CS2106: Introduction to Operating Systems

Lab Assignment 3 (A3) Synchronization Problems in Unix

Important

The deadline of submission through LumiNUS: Wed, 20 Oct, 2pm
The total weightage is 8% + [Bonus 2%]:

- Exercise 1: 1 % [Lab demo exercise]
- Exercise 2: 1 %
- Exercise 3: 2 %
- Exercise 4: 2 %
- Exercise 5: 2 %
- Exercise 6: 2 % (Bonus)

You must ensure the exercises work properly on the SoC Compute Cluster, hostnames: xcne0 – xcne7, Ubuntu 20.04, x86_64, gcc 9.3.0.

Section 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 two synchronisation problems (split into Sections 2 and 3 below) by implementing your own constructs. In Section 2, you are only allowed to use semaphores; but in Section 3, you are allowed to use any synchronisation mechanism (except busy waiting).

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.

Lab directory structure:

The skeleton code for each exercise is placed in a separate folder. In Section 2, the driver code for exercises 1 and 2 are identical. In Section 3, the driver code for exercises 4, 5, and 6 are identical.

Test cases in this writeup

In this writeup, you will find many test cases 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. We strongly recommend that you work on this lab using the SoC Compute Cluster.

Section 2. Ball Packing

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

2.1. 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. 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.

In each exercise, you will find the following files:

`Makefile` (do not modify)	Used to compile your files. Just run `make`.
`ex<#>.c` (do not modify)	Prewritten driver file. (It will be replaced when grading)
`packer.h` (do not modify)	Prewritten header file. Defines function prototypes.
`packer.c`	Implement your synchronisation mechanisms here.

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 **make** in the *ex1-3* 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:
$ ./ex[1, 2, or 3] < [test program]
where [test program] is an input file 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).

2.2. Exercise 1 (1% demo OR 1% submission)

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

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

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., allocate global variables and 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 (encoded as an integer). 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.

Below are some test cases to help you understand the requirements and test your code. However, 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.

Example sequential test case (ex1/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
```

Since there is a period ('.') separating every command, all the commands are executed sequentially, allowing 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: The output for each of the two balls in a matched pair may be printed in either order. Both outputs are correct.

Note 2: The driver will pause for 100ms after each '.' to wait for balls to be matched. There is a possibility that 100ms 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, 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() call in the driver. You should also check your solutions for deadlocks, as they could be a reason for not seeing the desired output.

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

Example parallel test case (ex1/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
```

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

Example tiny test case (ex1/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
```

This test case 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.

2.3. Exercise 2 (1%)

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 1, all test cases from exercise 1 are also valid for exercise 2.

The driver and header files for exercise 2 are identical to those from exercise 1.

Here is an example to illustrate this requirement:

Example ordering test case (ex2/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
```

Below are some other test cases to check your work. You are strongly encouraged to test your code with more test cases on your own.

Example sequential test case (ex2/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
```

Example parallel test case (ex2/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
```

2.4. Exercise 3 (1%)

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 semaphores on the SoC Compute Cluster 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 2, all test cases from exercises 1 and 2 are also valid for exercise 3. 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 assume 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 3 is modified from that of exercise 2, 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.

Example test case (ex3/test.in)

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
```

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.

Section 3. Restaurant

There are a total of **three exercises** (ex4 to ex6) in this section. Exercise 6 is a bonus exercise. In this section, you may use any synchronisation primitives from <semaphore.h> and <pthread.h>. This includes pthread mutexes, barriers, and condition variables. However, it is possible to solve all exercises with only semaphores.

Note that you are not allowed to use busy waiting.

3.1. Problem Setup

You operate a busy restaurant in the populous city of Singapore. There are many tables in your restaurant, and each table has 1, 2, 3, 4, or 5 seats. Groups of people (where each group has 1, 2, 3, 4, or 5 people) queue up outside your restaurant, waiting to be seated. You need to assign a table to each arriving group or keep them in the queue if there are no available tables for them. After a group has finished their meal, they will vacate the table that they have been assigned, and you can assign the vacated table to another group in the queue.

Each of the three exercises in this section comes with varying rules for how tables are to be assigned to groups. Each group is modelled as a thread, and you are to implement a synchronisation mechanism to queue the groups up and assign them to tables.

In each exercise, you will find the following files:

`Makefile` **`(do not modify)`**	Used to compile your files. Just run `make`.
`ex<#>.c` **`(do not modify)`**	Prewritten driver file. (It will be replaced when grading)
`restaurant.h`	Prewritten header file. You may add fields to the struct *group_state* definition, but you are not to modify the function declarations.
`restaurant.c`	Implement your synchronisation mechanism here.

You synchronisation mechanism should be implemented in **restaurant.c**. You may add fields to the struct *group_state* in **restaurant.h**. Changes to all other files will be discarded, and the driver file will be replaced with a different version for grading.

All three exercises in this section have identical drivers and skeleton code.

Functions you need to implement

The functions that you need to implement are identical for all exercises of this section. However, the required behaviours of the functions are different, and will be explained in each exercise.

You need to implement the following four functions:

- `void restaurant_init(int num_tables[5]);`
  This function is called once at the start of the program, providing you with the number of tables with each number of seats. For each *i* in {1, 2, 3, 4, 5}, *num_tables*[*i*-1] is the number of tables with *i* seats. Use this function to perform any setup necessary (e.g., allocate global arrays). You can assume all numbers in the array are positive integers.
- `void restaurant_destroy(void);`
  This function is called once at the end of the program. Use this function to free allocated resources that persist throughout the entire simulation.
- `int request_for_table(group_state *state, int num_people);`
  This function is called when a group enters the queue of your restaurant, giving the number of people in that group, which is an integer between 1 and 5 inclusive. This function should block until there is an available table that can be assigned to this group (according to exercise-specific rules that will be explained later). When that happens, this function should return the id of the table assigned to this group. Furthermore, once this group is in the queue, but before blocking, you need to call the `on_enqueue()` function (implemented by the driver) – see below for details. You can record information in *state* (you are allowed to add fields to *group_state* in *restaurant.h*); the same *state* pointer will be passed to the `leave_table()` invocation when the group leaves the table.
- `void leave_table(group_state *state);`
  This function is called when a group that is currently seated at a table leaves the restaurant. At this point, you should process the queue to see if any group can by assigned to the table vacated by this group. This function should not block.

The `request_for_table()` and `leave_table()` functions are called when you issue specific input commands to the driver. The driver input format will be clarified later in more detail.

Queueing

Groups queue up in a line to enter the restaurant. **A group may only be assigned a table if no other group in front of this group in the queue can be assigned a table.** This means that a group may jump the queue only if all groups in front of it are unable to be assigned a table. The examples in each exercise will make this requirement clear. Keep in mind that semaphore (and other POSIX synchronization primitive) implementations do not guarantee the order in which blocking threads are woken up, so you need to implement your own mechanisms to enforce this ordering.

Queue notification

For the purpose of grading, our grader needs to know the order in which the groups are queued. However, if our grader calls `request_for_table()` on two groups one after another and those two calls block, our grader will be unable to tell if the first group indeed queued up before the second group. This is because our grader does not know how long it should wait before sending the second group – the grader cannot be sure how long your synchronisation mechanism takes to block.

To resolve this situation, we have implemented the `on_enqueue()` function that takes in no parameters and returns void. In your implementation of `request_for_table()`, you are **required** to call `on_enqueue()` (exactly once) **before blocking**, once this group gets a position in the queue. More formally, given two groups, A and B, if group B invokes `request_for_table()` *after* group A's invocation of `on_enqueue()`, then group B must be behind group A in the queue. Our grader will wait until all arriving groups call the `on_enqueue()` function before issuing the next batch of commands (so if you do not call `on_enqueue()`, our grader will not issue any more commands and there will be a deadlock).

You **do not** need to synchronise calls to `on_enqueue()`, as the driver implements its own synchronisation mechanism for printing log messages. You should call `on_enqueue()` even if the group does not need to wait (i.e., there is an available table that can be immediately assigned to this group).

Compiling and running the exercises

To compile the exercises, run **make** in the *ex4-6* subdirectories. You will not be penalized for warnings, but you are strongly encouraged to resolve all warnings.

To run the exercises, simply run:
$ ./ex[4, 5, or 6]
and type in (or paste) batches of commands. Please refer to the next section for the expected input format. Due to nondeterminism (see the *Interactivity* section later), the validity of a command may depend on the response of previous commands.

Note that Valgrind might take a long time to run, because it is essentially a virtual machine that emulates every thread synchronously.

Driver input format

Similar to the ball packing exercises, the driver program takes input from stdin. The driver program expects the following input format:

The first line contains five **positive** (i.e., strictly greater than zero) integers n_1, n_2, n_3, n_4, and n_5, representing the number of tables with 1, 2, 3, 4, and 5 seats respectively. The five integers are separated by space. Tables are (implicitly) given ids starting from 0 – tables with 1 seat are given ids 0 to n_1-1 inclusive, tables with 2 seats are given ids n_1 to n_1+n_2-1 inclusive, and so on. Hence, the table ids overall range from 0 to $n_1+n_2+n_3+n_4+n_5-1$ inclusive.

For example, a line:

```
2 3 2 1 1
```

will create the following table configuration:

Table Size	Table IDs
1	0, 1
2	2, 3, 4
3	5, 6
4	7
5	8

Subsequent lines are in one of the following forms:

- `Enter <id> <num_people>` – this is a command indicating that a group with the given *id* and number of people *num_people* have arrived at your restaurant.
- `Leave <id>` – this is a command indicating that the group with the given *id* just vacated their assigned table; this command will only be issued if the specified group is already seated at a table
- `.` – a literal period indicates a synchronisation point for the driver (see below for details)

Note that the strings "Enter" and "Leave" must be capitalised exactly as shown.

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

- It is guaranteed that at the end of the input, the queue will be empty and there are no groups remaining in the restaurant.
- All group ids are unique within one simulation run.
- For each "Enter" command, 1 ≤ *num_people* ≤ 5.
- The input file is less than 1000 lines long.

Notice that the group id supplied in the input file is never given to your synchronisation mechanism. This is by design – the behaviour of your code should not depend on the group id.

Parallelism in the driver

Like in the previous section, input commands 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 group gets its own thread). The driver is designed to execute those commands at maximum concurrency.

Driver output format

Under normal operation, four types of log messages are printed by the driver:

- `Group <id> with <num_people> people arrived` – this message is printed immediately before `request_for_table()` is called
- `Group <id> is enqueued` – this message is printed after `on_enqueue()` is called, and should appear after the arrival messages, without the need to enter any further commands. Note that this message is always printed, even if the group is not blocked in the queue (as specified earlier, you should always call `on_enqueue()` in `request_for_table()`).
- `Group <id> is seated at table <table_id>` – this message is printed when `request_for_table()` returns.
- `Group <id> left` – this message is printed after `leave_table()` returns.

Driver behaviour overview

Upon receiving a batch of *Enter* or *Leave* commands, the driver does the following (in order):

- For every *Enter* command, prints the *group arrived* log message.
- For every *Enter* command, spawns a new thread and calls `request_for_table()`; for every *Leave* command, finds the associated thread and calls `leave_table()`.
- Sleeps for 100ms.
- Waits until the `on_enqueue()` function is called the same number of times as which `request_for_table()` was called (i.e., waits for all arriving groups in the batch to be either assigned a table or be blocked in the queue).
- For every *Enter* command, checks that `on_enqueue()` was called, and prints the *group enqueued* log message; for every *Leave* command, waits for `leave_table()` to complete, joins the thread, and prints the *group left* log message
- For every thread that newly returned from their call to `request_for_table()`, prints the *group seated* log message.

All log messages are printed from the main thread (i.e., the thread controlled solely by the driver and not used by any group). The driver implements the necessary synchronisation mechanisms to communicate the events between the group threads and the main thread for printing.

Due to the above flow, all *group arrived* log messages will be printed before all *group enqueued* and *group left* log messages, and those in turn will be printed before all *group seated* log messages.

For grading purposes, our grader that replaces this driver will figure out (through other means) the groups that should get assigned a table by the end of each batch of commands, and waits for all of them to return from their call to `request_for_table()` before proceeding. This guarantees that even if the system is slow and 100ms is not enough time for all groups to react to the incoming events, we will still be able to grade your solution correctly.

Interactivity

Note that test cases need to be adaptive – a later command may depend on the output of previous commands. Specifically, you can only issue a *Leave g* command, if group *g* has already been assigned a table (i.e., *Group g is seated at table t* has been printed). This means that some of the commands of our sample test cases may be invalid depending on your output (i.e., depending on how you order concurrently arriving groups in the queue), and you will need to type in the correct commands **depending on the output that has been printed**. The parallel test case for exercise 4 has comments that will make this clear.

During grading, we will use an adaptive grader that decides on the commands to issue based on the log messages that it has seen previously. **Hence, you will be able to assume that every command issued by our grader is valid.**

3.2. Exercise 4 (2%)

In this exercise, **each table may only be assigned to a group with the same number of people as seats at the table**. For example, a group with 3 people may only be assigned to a table with 3 seats.

The following example will help to illustrate this rule.

Example test case (ex4/seq_test.in)

```
Input                                                    Possible output
2 3 2 1 1
Enter 101 3
.
                                      Group 101 with 3 people arrived
                                                Group 101 is enqueued
                                       Group 101 is seated at table 5
Enter 102 3
.
                                      Group 102 with 3 people arrived
                                                Group 102 is enqueued
                                       Group 102 is seated at table 6
Enter 103 3
.
  (At this point, all tables with 3 seats are occupied, so group 103 has to wait in the queue)
                                      Group 103 with 3 people arrived
                                                Group 103 is enqueued
Enter 104 4
.
 (Even though group 103 is in front of group 104, group 103 cannot currently be assigned a table, so
                     group 104 gets to jump the queue and be assigned a table immediately)
                                      Group 104 with 4 people arrived
                                                Group 104 is enqueued
                                       Group 104 is seated at table 7
Enter 105 3
.
                                      Group 105 with 3 people arrived
                                                Group 105 is enqueued
Enter 106 4
.
                                      Group 106 with 4 people arrived
                                                Group 106 is enqueued
Leave 102
.
  (As group 102 leaves, their table with 3 seats becomes vacated, so group 103 takes that table)
                                                       Group 102 left
                                       Group 103 is seated at table 6
Leave 101
.
                                                       Group 101 left
```

```
                              Group 105 is seated at table 5
Enter 107 1
.
                             Group 107 with 1 people arrived
                                       Group 107 is enqueued
                              Group 107 is seated at table 0
Leave 104
.
                                              Group 104 left
                              Group 106 is seated at table 7
Leave 103
.
```

(The table remains vacated after group 103 leaves, since there are no groups with 3 people in the queue)

```
                                              Group 103 left
Leave 106
.
                                              Group 106 left
Leave 107
.
                                              Group 107 left
Leave 105
.
                                              Group 105 left
Enter 108 3
.
                             Group 108 with 3 people arrived
                                       Group 108 is enqueued
                              Group 108 is seated at table 5
Enter 109 3
.
                             Group 109 with 3 people arrived
                                       Group 109 is enqueued
                              Group 109 is seated at table 6
Enter 110 3
.
                             Group 110 with 3 people arrived
                                       Group 110 is enqueued
Leave 109
.
                                              Group 109 left
                              Group 110 is seated at table 6
Leave 110
.
                                              Group 110 left
Leave 108
```

(Ctrl+D pressed to end input stream)

```
                                              Group 108 left
```

Example test case (ex4/par_test.in)

Input | Possible output

```
1 2 3 2 1
Enter 150 2
Enter 185 3
.
```

(Note that the order within each pair of the three pairs of messages below do not matter)

```
Group 185 with 3 people arrived
Group 150 with 2 people arrived
          Group 185 is enqueued
          Group 150 is enqueued
 Group 150 is seated at table 1
 Group 185 is seated at table 3
```

(At this point, the queue contains no groups)

```
Enter 367 2
Enter 374 2
.
```

(As there is only one remaining available table with 2 seats, only one of the newly arrived groups can immediately sit at a table. As both groups arrived at the same time, either group may be placed before the other in the queue. The group that is placed before the other – in this case group 367 – should be seated immediately.)

```
Group 374 with 2 people arrived
Group 367 with 2 people arrived
          Group 374 is enqueued
          Group 367 is enqueued
 Group 367 is seated at table 2
```

(At this point, the queue contains just group 374)

```
Enter 776 2
Leave 150
Enter 777 2
.
```

(When group 150 leaves, group 374 will take the vacated table, since it arrived earlier than groups 776 and 777)

```
Group 777 with 2 people arrived
Group 776 with 2 people arrived
          Group 777 is enqueued
                 Group 150 left
          Group 776 is enqueued
 Group 374 is seated at table 1
```

(At this point, the queue contains groups 776 and 777, in an order known only to your implementation (but unknown to the reader of this output log))

```
Leave 367
Enter 420 5
.
```

(When group 367 leaves, either group 776 or 777, whichever is closer to the front of the queue, will take the vacated table; in this case group 776 is the group that is seated)
(Note that group 420 is seated even though group 777 is still in the queue and is closer to the front of the queue, since the table with 5 seats can only be assigned to group 420)

```
Group 420 with 5 people arrived
          Group 420 is enqueued
                 Group 367 left
 Group 776 is seated at table 2
```

```
                                               Group 420 is seated at table 8
```
(At this point, the queue contains just group 777)

```
Leave 374
.
                                                               Group 374 left
                                               Group 777 is seated at table 1
```
(At this point, the queue contains no groups)

```
Enter 418 5
Leave 777
.
                                              Group 418 with 5 people arrived
                                                               Group 777 left
                                                       Group 418 is enqueued
```
(At this point, the queue contains just group 418)

```
Leave 776
Leave 420
Leave 185
Enter 143 1
Enter 1523 1
.
```
(The three "leave" commands will make the restaurant completely empty. Group 418, and either one of group 143 or 1523, will be seated. In this test run, group 143 happens to be enqueued first, and hence get assigned the sole table with 1 seat.)

```
                                             Group 1523 with 1 people arrived
                                              Group 143 with 1 people arrived
                                                      Group 1523 is enqueued
                                                       Group 143 is enqueued
                                                               Group 185 left
                                                               Group 420 left
                                                               Group 776 left
                                               Group 418 is seated at table 8
                                               Group 143 is seated at table 0
```
(At this point, the queue contains just group 1523)

```
Leave 418
.
                                                               Group 418 left
```
(At this point, the queue contains just group 1523, and the restaurant only has one table occupied (by group 143))

```
Leave 143
Enter 144 1
Enter 111 1
Enter 419 1
.
```
(The table vacated by group 143 must be assigned to group 1523, since it queued up before the 3 groups that just arrived.)

```
                                              Group 419 with 1 people arrived
                                              Group 111 with 1 people arrived
                                              Group 144 with 1 people arrived
                                                       Group 419 is enqueued
                                                       Group 111 is enqueued
                                                       Group 144 is enqueued
                                                               Group 143 left
```

```
                                                      Group 1523 is seated at table 0
        (At this point, the queue contains groups 144, 111, and 419, in an order known only to your
                                                                            implementation)
Leave 1523
.
  (It turns out that group 111 was enqueued before groups 144 and 418, so group 111 gets the table
                                                                                     first)
                                                                      Group 1523 left
                                                        Group 111 is seated at table 0
            (At this point, the queue contains groups 144 and 419, in an order known only to your
                                                                            implementation)
Leave 111
.
                                   (It turns out that group 419 was enqueued before group 144)
                                                                       Group 111 left
                                                        Group 419 is seated at table 0
                                                (At this point, the queue contains just group 144)
Leave 419
.
                                                                       Group 419 left
                                                        Group 144 is seated at table 0
                                                     (At this point, the queue contains no groups)
Leave 144
(Ctrl+D pressed to end input stream)
                                                                       Group 144 left
```

Note that the last few commands may have to be modified depending on the actual order groups 144, 111, and 419 go into the queue, to ensure that we don't instruct a group that is not already seated to leave.

For example, after Group 1523 left, if Group 419 gets table 0 first (because it was enqueued earlier) and the driver prints "Group 419 is seated at table 0" first, then the valid next input would be "Leave 419" ("Leave 111" would no longer be valid, because Group 111 has not been assigned a table yet).

3.3. Exercise 5 (2%)

In this exercise, **each group may be assigned to a table that has at least as many seats as people in that group**. For example, a group with 3 people may be assigned to a table with 3, 4, or 5 seats.

However, if there are multiple available tables that a group can be assigned to, you should assign the group to a table with **minimum number of extra seats**. And if there are multiple such tables, you may pick any one of them. For example, suppose a group with 3 people arrives and there are two empty tables, one with 4 seats and one with 5 seats. You should assign the table with 4 seats to the group.

The given test inputs for exercise 4 are also valid for exercise 5, but the output will differ.

Example test case (ex4/seq_test.in)

```
Input                                                      Possible output
2 3 2 1 1
Enter 101 3
.
                                            Group 101 with 3 people arrived
                                                      Group 101 is enqueued
                                             Group 101 is seated at table 5
Enter 102 3
.
                                            Group 102 with 3 people arrived
                                                      Group 102 is enqueued
                                             Group 102 is seated at table 6
Enter 103 3
.
(At this point, all tables with 3 seats are occupied, so group 103 gets assigned to a table with 4 seats)
                                            Group 103 with 3 people arrived
                                                      Group 103 is enqueued
                                             Group 103 is seated at table 7
Enter 104 4
.
      (Since all tables with 4 seats are occupied, group 104 gets assigned to a table with 5 seats)
                                            Group 104 with 4 people arrived
                                                      Group 104 is enqueued
                                             Group 104 is seated at table 8
Enter 105 3
.
           (There are no empty tables with at least 3 seats, so group 105 waits in the queue)
                                            Group 105 with 3 people arrived
                                                      Group 105 is enqueued
Enter 106 4
.
                                            Group 106 with 4 people arrived
```

```
                                             Group 106 is enqueued
Leave 102
.
    (As group 102 leaves, their table with 3 seats becomes vacated; group 103 is at the front of the
                                                              queue and hence takes that table)
                                                    Group 102 left
                                     Group 105 is seated at table 6
Leave 101
.
 (As group 104 leaves, their table with 3 seats becomes vacated; but there are no groups with at most
                                       3 people in the queue, so the table remains empty)
                                                    Group 101 left
Enter 107 1
.
                                  Group 107 with 1 people arrived
                                             Group 107 is enqueued
                                     Group 107 is seated at table 0
Leave 104
.
  (As group 104 leaves, their table with 5 seats becomes vacated; group 106 is the only group in the
                queue at this point, and they can sit at this table, so they are assigned to it)
                                                    Group 104 left
                                     Group 106 is seated at table 8
Leave 103
.
 (The table remains vacated after group 103 leaves, since there are no groups with at most 4 table in
                                                                                        the queue)
                                                    Group 103 left
Leave 106
.
                                                    Group 106 left
Leave 107
.
                                                    Group 107 left
Leave 105
.
                                                    Group 105 left
Enter 108 3
.
            (Group 108 can be assigned to either table 5 or 6; in this example, they choose table 6)
                                  Group 108 with 3 people arrived
                                             Group 108 is enqueued
                                     Group 108 is seated at table 6
Enter 109 3
.
                                  Group 109 with 3 people arrived
                                             Group 109 is enqueued
                                     Group 109 is seated at table 5
Enter 110 3
.
```

(Note that group 110 must take table 7 – there are no empty tables with 3 seats. Group 110 cannot take table 8 because they must choose an available table with the minimum number of remaining empty seats.)

```
                                            Group 110 with 3 people arrived
                                                      Group 110 is enqueued
                                             Group 110 is seated at table 7
Leave 109
.
                                                            Group 109 left
Leave 110
.
                                                            Group 110 left
Leave 108
```
(Ctrl+D pressed to end input stream)
```
                                                            Group 108 left
```

The following is a test to check the correctness of your synchronisation mechanism under heavy load.

Example test case (ex5/load_test.in)

Input   Possible output
```
2 2 2 2 2
Enter 201 1
Enter 202 1
Enter 203 1
Enter 204 1
Enter 205 1
Enter 206 1
Enter 207 1
Enter 208 1
Enter 209 1
Enter 210 1
Enter 211 1
Enter 212 1
.
                                            Group 212 with 1 people arrived
                                            Group 211 with 1 people arrived
                                            Group 210 with 1 people arrived
                                            Group 209 with 1 people arrived
                                            Group 208 with 1 people arrived
                                            Group 207 with 1 people arrived
                                            Group 206 with 1 people arrived
                                            Group 205 with 1 people arrived
                                            Group 204 with 1 people arrived
                                            Group 203 with 1 people arrived
                                            Group 202 with 1 people arrived
                                            Group 201 with 1 people arrived
                                                      Group 212 is enqueued
                                                      Group 211 is enqueued
```

```
                                                    Group 210 is enqueued
                                                    Group 209 is enqueued
                                                    Group 208 is enqueued
                                                    Group 207 is enqueued
                                                    Group 206 is enqueued
                                                    Group 205 is enqueued
                                                    Group 204 is enqueued
                                                    Group 203 is enqueued
                                                    Group 202 is enqueued
                                                    Group 201 is enqueued
                                       Group 209 is seated at table 9
                                       Group 203 is seated at table 1
                                       Group 208 is seated at table 3
                                       Group 202 is seated at table 2
                                       Group 210 is seated at table 8
                                       Group 201 is seated at table 7
                                       Group 207 is seated at table 6
                                       Group 211 is seated at table 5
                                       Group 205 is seated at table 4
                                       Group 204 is seated at table 0
Enter 213 2
Enter 214 3
.
                                      Group 214 with 3 people arrived
                                      Group 213 with 2 people arrived
                                                    Group 214 is enqueued
                                                    Group 213 is enqueued
Leave 205
.
```

(Note that group 206 or 212 must take the vacated table (that has 3 seats) even though there are sufficient seats for groups 213 and 214, since groups 206 and 212 arrived earlier)

```
                                                           Group 205 left
                                       Group 212 is seated at table 4
Leave 212
.
```

(Similarly, group 206 must take the vacated table)

```
                                                           Group 212 left
                                       Group 206 is seated at table 4
Leave 206
.
```

(Now, either group 213 or 214 (whichever arrived first) should take the vacated table)

```
                                                           Group 206 left
                                       Group 213 is seated at table 4
Leave 213
.
                                                           Group 213 left
                                       Group 214 is seated at table 4
Leave 214
.
```

(There are no more groups in the queue, so the table vacated by group 214 remains empty)

```
                                                               Group 214 left
Leave 201
Leave 202
Leave 203
Leave 204
Leave 207
Leave 208
Leave 209
Leave 210
Leave 211
(Ctrl+D pressed to end input stream)
                                                               Group 211 left
                                                               Group 210 left
                                                               Group 209 left
                                                               Group 208 left
                                                               Group 207 left
                                                               Group 204 left
                                                               Group 203 left
                                                               Group 202 left
                                                               Group 201 left
```

You are also recommended to modify ex4/par_test adaptively, and create additional test cases to verify that your synchronisation mechanism is correct.

3.4. Exercise 6 (Bonus 2%)

This is a bonus exercise.

The restaurant now wants to admit as many groups as possible, and so will now **allow multiple groups to sit at the same table**. However, groups prefer not to share a table with other groups *unless* there are no other empty tables.

In this exercise, **each group may be assigned to a table that has at least as many empty seats as people in that group**. For example, a group with 3 people may be assigned to a table with 3, 4, or 5 empty seats. The table itself may have other non-empty seats that are used concurrently by another group.

However, if there are multiple possible tables that a group can be assigned to, you should do the following:

- If there are tables that are totally empty and have at least as many seats as people in the group, assign the group to such a table with **minimum number of extra seats**. (If there are multiple such tables, you may pick any one of them.)
- Otherwise, if there are tables that are partially occupied and have at least as many empty seats as people in the group, assign the group to such a table with the **minimum number of empty seats**. (If there are multiple such tables, you may pick any one of them.)
- Otherwise, keep this group in the queue.

The following example interaction will help to illustrate this rule.

Example test case (ex6/share_test.in)

```
Input                                          Possible output
1 2 1 1 1
Enter 601 1
.
                               Group 601 with 1 people arrived
                                         Group 601 is enqueued
                                Group 601 is seated at table 0
Enter 602 1
Enter 603 1
Enter 604 1
Enter 605 1
.
   (These four arriving groups must seat on tables 1, 2, 3, 4)
                               Group 605 with 1 people arrived
                               Group 604 with 1 people arrived
                               Group 603 with 1 people arrived
                               Group 602 with 1 people arrived
                                         Group 605 is enqueued
                                         Group 604 is enqueued
                                         Group 603 is enqueued
                                         Group 602 is enqueued
                                Group 605 is seated at table 4
                                Group 604 is seated at table 1
                                Group 602 is seated at table 3
                                Group 603 is seated at table 2
Enter 606 1
.
                             (Group 606 must seat on table 5)
                               Group 606 with 1 people arrived
                                         Group 606 is enqueued
                                Group 606 is seated at table 5
Enter 607 1
.
(As there are no empty tables, group 607 must share a table with someone else.  They must sit at
table 1 or 2, since those two tables have only one empty seat each)
                               Group 607 with 1 people arrived
                                         Group 607 is enqueued
                                Group 607 is seated at table 1
Enter 608 1
Enter 609 1
Enter 610 1
.
(The three arriving groups must sit at the one empty seat at table 2 and the two empty seats at table
                                                           3)
                               Group 610 with 1 people arrived
                               Group 609 with 1 people arrived
                               Group 608 with 1 people arrived
                                         Group 610 is enqueued
                                         Group 609 is enqueued
                                         Group 608 is enqueued
                                Group 608 is seated at table 2
                                Group 610 is seated at table 3
                                Group 609 is seated at table 3
Enter 611 1
Enter 612 1
Enter 613 1
.
(The three arriving groups must sit at the three empty seats at table 4)
                               Group 613 with 1 people arrived
                               Group 612 with 1 people arrived
                               Group 611 with 1 people arrived
                                         Group 613 is enqueued
                                         Group 612 is enqueued
                                         Group 611 is enqueued
                                Group 613 is seated at table 4
                                Group 612 is seated at table 4
                                Group 611 is seated at table 4
Leave 611
Leave 602
Leave 609
.
                                               Group 609 left
```

```
                                                     Group 602 left
                                                     Group 611 left
Enter 614 1
.
```

(Group 614 must sit at table 4, since all tables are not totally empty and table 4 is the only table with exactly one empty seat. Note that even though table 3 has a smaller *total* number of seats than table 4, they must sit at table 4 because table 4 has a smaller number of *empty* seats than table 3.)

```
                                   Group 614 with 1 people arrived
                                             Group 614 is enqueued
                                    Group 614 is seated at table 4
Enter 615 3
.
                                   Group 615 with 3 people arrived
                                             Group 615 is enqueued
                                    Group 615 is seated at table 5
Enter 616 3
.
                                   Group 616 with 3 people arrived
                                             Group 616 is enqueued
Enter 617 2
.
                                   Group 617 with 2 people arrived
                                             Group 617 is enqueued
                                    Group 617 is seated at table 3
Leave 612
Leave 613
.
```

(After groups 612 and 613 leave, there are two empty spaces at table 4; however, group 616 has 3 people so they must still remain in the queue)

```
                                                     Group 613 left
                                                     Group 612 left
Leave 614
.
```

(After group 614 leaves, there are three empty spaces at table 4, so group 616 is assigned to that table (which is shared with group 605))

```
                                                     Group 614 left
                                    Group 616 is seated at table 4
Leave 601
.
                                                     Group 601 left
Enter 618 2
.
                                   Group 618 with 2 people arrived
                                             Group 618 is enqueued
Leave 604
Leave 607
.
```

(After groups 604 and 607 leave, table 1 (with 2 seats) becomes totally empty, so group 618 is assigned to table 1)

```
                                                     Group 607 left
                                                     Group 604 left
```

```
                                 Group 618 is seated at table 1
Enter 619 1
Enter 620 1
Enter 621 2
Enter 622 1
.
                                Group 619 with 1 people arrived
                                Group 620 with 1 people arrived
                                Group 621 with 2 people arrived
                                Group 622 with 1 people arrived
                                          Group 619 is enqueued
                                          Group 620 is enqueued
                                          Group 621 is enqueued
                                          Group 622 is enqueued
                                 Group 619 is seated at table 0
                                 Group 622 is seated at table 5
Leave 615
.
    (As group 615 leaves, 3 seats are vacated, and both groups 620 and 621 can take those seats)
                                                 Group 615 left
                                 Group 620 is seated at table 5
                                 Group 621 is seated at table 5
Leave 619
Leave 620
Leave 621
Leave 603
Leave 608
Leave 610
Leave 617
Leave 605
Leave 616
Leave 606
Leave 618
Leave 622
(Ctrl+D pressed to end input stream)
                                                 Group 622 left
                                                 Group 618 left
                                                 Group 606 left
                                                 Group 616 left
                                                 Group 605 left
                                                 Group 617 left
                                                 Group 610 left
                                                 Group 608 left
                                                 Group 603 left
                                                 Group 621 left
                                                 Group 620 left
                                                 Group 619 left
```

You are also recommended to test out your implementation with the test cases from exercises 4 and 5.

Section 4. Check your archive before submission

Before you submit your lab assignment, run our check archive script named **`check_zip.sh`.** The script checks the following:

a. The name or the archive you provide matches the naming convention mentioned in Section 5.
b. Your zip file can be unarchived, and the folder structure follows the structure presented in Section 5.
c. All files for each exercise with the required names are present.
d. Each exercise can be compiled.

Once you have the zip file, you will be able to check it by doing:

```
$ chmod +x ./check_zip.sh
$ ./check_zip.sh E0123456.zip (replace with your zip file name)
```

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

If you do not want to make a submission for any of the exercises (including the bonus exercise), just leave out the entire folder for that exercise, or zip the unmodified skeleton files that we have provided to you.

Expected Successful Output

```
Checking zip file....
Unzipping file: E0123456.zip
Transferring necessary skeleton files
ex1: Success
ex2: Success
ex3: Success
ex4: Success
ex5: Success
ex6: Success
```

Note that the script does not run your code on any of the test cases we have provided, because the output of the test cases may be nondeterministic. The script merely checks that we can unzip and compile each exercise. Do check that you have removed any debugging output, because extra output may interfere with grading.

4.1. FAQ

Frequently asked questions (FAQ) received from students for this assignment will be answered here. The most recent questions will be added at the beginning of the file, preceded by the date label. **Check this file before asking your questions.** If there are any questions regarding the assignment, please post on the LumiNUS forum.

Section 5. Submission through LumiNUS

Zip the following files as `E0123456.zip` (**use your NUSNET id, NOT your student no A012…B, and use capital 'E' as prefix**).

Do **not** add any additional folder structure during zipping.

`E0123456.zip` should contain 1 folder per exercise, containing the files you need to submit for that exercise:

```
ex1/
        packer.c
ex2/
        packer.c
ex3/
        packer.c
ex4/
        restaurant.h
        restaurant.c
ex5/
        restaurant.h
        restaurant.c
ex6/
        restaurant.h
        restaurant.c
```

*The bolded names are folders.

If there are exercises which you did not attempt, either leave out the folder or use the skeleton files for that exercise.

Upload the zip file to the "Student Submissions Lab 3" folder on LumiNUS. Note that the deadline for the submission is **Wed, 20 Oct, 2pm**.

Please ensure that you follow the instructions carefully (output format, how to zip the files etc.). Deviations will be penalized.