CS 502 Operating Systems
Craig E. Wills
Assigned: Monday, March 11, 2024

WPI, Spring 2024
Project 5 (30 pts)
Due: Monday, April 1, 2024

Introduction

This assignment is intended to help you put into practice the concepts of thread coordination and resource sharing by building a conceptual maze that is traversed by a set of "rat" threads within the same process. You will use the facilities of the *pthreads()* library for thread and synchronization routines. Your program must be coded in C or C++.

Problem

The basic idea of this assignment is to create a number of rats, each a thread, that will traverse a maze of interconnected rooms. Figure 1 shows a sample three-room maze where all rooms have connectors to all other rooms and any room can be used by a rat to enter or leave the set of rooms.

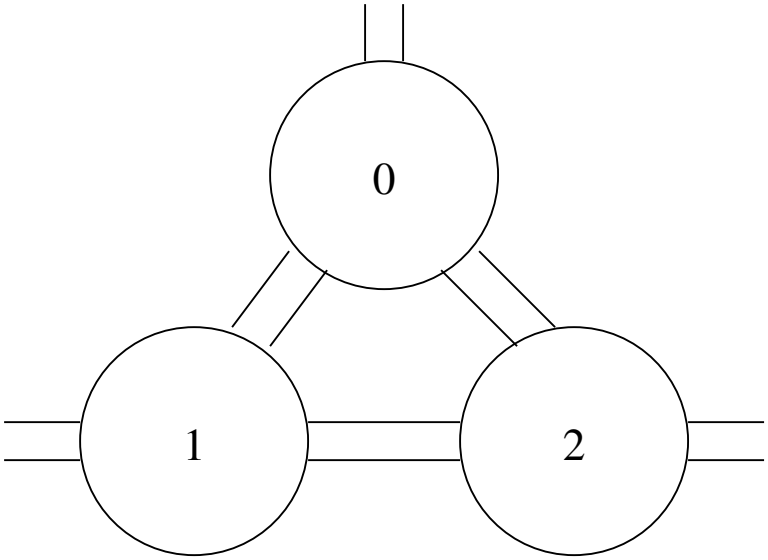

Figure 1: Sample Three-Room Maze of Rooms

Once a rat has entered one of the rooms it may not leave the set of rooms until it has visited all of the rooms. The maximum number of rats in the system at one time is controlled by a command line parameter, but this number can be no more than `MAXRATS`, which is defined as follows:

```
#define MAXRATS 5
```

Each rat thread will have its own numeric identifier with the first rat known as "rat 0", the second as "rat 1" and so forth. The main thread in your program will create all of the rat threads using the routine *pthread_create()*. The index of the thread is passed as the argument. In creating the threads, your program will need to remember the thread id stored in the first argument to *pthread_create()* for later use. You might call the routine executed by each child thread *Rat()* because it will act like a rat traversing the maze of rooms.

Each rat thread acts independently as it traverses the set of rooms. While all rooms are interconnected, they are of different sizes and take different amounts of time to traverse. The room configuration is specified by a configuration filed call `rooms`, which needs to be initially read by the main thread before it creates any child rat threads. A sample `rooms` file is given below with each line giving the capacity and traversal time (delay in seconds) for a room.

```
3 1
2 2
1 2
```

Each room is given a numeric identifier beginning with zero, so in the sample configuration Room 0 can hold three rats and it takes one second for a rat to traverse the room. Room 1 can hold up to two rats at a time with each rat taking two seconds to traverse the room. The final room, Room 2, can hold only one rat and it takes the rat two seconds to traverse this room. A valid `rooms` file will contain two positive integers on each line with no more than `MAXROOMS` lines. This value is defined as

```
#define MAXROOMS 8
```

If a file named "rooms" is missing in the current directory or is not a valid description then your program should report an error message and terminate the program.

Your main thread needs to create appropriate synchronization primitives (semaphores!) to ensure that no more rat threads are allowed into a room than the capacity of a room. If a rat thread tries to enter a room that is already full of rats then the rat thread must wait for a rat thread to leave the room before it can enter. There is no limit on the number of rat threads that can wait in the connector to enter a room.

Each room also contains a visitors book so each rat leaving a room enters its id, its time of entry and its time of departure as the next log entry. You can use the following structure for each entry in the visitors book.

```
struct vbentry {
    int iRat;    /* rat identifier */
    int tEntry;  /* time of entry into room */
    int tDep;    /* time of departure from room */
};
```

The current time (in seconds) can by found using the Unix library call *time()*, found in Section 3 of the man pages, which maintains a running count of the time in seconds since January 1, 1970. Your program should record the current time in a global variable when it starts execution so that all time is recorded relative to the beginning of your program execution.

Because each rat thread must sign the visitors book of each room, these visitors books must be shared amongst all rat threads—hence must be global. The visitors books for all rooms are represented by the `RoomVB` two-dimensional array, which can be declared as follows:

```
struct vbentry RoomVB[MAXROOMS][MAXRATS];  /* array of room visitors books */
```

To access the `iRat` field of the *jth* entry for the *ith* room use the expression `RoomVB[i][j].iRat`.

In addition, you will need a global variable `VisitorCount[MAXROOMS]` to keep track of the total number of visitors for each room. Then use `VisitorCount[i]` to find the visitor count (thus far) for the *ith* room. Note: you will need to use this variable to determine the next location in the `RoomVB` array to enter a visitor. Beware that more than one rat thread may be trying to log information to the visitors book at a time.

Similarly, semaphores should be created to control access to each room. Semaphores should be created using the routine *sem_init()*. These semaphores should also be created by the main thread before it creates the other threads.

To handle access to each room by a rat thread you must write two procedures: *EnterRoom()* and *LeaveRoom()*. These procedures have the following interfaces

```
EnterRoom(int iRat, int iRoom)
/* iRat - id of rat entering the room */
/* iRoom - id of room being entered */

LeaveRoom(int iRat, int iRoom, int tEnter)
/* iRat - id of rat leaving the room */
/* iRoom - id of room being left */
/* tEnter - time Rat entered room */
```

When a rat thread tries to enter a room it should block until room is available in the room for it to enter. The `LeaveRoom()` routine should enter information about the rat thread in the visitors book for the room.

The time to traverse a room once successfully entered is simply a delay that your rat thread must incur. Your thread should delay the given number of seconds for a room using the Unix library call *sleep()*.

If you write your program in C++, you could think of having "rat" and/or "room" classes with appropriate methods. If you take this approach then your parameters may vary from those given in this project description.

Basic Objective

You are to create a program to control the traversal of rat threads through the given maze of rooms. The name of your executable should be *maze* and it should take two command-line arguments. The first is the number of rats in the maze. If more than `MAXRATS` is specified then your program should report an error and terminate. The second argument is the algorithm your rats should use in traversing the maze. The argument can either be "i" for in-order or "d" for distributed.

The in-order algorithm causes all rat threads to take the same order through the maze beginning with room 0, followed by room 1 and ending with the last room in the room description file. The distributed algorithm also has the rats moving through the rooms in ascending order, but the starting rooms for the rats are distributed corresponding to their own identifier. Thus rat 2 first enters room 2 and eventually leaves the maze from room 1. Rats should loop around from the highest numbered room to room 0 if they have not yet completed the maze. If there are more rats than rooms for the distributed algorithm then take the rat's identifier modulo the number of rooms to determine the initial room number.

After your main thread has created all of the semaphores and rat threads it waits for all of the rat threads to terminate using *pthread_join()*. As each rat finishes the maze it should print its completion time in seconds (relative to the start of the program). Each rat should also add its time to a global total time variable. After all rats have completed, the main thread should print (one line per room), the capacity and delay along with the visitor book contents for each room. The last printed line should be the total traversal time for all rats along with the ideal traversal time, which is simply the total of all room delays multiplied by the number of rats. This line shows the relative "goodness" of the algorithm of all rats traversing the maze.

As a starting point it is suggested that you begin your testing with a single room then increase the number of rooms once your program is working properly with one room.

Sample Execution

Using the sample room description in this handout, here is a sample execution of the program. The order of your rat threads may vary.

```
% ./maze 3 i
Rat 0 completed maze in 5 seconds.
Rat 1 completed maze in 7 seconds.
Rat 2 completed maze in 9 seconds.
Room 0 [3 1]: 0 0 1; 1 0 1; 2 0 1;
Room 1 [2 2]: 0 1 3; 1 1 3; 2 3 5;
Room 2 [1 2]: 0 3 5; 1 5 7; 2 7 9;
Total traversal time: 21 seconds, compared to ideal time: 15 seconds.
```

Additional Work

Satisfactory completion of the basic objective of this assignment is worth 26 of the 30 points. For two additional points, you need to implement another send primitive *TryToEnterRoom()* with the same arguments as *EnterRoom()* except that it should not block if the room is already full. In this case *TryToEnterRoom()* should return immediately with a return code of -1. It should return a value of zero if the rat is able to enter the room. In building this routine, be aware of creating race conditions if multiple rat threads try to call this same routine for the same room. You may need to investigate additional routines, such as *sem_getvalue()* (read carefully what it returns), available for semaphores.

The final two points for the assignment can be earned by demonstrating the use of the *TryToEnterRoom()* routine in this assignment. To do this you need to implement a non-blocking algorithm (denoted by "n" on the command line), which provides at least as good total traversal time as the in-order and distributed algorithms. Your rat threads may take any order through the maze, but must visit all rooms and use the *TryToEnterRoom()* routine. Your non-blocking algorithm cannot use prior knowledge of the room descriptions in making its decisions. The best algorithm will yield a total traversal time as close as possible to the ideal for all room configurations. Describe the algorithm you use in comments with the code.

Make

A useful program for maintaining large programs that have been broken into many software modules is *make*. The program uses a file, usually named `Makefile`, that resides in the same directory as the source code. This file describes how to "make" a number of targets specified. The following is a portion of the `Makefile` linked on the course Web page. Typing "`make maze`" causes the executable file *maze* to be created from `maze.cpp`.

You should copy the sample `Makefile` and modify it for your project. Note the sample `Makefile` is used to compile a C++ version of the sample program. You will need to modify if you are using C.

As a matter of explanation the first three lines below simply define and give values to *make* variables. The remaining text describes how to make the target "maze". WARNING: The operation line(s) for a target MUST begin with a TAB (do not use spaces). See the

man page for *make*, *g++* and *gcc* for additional information.

```
LIB=-lpthread
CC=g++

maze: maze.cpp
        $(PP) maze.cpp -o maze $(LIB)
```

Submission of Project

Please compress all the files together as a single .zip archive for submission. In addition to source files and makefile that compiles your *maze* code, you should include a `typescript` file (created using *script* program) showing sample execution of your program on various test inputs.

Please upload your .zip archive via InstructAssist with the project name of *proj5*.