SI411 Lab, Fall 2025 (AY26) Name _____George Prielipp_____

Synchronization and Deadlock Lab

Summary: In this lab, you'll practice using semaphores to implement mutual exclusion and synchronization on POSIX threads in C.

Learning Objectives: Understand the conditions that lead to deadlock. Understand the requirements of mutual exclusion. Implement semaphores to solve an example problem.

Instructions:

- This lab may be completed with a Linux lab machine, or with WSL, or with a VM.
- Download this file, and enter each answer in the box or space provided
  - All answers must be typed.
- Use your own words. Do *not* copy and paste text from elsewhere. Ok to paste program output.
- Deliverables
  - **Hard-copy of this worksheet,** due at start of class on the due date. **(**staple required for full credit**)**
  - Submit `player.c` and `player_detect.c` to the online submission system.

Requirements: Submit `player.c` and `player_detect.c` to the online submission system. Also submit this sheet with competed answers.

Lab Total: 20 points (9 points for Part 1, and 11 points for Part 2)

Note: your OS will need to be able to run multiple threads! If you are running the lab on a Linux VM and have assigned it only one CPU 'core', you will probably need to increase its allocation to 2 or more cores (the more, the better) to observe the deadlock behavior. Ask your instructor if you don't know how to do this!

Overview

The baseball playoffs are jeopardy! As a result of some movers boxing up all the wrong stuff, the ball players don't have any baseballs, bats, or gloves. Your job is to help them acquire the resources they need. Once a ballplayer has a ball, bat, and glove, he can play ball.

There is contention for resources – the number of players exceeds the number of balls, bats, and gloves. The players will have to take turns, since not everyone can play at the same time. Once a player thread is created, it randomly chooses the order in which to acquire the needed resources (ball, bat, glove). Each player only needs one of each resource type.

Part 1: Eliminating Deadlocks

The `player.c` file implements all of the above functionality, including data structures to track resource allocation. Unfortunately, the code deadlocks. Your tasks are as follows:

(1) Output. Verify that the existing code often deadlocks by compiling it and running it a few times. It might not deadlock 100% of the time, but it should more often than not:

- Note: the variables created, near the top, under the heading "Deadlock detection data structures" don't matter for this part – you'll use them in Part 2.
- Find `main()` and read through it. Identify how you will know if the program successfully completes? (and what might deadlock look like instead?)
- Properly compile and link player.c
- Run the program until you see the program deadlock. Copy a sample output, showing the deadlock, below. **(see below if it doesn't deadlock for you**)

```
player 1 got bat 0
player 1 got glove 2
player 1 got ball 0
player 4 got bat 2
player 0 got glove 0
player 8 got glove 5
player 7 got bat 3
player 2 got bat 1
player 5 got glove 3
player 3 got glove 1
player 10 got bat 5
player 9 got bat 4
player 8 got ball 1
player 5 got ball 2
player 4 got glove 6
player 6 got glove 4
player 8 got bat 6
player 4 got ball 3
player 5 got bat 7
player 12 got ball 6
player 9 got glove 7
player 0 got ball 4
player 6 got ball 5
player 13 got ball 7
player 1 released glove 2
player 1 released bat 0
player 1 released ball 0
player 1 ..... done
player 11 got glove 2
player 14 got ball 0
player 12 got bat 0
player 5 released glove 3
player 4 released glove 6
player 4 released bat 2
player 5 released bat 7
player 5 released ball 2
player 4 released ball 3
player 4 ..... done
```

player 5 ..... done
player 2 got glove 6
player 8 released glove 5
player 19 got bat 2
player 12 got glove 3
player 15 got glove 5
player 17 got ball 3
player 8 released bat 6
player 8 released ball 1
player 8 ..... done
player 16 got ball 2
player 6 got bat 6
player 0 got bat 7
player 9 got ball 1
player 12 released glove 3
player 12 released bat 0
player 12 released ball 6
player 12 ..... done
player 7 got glove 3
player 7 got ball 6
player 13 got bat 0
player 6 released glove 4
player 6 released bat 6
player 0 released glove 0
player 13 got glove 4
player 10 got glove 0
player 22 got bat 6
player 21 got ball 5
player 0 released bat 7
player 0 released ball 4
player 0 ..... done
player 6 released ball 5
player 6 ..... done
player 9 released glove 7
player 9 released bat 4
player 14 got bat 7
player 17 got bat 4
player 3 got ball 1
player 9 released ball 1
player 9 ..... done
player 23 got ball 4
player 19 got glove 7
player 7 released glove 3
player 7 released bat 3
player 7 released ball 6
player 7 ..... done
player 22 got glove 3
player 11 got ball 6
player 16 got bat 3

```
player 13 released glove 4
player 13 released bat 0
player 18 got glove 4
player 21 got bat 0
player 13 released ball 7
player 13 ..... done
player 15 got ball 7
```

If the program does NOT seem to deadlock very often for you (will depend on the machine you use to run), make some change to one of these statements:

```
#define NUM_PLAYERS     24
#define NUM_BALLS       8
#define NUM_BATS        8
#define NUM_GLOVES      8
#define SHORT_WAIT      .01
```

… to make it deadlock. What did you change to make it deadlock more often? (briefly explain)

_____

(1) Analysis. Observe that the players choose randomly what order to grab resources. Let's investigate the importance of that.

- Create a version that does NOT use randomness. Run it to see if deadlock still occurs. (Tip – you're going to need to undo this, so just comment out the code that causes randomness and add your own non-random version of it in the same spot)
- Answer below: is the randomness important in making deadlock occur?
- Also answer: based on your observations and what we've learned in class, **why** the randomness does or does not affect whether the program deadlocks.

The randomness is very important for making the deadlock occur. It is affecting the program by sometimes getting unlucky and requesting a resource that has not been freed and won't be freed because the other thread is needing the resource that the initial thread acquired. The randomness in essence is only in play because it changes the order that the threads acquire the locks.

Now **put the randomness back into the program** before continuing!

(1) Analysis. Does the length of time holding resources affect how quickly the program deadlocks? Why or why not? [to answer, try changing the SHORT_WAIT length and running it]

Yes. Longer waits seem to have higher rates of deadlocks. I think this is happening because the OS is shifting the thread out while it waits and letting the next thread acquire its first resource. This causes more threads to get a chance at acquiring the precious resources out of order. Which results in deadlocks.

(1) Analysis. Does the comparative number of resources affect whether deadlock is possible? For example, suppose there were more than one ball, bat, and glove for each player – could deadlock still occur? [be sure to answer separately answer **both** questions here]

As the ratio of equipment per player increases, the chance for deadlock decreases. This is because as more resources become available, there is a lower chance that each player will be waiting on something that someone else has.

(5) Identify basic solution. This example problem can be corrected with the use of one or more **new** semaphores. The object is to restrict the threads so that only one thread at a time may get new resources. If only one thread at a time can gather resources, and that player thread is allowed is allowed to get a ball, bat, and glove all at once, before another thread can grab anything, deadlock will be avoided. Make the necessary changes to `player.c` so it never deadlocks. The challenge is to figure out exactly where the changes should go. AND you must do this in such a way that multiple threads can still run simultaneously (but only one at a time is grabbing a ball/bat/glove).

Start by searching the code for all uses of `gloves_sem`, to make sure you understand the basic operations that this code is using for semaphores!

Code should be appropriately commented. You may not change the number of players or the quantity of any resource for this part, or relax any of the restrictions specified above. All changes should come through proper implementation of one or more new semaphores.

Part 2: Detecting Deadlocks

(10) Your job is to complete an implementation of the Deadlock Detection algorithm from the text, as discussed in the notes. You are given most of the code as a template in the provided file `player_detect.c`. This code contains data structures necessary to implement deadlock detection in a separate thread. Do not add semaphores for this part. For it to work, the basic code must deadlock, so leave the synchronization as it is. Using the TODOs and the text and notes as guides, finish implementing the deadlock detection algorithm. Your code should compile, run, and produce output similar to the example on the next page. The code must correctly implement the algorithm.

WARNING – just because running your code produces a message that deadlock was detected does not necessarily mean that you properly implemented the algorithm! (it's very possibly to write code that prematurely "detects" deadlock, before it has really occurred). So make sure you understand the algorithm and the starter code before you begin, and analyze the results when finished.

(1) Copy and paste an example of your correct output from Part 2 below.

Example (excerpt) of correct output from Part 2:

NO DEADLOCK EXAMPLE OUTPUT:

```
…
player 4 released glove 5
player 4 released bat 2
player 4 released ball 3
player 4 ..........  done
player 1 released glove 6
player 1 released bat 1
player 1 released ball 0
player 1 ..........  done
player 22 got ball 0
                          **No deadlock**
```

DEADLOCK EXAMPLE OUTPUT:

```
…
player 10 got bat 1
player 7 got glove 5
player 6 got bat 2
player 10 got glove 6
player 14 got ball 3
                          **Deadlock detected!**
             C        R
thread  0: 0 1 1    1 0 0
thread  2: 1 1 0    0 0 1
thread  3: 1 1 0    0 0 1
thread  5: 0 0 1    1 1 0
thread  6: 0 1 0    1 0 1
thread  7: 0 1 1    1 0 0
thread  8: 0 1 0    1 0 1
thread  9: 0 1 1    1 0 0
thread 10: 0 1 1    1 0 0
thread 11: 0 0 1    1 1 0
thread 12: 1 0 0    0 1 1
thread 13: 0 0 1    1 1 0
thread 14: 1 0 0    0 1 1
thread 15: 1 0 0    0 1 1
thread 16: 1 0 0    0 1 1
thread 19: 0 0 1    1 1 0
thread 20: 1 0 0    0 1 1
thread 22: 1 0 0    0 1 1

E: 8 8 8    A: 0 0 0
```