer_thread.start()


    # Wait for both threads to finish
    producer_thread.join()
    consumer_thread.join()
```

Explanation:

The provided code defines a class ITStudent to hold student information, including name, ID, program, courses, and marks. The producer function generates random student information using the generate_student_info function and writes it to XML files using the write_to_xml function. The consumer function reads the XML files, calculates the average marks, and determines the pass/fail status using the read_from_xml and calculate_average functions, respectively.

The Buffer class implements a shared buffer with a maximum size of 10. It uses semaphores (semaphore_producer and semaphore_consumer) to control access to the buffer and a mutex (mutex) to ensure mutual exclusion when adding or removing elements from the buffer.

In the producer function, a loop is used to generate 10 students and their information. Each student's data is wrapped into an XML file (e.g., student1.xml, student2.xml, etc.), and the corresponding file number (1 to 10) is added to the buffer using the add_to_buffer method.

In the consumer function, an infinite loop is used to continuously consume student data from the buffer. The file number is obtained from the buffer using the remove_from_buffer method, and the corresponding XML file is read and processed. The student's information is then displayed on the screen, including average marks and pass/fail status.

The producer and consumer functions are run in separate threads (producer_thread and consumer_thread) using Python's threading module. This enables concurrent execution of the producer and consumer processes.

The program waits for both the producer and consumer threads to finish using the join method, ensuring that the entire process is completed.