Project Report

Single-Lane Bridge Problem

Course Title : Operating Systems Lab
Course Code : CSE-3632

Submitted To

Mohammad Zainal Abedin
Assistant Professor
Dept. of Computer Science and Engineering
IIUC

Submitted By - Group 3

Tehsim Fariha (C193207)
Raisa Ahmed (C193220)
Efter Jahan Ema (C193229)

INDEX

INTRODUCTION

1.1 INTRODUCTION

Why this synchronization project ?

Synchronization is an important part of modern computer applications that have multiple threads of execution within them. For example, an Internet browser tool such as Mozilla Firefox would have multiple threads to manage various user and system tasks simultaneously. Most operating systems and programming languages provide some primitives for synchronization. Java, for instance, provides several high-level constructs for synchronization. C, on the other hand, does not provide any constructs by itself. However, there are various library-based or OS-based solutions for synchronization that can be invoked from a C program.

POSIX Threads, usually referred to as pthreads, is an execution model that exists independently from a language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API.

This project is to build high-level constructs using some basic primitives supported by Pthreads/UNIX within C.

DESCRIPTION

1.2 DESCRIPTION

The Problem Statement: A single-lane bridge connects the two Vermont villages of North Tunbridge and South Tunbridge. Farmers in the two villages use this bridge to deliver their produce to the neighboring town. The bridge can become deadlocked if a northbound and a southbound farmer get on the bridge at the same time. (Vermont farmers are stubborn and are unable to back up.) Using semaphores and/or mutex locks, design an algorithm in pseudocode that prevents deadlock.

Implement your solution to using POSIX synchronization. In particular, represent northbound and southbound farmers as separate threads. Once a farmer is on the bridge, the associated thread will sleep for a random period of time, representing traveling across the bridge. Design your program so that you can create several threads representing the northbound and southbound farmers.

We have implemented a C++ program to help us understand a solution to this problem. The concepts used in the development of the project and a brief description about each of them are discussed in the following subsections.

1.2.1 Threads

A thread is a path of execution within a process. A process can contain multiple threads. The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces. Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources. But, like the process, a threadhas its own program counter (PC), registerset, and stack space.

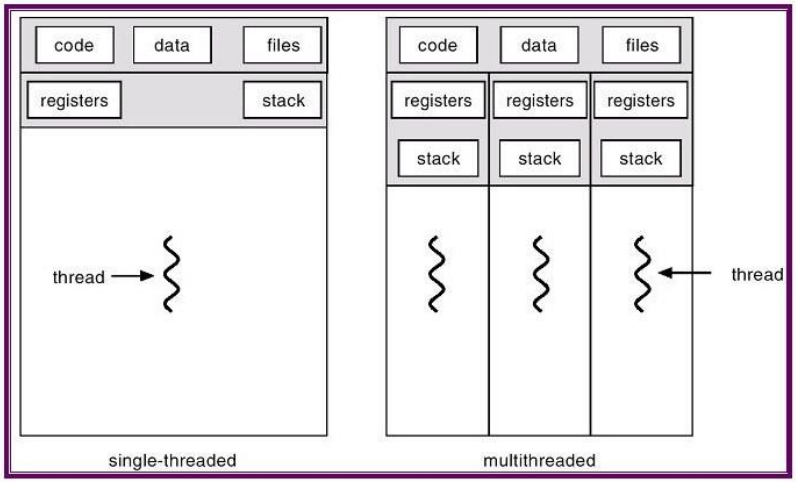

Figure 1.2.1.1
Single- and Multithreaded Processes

Advantages of threads over processes:

Responsiveness

Faster context switch

Effective utilization of multiprocessor system

Resource sharing

Communication

Enhanced throughput of the system

1.2.2 Race condition & Critical section problem

A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in the critical section differs according to the order in which the threads execute. Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Critical section is a code segment that can be accessed by only one process at a time.All the processes have to wait to execute in their critical sections. Critical section contains shared variables which need to be synchronized to maintain consistency of data variables.

1.2.3 Semaphore & Mutex lock

Semaphore is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment. Semaphores are of two types:

1. **Binary Semaphore** – This is also known as mutex lock. It can have only two values 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.
2. **Counting Semaphore** – Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

1.2.4 Deadlock & Starvation

Deadlock- occurs when each process holds a resource and waits for another resource held by any other process. In this no process holding one resource and waiting for another gets executed.

Starvation- is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. In starvation resources are continuously utilized by high priority processes. Problem of starvation can be resolved using Aging. Aging priority of long waiting processes is gradually increased.

In our problem starvation occurs when villagers enter the bridge continuously while the other villagers wait for the bridge to be free. So to solve this we give the other villagers a chance to cross by blocking after a fixed number of villagers cross continuously.

TOOLS &TECHNOLOGIES USED

1.3 TOOLS & TECHNOLOGIES USED

The tools and technologies used in the development of the project and a brief description about each of them are discussed in the following subsections.

1.3.1 C++

C++ is a multi-paradigm programming language encompassing statically typing, free form, compiled. It is regarded as an intermediate-level language, as it comprises both high-level and low-level language features. It was developed by Bjarne Stroustrup in 1979 at Bell Labs. C++ was originally named C with Classes, adding object-oriented features, such as classes, and other enhancements to the C programming language. The language was renamed C++ in 1983, involving the increment operator. It is an efficient compiler to native code, and its application domains include systems software, application software, device drivers, embedded software, high performance server and client applications, and entertainment software such as video games. C++ 5 is provided by several groups as both free and proprietary C++ compiler software, including the GNU Project, LLVM, Microsoft and Intel. C++ has greatly influenced many other popular programming languages, most notably C# and Java. The FAST decoding logic is written using C++14.

1.3.2 Visual Studio Code

Visual Studio Code is a free and open-source, source code editor developed by Microsoft for Windows, Linux and macOS. Visual Studio Code was announced on April 29, 2015 by Microsoft at the 2015 Build conference. On November 18, 2015, Visual Studio Code was released under the MIT License. On April 14, 2016, Visual Studio Code graduated the public preview stage and was released to the web. It includes support for debugging, embedded Git control, syntax highlighting, intelligent code completion, snippets, and code refactoring. It comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages (such as C++, C#, Python, PHP, Go) and runtimes (such as .NET and Unity).

1.3.3 Online GDB C Compiler

OnlineGDB is online IDE with **c compiler**. Quick and easy way to compile c program online. It supports gcc compiler for c

DESIGN PHASE

1.4 Design Phase

1.4.1 Algorithm

```
mutex lock;
semaphore northbound;
semaphore southbound;

northbound_farmer_thread()
{
   lock.acquire();
   northbound.increment();
   lock.release();
   northbound.wait();
   cross_bridge();
   northbound.decrement();
}

southbound_farmer_thread()
{
   lock.acquire();
   southbound.increment();
   lock.release();
   southbound.wait();
   cross_bridge();
   southbound.decrement();
}

bridge_thread()
{
   while (true)
   {
      lock.acquire();
      if (northbound.value() > 0 && southbound.value() == 0)
      {
```

```
            northbound.signal();
        }
        else if (northbound.value() == 0 && southbound.value() > 0)
        {
            southbound.signal();
        }
        lock.release();
    }
}
```

This algorithm uses a mutex lock to synchronize access to the semaphores, which represent the northbound and southbound farmers waiting to use the bridge. The northbound_farmer_thread and southbound_farmer_thread functions represent the threads for the northbound and southbound farmers, respectively. These threads acquire the mutex lock, increment the appropriate semaphore to indicate that a farmer is waiting to cross the bridge, and then release the lock. They then wait on the semaphore until they are signaled by the bridge_thread function.

The bridge_thread function runs in an infinite loop, continuously checking for farmers waiting to cross the bridge. It acquires the mutex lock, checks the values of the northbound and southbound semaphores, and then releases the lock. If there are northbound farmers waiting and no southbound farmers waiting, it signals the northbound semaphore to allow a northbound farmer to cross the bridge. If there are southbound farmers waiting and no northbound farmers waiting, it signals the southbound semaphore to allow a southbound farmer to cross the bridge. If there are farmers waiting in both directions, or no farmers waiting at all, it does nothing and continues to the next iteration of the loop.

This algorithm ensures that only one direction of farmers can cross the bridge at a time, preventing deadlock. It does not, however, address the

issue of starvation, where farmers in one direction may be prevented from using the bridge indefinitely. To address this issue, you may want to add additional logic to the bridge_thread function to alternately signal the northbound and southbound semaphores, allowing farmers in both directions to eventually use the bridge.

1.4.2 Checking if Starvation-free or not:

Starvation is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. In starvation resources are continuously utilized by high priority processes. Problem of starvation can be resolved using Aging. Aging priority of long waiting processes is gradually increased.

In our problem starvation occurs when villagers enter the bridge continuously while the other villagers wait for the bridge to be free. So to solve this we give the other villagers a chance to cross by blocking after a fixed number of villagers cross continuously.

So the problem became starvation free.

```
monitor bridge {
int num_waiting_north = 0;
int num_waiting_south = 0;
int on_bridge = 0;
condition ok_to_cross;
int prev = 0;
void enter_bridge_north() {
num_waiting_north++;
while (on_bridge ||(prev == 0 &&
num_waiting_south > 0))
ok_to_cross.wait();
on_bridge=1;
num_waiting_north--;
prev = 0;
}
void exit_bridge_north() {
on_bridge = 0;
ok_to_cross.broadcast();
}
void enter_bridge_south() {
num_waiting_south++;
while (on_bridge ||(prev == 1 &&
num_waiting_north > 0))
ok_to_cross.wait();
on_bridge=1;
num_waiting_south--;
prev = 1;
}
void exit_bridge_south() {
on_bridge = 0;
ok_to_cross.broadcast();
}
```

1.4.3 Implementation:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include<time.h>


// Semaphores to represent the northbound and southbound directions on the bridge
pthread_mutex_t northboundLock;
pthread_mutex_t southboundLock;

// Mutex to ensure that only one farmer is on the bridge at a time
pthread_mutex_t mutex;

// Function to simulate a northbound farmer crossing the bridge
void* northboundFarmer(void* arg) {
  pthread_mutex_lock(&northboundLock);
  pthread_mutex_lock(&mutex);
  printf("northbound farmer is on the bridge\n");
  // Sleep for a random period of time to simulate crossing the bridge
  sleep(rand() % 5 + 1);
  printf("northbound farmer has crossed the bridge\n");
  pthread_mutex_unlock(&mutex);
```

```
  pthread_mutex_unlock(&northboundLock);
  return NULL;
}

// Function to simulate a southbound farmer crossing the bridge
void* southboundFarmer(void* arg) {
  pthread_mutex_lock(&southboundLock);
  pthread_mutex_lock(&mutex);
  printf("southbound farmer is on the bridge\n");
  // Sleep for a random period of time to simulate crossing the bridge
  sleep(rand() % 5 + 1);
  printf("southbound farmer has crossed the bridge\n");
  pthread_mutex_unlock(&mutex);
  pthread_mutex_unlock(&southboundLock);
  return NULL;
}

int main(int argc, char* argv[]) {
  // Seed the random number generator
  srand(time(NULL));

  // Initialize the Semaphores and Mutex
  pthread_mutex_init(&northboundLock, NULL);
  pthread_mutex_init(&southboundLock, NULL);
  pthread_mutex_init(&mutex, NULL);

// Create several northbound and southbound farmer threads
```

```
for (int i = 0; i < 4; i++) {
pthread_t t1, t2;
pthread_create(&t1, NULL, northboundFarmer, NULL);
pthread_create(&t2, NULL, southboundFarmer, NULL);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
}

// Destroy the Semaphores and Mutex
pthread_mutex_destroy(&northboundLock);
pthread_mutex_destroy(&southboundLock);
pthread_mutex_destroy(&mutex);

return 0;
}
```

SYSTEM CALLS USED

1.5 SYSTEM CALLS USED

In computing, a system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to interact with the operating system. A computer program makes a system call when it makes a request to the operating system's kernel. System call provides the services of the operating system to the user programs via Application Program Interface(API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

In a Unix/Linux operating system, the C/C++ languages provide the POSIX thread (pthread) standard API (Application program Interface) for all thread related functions. It allows us to create multiple threads for concurrent process flow. It is most effective on multiprocessor or multi-core systems where threads can be implemented on a kernel level for achieving the speed of execution.

We must include the pthread.h header file at the beginning of the script to use all the functions of the pthreads library. To execute the c file, we have to use the -pthread or -lpthread in the command line while compiling the file.

```
g++ -pthread file.cpp or  g++ -lpthread file.cpp
```

1.5.1 Thread Creation

pthread_create : used to create a new thread

```
int pthread_create(pthread_t * thread, const pthread_attr_t * attr,
                   void * (*start_routine)(void *), void *arg);
```

Parameters:

- **thread:** pointer to an unsigned integer value that returns the thread id of the thread created.
- **attr:** pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
- **start_routine:** pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void *. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
- **arg:** pointer to void that contains the arguments to the function defined in the earlier argument

1.5.2 Thread Destroy

pthread_destroy: used to terminate a thread

void pthread_destroy(void *retval);

Parameters: This method accepts a mandatory parameter **retval** which is the pointer to an integer that stores the return status of the thread terminated. The scope of this variable must be global so that any thread waiting to join this thread may read the return status.

1.5.3 Thread join

pthread_join: used to wait for the

termination of a thread.

```
int pthread_join(pthread_t th, void
**thread_return);
```

Parameter: This method accept following

parameters:

- **th:** thread id of the thread for which the current thread waits.
- **thread_return:** pointer to the location where the exit status of

the thread mentionedin th is stored.

1.5.4 Mutex initialization

pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr) Creates a mutex, referenced by mutex, with attributes specified by attr. If attr is NULL, the default mutex attribute (NONRECURSIVE) is used.

Returned value

If successful, pthread_mutex_init() returns 0, and the state of the mutex becomes initialized andunlocked.

If unsuccessful, pthread_mutex_init() returns 1.

1.5.5 Mutex lock

int pthread_mutex_lock(pthread_mutex_t *mutex)

Locks a mutex object, which identifies a mutex. If the mutex is already locked by another thread, the thread waits for the mutex to become available. The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it. When the mutex has the attribute of recursive, the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then a count is incremented and no waiting thread is posted. The owning thread

must call pthread_mutex_unlock() the same number of times to decrement the count to zero.

Returned value

If successful, pthread_mutex_lock() returns 0, if unsuccessful, pthread_mutex_lock() returns 1.

1.5.6 Mutex Unlock

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Releases a mutex object. If one or more threads are waiting to lock the mutex, pthread_mutex_unlock() causes one of those threads to return from pthread_mutex_lock() with the mutex object acquired. If no threads are waiting for the mutex, the mutex unlocks with no current owner. When the mutex has the attribute of recursive the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then unblock will decrement the count and no waiting thread is posted to continue running with the lock. If the count is decremented to zero, then the mutex is released and if any thread is waiting for it is posted.

Returned value

If successful, pthread_mutex_unlock() returns 0 and returns 1 if unsuccessful.

WORKFLOW

1.6 WORKFLOW

1.6.1 Headers included:

<pthread.h>

<stdio.h>

<stdlib.h>

<unistd.h>

<time.h>

1.6.2 Variables:

pthread_mutex_t northboundLock;
pthread_mutex_t southboundLock;
pthread_mutex_t mutex;
pthread_t t1, t2;

1.6.3 Functions:

1.6.3.1 MAIN

- All variables are initialized
- villagers(threads) are created

1.6.3.2 THREAD FUNCTION

- Two thread functions are created – northboundFarmer and southboundFarmer

Stage 1: WAITING

- Lock the bridge

Stage 2: GOT PERMISSION

- Lock mutex
    - Increment the northboundLock
    - If (this thread is the 1st thread getting permission)
        - Then lock opposite bridge
    - Else
        - southboundLock made zero
- Unlock mutex

Stage 3: CROSSING

- Sleep for random time

Stage 4: CROSSED

- Lock mutex
    - Decrement northboundLock
    - If no one on bridge
        - Unlock opposite bridge
- Unlock mutex
- If opposite bridge is locked
    - Unlock our bridge and opposite bridge
- Exit thread

1.6.3.3 DISPLAY

- Displays the stage of each thread concurrently.

RESULT
&
DISCUSSION

1.7 RESULT AND DISCUSSION

To compile : g++ main.cpp –lpthread

Output

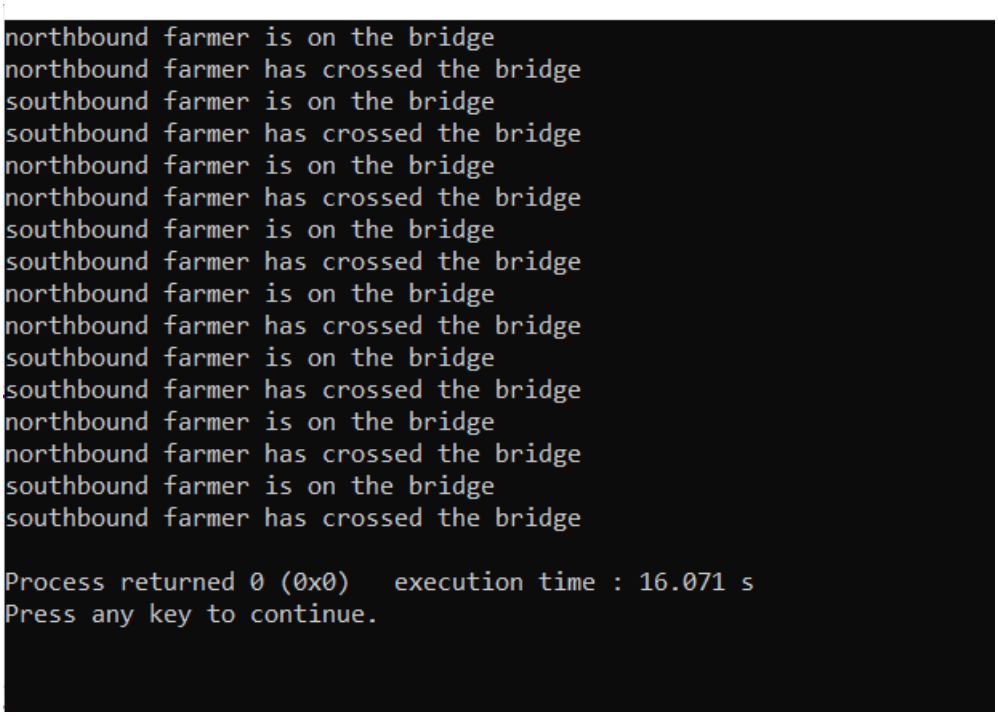

Figure 1.7.1

Figure 1.7.1 is a sample output of three villagers from one side and one villager from another village.

Using Threads and mutex to solve this type of problems are efficient and lessmemory consuming and are real time execution of the situation.

CONCLUSION

1.8 CONCLUSION

During the course of the project, the basics behind every concept were understood and

this knowledge was used in helping build the system. The understanding of the working of the different technologies helped in implementing the project very much. The project gave opportunities to learn new tools and technologies and implement them. It also provided training to face critical issues and resolve them immediately.

Future work includes making the system fail-proof that might come in later.

REFERENCE

1.9 REFERENCE

Books

1. Chapter 4 ,Silberschatz, Gavin, Gagne, “Operating System Concepts”, Wiley, 2011.
2. Chapter 5, Mutex locks, Silberschatz, Gavin, Gagne, “Operating System Concepts”, Wiley,2011.
3. Chapter 6, Thread Scheduling ,Silberschatz, Gavin, Gagne, “Operating System Concepts”,Wiley, 2011.
4. Chapter 7, Silberschatz, Gavin, Gagne, “Operating System Concepts”, Wiley, 2011.

Websites

1. http://www.cs.rpi.edu/academics/courses/fall04/os/c6/index.html
2. https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/
3. https://www.geeksforgeeks.org/thread-functions-in-c-c/
4. https://www.geeksforgeeks.org/multithreading-c-2/
5. https://www.geeksforgeeks.org/thread-in-operating-system/?ref=rp
6. http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html