# CS2106 Lab 3

LAB 9 & 13

# Synchronization in Practice

- Used when there are multiple threads
- Ensures correct execution of the overall program
- For POSIX: phthreads library
  - <semaphore.h> – general semaphores
  - <pthread.h> – mutexes, barriers, condition variables

# semaphore.h

- General semaphore
- Able to solve most synchronization problems (might be a bit more complicated)
- Functions available:
  - **sem_init** – initialize the semaphore
  - **sem_wait** – decrease the value of the semaphore if possible
    - sem_trywait – if cannot decrease, return an error
    - sem_timedwait – only “waits” for a certain amount of time
  - **sem_post** – increase the value of the semaphore
  - **sem_destroy** – clean up the semaphore

# semaphore.h usage

```
// declare the semaphore struct
sem_t s;

// initialize before using it
sem_init(&s, ...);

// wait/post as desired in your synchronisation
algorithm, in any/all your threads
```

```
// thread 1                         | // thread 2
... sem_wait(&s); ...               | ... sem_post(&s); ...
```

```
// de-initialize after using it
sem_destroy(&s);
```

# Problems

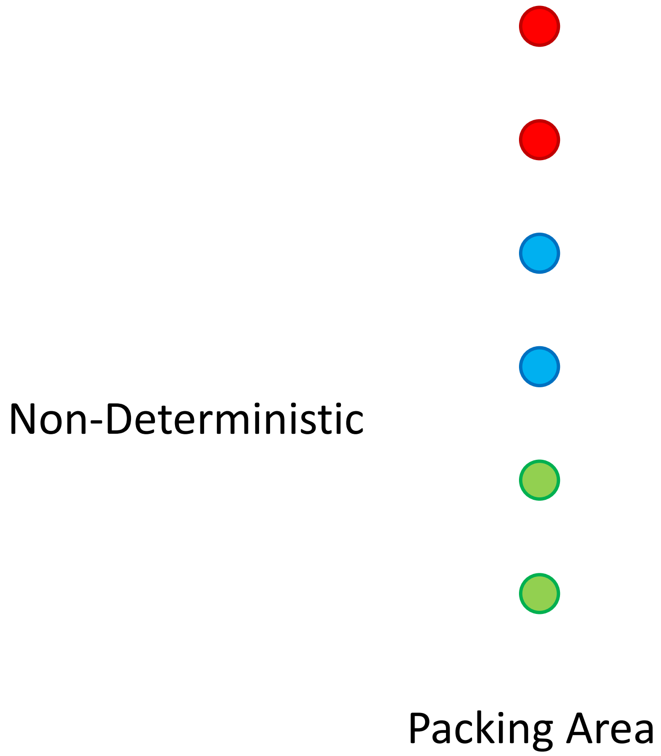
- Ball picking
  - Ex1, Ex2, Ex3
  - Can only use semaphore.h
- Restraunt
  - Ex4, Ex5, Ex6
  - Free to use any synchronization method (except **busy waiting**)

# Ball Packing

- Pack balls (red, green, blue) into boxes
- Each box should have $N$ balls of the same colours $(N \geq 2)$
  - Ex1 & Ex2: $N = 2$
  - Ex3: $2 \leq N \leq 64$
- Balls “wait” in packing area until there are $N$ balls of the same colours in the packing area
- Each ball has a unique ID and is modelled as a thread
- At the end of the input, all balls can be packed
  - There will be multiple of $N$ balls of each colour
- Multiple balls may arrive at the same time, you need to synchronize them

# Visualization (Ex1)

- At most two balls of each colour



Non-Deterministic

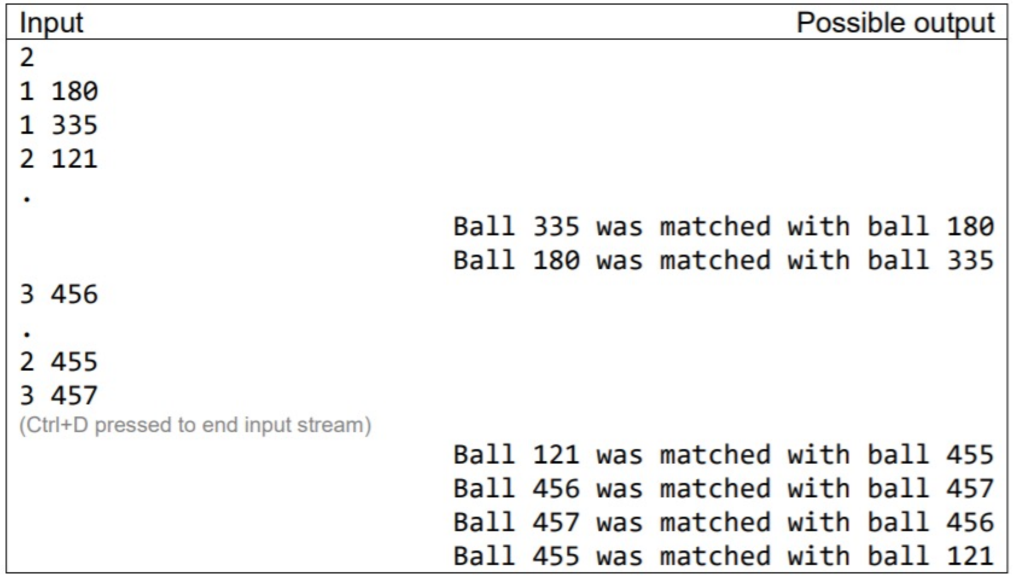
Packing Area

# Functions to Implement

- Only need to edit `packer.c` and `packer.h` (if needed)
- `void packer_init(void);`
- `void packer_destroy(void);`
- `int pack_balls(int colour, int id);`
- `void pack_balls(int colour, int id, int* other_ids);`

# Driver Input

- First line: Integer containing $N$
- Subsequent lines:
  - <colour> <id> – indicate ball with given colour and id has arrived at the packing area
  - . – period indicates a synchronisation point for the driver (all prior command executed in parallel here)



```
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
```

# Driver Overview

```
// for you to do initialisation
packer_init();

// spawn all the balls (i.e. threads)
pthread_create(...);
...

// each ball arrives at the packing area (at arbitrary times) ...
```

| | | | |
|---|---|---|---|
| `// thread 1`<br>`pack_ball(colour, id)` | `// thread 2`<br>`pack_ball(colour, id)` | `// thread 3`<br>`pack_ball(colour, id)` | … |

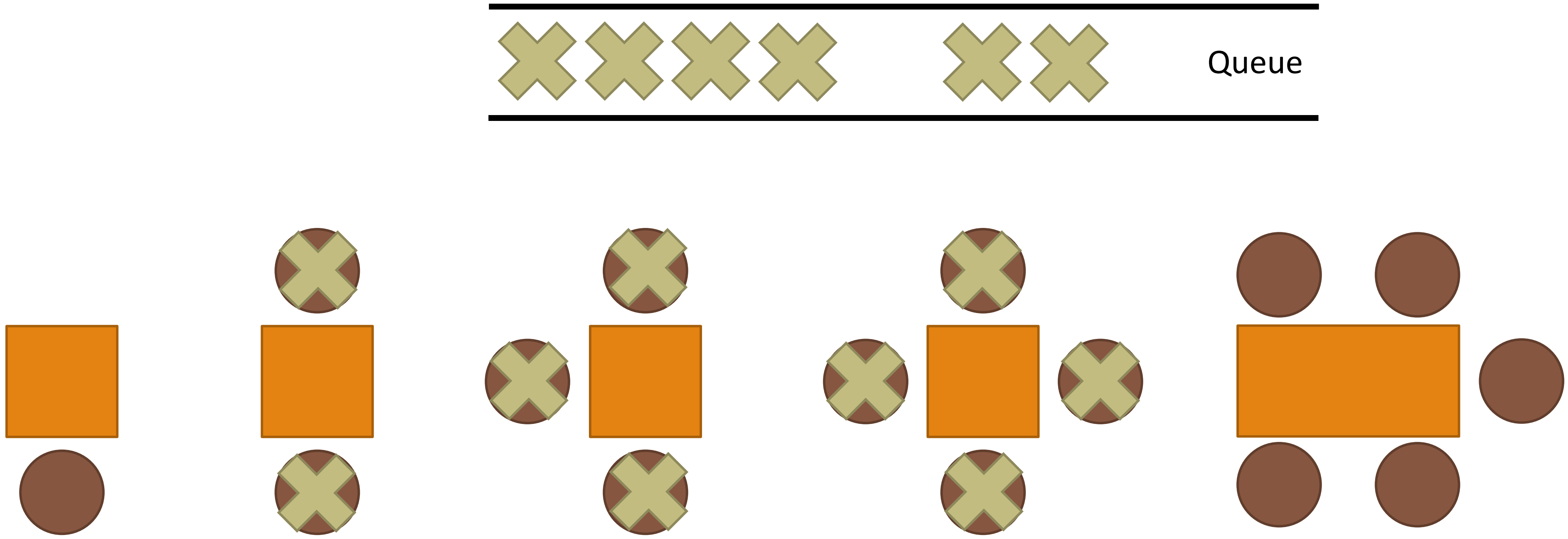
```
// join all the threads
pthread_join(...);
...

// for you to do de-initialisation
packer_destroy();
```

# Restaurant

- Sit groups of guests (1 – 5) into tables (1 – 5 seats)
- Each group has a unique ID and is modelled as a thread
- Assign table to each arriving group or keep them in the queue if there are no available tables
  - Group can jump queue only if all groups in front of it are unable to be assigned a table
  - Need own mechanism to enforce queue order
- After a group finish their meal, they will vacate the table, freeing it.
- Different constraints for each exercise:
  - Ex4: Groups can only sit at tables with exact number of seats
  - Ex5: Groups can sit at tables with at least as many seats (choose min number of seats)
  - Ex6: Multiple groups can share table (Bonus)

# Visualization (Ex4)

- Only sit guest at tables with exact number of seats

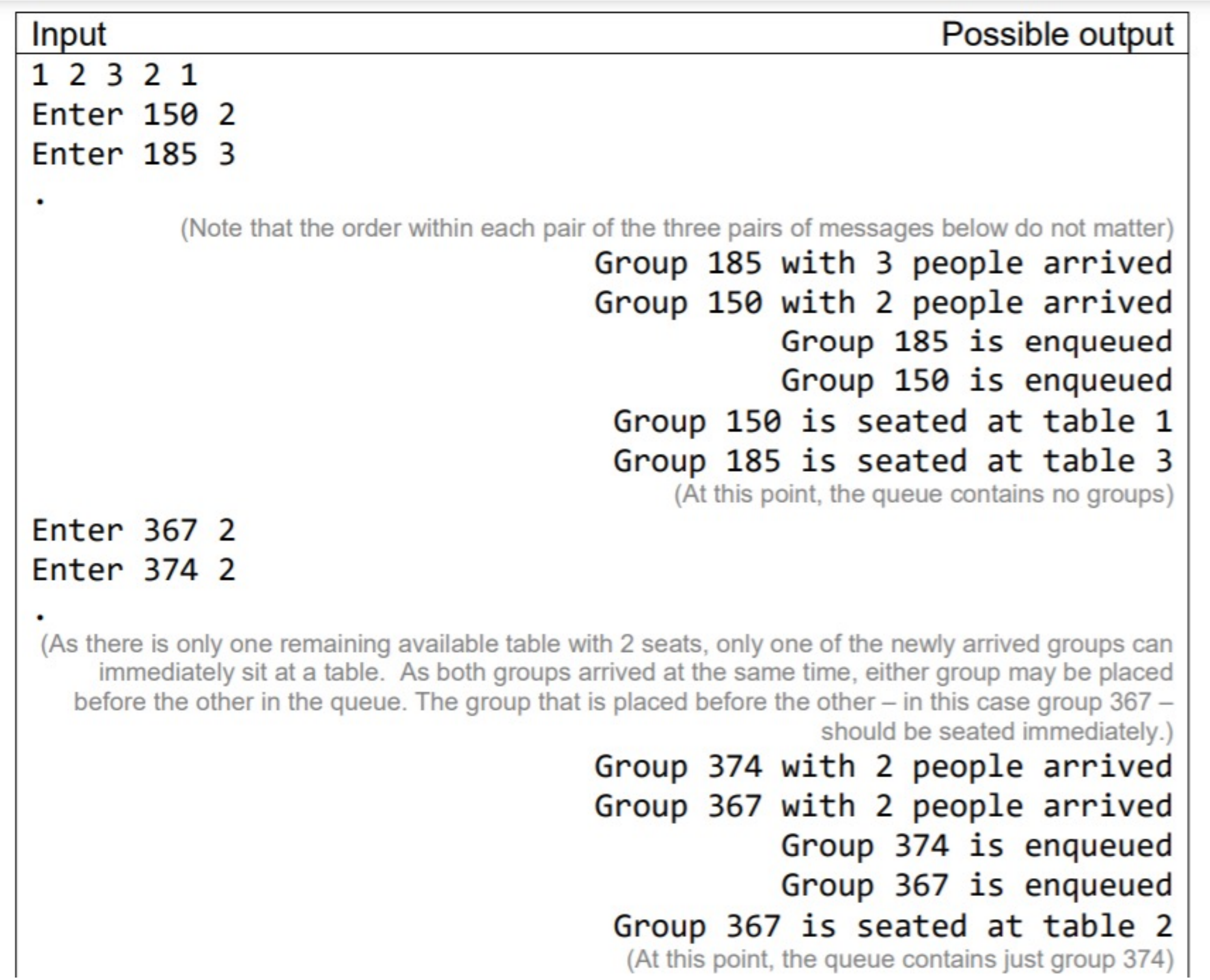


# Functions to Implement

- Only need to modify `restaurant.c` and `restaurant.h`
- `void restaurant_init(int num_tables[5]);`
  - Assume all values are positive
- `void restaurant_destroy(void);`
- `int request_for_table(group_state* state, int num_people)`
  - Make a call to **`on_enqueue()`** once the group gets a position in the queue, before blocking
  - All groups must be enqueue even if there is an available table
  - Do not need to worry about `on_enqueue()` as it has its own synchronization
- `void leave_table(group_state* state);`

# Driver Input

- First line: 5 positive $(> 0)$ integers representing the number of each type of tables
- Subsequent lines:
  - Enter <id> <num_people> (case-sensitive)
  - Leave <id> (case-sensitive)
  - . – period indicates a synchronisation point for the driver (all prior command executed in parallel here)



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)

# Driver Overview

```
// for you to do initialisation
restaurant_init(num_tables);

// spawn all the groups (i.e. threads)
pthread_create(...);
...

// each group arrives (at arbitrary times) ...
```

| | | |
|---|---|---|
| // thread 1<br>// arrive at restaurant<br>request_for_table(state, group_size);<br>…<br>leave_table(state); | // thread 2<br>// arrive at restaurant<br>request_for_table(state, group_size);<br>…<br>leave_table(state); | … |

```
// join all the threads
pthread_join(...);
...

// for you to do de-initialisation
restaurant_destroy();
```

# Final Remarks

- Very long lab that has a lot of details – read carefully
- Start early and start small
- Think about what is the shared resource(s)
- Synchronization labs might have some edge cases, make sure to test for them
- Deadline: 20/10 1400