CS307 - Operating Systems
PA3 Report
Synchronization Sightseeing: Managing a Tourist Attraction with Semaphores

Barış Pome
Student ID: 31311
Sabancı University

Fall 2024-2025

1 Introduction

In modern multi-threaded programming, synchronization mechanisms like semaphores, mutexes, barriers, and condition variables play a pivotal role in ensuring coordination and correctness. This report presents the implementation of a C++ class for managing a tourist attraction site, as part of Programming Assignment 3 (PA3) for the CS307 Operating Systems course. The primary objective of this assignment is to simulate a synchronized environment where visitors, represented as threads, interact with a shared resource under predefined constraints.

The simulation models visitors arriving at a landmark, waiting for a tour to begin, and departing either after completing the tour or leaving due to specific conditions. The implementation ensures synchronization among threads using semaphores, mutexes, and barriers, avoiding busy-waiting and guaranteeing correctness.

This document details the implementation of the 'Tour' class, focusing on its synchronization methods ('arrive' and 'leave'), the rationale for the choice of synchronization primitives, and their correctness. Additionally, the flow of the implemented methods is presented as pseudocode, followed by a formal discussion of how the synchronization mechanisms meet the specified criteria.

2 Program Overview

The program simulates the management of a tourist attraction site with synchronization between multiple threads representing visitors and, optionally, a guide. The shared resource is the attraction site, and synchronization is achieved using semaphores, mutexes, and barriers to ensure correct thread behavior and prevent race conditions.

The primary component of the program is the 'Tour' class, which encapsulates the logic for managing the attraction site. The following methods are implemented to coordinate visitor and guide activities:

- **Constructor:** The 'Tour' constructor initializes the shared state of the attraction site, including the maximum group size, presence of a guide, semaphores for entry and guide synchronization, mutexes for critical sections, and a barrier for coordinating the end of the tour.

- `arrive`**:** This method handles the arrival of visitors at the site. It uses semaphores to manage entry to the site and ensures that visitors do not enter during an ongoing tour. Mutexes are used to increment the

group size atomically, and conditional checks determine whether a tour can start or if visitors must wait.

- `leave`**:** This method manages the departure of visitors and the guide. Visitors leaving early or after a tour are synchronized using a barrier. The last visitor to leave notifies waiting visitors to enable the next group to enter. Semaphores and mutexes are employed to ensure orderly exits and state transitions.

- **Private Methods:** Several helper methods encapsulate specific actions:

  - `handle_group_complete`: Manages the transition to an active tour when the required group size is met.
  - `handle_early_leave`: Handles visitors leaving before a tour starts.
  - `handle_visitor_departure`: Synchronizes visitors leaving after a tour and resets the state for the next group.
  - `handle_guide_departure`: Ensures the guide announces the end of the tour before visitors can leave.
  - `reset_tour_state`: Resets the state to allow new visitors to enter after a tour ends.

The program uses thread-safe logging to print the status and behavior of threads at various stages. Synchronization primitives like barriers and semaphores are employed to prevent busy-waiting and ensure correctness, adhering to the assignment's requirements.

3 Detailed Explanation of the Code

This section provides an in-depth explanation of the 'Tour' class implementation, focusing on its methods and their interactions with helper functions and synchronization mechanisms.

3.1 Core Methods and Logic

3.1.1 `arrive` Method

The `arrive` method handles visitor arrivals, ensuring they wait if the site is full or if a tour is ongoing. Synchronization mechanisms include semaphores for controlling entry and mutexes for safe updates to shared state.

Pseudocode:

```
function arrive()
    log "Arrived at the location"
    wait(_entry_sem)  // Wait for entry permission
    lock(_state_mutex)
    if current_group_size == max_group_size:
        start tour and set guide if needed
    else:
        increment current_group_size
        log "Only n visitors inside, starting solo shots"
        signal(_entry_sem)
    unlock(_state_mutex)
```

Code:

```
1  void arrive()
2  {
3      pthread_t tid = pthread_self();
4      print_info("Arrived at the location.", tid);
5
6      sem_wait(&_entry_sem);
7      pthread_mutex_lock(&_state_mutex);
8
9      process_visitor_entry();
10
11     pthread_mutex_unlock(&_state_mutex);
12 }
```

The method relies on `process_visitor_entry` to increment the group size and decide whether the tour can start. If the group is incomplete, it uses the semaphore to allow the next visitor to enter.

Helper Function: process_visitor_entry

```
void process_visitor_entry()
{
    current_gropu_size++;
    if (current_gropu_size == max_group_size)
    {
        handle_group_complete(pthread_self());
    }
    else
    {
        handle_partial_group();
    }
}
```

Helper Function: handle_group_complete

```
void handle_group_complete(pthread_t tid)
{
    is_tour_active = true;
    if (has_guide)
    {
        guide_thread = tid;
    }
    print_info("There are enough visitors, the tour is
       starting.", tid);
}
```

Helper Function: handle_partial_group

```
void handle_partial_group()
{
    print_info("Only " + to_string(current_gropu_size) +
               " visitors inside, starting solo shots.",
                  pthread_self());
    sem_post(&_entry_sem);
}
```

3.1.2 `leave` Method

The `leave` method handles visitor departures, ensuring no visitor leaves while the tour is ongoing unless explicitly allowed. Synchronization mechanisms like barriers and semaphores ensure orderly behavior.

Pseudocode:

```
function leave()
    lock(_state_mutex)
    if not in_tour:
        log "Leaving early due to camera memory issue"
        decrement current_group_size
        if current_group_size == 0:
            signal(_entry_sem)
        unlock(_state_mutex)
        return
    unlock(_state_mutex)

    wait(_tour_barrier)

    lock(_state_mutex)
    if has_guide:
        handle_guide_departure()
    log "I am a visitor and I am leaving"
    decrement current_group_size
    if current_group_size == 0:
        log "All visitors have left, the new visitors can come"
        reset_tour_state()
    unlock(_state_mutex)
```

Code:

```
void leave()
{
    pthread_t tid = pthread_self();
    pthread_mutex_lock(&_state_mutex);
    bool in_tour = is_tour_active;

    if (!in_tour)
    {
        handle_early_leave();
        pthread_mutex_unlock(&_state_mutex);
        return;
    }
    pthread_mutex_unlock(&_state_mutex);

    pthread_barrier_wait(&_tour_barrier);

    pthread_mutex_lock(&_state_mutex);
    if (has_guide)
    {
        handle_guide_departure();
    }
    handle_visitor_departure();
    pthread_mutex_unlock(&_state_mutex);
}
```

Helper Function: handle_early_leave

```
void handle_early_leave()
{
    print_info("My␣camera␣ran␣out␣of␣memory␣while␣waiting,␣I␣
       am␣leaving.", pthread_self());
    current_gropu_size--;
    if (current_gropu_size == 0)
    {
        sem_post(&_entry_sem);
    }
}
```

Helper Function: handle_guide_departure

```
void handle_guide_departure()
{
    if (is_guide(pthread_self()))
    {
        print_info("Tour guide speaking, the tour is over.",
            pthread_self());
        sem_post(&_guide_sem);
    }
    else
    {
        pthread_mutex_unlock(&_state_mutex);
        sem_wait(&_guide_sem);
        sem_post(&_guide_sem);
        pthread_mutex_lock(&_state_mutex);
    }
}
```

Helper Function: handle_visitor_departure

```
void handle_visitor_departure()
{
    if (!is_guide(pthread_self()))
    {
        print_info("I am a visitor and I am leaving.",
            pthread_self());
    }
    current_gropu_size--;

    if (current_gropu_size == 0)
    {
        print_info("All visitors have left, the new visitors
            can come.", pthread_self());
        reset_tour_state();
    }
}
```

3.1.3 reset_tour_state

```
void reset_tour_state()
{
    is_tour_active = false;
    sem_post(&_entry_sem);
}
```

3.2 Synchronization Mechanisms

The program employs several synchronization mechanisms to ensure thread safety and proper coordination between visitor and guide threads. These mechanisms address challenges such as shared state consistency, thread scheduling, and the avoidance of busy-waiting.

- **Semaphores:**
  - `_entry_sem`: This semaphore controls visitor entry into the attraction site. It ensures that no new visitors can enter while a tour is in progress or if the site is full. When a visitor enters, the semaphore decrements its value, and when a tour ends, the last visitor resets the semaphore to allow the next group to enter.
  - `_guide_sem`: This semaphore facilitates synchronization between the guide and the visitors. It ensures that visitors wait for the guide to announce the end of the tour before leaving. The guide signals this semaphore upon completing their task, allowing the visitors to proceed with their departure.

  Semaphores are critical for maintaining the correct order of operations and preventing unwanted interleaving of thread execution.

- **Mutexes:**
  - `_state_mutex`: This mutex ensures atomic updates to shared variables like `current_group_size`, `is_tour_active`, and `has_guide`. By locking this mutex during updates, race conditions are avoided, ensuring thread-safe operations.
  - `_log_mutex`: This mutex synchronizes logging operations to avoid garbled or interleaved output from multiple threads. By locking the mutex during logging, the program ensures that each message is printed sequentially.

  Mutexes are essential for protecting critical sections and ensuring consistent updates to shared state variables.

- **Barriers:**
  - `_tour_barrier`: The barrier ensures that all visitors in a tour reach the same point before proceeding. Specifically, at the end of a tour, all threads wait at the barrier until the guide (or the last visitor, if there is no guide) announces the end of the tour.

Once all threads reach the barrier, they are released together to continue their respective operations.

Barriers provide a convenient way to synchronize multiple threads at a specific point, ensuring coordinated progression of the program.

Summary of Synchronization Roles:

- Semaphores manage thread entry and guide signaling, preventing improper access or premature execution.
- Mutexes protect critical sections and ensure safe concurrent updates to shared variables.
- Barriers synchronize threads at key stages, such as at the end of a tour, allowing them to proceed together without causing inconsistencies.

These mechanisms collectively ensure correctness, efficiency, and adherence to the assignment's constraints. They eliminate issues like race conditions, deadlocks, and busy-waiting, providing a robust implementation for managing the attraction site.

4 Conclusion

This project involved implementing a synchronized system for managing a tourist attraction site using C++ threads and synchronization primitives. The key objectives were to ensure correctness, efficiency, and adherence to the specified constraints, such as preventing busy-waiting and coordinating thread behavior.

The 'Tour' class encapsulates the core logic for managing visitor and guide threads. The main methods, `arrive` and `leave`, handle visitor entry and departure, using semaphores, mutexes, and barriers to ensure safe and orderly execution. Helper functions were incorporated to modularize the code, making it easier to maintain and debug.

The use of semaphores enabled controlled access to the shared resource, while mutexes ensured atomic updates to critical shared state variables. Barriers facilitated thread synchronization at key stages, such as the end of a tour. These mechanisms collectively addressed the challenges of race conditions, deadlocks, and thread scheduling.

The program logic ensures that visitors and guides interact seamlessly, adhering to the assignment's requirements. This implementation demonstrates the effective use of synchronization primitives in solving complex multi-threading problems, providing valuable insights into the practical application of operating systems concepts.

5 Source Code

Below is the complete source code of the `Tour` class:

5.1 Header File: `Tour.h`

```
#ifndef TOUR_H
#define TOUR_H

#include <pthread.h>
#include <semaphore.h>
#include <iostream>
#include <stdexcept>
#include <unistd.h>

using namespace std;

class Tour
{
private:
    int max_group_size;
    int current_gropu_size;
    bool has_guide;
    bool is_tour_active;
    pthread_barrier_t _tour_barrier;
    pthread_mutex_t _state_mutex;
    pthread_mutex_t _log_mutex;
    pthread_t guide_thread;
    sem_t _entry_sem;
    sem_t _guide_sem;

    void argument_check(int group_size, int has_guide);
    void print_info(const std::string &message, pthread_t tid
        );
    bool is_guide(pthread_t tid);
    void handle_group_complete(pthread_t tid);
    void reset_tour_state();
    void process_visitor_entry();
    void handle_partial_group();
    void handle_early_leave();
    void handle_guide_departure();
    void handle_visitor_departure();

public:
    void start();
    Tour(int group_size, int has_guide_param);
    void arrive();
    void leave();
```

```
42      ~Tour();
43  };
44
45  #endif // TOUR_H
```

5.2 Implementation: Tour.cpp

```
1   #include "Tour.h"
2
3   void Tour::argument_check(int group_size, int has_guide)
4   {
5       if (group_size <= 0 || (has_guide != 0 && has_guide != 1)
            )
6       {
7           throw invalid_argument("An␣error␣occurred.");
8       }
9   }
10
11  void Tour::print_info(const std::string &message, pthread_t
        tid)
12  {
13      pthread_mutex_lock(&_log_mutex);
14      cout << "Thread␣ID:␣" << tid << "␣|␣Status:␣" << message
            << endl;
15      pthread_mutex_unlock(&_log_mutex);
16  }
17
18  bool Tour::is_guide(pthread_t tid)
19  {
20      return has_guide && pthread_equal(tid, guide_thread);
21  }
22
23  void Tour::handle_group_complete(pthread_t tid)
24  {
25      is_tour_active = true;
26      if (has_guide)
27      {
28          guide_thread = tid;
29      }
30      print_info("There␣are␣enough␣visitors,␣the␣tour␣is␣
            starting.", tid);
31  }
32
33  void Tour::reset_tour_state()
34  {
35      is_tour_active = false;
36      sem_post(&_entry_sem);
37  }
```

```
38
39 void Tour::process_visitor_entry()
40 {
41     current_gropu_size++;
42     if (current_gropu_size == max_group_size)
43     {
44         handle_group_complete(pthread_self());
45     }
46     else
47     {
48         handle_partial_group();
49     }
50 }
51
52 void Tour::handle_partial_group()
53 {
54     print_info("Only " + to_string(current_gropu_size) +
55                " visitors inside, starting solo shots.",
                      pthread_self());
56     sem_post(&_entry_sem);
57 }
58
59 void Tour::handle_early_leave()
60 {
61     print_info("My camera ran out of memory while waiting, I
          am leaving.", pthread_self());
62     current_gropu_size--;
63     if (current_gropu_size == 0)
64     {
65         sem_post(&_entry_sem);
66     }
67 }
68
69 void Tour::handle_guide_departure()
70 {
71     if (is_guide(pthread_self()))
72     {
73         print_info("Tour guide speaking, the tour is over.",
              pthread_self());
74         sem_post(&_guide_sem);
75     }
76     else
77     {
78         pthread_mutex_unlock(&_state_mutex);
79         sem_wait(&_guide_sem);
80         sem_post(&_guide_sem);
81         pthread_mutex_lock(&_state_mutex);
82     }
83 }
```

```
84
85  void Tour::handle_visitor_departure()
86  {
87      if (!is_guide(pthread_self()))
88      {
89          print_info("I␣am␣a␣visitor␣and␣I␣am␣leaving.",
                pthread_self());
90      }
91      current_gropu_size--;
92
93      if (current_gropu_size == 0)
94      {
95          print_info("All␣visitors␣have␣left,␣the␣new␣visitors␣
                can␣come.", pthread_self());
96          reset_tour_state();
97      }
98  }
99
100 Tour::Tour(int group_size, int has_guide_param)
101 {
102     argument_check(group_size, has_guide_param);
103     max_group_size = group_size;
104     has_guide = has_guide_param;
105     current_gropu_size = 0;
106     is_tour_active = false;
107
108     if (has_guide)
109     {
110         max_group_size++;
111     }
112
113     pthread_barrier_init(&_tour_barrier, nullptr,
            max_group_size);
114     sem_init(&_entry_sem, 0, 1);
115     sem_init(&_guide_sem, 0, 0);
116
117     pthread_mutex_init(&_state_mutex, nullptr);
118     pthread_mutex_init(&_log_mutex, nullptr);
119 }
120
121 void Tour::arrive()
122 {
123     pthread_t tid = pthread_self();
124     print_info("Arrived␣at␣the␣location.", tid);
125
126     sem_wait(&_entry_sem);
127     pthread_mutex_lock(&_state_mutex);
128
129     process_visitor_entry();
```

```
130
131     pthread_mutex_unlock(&_state_mutex);
132 }
133
134 void Tour::leave()
135 {
136     pthread_t tid = pthread_self();
137     pthread_mutex_lock(&_state_mutex);
138     bool in_tour = is_tour_active;
139
140     if (!in_tour)
141     {
142         handle_early_leave();
143         pthread_mutex_unlock(&_state_mutex);
144         return;
145     }
146     pthread_mutex_unlock(&_state_mutex);
147
148     pthread_barrier_wait(&_tour_barrier);
149
150     pthread_mutex_lock(&_state_mutex);
151     if (has_guide)
152     {
153         handle_guide_departure();
154     }
155     handle_visitor_departure();
156     pthread_mutex_unlock(&_state_mutex);
157 }
158
159 Tour::~Tour()
160 {
161     pthread_barrier_destroy(&_tour_barrier);
162
163     pthread_mutex_destroy(&_state_mutex);
164     pthread_mutex_destroy(&_log_mutex);
165
166     sem_destroy(&_entry_sem);
167     sem_destroy(&_guide_sem);
168 }
```