Aim: Shell programming.

Description: This experiment introduces fundamental shell commands, including printing text, basic arithmetic, conditional checks, and using loops (for, while).

Code:

1. Print a simple message using the echo command

```
#!/bin/bash
# Script to print a simple message

echo "Hello, welcome to basic shell scripting!"
```

2. Basic arithmetic using shell variables

```
#!/bin/bash
# Script to perform basic arithmetic operations

num1=10
num2=5
sum=$((num1 + num2))
difference=$((num1 - num2))
product=$((num1 * num2))
quotient=$((num1 / num2))

echo "Sum: $sum"
echo "Difference: $difference"
echo "Product: $product"
```

3. Check if a number is even or odd using if-else

```
#!/bin/bash
# Script to check if a number is even or odd

read -p "Enter a number: " num

if [ $((num % 2)) -eq 0 ]
then
  echo "$num is even"
else
  echo "$num is odd"
fi
```

4. Display numbers 1 to 5 using a for loop

```
#!/bin/bash
# Script to print numbers from 1 to 5 using for loop

for i in {1..5}
do
  echo "Number: $i"
done
```

5. Prompt user for a password and validate using while loop

```
#!/bin/bash
# Script to validate password input

password="secret"

while true
do
  read -sp "Enter password: " input
  echo
  if [ "$input" == "$password" ]
  then
    echo "Access granted!"
    break
  else
    echo "Incorrect password, try again."
  fi
done
```

Output:

1.

Hello, welcome to basic shell scripting!

2.

Sum: 15

Difference: 5

Product: 50

Quotient: 2

3.

Input: Enter a number: 4

- **Output:** 4 is even

Input: Enter a number: 7

- **Output:** 7 is odd

4.

Number: 1

Number: 2

Number: 3

Number: 4

Number: 5

5.

Input: Enter password: wrong_password

- **Output:** Incorrect password, try again.

Input: Enter password: secret

- **Output:** Access granted!

Aim: To implement page replacement algorithms: (a) FIFO (First-In-First-Out) and (b) LRU (Least Recently Used).

Description: Page replacement algorithms are crucial in memory management for operating systems. When a page fault occurs, these algorithms determine which memory page should be replaced to make room for a new page.

- **FIFO (First-In-First-Out):** This algorithm replaces the oldest page in memory, based on the order of arrival.
- **LRU (Least Recently Used):** This algorithm replaces the page that has not been used for the longest time, based on usage history.

In this implementation, each algorithm takes a sequence of page references and a frame capacity, then calculates the number of page faults by determining which pages to replace when a fault occurs.

Codes:

```
# FIFO Page Replacement Algorithm
def fifo(pages, frame_capacity):
    frames = []
    page_faults = 0

    for page in pages:
        if page not in frames:
            if len(frames) == frame_capacity:
                frames.pop(0)  # Remove the oldest page
            frames.append(page)
            page_faults += 1  # Page fault occurs

    return page_faults

# LRU Page Replacement Algorithm
def lru(pages, frame_capacity):
    frames = []
    page_faults = 0
    recent_usage = {}

    for i, page in enumerate(pages):
        if page not in frames:
```

```
            if len(frames) == frame_capacity:
                # Find the least recently used page
                lru_page = min(recent_usage, key=recent_usage.get)
                frames.remove(lru_page)
                del recent_usage[lru_page]
            frames.append(page)
            page_faults += 1  # Page fault occurs

        # Update the recent usage time for the page
        recent_usage[page] = i

    return page_faults

# Test the algorithms
pages = [1, 3, 0, 3, 5, 6, 3, 1, 3, 0, 6]
frame_capacity = 3

print("FIFO Page Faults:", fifo(pages, frame_capacity))
print("LRU Page Faults:", lru(pages, frame_capacity))
```

Output:

FIFO Page Faults: 7
LRU Page Faults: 6

Aim: Programs to illustrate threads.

Description: Threads allow a program to execute multiple tasks concurrently, making it possible to perform time-consuming operations without blocking other parts of the program. Python provides a built-in threading module to work with threads, enabling multitasking. The two examples below demonstrate:

1. A program that creates two threads to print messages simultaneously.
2. A program that uses multiple threads to calculate the sum of numbers in different ranges concurrently.

Code:

Program 1: Printing Messages Simultaneously Using Threads

```
import threading
import time

# Function to print messages
def print_message(message, delay):
    for _ in range(5):
        time.sleep(delay)
        print(message)

# Creating two threads
thread1 = threading.Thread(target=print_message, args=("Hello from Thread 1", 1))
thread2 = threading.Thread(target=print_message, args=("Hello from Thread 2", 1.5))

# Starting threads
thread1.start()
thread2.start()

# Waiting for both threads to finish
thread1.join()
thread2.join()

print("Both threads have finished execution.")
```

Program 2: Summing Numbers in Ranges Using Threads

```
import threading

# Function to calculate sum of a range
def calculate_sum(start, end, result, index):
    result[index] = sum(range(start, end + 1))

# Range splits and result storage
ranges = [(1, 50), (51, 100), (101, 150)]
result = [0] * len(ranges)
threads = []

# Creating threads for each range
for i, (start, end) in enumerate(ranges):
    thread = threading.Thread(target=calculate_sum, args=(start, end, result, i))
    threads.append(thread)
    thread.start()

# Waiting for all threads to finish
for thread in threads:
    thread.join()

# Printing results
total_sum = sum(result)
print(f"Partial sums: {result}")
print(f"Total sum from 1 to 150: {total_sum}")
```

Output:

1.

Hello from Thread 1

Hello from Thread 2

Hello from Thread 1

Hello from Thread 1

Hello from Thread 2

Hello from Thread 1

Hello from Thread 2

Hello from Thread 1

Hello from Thread 2

Hello from Thread 1

Both threads have finished execution.

2.

Partial sums: [1275, 3775, 6275]

Total sum from 1 to 150: 11325

Aim: To implement classical synchronization problems in Python: (a) Dining Philosopher Problem and (b) Producer-Consumer Problem.

Description: Synchronization problems arise in concurrent programming where multiple processes or threads need to access shared resources. Here, we implement:

1. **Dining Philosopher Problem**: This problem involves philosophers who need to access limited shared resources (chopsticks) without causing deadlock.
2. **Producer-Consumer Problem**: This problem involves a producer that generates data and a consumer that uses it, requiring synchronization to prevent overproduction and underconsumption.

Code:

Program 1: Dining Philosopher Problem

```
import threading
import time

class Philosopher(threading.Thread):
    def __init__(self, name, left_fork, right_fork):
        threading.Thread.__init__(self)
        self.name = name
        self.left_fork = left_fork
        self.right_fork = right_fork

    def run(self):
        for _ in range(3):
            self.think()
            self.eat()

    def think(self):
        print(f"{self.name} is thinking.")
        time.sleep(1)

    def eat(self):
        # Acquire left fork first, then right fork
        with self.left_fork:
            with self.right_fork:
                print(f"{self.name} is eating.")
                time.sleep(1)
```

```
# Forks (one for each philosopher)
forks = [threading.Lock() for _ in range(5)]

# Creating philosophers and assigning forks
philosophers = [Philosopher(f"Philosopher {i+1}", forks[i], forks[(i+1) % 5]) for i in
range(5)]

# Starting philosopher threads
for philosopher in philosophers:
    philosopher.start()

# Waiting for all threads to complete
for philosopher in philosophers:
    philosopher.join()

print("Dining Philosopher Problem Simulation Complete.")
```

Program 2: Producer-Consumer Problem

```
import threading
import time
import queue

buffer = queue.Queue(maxsize=5)

# Producer Class
class Producer(threading.Thread):
    def run(self):
        for i in range(10):
            item = f"Item {i}"
            buffer.put(item)
            print(f"Producer produced: {item}")
            time.sleep(1)

# Consumer Class
class Consumer(threading.Thread):
    def run(self):
        for i in range(10):
            item = buffer.get()
            print(f"Consumer consumed: {item}")
            buffer.task_done()
            time.sleep(2)

# Create and start producer and consumer threads
producer = Producer()
consumer = Consumer()
```

```
producer.start()
consumer.start()

# Wait for both threads to finish
producer.join()
consumer.join()

print("Producer-Consumer Problem Simulation Complete.")
```

Output:

Program 1: Dining Philosopher Problem

Philosopher 1 is thinking.

Philosopher 2 is thinking.

Philosopher 3 is thinking.

Philosopher 4 is thinking.

Philosopher 5 is thinking.

Philosopher 4 is eating.

Philosopher 5 is eating.

...

Dining Philosopher Problem Simulation Complete.

Program 2: Producer-Consumer Problem

Producer produced: Item 0

Consumer consumed: Item 0

Producer produced: Item 1

Consumer consumed: Item 1

Producer produced: Item 3

Producer-Consumer Problem Simulation Complete.