CS2106: Introduction to Operating Systems

Lab Assignment 3 (A3)

Synchronization Problems in Unix

IMPORTANT

The deadline of submission through Canvas: **25th March, 2023, 11.59 PM Saturday**

The total weightage is 6% (+2% bonus):

- Exercise 1
  - Exercise 1a: 2% [Lab demo exercise]
  - Exercise 1b: 2%
- Exercise 2: 2%
- Exercise 3: 2% (Bonus)

You must ensure the exercises work properly on the SoC Cluster.

1 Introduction

When writing a program that uses multiple threads, ensuring correct execution often requires some form of synchronisation. On most Unix-like operating systems, POSIX threads, or more commonly known as pthreads, is the usual threading library to use.

Many synchronisation primitives are available in the pthreads library (such as mutexes, barriers, and condition variables), and they are declared in the `<pthread.h>` header file. Semaphores are also available in the pthreads library, and are declared in `<semaphore.h>`. Semaphores are regarded as the fundamental building block of any synchronisation mechanism, since all the synchronisation primitives can be built on top of semaphores.

In this lab, you'll be solving a synchronisation problems by implementing your own constructs.

Problem setup

You are the manager of a factory that manufactures coloured balls. Each ball comes in one of three colours — red, green, or blue. The balls are to be packed into boxes of N balls each (where N is at least 2), and all balls in a box must have the same colour (***with the exception of exercise 3***). The balls enter the packing area at arbitrary times, and they should stay at the packing area until there are N balls of the same colour. When there are N balls of the same colour, those N balls should be immediately released (at the same time) from the packing area.

Each ball has a **unique** id and is modelled as a thread. You are to implement a synchronisation mechanism to block each ball until it can be released from the packing area.

There are a total of **three exercises** (ex1 to ex3) in this lab. **You may only use POSIX semaphores (i.e., those in `<semaphore.h>`, such as `sem_wait()`/`sem_post()`) in this lab.** You are not allowed to use any other synchronisation mechanism, including those in `<pthread.h>`, nor busy waiting.

Take a look here to get you started. Another useful reference to understand the different concepts of synchronization is this. To solve this lab, you will need to understand the concept of mutex, barrier semaphores, and queues.

Lab directory structure

When you unzip `lab3.tar.gz`, you should find three sub-directories `ex{1..3}`.

To help you focus on the implementation of synchronisation mechanisms, a driver file (named `ex<#>.c`) has been provided for you. The driver file contains the `main()` function, and handles reading from the input file, spawning all the threads, and printing out the events that occur. Your task will be implementing functions that the driver calls.

`Makefile`	Used to compile your files. Just run make. (It will be replaced when grading)
`ex<#>.c`	Prewritten driver file. # range from 1 to 3. (It will be replaced when grading)
`packer.h`	Prewritten header file. Defines function prototypes. (Do not modify)
`packer.c`	Implement your synchronisation mechanisms here.
`*.in`	Input of testcases. (It will be replaced when grading)

You should only modify `packer.c`. Changes to all other files will be discarded, and the driver file will be replaced with a different version for grading.

Compiling and running the exercises

To compile the exercises, run **`srun make`** in the `ex<#>` subdirectory. You will not be penalized for warnings, but you are strongly encouraged to resolve all warnings. You are also advised to use *valgrind* to make sure your program is free of memory errors.

To run the exercises, simply run:

```
$ srun ./ex<#> < [test program]
```

where `[test program]` is an input file (one of the `*in` files) for the driver program. Please see the next section for the expected input format. You can also use stdin to input commands manually.

Driver input format

The driver program takes input from stdin. You are encouraged to write test commands into a file, and use input redirection to feed the test file to the driver program. We have also provided several sample test files for each exercise. The driver program expects the following input format:

- The first line contains a single integer, N, specifying the number of balls in each box (N >= 2).
- Subsequent lines are in one of the following forms:
  - **`<colour> <id>`**: this is a command indicating that a ball with the given *colour* and *id* has arrived at the packing area.
  - **`.`**: a literal period indicates a synchronisation point for the driver (see below for details)

You can make the following **assumptions** about the input we will use for grading:

- It is guaranteed that at the end of the input, all balls can be packed (i.e., there should be no balls remaining in the packing area).
- All balls in one simulation run will have **unique** ids.
- The input file is less than 1000 lines long.

Parallelism in the driver

As per the input format above, input commands (i.e., balls arriving in the packing area) are separated by synchronisation points. The driver batches together all commands until it encounters a synchronisation point or EOF, and then it executes those commands simultaneously in parallel (each ball gets its own thread). The driver is designed to execute those commands at maximum concurrency.

Nondeterminism

Note that the output of your program may be nondeterministic when multiple balls arrive at the same time. This is because when multiple balls arrive at the same time, any one of them might be processed first (e.g., to complete a box).

Test cases in this writeup

In this writeup, you will find many samples for the various exercises. Input text are left-aligned, output text are right-aligned, and parenthesised text in small font are comments (that are not visible when running your program).

Reminder for Mac users

Please note that POSIX semaphores do not work on macOS. Compiling programs with semaphores generally does not throw an error, but the semaphore functions will silently fail.

2 Exercises in Lab 3

2.1 Exercise 1

In this first exercise, we impose a constraint on the problem setup to simplify your implementation:

N = 2 (i.e., balls are packed in pairs)

Functions you need to implement:

- `void packer_init(void);`
  This function is called once at the start of the program. Use this function to perform any necessary setup (e.g., initialize semaphores).
- `void packer_destroy(void);`
  This function is called once at the end of the program. Use this function to free allocated resources that persist throughout the simulation.
- `int pack_ball(int colour, int id);`
  This function is called when a ball enters the packing area, giving the colour and id of this ball. The colour is an integer between 1 and 3 inclusive, giving the colour of the ball (red is 1, green is 2, and blue is 3). The id of a ball may be any integer. This function should block until there is another ball of the same colour entering the area. When that happens, this function should return the id of **the other ball** that should be packed with it in the same box.

As stated in the assumptions, it is guaranteed that no two balls will have the same id. However, ids are arbitrary and may not be issued monotonically.

2.1.1 Exercise 1a: At most Two Balls Packed (1% demo + 1% submission OR 2% submission)[OPTIONAL DEMO]

In this first part, we require that there will be at most two balls of each colour in the entire simulation. Note that this means that there will be at most 6 balls in the entire simulation, and at most one box of balls of each colour.

Here are some test cases through which we elaborate the requirements. You may use them to help to test your code, but you are strongly encouraged to come up with more test cases. Note that due to non-determinism, your output may differ from the example runs below.

A Sample 1: Sequential Test Case

In Sample 1, all the commands are executed sequentially, because there is a period (‘`.`’) separating every command. That allows balls to be matched (if any) before issuing the next command.

The line “Ball X was matched with ball Y” indicates that the `pack_ball()` invocation by ball X has returned and your synchronisation mechanism has decided that balls X and Y should go into the same box.

Note:

1. The output for each of the two balls in a matched pair may be printed in either order. Both outputs are correct.
2. The driver will pause for 100 ms after each ‘`.`’ to wait for balls to be matched. There is a possibility

Sample 1: Example sequential test case (`ex1/test_1a/seq_test.in`)

Input	Possible Output
`2`	
`1 14`	
`.`	
`2 12`	
`.`	
`2 12345`	
`.`	
	`Ball 12 was matched with ball 12345`
	`Ball 12345 was matched with ball 12`
`3 333`	
`.`	
`1 25`	
`.`	
	`Ball 25 was matched with ball 14`
	`Ball 14 was matched with ball 25`
`3 87878`	
`(Ctrl+D pressed to end input stream)`	
	`Ball 333 was matched with ball 87878`
	`Ball 87878 was matched with ball 333`

that 100 ms is not long enough for the balls to be matched, especially if your system has a high load. Based on our testing on the SoC compute cluster/VM, we do not expect this to happen. In the unlikely event that this actually happens, you can try increasing the time limit of the `usleep()` in call in the `ex1.c` file. You should also check your solutions for deadlocks, as they could be a reason for not seeing the desired output.

3. The very first integer in the input is the value of N, and it is always "2" for this exercise.

B Sample 2: Parallel Test Case

In Sample 2, commands that are not separated by periods are batched together, meaning `pack_ball()` will be called on each arriving ball concurrently. You need to ensure that your synchronisation mechanism handles these cases.

C Sample 3: Tiny Test Case

The test case Sample 3 highlights that it is not necessary to receive exactly two balls of each colour. It could be possible to receive zero balls of some colours. Note that it is impossible to only receive one ball of some colour, because that would violate the guarantee that all balls can be packed at the end of the simulation.

Sample 2: Example parallel test case (`ex1/test_1a/par_test.in`)

Input **Possible Output**

```
2
1 180
1 335
2 121
.
                                        Ball 335 was matched with ball 180
                                        Ball 180 was matched with ball 335
3 456
.
2 455
3 457
 (Ctrl+D pressed to end input stream)
                                        Ball 121 was matched with ball 455
                                        Ball 456 was matched with ball 457
                                        Ball 457 was matched with ball 456
                                        Ball 455 was matched with ball 121
```

Sample 3: Example tiny test case (`ex1/test_1a/tiny_test.in`)

Input **Possible Output**

```
2
1 1
.
1 2
 (Ctrl+D pressed to end input stream)
                                    Ball 1 was matched with ball 2
                                    Ball 2 was matched with ball 1
```

2.1.2 Exercise 1b: At least Two Balls Packed (2%)

In this exercise, we still have the constraint that N = 2. However, there may now be more than two balls of each colour. As such, you will need to be careful about which balls, amongst those of the same colour, get packed. In particular, a ball that arrives earlier than another ball of the same colour must be packed no later than that other ball. (Remember that when we say that ball A "arrives earlier" than ball B, it means that the command for ball A appears before the command for ball B and is separated by at least one synchronisation point.)

As this exercise is a superset of exercise 1a, all test cases from exercise 1a are also valid for exercise 1b. **Note: for exercises 1a and 1b, you will submit only a single .c file that satisfies the test cases of both exercises.**

Here is Sample 4 to illustrate this requirement:

Sample 4: Example ordering test case (`ex1/test_1b/order_test.in`)

Input **Possible Output**

```
2
1 101
.
1 102
1 103
.
          (Since ball 101 arrived before balls 102 and 103, ball 101 must be packed with one of them.
                              Either ball 102 or ball 103 may be chosen to be packed with ball 101.)
                                                  Ball 101 was matched with ball 102
                                                  Ball 102 was matched with ball 101
1 104
1 105
.
                 (The remaining unpaired ball (ball 103) must be packed with either ball 104 or 105.)
                                                  Ball 103 was matched with ball 105
                                                  Ball 105 was matched with ball 103
1 106
1 107
1 108
 (Ctrl+D pressed to end input stream)
                (The remaining unpaired ball (ball 104) may be paired with any of the three new balls.)
                                                  Ball 106 was matched with ball 108
                                                  Ball 108 was matched with ball 106
                                                  Ball 104 was matched with ball 107
                                                  Ball 107 was matched with ball 104
```

Sample 5: Example sequential test case (`ex1/test_1b/seq_test.in`)

Input	**Possible Output**
2	
1 123	
.	
2 11	
.	
1 456	
.	
	Ball 123 was matched with ball 456
	Ball 456 was matched with ball 123
2 1000	
.	
	Ball 11 was matched with ball 1000
	Ball 1000 was matched with ball 11
3 145	
.	
3 144	
.	
	Ball 145 was matched with ball 144
	Ball 144 was matched with ball 145
3 2000	
.	
2 2001	
.	
1 2002	
.	
3 2003	
.	
	Ball 2000 was matched with ball 2003
	Ball 2003 was matched with ball 2000
2 43	
.	
	Ball 2001 was matched with ball 43
	Ball 43 was matched with ball 2001
1 5	
(Ctrl+D pressed to end input stream)	
	Ball 2002 was matched with ball 5
	Ball 5 was matched with ball 2002

Sample 6: Example parallel test case (`ex1/test_1b/par_test.in`)

Input	**Possible Output**
2	
1 101	
2 102	
3 103	
.	
1 104	
3 105	
.	
	Ball 103 was matched with ball 105
	Ball 101 was matched with ball 104
	Ball 104 was matched with ball 101
	Ball 105 was matched with ball 103
2 106	
2 107	
.	
	Ball 102 was matched with ball 107
	Ball 107 was matched with ball 102
2 108	
2 109	
.	
	Ball 106 was matched with ball 109
	Ball 109 was matched with ball 106
2 110	
2 111	
1 112	
3 113	
1 114	
.	
	Ball 108 was matched with ball 110
	Ball 110 was matched with ball 108
	Ball 114 was matched with ball 112
	Ball 112 was matched with ball 114
3 115	
.	
	Ball 113 was matched with ball 115
	Ball 115 was matched with ball 113
2 116	
(Ctrl+D pressed to end input stream)	
	Ball 111 was matched with ball 116
	Ball 116 was matched with ball 111

2.2 Exercise 2: N (>=2) Balls Packed (2%)

In this exercise, we no longer have the constraint that N = 2. This means that the number of balls to be packed in each box may be any integer *at least 2*. For grading purposes, we guarantee that N is no more than 64. POSIX semaphore can be incremented to a value of more than 64 without issue, so you do not need to worry about semaphore limits.

As this exercise is a superset of exercise 1, all test cases from exercises 1a and 1b are also valid for exercise 2. You are recommended to test your code with those test cases as well.

There are some changes to the API between the driver and your code, to allow returning the ids of all the other balls to be packed together. Specifically, the `pack_ball()` function now has the following signature:

```
void pack_ball(int colour, int id, int *other_ids);
```

The colour and id of the arriving ball are provided to you in the same way. The `other_ids` parameter points to an array of length N-1, and you should write the ids of the N-1 other balls that should be packed with this ball into the array before returning from this function. Those N-1 ids may be written in any order. (Note that the array has length N-1 because you do not include the id of the current ball itself.)

Notice that instead of returning the id of the other ball via the return value, we are now returning the ids of the other balls via an output array parameter. Note that like previous exercises, you can still that all balls have **unique** ids. Also note that `other_ids` points to memory that is owned by the driver — you do not need to allocate extra memory to pass the N-1 ids.

The driver for exercise 2 is modified from that of exercise 1, to handle the varying value of N and the different API. However, the behaviour of this driver is very similar to that of the previous exercises.

Note that the number of balls per box, N, is given at the very start of the input file, and it is fixed for the entire simulation.

Input **Possible Output**

```
4                                                                    
1 123                                                                
1 234                                                                
.                                                                    
1 111                                                                
3 561                                                                
1 323                                                                
1 888                                                                
.                                                                    
                          Ball 123 was matched with balls 234, 323, 888
                          Ball 234 was matched with balls 323, 888, 123
                          Ball 323 was matched with balls 234, 888, 123
                          Ball 888 was matched with balls 234, 323, 123
2 606                                                                
2 607                                                                
2 608                                                                
```

```
2 609
2 610
2 611
2 612
2 613
2 614
2 615
.
                            Ball 611 was matched with balls 614, 610, 615
                            Ball 610 was matched with balls 614, 615, 611
                            Ball 615 was matched with balls 614, 610, 611
                            Ball 614 was matched with balls 610, 615, 611
                            Ball 607 was matched with balls 609, 613, 612
                            Ball 613 was matched with balls 609, 612, 607
                            Ball 612 was matched with balls 609, 613, 607
                            Ball 609 was matched with balls 613, 612, 607
3 616
3 432
.
1 700
.
2 708
1 705
.
1 360
2 370
3 380
 (Ctrl+D pressed to end input stream)
                                Ball 561 was matched with balls 432, 616, 380
                                Ball 608 was matched with balls 606, 708, 370
                                Ball 616 was matched with balls 432, 380, 561
                                Ball 606 was matched with balls 708, 370, 608
                                Ball 111 was matched with balls 700, 705, 360
                                Ball 708 was matched with balls 606, 370, 608
                                Ball 432 was matched with balls 616, 380, 561
                                Ball 705 was matched with balls 700, 360, 111
                                Ball 370 was matched with balls 606, 708, 608
                                Ball 700 was matched with balls 705, 360, 111
                                Ball 380 was matched with balls 432, 616, 561
                                Ball 360 was matched with balls 700, 705, 111
```

2.3 Exercise 3: At most Two Balls of Different Colours Packed (2%) [BONUS]

In this exercise, we go back to the constraint of N=2 and there may be more than one balls of each colour arriving in the packing area. However, we now consider 4 colour balls (red, blue, green, **and black**). Now, the N balls packed into boxes have different colours. Specifically, *a red colour ball is always packed with a green colour ball, whereas a blue colour ball is always packed with a black colour ball.* The function changes as follows:

```
int pack_ball(int colour, int id);
```

This function is called when a ball enters the packing area, giving the colour and id of this ball. The colour is an integer **between 1 and 4 inclusive**, giving the colour of the ball (red is 1, green is 2, blue is 3, and black is 4). The id of a ball may be any integer. This function should block until **there is another ball of the corresponding paired colour** entering the area. For example, if a red ball (encoded as 1) is already in the packing area, then block till a green ball (encoded as 2) arrives. When that happens, this function should return the id of the **other ball** that should be packed with it in the same box. Similarly, blue ball (encoded as 3) is matched with black ball (encoded as 4).

As stated in the assumptions, it is guaranteed that no two balls will have the same id. The driver and header files for exercise 2 are modified to handle the fourth ball colour. The value of N will always be "2" for this exercise.

You are allowed to use the data structures provided by `sys/queue.h` if needed. Please refer to its manual and some blog.

Sample 7: Example test case (`ex3/tiny_test.in`)

Input	**Possible Output**
2 4 101 . 3 102	Ball 102 was matched with ball 101 Ball 101 was matched with ball 102

Sample 8: Example test case (`ex3/small_test.in`)

Input **Possible Output**

```
2
1 103
1 109
1 102
2 105
2 107
1 110
.
                                    Ball 107 was matched with ball 110
                                    Ball 110 was matched with ball 107
                                    Ball 105 was matched with ball 102
                                    Ball 102 was matched with ball 105
2 112
.
                                    Ball 109 was matched with ball 112
                                    Ball 112 was matched with ball 109
2 101
                                    Ball 103 was matched with ball 101
                                    Ball 101 was matched with ball 103
```

Sample 9: Example test case (`ex3/test.in`)

Input	**Possible Output**
2	
2 110	
3 115	
1 101	
3 114	
2 105	
1 109	
4 103	
.	
	Ball 109 was matched with ball 110
	Ball 110 was matched with ball 109
	Ball 115 was matched with ball 103
	Ball 103 was matched with ball 115
	Ball 105 was matched with ball 101
	Ball 101 was matched with ball 105
4 106	
4 112	
.	
	Ball 114 was matched with ball 106
	Ball 106 was matched with ball 114
2 108	
.	
3 107	
1 113	
.	
	Ball 112 was matched with ball 107
	Ball 108 was matched with ball 113
	Ball 107 was matched with ball 112
	Ball 113 was matched with ball 108
3 111	
1 104	
4 116	
2 102	
	Ball 111 was matched with ball 116
	Ball 102 was matched with ball 104
	Ball 104 was matched with ball 102
	Ball 116 was matched with ball 111

2.4 Exercise 4: Check and Zip Your Solution for Submission (0%)

A Remove ex3 if you haven't attempted it

Your current directory should have the following file structure:

```
ex1
ex1/packer.c
ex2
ex2/packer.c
ex3
ex3/packer.c
check_zip.sh
```

Include bonus `ex3` folder only if you have attempted it. Otherwise rename it (or remove it) from the current working directory, so that it won't be included in the archive.

B Check and archive your solution

Once you have the folders and files as above, you will be able to check it and zip by executing `check_zip.sh`. **We have updated this script so that it does the zipping of the folder along with the checking.** This is to avoid any folder archiving errors.

```
$ chmod +x ./check_zip.sh
$ ./check_zip.sh E0123456 # (replace with your NUSNET id file name)
    or
$ ./check_zip.sh E0123456_E0123457.zip
```

Use your NUSNET ID, NOT your student number 'A012...B', and use capital 'E' as prefix.

The script checks the following:

a. You provided a valid NUSNET ID starting with a capital 'E'.
b. Your current directory contains all the folder and files that needs to be zipped. The folder structure follows the structure presented above.
c. Each exercise can be compiled properly.

During execution, the script prints if the checks have been successfully conducted, and which checks failed. Successfully passing checks ensures that we can grade your assignment. **Points might be deducted if you fail these checks.**

E0123456.zip will be created and contains a top folder with 2 or 3 subdirectories, following one of these file structures depending on whether you have done the bonus part.

```
E0123456                    Or    E0123456
E0123456/ex1                      E0123456/ex1
E0123456/ex1/packer.c             E0123456/ex1/packer.c
E0123456/ex2                      E0123456/ex2
E0123456/ex2/packer.c             E0123456/ex2/packer.c
E0123456/ex3
E0123456/ex3/packer.c
```

*The bolded names are folders.

3 Submission through LumiNUS

Before you submit your lab assignment, run our archive script named **`check_zip.sh`** as mentioned in Section 2.4. Please note that **we have updated this script so that it does the zipping of the folder along with the checking.**

You can check your file structure matches the above format by using the following command:

```
$ unzip -l E0123456.zip          % replace with your zip file name
```

Please ensure that you follow the instructions carefully. Deviations will be penalized.

Upload the created **zip file** to the "Student Submissions Lab 3" folder on Canvas. Note the deadline for the submission is **25th March, 2023, 11.59 PM Saturday**.