Simulation and Comparative Analysis of Synchronization Mechanisms in Concurrent Resource Access

Arash Rahmani
Department of Software Engineering
University of Europe for Applied Sciences
Berlin, Germany
arash.rahmani@ue-germany.de

***Abstract*—Synchronization is a critical aspect of concurrent programming, ensuring safe access to shared resources in multi-threaded environments. This report presents the simulation and comparative analysis of synchronization mechanisms, specifically mutex locks and semaphores, in the context of concurrent access to printer and scanner resources. By simulating job execution with and without synchronization, we evaluate the effectiveness of these mechanisms in preventing race conditions and ensuring predictable system behavior. Our findings highlight the importance of proper synchronization and provide insights into the performance implications of different synchronization strategies.**

***Index Terms*—Synchronization, Concurrent Programming, Mutex Locks, Semaphores, Resource Sharing, Threading, Simulation**

I. Introduction

In multi-threaded systems, shared resources such as printers and scanners are accessed concurrently by multiple threads. Without proper synchronization, this concurrent access can lead to race conditions, data corruption, and unpredictable system behavior. Ensuring that only one thread accesses a shared resource at a time is essential for maintaining data integrity and system stability.

This project aims to simulate concurrent access to shared resources in a multi-threaded environment and analyze the impact of different synchronization mechanisms. We focus on two primary synchronization techniques: mutex locks and semaphores. By comparing job execution with and without synchronization, we demonstrate the importance of synchronization and evaluate the effectiveness of each mechanism.

The rest of the document is organized as follows: Section II provides an overview of the synchronization mechanisms used. Section III describes the methodology of the simulation. Section IV presents the simulation results and analysis. Section V concludes the report with key findings and suggestions for future work.

II. Synchronization Mechanisms Overview

A. Mutex Locks

A mutex (mutual exclusion) lock is a synchronization primitive used to protect shared resources from concurrent access. It ensures that only one thread can acquire the lock and access the resource at any given time. Mutex locks are simple to implement and are widely used for enforcing exclusive access in multi-threaded applications.

B. Semaphores

Semaphores are signaling mechanisms that control access to shared resources by maintaining a counter representing the number of available units. A binary semaphore (with a value of 0 or 1) functions similarly to a mutex lock. Counting semaphores allow multiple threads to access a limited number of resource instances concurrently. Semaphores provide more flexibility than mutexes and can be used for more complex synchronization scenarios.

C. Peterson's Algorithm

Peterson's algorithm is a classical software-based synchronization solution for two threads. It uses shared variables to ensure mutual exclusion without requiring special hardware instructions. However, its applicability is limited to two-thread scenarios and is not suitable for modern multi-threaded systems with more than two threads.

III. Methodology

A. Job Generator

To simulate concurrent access, we generated jobs for five users (`P1` to `P5`). Each user is assigned two jobs, which can be either a 'Print' or a 'Scan' job. Job attributes are randomly generated:

- **Job Type**: 'Print' or 'Scan'.
- **Number of Pages**: Randomly selected from predefined ranges for 'Short', 'Medium', or 'Large' jobs.
- **Arrival Time**: Incrementally increased with a random interval to simulate job submissions over time.

Jobs are stored in a job queue and processed in the order of their arrival times.

B. Unsynchronized Execution

In the unsynchronized execution, jobs are processed concurrently without any synchronization mechanisms. Each job is handled by a separate thread, invoking either the `printer` or `scanner` function based on the job type. Threads execute independently, leading to potential simultaneous access to the shared printer and scanner resources.

C. Synchronized Execution

1) Using Mutex Locks (First Attempt): In the first attempt, we implemented synchronization using mutex locks. Two mutex locks, `printer_lock` and `scanner_lock`, were used to control access to the printer and scanner resources, respectively. Threads attempting to access a resource acquire the corresponding lock before proceeding and release it after completing the job. This ensures that only one thread can use a resource at a time.

2) Using Semaphores (Second Attempt): In the second attempt, we used semaphores for synchronization. Binary semaphores (`printer_semaphore` and `scanner_semaphore`) were initialized with a value of 1. Similar to mutex locks, threads must acquire the semaphore before accessing the resource and release it afterward. This approach allows us to compare the effectiveness of semaphores against mutex locks in enforcing mutual exclusion.

D. Event Logging and Visualization

During the simulation, events such as job start, page processing, and job completion are logged with timestamps. This event log is used to create visual timelines of resource usage, helping us analyze overlapping access and the impact of synchronization.

We generated four visuals:

1) **Unsynchronized 1**: Timeline from the first attempt without synchronization.
2) **Synchronized 1**: Timeline from the first attempt using mutex locks.
3) **Unsynchronized 2**: Timeline from the second attempt without synchronization.
4) **Synchronized 2**: Timeline from the second attempt using semaphores.

These visuals are included in the results section to support our analysis.

IV. Simulation Results

A. Unsynchronized Execution

In the unsynchronized execution, multiple threads accessed the printer and scanner resources simultaneously. This led to overlapping usage, as evidenced by the event logs and visual timelines.

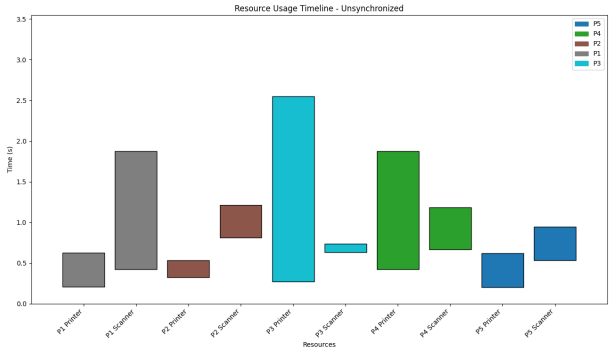

Fig. 1: Resource Usage Timeline - Unsynchronized Execution (First Attempt)

As shown in Figure 1, several users were printing or scanning at the same time. For example, `P1`, `P3`, and `P5` were printing concurrently. This simultaneous access can cause data corruption or unexpected behavior in real-world systems.

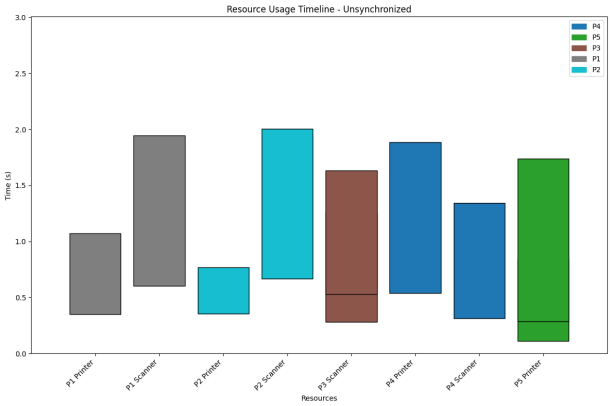

Fig. 2: Resource Usage Timeline - Unsynchronized Execution (Second Attempt)

Figure 2 further illustrates the overlapping access in the second attempt. Again, multiple users accessed the same resource simultaneously, confirming the need for synchronization.

B. Synchronized Execution

1) Using Mutex Locks (First Attempt): When mutex locks were implemented, overlapping access was eliminated. Threads had to acquire the lock before accessing the resource, ensuring exclusive access.

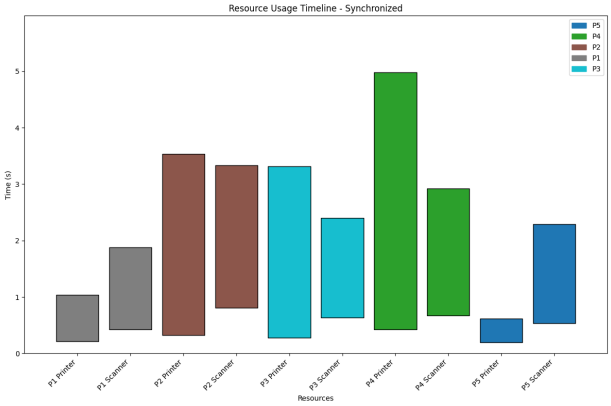

Fig. 3: Resource Usage Timeline - Synchronized Execution with Mutex Locks (First Attempt)

Figure 3 shows that no two users accessed the same resource simultaneously. However, serialization of access led to increased waiting times for some users.

2) Using Semaphores (Second Attempt): Using semaphores yielded similar results to mutex locks. Exclusive access was maintained, and overlapping usage was prevented.

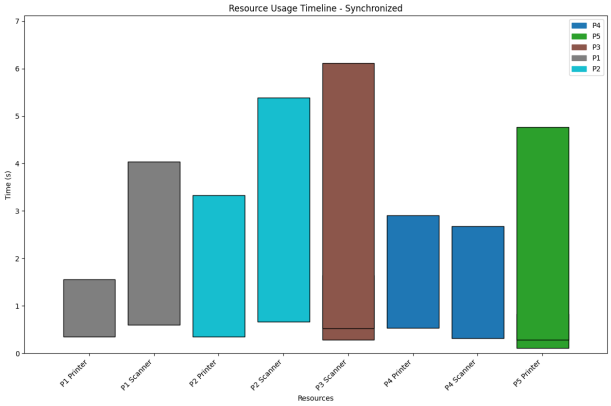

Fig. 4: Resource Usage Timeline - Synchronized Execution with Semaphores (Second Attempt)

As depicted in Figure 4, resource usage is orderly, with one user accessing the resource at a time. This confirms the effectiveness of semaphores in enforcing mutual exclusion.

C. Comparative Analysis

1) Resource Utilization: In both synchronized executions, the utilization of resources remained high, but access was serialized. While this prevents conflicts, it may lead to increased waiting times, as threads have to wait for the resource to become available.

2) Overlapping Access: The unsynchronized executions showed significant overlapping access to resources. This can cause race conditions and unpredictable behavior. Synchronization effectively eliminated overlapping access, ensuring safe and predictable operation.

3) Performance Metrics: We measured the total execution time in both scenarios. The synchronized executions took longer to complete due to the enforced waiting periods. However, the trade-off is acceptable given the importance of data integrity and system stability.

4) Effectiveness of Synchronization Mechanisms: Both mutex locks and semaphores effectively enforced mutual exclusion. No significant differences were observed between them in the context of this simulation. However, semaphores offer more flexibility and can be extended for scenarios requiring counting semaphores.

V. Conclusion

This study demonstrates the critical role of synchronization in concurrent programming. Unsynchronized access to shared resources leads to overlapping usage, which can cause race conditions and data corruption. Implementing synchronization mechanisms like mutex locks and semaphores effectively prevents these issues.

While synchronization may introduce some performance overhead due to increased waiting times, the benefits of ensuring data integrity and predictable system behavior outweigh the drawbacks. Choosing the appropriate synchronization mechanism depends on the specific requirements of the system.

A. Future Work

Future research could explore other synchronization methods, such as monitors or condition variables. Additionally, scaling the simulation to include more users or resources could provide further insights into the performance implications of synchronization in larger systems.

VI. Source Code

A. First Attempt Code (Using Mutex Locks)

```
import random
import threading
import time
import sys
import json

# Constants for testing
PAGE_PROCESSING_TIME = 0.1  # seconds

# Job Generator
users = ['P1', 'P2', 'P3', 'P4', 'P5']
job_types = ['Print', 'Scan']
job_length_ranges = {
    'Short': (1, 5),
    'Medium': (6, 15),
    'Large': (16, 50)
}
job_queue = []
for user in users:
    arrival_time = 0
    # Assign one 'Print' and one 'Scan' job to
        each user
    for job_type in job_types:
        length_category = random.choice(list(
            job_length_ranges.keys()))
```

```
        pages = random.randint(*job_length_ranges[length_category])
        arrival_interval = random.uniform(0.1, 0.5)
        arrival_time += arrival_interval
        job = {
            'user': user,
            'job_type': job_type,
            'pages': pages,
            'arrival_time': arrival_time
        }
        job_queue.append(job)

# Function to log events for analysis
event_log = []

def log_event(event):
    event_log.append(event)

# Part 2: Task Implementation
start_time = time.time()

# Shared resources without synchronization
def printer(job):
    # Log job start
    log_event({'timestamp': time.time()-start_time, 'resource': 'Printer', 'user': job['user'], 'action': 'start', 'execution': 'Unsynchronized'})
    for page in range(1, job['pages'] + 1):
        print(f"{time.time()-start_time:.2f}s: {job['user']} printing page {page} "
              f"of {job['pages']}")
        log_event({'timestamp': time.time()-start_time, 'resource': 'Printer', 'user': job['user'], 'action': 'printing', 'page': page, 'execution': 'Unsynchronized'})
        time.sleep(PAGE_PROCESSING_TIME)
    # Log job completion
    log_event({'timestamp': time.time()-start_time, 'resource': 'Printer', 'user': job['user'], 'action': 'end', 'execution': 'Unsynchronized'})

def scanner(job):
    # Log job start
    log_event({'timestamp': time.time()-start_time, 'resource': 'Scanner', 'user': job['user'], 'action': 'start', 'execution': 'Unsynchronized'})
    for page in range(1, job['pages'] + 1):
        print(f"{time.time()-start_time:.2f}s: {job['user']} scanning page {page} "
              f"of {job['pages']}")
        log_event({'timestamp': time.time()-start_time, 'resource': 'Scanner', 'user': job['user'], 'action': 'scanning', 'page': page, 'execution': 'Unsynchronized'})
        time.sleep(PAGE_PROCESSING_TIME)
    # Log job completion
    log_event({'timestamp': time.time()-start_time, 'resource': 'Scanner', 'user': job['user'], 'action': 'end', 'execution': 'Unsynchronized'})

# Synchronization using mutexes
printer_lock = threading.Lock()
scanner_lock = threading.Lock()

def synchronized_printer(job):
    log_event({'timestamp': time.time()-start_time, 'resource': 'Printer', 'user': job['user'], 'action': 'start', 'execution': 'Synchronized'})
    with printer_lock:
        for page in range(1, job['pages'] + 1):
            print(f"{time.time()-start_time:.2f}s: {job['user']} printing page {page} "
                  f"of {job['pages']}")
            log_event({'timestamp': time.time()-start_time, 'resource': 'Printer', 'user': job['user'], 'action': 'printing', 'page': page, 'execution': 'Synchronized'})
            time.sleep(PAGE_PROCESSING_TIME)
    log_event({'timestamp': time.time()-start_time, 'resource': 'Printer', 'user': job['user'], 'action': 'end', 'execution': 'Synchronized'})

def synchronized_scanner(job):
    log_event({'timestamp': time.time()-start_time, 'resource': 'Scanner', 'user': job['user'], 'action': 'start', 'execution': 'Synchronized'})
    with scanner_lock:
        for page in range(1, job['pages'] + 1):
            print(f"{time.time()-start_time:.2f}s: {job['user']} scanning page {page} "
                  f"of {job['pages']}")
            log_event({'timestamp': time.time()-start_time, 'resource': 'Scanner', 'user': job['user'], 'action': 'scanning', 'page': page, 'execution': 'Synchronized'})
            time.sleep(PAGE_PROCESSING_TIME)
    log_event({'timestamp': time.time()-start_time, 'resource': 'Scanner', 'user': job['user'], 'action': 'end', 'execution': '
```

```
                    Synchronized'})

# Execute jobs with synchronization
threads = []
for job in job_queue:
    if job['execution'] == 'Synchronized':
        if job['job_type'] == 'Print':
            t = threading.Thread(target=
                synchronized_printer, args=(
                job,))
        else:
            t = threading.Thread(target=
                synchronized_scanner, args=(
                job,))
    else:
        if job['job_type'] == 'Print':
            t = threading.Thread(target=
                printer, args=(job,))
        else:
            t = threading.Thread(target=
                scanner, args=(job,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

Listing 1: Simulation Code with Mutex Locks

B. Second Attempt Code (Using Semaphores)

```
import random
import threading
import time
import sys
import json
from threading import Semaphore

# Constants for testing
PAGE_PROCESSING_TIME = 0.1  # seconds

# Job Generator (Same as before)

# Function to log events for analysis (Same as
    before)

# Part 2: Task Implementation
start_time = time.time()

# Synchronization using semaphores
printer_semaphore = Semaphore(1)
scanner_semaphore = Semaphore(1)

def semaphore_printer(job):
    log_event({'timestamp': time.time()-
        start_time, 'resource': 'Printer',
               'user': job['user'], 'action':
                   'start', 'execution': '
                   Synchronized'})
    printer_semaphore.acquire()
    try:
        for page in range(1, job['pages'] + 1)
            :
            print(f"{time.time()-start_time:.2
                f}s: {job['user']} printing
                page {page} "
                  f"of {job['pages']}")
            log_event({'timestamp': time.time
                ()-start_time, 'resource': '
                Printer',
                       'user': job['user'], '
                           action': 'printing'
                           , 'page': page, '
                           execution': '
                           Synchronized'})
            time.sleep(PAGE_PROCESSING_TIME)
    finally:
        printer_semaphore.release()
    log_event({'timestamp': time.time()-
        start_time, 'resource': 'Printer',
               'user': job['user'], 'action':
                   'end', 'execution': '
                   Synchronized'})

def semaphore_scanner(job):
    log_event({'timestamp': time.time()-
        start_time, 'resource': 'Scanner',
               'user': job['user'], 'action':
                   'start', 'execution': '
                   Synchronized'})
    scanner_semaphore.acquire()
    try:
        for page in range(1, job['pages'] + 1)
            :
            print(f"{time.time()-start_time:.2
                f}s: {job['user']} scanning
                page {page} "
                  f"of {job['pages']}")
            log_event({'timestamp': time.time
                ()-start_time, 'resource': '
                Scanner',
                       'user': job['user'], '
                           action': 'scanning'
                           , 'page': page, '
                           execution': '
                           Synchronized'})
            time.sleep(PAGE_PROCESSING_TIME)
    finally:
        scanner_semaphore.release()
    log_event({'timestamp': time.time()-
        start_time, 'resource': 'Scanner',
               'user': job['user'], 'action':
                   'end', 'execution': '
                   Synchronized'})

# Execute jobs with synchronization (Same as
    before, but using semaphore functions)
```

Listing 2: Simulation Code with Semaphores

VII. Conclusion

Our simulation and analysis demonstrate the importance of synchronization in concurrent resource access. Both mutex locks and semaphores effectively prevent overlapping usage of shared resources, ensuring data integrity and predictable system behavior. While synchronization introduces some performance overhead, it is essential for the stability and reliability of multi-threaded systems.