Project Lunch [redacted]

You went to your favorite restaurant for lunch. While waiting, you observed their service process and thought you could write a program to simulate it. The process you observed is like the following: a customer took a ticket first; after a random while looking around, the customer was ready to order and the ticket number was displayed on a screen showing "Now Serving". The order was processed ~~by either an available server who finished the previous customer and was ready to serve next customer, or~~ by a new server being added (see **Additional Requirements**). Otherwise, the customer waits for the order to be processed. On the other hand, when there was no customer, a server had to wait. Note: in this project,

In this project, you will write code to simulate the above process using synchronization primitives of locks and semaphores. They are available in pthread library in C. Use only locks and semaphores, i.e., no condition variables or other synchronization primitives are allowed.

For the simulator to run, the total numbers of customers and the servers should be input from command line. If input is wrong, prompt should be given for try again with correct format. Sleep function for a random time should be used to simulate the time that a customer spending ~~waiting~~ looking around after getting the ticket. Sleep function for a random time should also be used to simulate the time that a server ~~serving~~ waiting to serve a current customer.

Hint: you could define a structure `struct lunch`, plus four functions described below. `struct lunch` should contain needed information for each client, such as TID, ticket # and be used in functions to pass information. You can use a globe ticket # counter. Also, an order list should be used to store the submitted tickets. In this way, the orders will be served first-come-first-serve. You can implement the list with your own preference. If you implement using circular array, the buffer size can be fixed at 50 as the MAX.

(1) The initialization function for the `struct lunch`, invoked once, starting with there are no customers and no servers.

```
void lunch_init(struct lunch *lunch)
```

(2) When a new customer arrives, he invokes the function

```
int lunch_get_ticket(struct lunch *lunch)
```

This function will return the customer's ticket number, which is also in the lunch variable as well, which is the smallest positive integer that has not been returned to any previous customer (1 for the first customer, 2 for the second, and so on). No two customers will receive the same ticket number. ~~Suggest to use a FIFO queue for the ticket numbers. The queue can be within the struct lunch.~~ After getting a ticket, the customer is free to look around (simulate by random sleep). The following outputs are needed on this function.

<ThreadId Customer> enter <function name>
<ThreadId Customer> get ticket <ticket number>
<ThreadId Customer> leave <function name>

(3) Eventually, the customer will invoke the function

```
lunch_wait_turn(struct lunch *lunch~~, int ticket~~)
```

~~where `ticket` is the value returned by `lunch_get_ticket`~~. This function must not return until there is an available server and the "Now Serving" screen shows a ticket number of the client~~at least as high as `ticket`~~. Once `lunch_wait_turn` returns, the customer will be served by a server (no need to implement this mechanism). The following outputs will be print out.

<ThreadId Customer> enter <function name> with <ticket number>
<ThreadId Customer> leave <function name> after ticket < ticket number> served.

(4) When a new server arrives, ~~or when an existing server finishes with the current customer and is ready to serve a new customer,~~ he will invoke the function

```
lunch_wait_customer(struct lunch *lunch)
```

It must not return until there is a customer needing for service. ~~This function must update the value displayed in the "Now Serving" screen.~~ The order list is helpful to assure clients are served first-come-first-serve. Once this function returns the server will service the customer (you do not need to implement that mechanism). The outputs from the function are:

<ThreadId server > enter <function name>

<ThreadId server > after served ticket < ticket number>
<ThreadId server > leave <function name>

Requirements:

~~(i)You may not use more than one lock in each `struct lunch`.~~

(ii) Don't forget to write a function `Show_serving(int number)`, which will cause `number` to be displayed in the "Now Serving" screen. Invoke this function as needed in your code. The output format: "Serving `<number>`".

(iii) Your solution must not use busy-waiting, e.g., waiting in a while loop.

Additional Requirements:

(a) ~~For the server, you should simulate that each server repeatedly services customers one after another (we will ctrl-c).~~ Read from command line the # of servers and # clients. One server only serves once.

In main program, don't forget join customers and servers.

~~(b) Output necessary info on the activities of the execution of the above four functions are suggested above. In addition, whenever a waiting will happen, printout "`<ID> will wait on <xxx>`".~~

(c) Use an array for each type of threads, also a loop and sleep with a random number. A client sleeps only after getting the ticket. A server sleeps first before serving a client. Make sure that when using the random sleep duration, set a short sleep time. Before sleep, print out "`Sleeping Time: X sec; Thread Id = ThreadId`".

(d) You need to call the specific function mytime() (see files mytime.c, mytime.h) to obtain a random time to be the input to the sleep function. The makefile is provided.

(e) Feel free to use more functions wherever you see fit. The printouts should be used following the same format.

(f) Example of starting multiple threads are given in an early post, file name: PC-inputs-main.c

Submission and Grading:

(1) Your code will use the filename lunch.c. At the start of the code, use comments to write your name, class and the project name. Add other information related to the project as you see fit.

(2) Before submission, use the provided makefile to compile your code at cs-intro server. Make sure the compilation is successful.

(3) After successful compilation, **rename your lunch.c as "Firstname_Lastname_lunch.c"**. You only need to submit **Firstname_Lastname_lunch.c**. This helps TA to identify your individual submission. We have the rest files.

(4) Grading will use cs-intro.