



THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC 3150

OPERATING SYSTEMS

---

## Assignment 2

---

*Author:*  
Shi Wenlan

*Student Number:*  
119010265

October 24, 2021

## Contents

<b>1</b>	<b>Task 1</b>	<b>2</b>
1.1	How did I design the program . . . . .	2
1.2	The environment of running my program . . . . .	3
1.3	The steps to execute my program . . . . .	3
1.4	Screenshots of my program output . . . . .	4
1.5	What did I learn from the task . . . . .	5
<b>2</b>	<b>Bonus</b>	<b>5</b>
2.1	How did I design the program . . . . .	5
2.2	The environment of running my program . . . . .	5
2.3	The steps to execute my program . . . . .	6
2.4	Screenshots of my program output . . . . .	6
2.5	What did I learn from the task . . . . .	6

# 1 Task 1

## 1.1 How did I design the program

**Core design idea:** 1. I used a total of two threads. One thread controls the movement of logs and the movement of frog along with the logs (*logs\_move*), and the other thread controls the movement of frog when it accepts keyboard input (*frog\_move*). I used *pthread\_create()* function to create multiple threads. 2. There are two pieces of data shared by the two threads, one is the map, and the other is the location information of the frog. In order to ensure that the reading and writing of shared data do not conflict, I use the *pthread\_mutex\_init()* function to construct two locks, *mutex* is used to protect the map, and *mutex2* is used to protect the positional parameters of frog. 3. Only the *printer()* function for printing results and the *logs\_move()* function for controlling the movement of logs are the only ones with the permission to modify the map. I use the same mutex to lock the the part of the code that modifies the map in these two functions. 4. The position of frogs is determined by *frog.x* and *frog.y*. I wrote a *mover()* function to modify these two parameters, and ensure that both threads can only modify the positional parameters of frogs through *mover()*. Then I use the *pthread\_mutex\_lock()* function and *pthread\_mutex\_unlock()* to lock the function body of *mover()* to ensure that only one thread can modify the positional parameters of frog at the same time. 5. Every time the logs or the frog moves, the *printer()* would be called to make the movement immediately visible.

**For moving of logs:** I set two local array variables *start*[9] and *length*[9] in *logs\_move()* to store the starting location of each log and the length of the logs. The starting position of each log is randomly determined by the *rand()* function. Whenever the logs move, data is read from these two arrays to determine where the head and end of the logs are(head stored in start array, end is the remainder of (head + length) to (COL - 1)), so that one side is changed to space and the other side is added = to achieve the visual effect of the movement. Every time the logs move, the corresponding start data will also be changed synchronously.

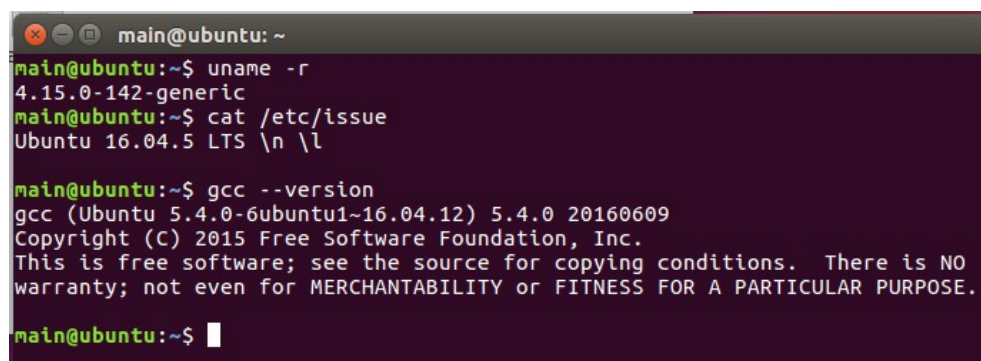
**For printing of frog position:** Because the change of the absolute position of the frog has two sources (movement with the log and keyboard input by the player), the operation of repeatedly modifying the position of 0 in the map becomes very cumbersome (I need to change the original position of the frog to = or space, where I need to consider many branch conditions, and then change the new position of frog to 0). So I changed the method. I chose to change only *frog.x* and *frog.y* in *logs\_move()* and *frog\_move()*, until every time *printer()* was called to print the result, I store the content in the position (*frog.x*, *frog.y*) of map in *tmp*, then change

the content to 0, and then change the content back to tmp after printing. This avoids the use of cumbersome branch judgments to determine what content should be used to fill the original position occupied by frog after frog moves.

**For judging the state of the game :** I used 3 global variables *isQuit*, *isWin*, and *isLose* to control the game state. Both threads have a while loop conditioned on these three variables, and will continue to execute only when the three variables are all 0. When q on the keyboard is pressed, *isQuit* will be set to 1. When frog.x equals 0, *isWin* will be set to 1. When frog.y is less than 0 or frog.y is larger than 49 or the content in the position (frog.x, frog.y) of map is space(not on the logs), *isLose* will be set to 1.

## 1.2 The environment of running my program

The code running environment is shown in the following figure:



```
main@ubuntu: ~  
main@ubuntu:~$ uname -r  
4.15.0-142-generic  
main@ubuntu:~$ cat /etc/issue  
Ubuntu 16.04.5 LTS \n \l  
  
main@ubuntu:~$ gcc --version  
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609  
Copyright (C) 2015 Free Software Foundation, Inc.  
This is free software; see the source for copying conditions. There is NO  
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  
  
main@ubuntu:~$
```

Figure 1: The environment of running my program

## 1.3 The steps to execute my program

Enter *source* folder in terminal and execute the following instructions in sequence so start the game.

```
g++ hw2.cpp -lpthread  
./a.out
```

Figure 2: Instuctions

In the game, you can use following keys to operate.

```
W: UP
S: DOWN
A: LEFT
D: RIGHT
Q: QUIT the game
K: SPEED DOWN
L: SPEED UP
```

Figure 3: How to play

## 1.4 Screenshots of my program output

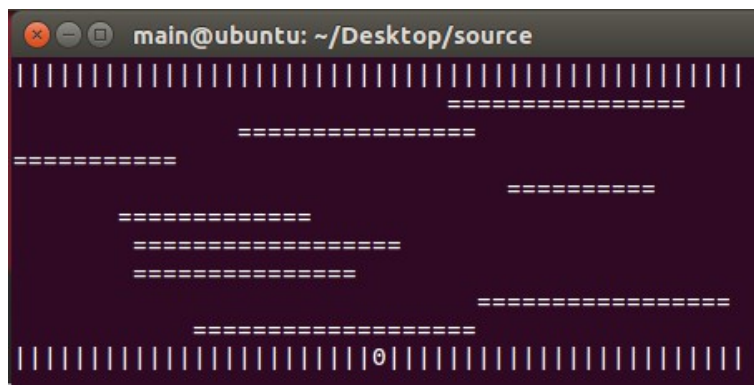


Figure 4: When you start the game(without side bar ver.)

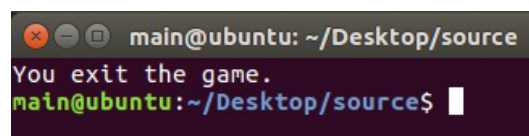


Figure 5: When you quit

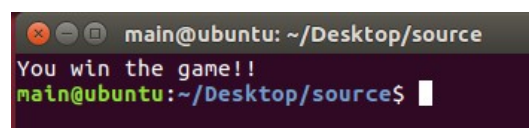


Figure 6: When you win

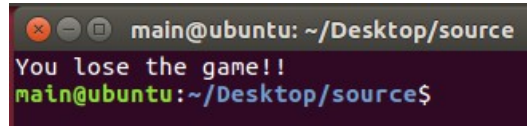


Figure 7: When you lose

## 1.5 What did I learn from the task

I preliminarily mastered the use of thread creation and operation. I learned how to use mutexes and signals to control multiple threads. I can distinguish the similarities and differences between threads and processes.

## 2 Bonus

### 2.1 How did I design the program

I aim to complete two challenges, one is to achieve random generation of logs length, and the other is to design a slide bar that adjusts the speed of floating logs.

**For generating logs length in random:** When initializing the logs, I used `rand()` to randomly generate a random number between 10 and 20 as the length of the logs.

**For designing the slide bar:** I set a variable *speed* as the parameter passed in `usleep()` to control the refresh interval of logs movement, thereby indirectly changing the movement speed of logs. I modified the content of `frog_move()` to additionally react to the input of K and L. When the player presses the K key, the speed parameter increases, the log refresh interval increases, and the log movement speed decreases. When the player presses the L key, the speed parameter decreases, the logs refresh interval decreases, and the logs movement speed increases. In addition, when printing the map, I added the printing of the slide bar to visualize the speed change. The slide bar has a total of 11 speeds. The initial speed is 5. Press K once to decrease by 1, 0 to cover the bottom, press L to increase by 1 and 10 to cover the top.

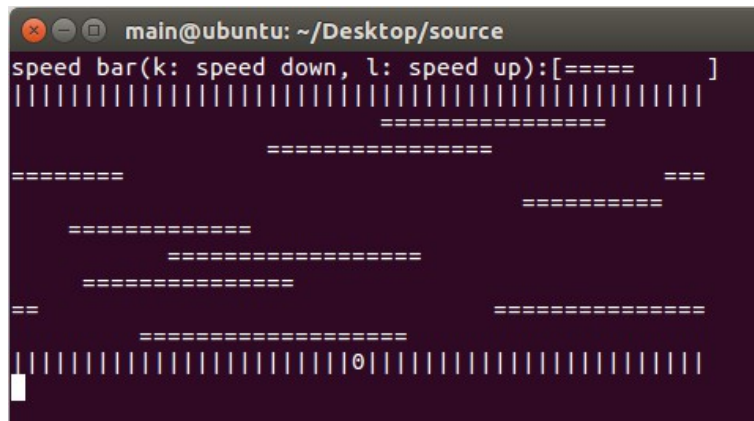
### 2.2 The environment of running my program

The code running environment is the same as task 1, see Figure 1.

## 2.3 The steps to execute my program

The steps to execute my program is the same as task 1, see Figure 2 and 3.

## 2.4 Screenshots of my program output



**Figure 8:** When you start the game (bonus)

## 2.5 What did I learn from the task

I have improved my proficiency in the use of various functions of multithreading.