COMP2401 - Assignment #5

(Due: **Wednesday, March 27th, 2024 @ 11pm**)

In this assignment, you will make a simulator for a Drive-thru Restaurant in which customers get into lineups to order and pick up their food. The customers get into line (if space is available in the drive-thru) and then pull up to the "order window" to place their order. Then they move up in the line to wait for their order to be ready. Once the order is ready, they take it and leave.

To begin this assignment, you should download the following files:

- **makefile** – the file that you will use to compile everything.
- **restaurant.h** – definitions and structs that will be used throughout your code.
- **display.h** – a few display definitions that you will use for display.
- **restaurantServer.c** – the program that will simulate the restaurant simulator.
- **lineCoordinator.c** – code for a thread that will handle incoming customer lineup.
- **display.c** – contains the window/drawing code that you will use to display the cars.
- **orderTaker.c** – contains code for a thread that will run the person taking customer orders.
- **timer.c** – contains code for a thread that will countdown time for customer orders to be ready.
- **customer.c** – a program that represents a customer process.
- **generator.c** – a program that will generate random customers.
- **stop.c** – a program used to shut down the server.

You will generate (and use) 4 executables in this assignment:

1. **restaurantServer** – opens a widow to display everything as well as set up the server threads. It is the main program simulator.
2. **stop** – shuts down the server.
3. **customer** – runs a process that will simulate a single customer at the store.
4. **generator** – generates random customers for testing.

When compiling the files, you will need to include the `-lpthread` and `-lX11` libraries. (but the `-lX11` is only needed for the **restaurantServer** program since it uses a window and graphics.)

(1) Examine the **restaurant.h** file. It contains the following definitions and constants that represent the restaurant that we will be using:

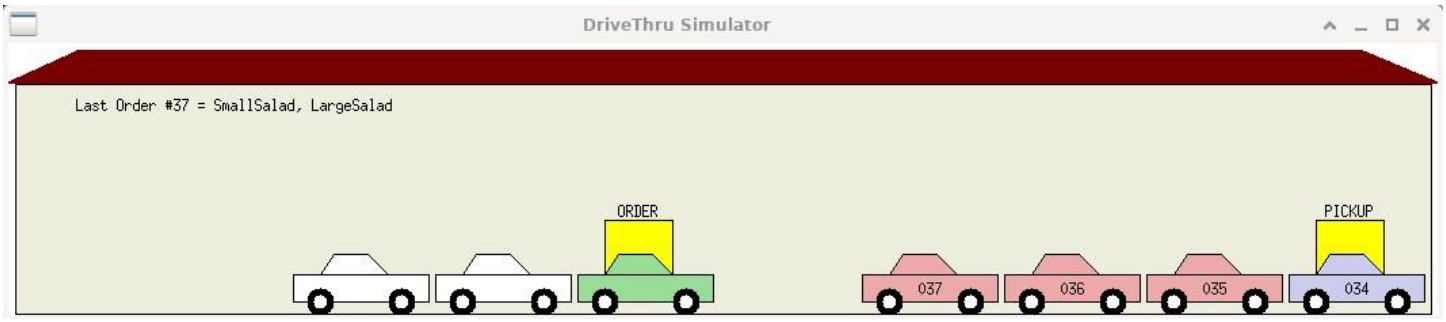
```
#define   MAX_ITEMS_PER_ORDER      10 // Maximum # of items that can be ordered at once
#define   MAX_CUST_WAIT_TO_ORDER    5 // Maximum # of customers allowed to wait to order
#define   MAX_CUST_WAIT_FOR_FOOD    5 // Maximum # of customers allowed to wait to pick up food
#define   MAX_CUSTOMERS            10 // Maximum # of customers being dealt with at any time
#define   MAX_ORDERS                6 // Maximum number of orders being processed at any time
#define   NUM_MENU_ITEMS           11 // The # of menu items available
#define   MAX_ITEM_NAME_SIZE       15 // The maximum # of characters in a menu item name


// Commands that a Restaurant Server may get
#define   ARRIVED                1 // Inform the restaurant that a customer has arrived
#define   PLACE_ORDER            2 // Place an order at the drivethru window
#define   SHUT_DOWN              3 // Shut down the restaurant
```

```
// Responses that a Restaurant Server may send back
#define   ACCEPTED                4 // Customer able to get in line
#define   DENIED                  5 // Customer not able to get in line

// Order-related constants
#define   NO_ORDER_YET          255 // Indicates that no order has been made yet
#define   PACK_TIME               2 // Time it takes (in seconds) to pack an item into the bag

// Settings for Restaurant Server setup
#define   SERVER_IP     "127.0.0.1"  // IP address of RestaurantServer
#define   SERVER_PORT          6000  // PORT of the RestaurantServer

// This represents an item that can be ordered
typedef struct mi {
   const char  *name;  // Name of the item ordered
   float        cost;  // Cost of the item ordered
} MenuItem;


// This represents an order that a customer makes
typedef struct ord {
   unsigned char   orderNumber;                  // Unique order number for each customer
   MenuItem        items[MAX_ITEMS_PER_ORDER];   // The items ordered
   unsigned char   numItems;                     // # of items ordered
   short           secondsUntilReady;            // Time until food is ready
} Order;


// This represents a customer in line
typedef struct cust {
   unsigned int   customerPid;  // process ID of this customer
   Order          order;        // order that customer made, NULL if not made yet
   char           startedOrder; // set to true if we told the customer to start ordering
} Customer;


// This represents the shared data in the restaurant
typedef struct rest {
   Customer        driveThruLine[MAX_CUSTOMERS]; // Customers in line
   unsigned int    nextOrderNumber;              // Global customer count
   sem_t           LineupSemaphore;               // Allows us to lock up line
} Restaurant;


// External Global Variables
extern const char       *ItemNames[NUM_MENU_ITEMS];
extern const float       ItemPrices[NUM_MENU_ITEMS];
extern const float       ItemCookTime[NUM_MENU_ITEMS];
extern const float       ItemPrepTime[NUM_MENU_ITEMS];
extern const float       ItemFillTime[NUM_MENU_ITEMS];
```

The 5 external global variables mentioned above and defined in **restaurantServer.c** as follows:

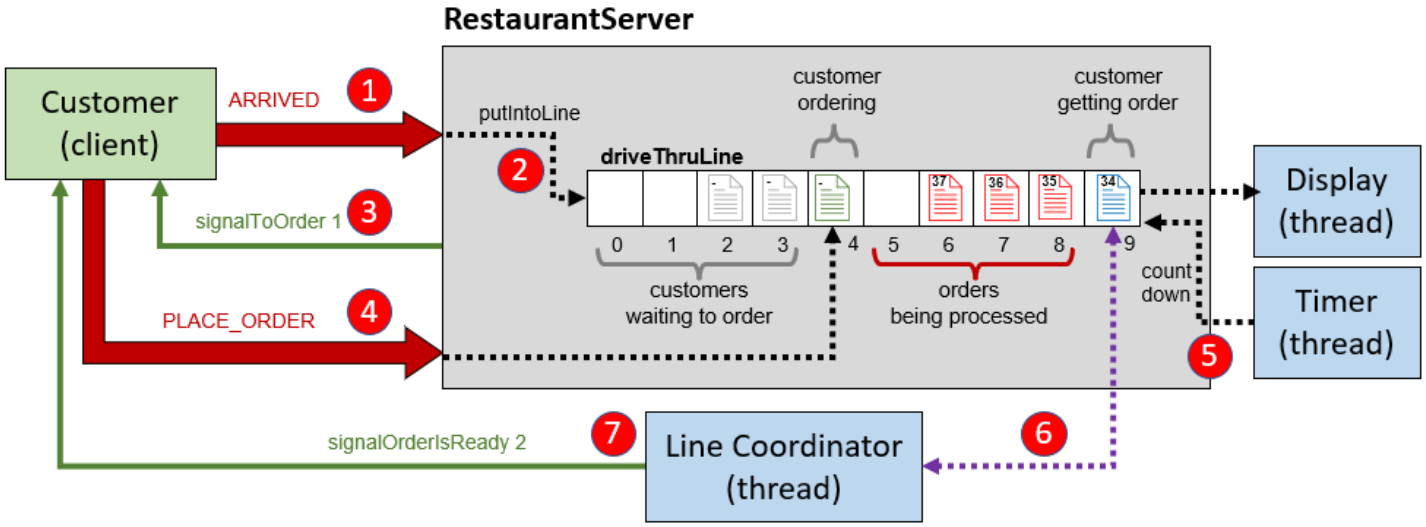
```
const char *ItemNames[NUM_MENU_ITEMS] = {"Hamburger", "CheeseBurger", "VeggieBurger", "ChickenFingers",
                                         "SmallFries", "LargeFries", "SmallSalad", "LargeSalad",
                                         "SmallDrink", "LargeDrink", "IceCreamSundae"};

// These are the prices of the menu items
const float ItemPrices[NUM_MENU_ITEMS] = {3.99, 4.99, 5.29, 6.99, 2.99, 4.19, 4.99, 7.99, 2.29, 3.79, 5.99};

// These are the preparation (e.g., cooking/preparation/filling time) for the menu items
const float ItemCookTime[NUM_MENU_ITEMS] = {120, 120, 115, 180, 90, 90, 0, 0, 0, 0, 0};
const float ItemPrepTime[NUM_MENU_ITEMS] = {10, 12, 10, 8, 8, 10, 2, 2, 2, 2, 3};
const float ItemFillTime[NUM_MENU_ITEMS] = {0, 0, 0, 0, 0, 0, 0, 0, 10, 15, 30};
```

Here is a screenshot of what the **restaurantServer** may look like while running:



It shows **7** customers/cars in line. **White** indicates “waiting to order”. **Green** indicates “currently ordering”. **Red** indicates “waiting for order to be ready”. **Periwinkle** indicates “picking up order”. Each car that has ordered shows the order number on it. The last order placed is also displayed at the top of the restaurant window. As cars arrive, place orders and wait … they continually move forward in the lineup … to the right.

It is important to understand how everything will work. Therefore, here is a diagram that shows what needs to be done. Understand this carefully, before you continue on the assignment.



The **driveThruLine** is the key to coordinating things. It is an array that contains exactly **10** Customer structs/objects. If no customer is waiting at a specific index in the array, the **customerPid** for that customer struct/object will be set to **0**.

Notice that there are three threads:

- The **Display** thread has been written for you already. It simply displays the cars from the **driveThruLine**. You must NOT alter the code in **display.c** nor **display.h**.
- The **Timer** thread looks through all orders that have been made (4 of them in the example above) and simply continues to reduce the **secondsUntilReady** values for those orders.
- The **LineCoordinator** thread continuously examines the **driveThruLine** and moves customers/cars up as they order and as other cars move up and space permits them to advance.

Looking at the red circles, this indicates the order that things have to happen in:

1. A customer/car starts everything by sending an ARRIVED message to the **orderTaker** (which is the main loop of the **restaurantServer**). Along with the message, it sends its process id (which is a 4-byte value). It then waits for a response from the **orderTaker** and then disconnects from the server and goes into a waiting loop until they are signaled to place an order.
2. The **orderTaker** puts the customer into the lineup at position **0** in the array if nobody is waiting there already (i.e., sets the **customerPid** for that customer in the array) and then replies to the customer that they are ACCEPTED to the lineup. However, if there is a customer/car already at that leftmost location in the lineup, then a DENIED response is sent back to the customer.
3. As the **lineCoordinator** moves customers up in the line, it detects when a customer has just moved up to the order window. At that point, the **lineCoordinator** thread sends a **SIGUSR1** signal to the customer to inform them to place their order.
4. Once the customer receives the signal to order, it re-connects to the server and sends a PLACE_ORDER message to the **orderTaker**. As part of the message, it sends a sequence of bytes in which each byte is a number from **0** to **NUM_MENU_ITEMS** and represents an item to be ordered from the menu. Each number represents an index in the **ItemNames** array defined in **restaurantServer.c** file. The customer does not wait for a response from the server, but instead disconnects from the server and then goes into a waiting loop until they are signaled that their order is ready to be picked up.
5. From step 4 above, the **orderTaker** assigns a unique **orderNumber** (based on the **nextOrderNumber** from the **Restaurant** struct) and stores the ordered **MenuItems** into the customer's items array at that position in the array. It also assigns a **secondsUntilReady** value for that order. The timer thread repeatedly decreases the **secondsUntilReady** value at fixed intervals.
6. The **lineCoordinator** continuously examines the customer in the rightmost position in the array to determine if the **secondsUntilReady** value has reached **0**.
7. When **secondsUntilReady** value for the rightmost customer has reached **0** the **lineCoordinator** sends a **SIGUSR2** signal to the customer whose **customerPid** is set for this rightmost customer in the lineup to inform the customer that the order is ready to be picked up. The customer receives that signal and then quits.

It will be important to complete this assignment in stages. Try to follow the order that is given in the next parts of the assignment. You MUST follow the directions/instructions mentioned above, even though you may feel that there is a better way to do things. This will be necessary for consistency in grading the assignments.

(2) Modify the code in the **orderTaker.c** file so that it accepts an incoming **SHUT_DOWN** request. When it receives such a request, it should go offline and print a message saying that it is shutting down. Make sure that any client sockets that were opened are closed before you shut down the server.

Write code in the **stop.c** file so that it sends a single-byte **SHUT_DOWN** command to the **orderTaker** and then quits. Make sure to properly handle any errors in connecting to the server. Note that there will be no loops in this program. You send one command and then disconnect and quit.

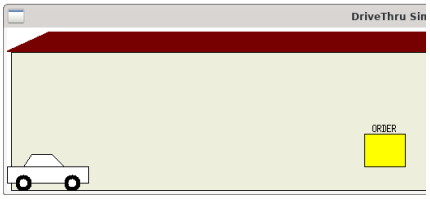
At this point … make sure that your code compiles and runs properly. Open two terminal windows. Start the **restaurantServer** in one terminal window. You should see the restaurant displayed with

no cars. Run the **stop** program in the other terminal window. It should shut down the **restaurantServer** with no errors and the display window should be gone. Run **ps** in both terminal windows to make sure that no processes are still running.

(3) Now it is time to do something interesting. Add code to the **while** loop in the **customer.c** file so that the customer sends an **ARRIVED** message to the restaurantServer/orderTaker. It should also send its 4-byte process ID as part of the message. The customer should then accept a single-byte reply which must be **ACCEPTED** or **DENIED**. Either way, the customer should close connection to the server. If it was DENIED, the customer should wait 2 seconds and then go back up the loop to try again. Otherwise, if ACCEPTED, it should leave the loop.

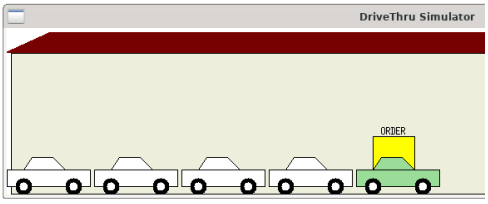
Adjust the **orderTaker.c** code so that it accepts an incoming ARRIVED message from the customer. The code should examine the **driveThruLine** array for the restaurant at position **0**. If a customer is already at position **0** in the array (check to see if the **customerPid** is non-zero), then a **DENIED** should be sent back to the customer. Otherwise, the customer should be sent **ACCEPTED** and the customer should be placed at position **0** in the **driveThruLine** array, **startedOrder** should be set to **0** and the **orderNumber** should be set to **NO_ORDER_YET**. Make sure to close the client socket before restarting the **while** loop. You may want to add some **printf()** statements for debugging to make sure that the messages re being sent & receiver properly.

Test your code by running the **restaurantServer** in one window and the **customer** in the other. You should see a car appear on the left (see image here) if you successfully added the customer to the lineup. Leave the **restaurantServer** running and then run the **customer** again. If you put appropriate print statements in, you should see that the customer is DENIED and it should attempt again every two seconds.



(4) Add code to the **while** loop of the **lineCoordinator.c** (after the 1 second delay) so that the new customer is moved up in the line. To do this, just check if there is a non-zero **customerPid** at location **0** in the array. This indicates that a customer is at the leftmost position in the array. To see if it should move forward, you just need to check where it should go. Check backwards starting from position 4 in the array down to position 2 to search for a “free position” that has no car. If there is a “free position” at position **i** (i.e., the **customerPid** is zero), then move the customer at position **0** to position **i** by simply copying over the customer in the array. Make sure to set the **customerPid** at position **0** in the array to **0** now so that it is a “free position” for any new customers.

Test your code by running the **restaurantServer** in one window and then 6 **customers** (one at a time) in the other. You should be able to add 5 cars and then the 6th one should be DENIED (see image here).



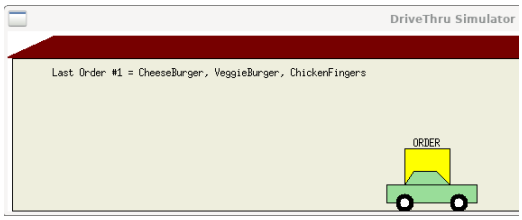
(5) Adjust the **customer.c** code so that after a customer gets an **ACCEPTED** signal (i.e., after the **while** loop), it the waits for a **SIGUSR1** signal. If you decide to make an infinite loop to wait, make sure to put a **sleep(1)** in there to wait for a second … otherwise the loop will take up too much CPU time. Once the signal is received, the customer should wait for `usleep(1000000 + 200000*n)` microseconds, where **`n`** is the number of items being ordered. This will simulate a delay in the ordering process based on the number of items being ordered. The number of items being ordered should be the number of integers entered as command-line parameters to the **customer** program.

After this delay, the **customer** should connect to the **restaurantServer** and send a PLACE_**ORDER** message to the order taker which should be accompanied by a single byte representing each item ordered (which are the command-line parameters). So, if the program was run as follows: `./customer 3 1 6`, then the byte values **3**, **1** and **6** should be sent representing **ChickenFingers**, a **Cheeseburger** and a **SmallSalad** (according to the defined **ItemNames** array). The socket connection should then be closed.

In the **orderTaker.c** code … you should handle the incoming **PLACE_ORDER** message along with the items to be ordered. The car currently at the order window (there should always be one there when this message arrives) should have the **numItems** set to the number of items ordered. For that car/customer, the items array should then be filled in with the **ItemNames** and **ItemPrices** (see **RestaurantServer.c** code) for all the items that are being ordered. Finally, a brand new **orderNumber** should be given to that order based on the **nextOrderNumber** for the restaurant. Make sure to close the client socket when you are done.

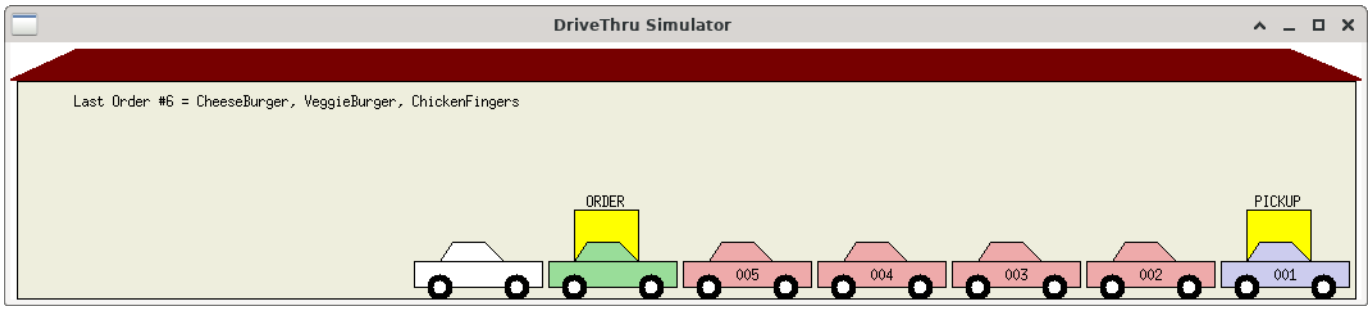
Finally, in the **while** loop of the **lineCoordinator.c** code, after the code that you wrote for moving up customer (i.e., at the end of the **while** loop), add code that informs the customer at the order window to place the order. To do this, first check to see if someone is waiting to order (i.e., there is a car at the order window position … just check for the **customerPid** being non-zero. Also, this customer must not have ordered already … check the **startedOrder** Boolean). If there is indeed a customer waiting to order, then send it the **SIGUSR1** signal and set the **startedOrder** flag to true.

Test your code by running the **restaurantServer** in one window and then **./customer 1 2 3** in the other. If you placed appropriate **printf()** statements, you should see that the customer sends the PLACE_ORDER message and that the server received it properly. The order should appear as `Last Order #1 = Cheeseburger, VeggieBurger, ChickFingers` in the display window(see image here).



(6) Now we will get the customer to move up and wait for its order to be ready. Add code in the while loop of the **lineCoordinator.c** code to move the customer forward after ordering. Do this before the other code in the **while** loop (but after the **sleep(1)**). Just check if the customer at the order window has an **orderNumber** assigned to it. If so, then check backwards from the last position in the **driveThruLine** (i.e., the pickup window) back to the position after the order window to see if the car can move forward. As you check each position (in backwards ordering), just check to see if there is an empty spot (i.e., the **driveThruLine** location has **NO_ORDER_YET** at that location). If so, copy the customer at the order window into that new location then remove it from the order window location by setting the **customerPid** to **0** and **orderNumber** to **NO_ORDER_YET**.

Test your code by running the **restaurantServer** in one window and then **./customer 1 2 3** in the other. The customer should go to the window, place the order and then end up at the PICKUP location. Run 6 more customers the same way. You should see them all line up as shown in the image below, each with their own unique order number, except for the last two. The last customer ran (i.e., the white car) should still be running because it is waiting to be asked to place an order. The green car will have placed the order but the order number will not appear. However, you should be able to see the order displayed at the top of the restaurant display showing Last Order #6.



(7) Now we need to get customers to pick their order up. Go back to the **customer.c** code and add code at the end to wait for a **SIGUSR2** signal. Make sure to use `sleep(1)` within any infinite loops. Once the **SIGUSR2** signal is encountered, the customer program should quit.

Insert code in your **lineCoordinor.c** code's **while** loop to allow the customer to pick up the order. To do this, add code at the top of the **while** loop (after the `sleep(1)`) that checks to see if there is a customer waiting at the PICKUP window. If so, it should send a **SIGUSR2** signal to that customer and then erase the customer from that position in the lineup by setting the **customerPid** to **0** and order number to **NO_ORDER_YET**.

Test your code by running the **restaurantServer** in one window and then **./customer 1 2 3** in the other. The customer should go to the window, place the order, move to the PICKUP location and then drive off. Run a few customers … each should do the same thing.

(8) Now we need to adjust the code so that the food takes a while to prepare and get picked up. Go back to your **orderTaker** code. Adjust your code that fills in the order's item names and prices so that it also computes the food preparation time for that order. For each item ordered, you need to add some time to a **total** time. For each item, add the **ItemPrepTime** and the **ItemFillTime** for that item (see **RestaurantServer.c** code) to the total time. Also add a standard **PACK_TIME** for each item. As you loop through the items, you should also check the **ItemCookTime** for that item and determine the maximum **ItemCookTime** for all the items. Do NOT add the **ItemCookTime** to the total for each item. Once you have the **total** time as well as the **maximum** cook time, set the **secondsUntilReady** for that order to be the addition of these two values

Go back to the **lineCoordintor.c** code and adjust the code from part 7 of the assignment so that a customer is not told to pick up the order unless the **secondsUntilReady** for the customer at the pickup location is less than or equal to **0**.

In the **timer.c** code, add code to the **while** loop (after the `usleep(30000)`). You should go through each customer that has placed an order and decrement the **secondsUntilReady** value by 1.

Test your code by running the **restaurantServer** in one window and then try running a few customers (1 at a time) in the other window. Each one should have a different delay depending on the items ordered. Try these:

- **./customer 7** - should not even turn purple but should drive off quickly
- **./customer 2** - should turn purple for about 2.5 seconds
- **./customer 3 3 3** - should turn purple for about 4 seconds
- **./customer 0 0 5 5 9 9 10** - should turn purple for about 6.5 seconds

(9) At the moment, we are running one customer at a time, but we still have not moved up any customers in the line when a customer leaves. If the car picking up the order has left, we need to move forward (in the lineup) any customers who have ordered. In the while loop of the **lineCoordintor.c** code … after the code that checks for customers pickup up food and signaling them to leave … we need to add code to move up the customers. Check to see if the customer at the pickup window is gone (i.e., there is NO_ORDER_YET for that customer). To do this, check (in reverse order) from the pickup window back towards the order window. If there is a customer waiting to pickup food (i.e., there is an **orderNumber** for that customer position) then move that customer forward in the lineup by copying over its information in the array. Make sure that you code moves all waiting customers (who have ordered) up by one. Then for the last customer moved forward, make sure to set the **customerPid** to **0** and order to be **NO_ORDER_YET** for the location that it just moved from.

We also need to do something similar when a customer orders … by moving customers forward towards the ordering window. In that same **while** loop, just before the code that moved the newly-arriving customer up to the order window (i.e., from part 4 of the assignment), add code that checks to see if there is nobody at the order window (i.e., check **customerPid**). In that case, as you just did above, check backwards from the order window back to the leftmost position in line and see if there are any customers (i.e., **customerPid** is non-zero) that should move up. As you did above, copy the customer forward in the line and set the **customerPid** for the last customer moved up to be **0**. We will test this code when you complete the next step.

(10) Write the **generator.c** code so that it creates and runs 20 customer processes. For each customer process, it should create an order of a randomly-chosen number of items … between 1 and 10. It should then randomly choose each item. Each process should run as a background process. You should set a delay of `usleep(rand()%10 * 500000)` between processes so that there is a bit of time between them, chosen randomly.

Test your code by running the **restaurantServer** in one window and then running the generator in the other window. You should see customers lining up, ordering, waiting for food and then picking up food and leaving. Depending on the randomness, you may see longer lineups forming of customers that are waiting for their food and sometimes longer lines of ones waiting to order. Eventually, you should see that exactly 20 orders are placed and all customers are eventually gone.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit:

1. A **Readme** text file containing
   - your name and studentNumber
   - a list of source files submitted
   - any specific instructions for compiling and/or running your code
2. Your **makefile** and ALL the **.c** and **.h** files needed to compile and run.

The code **MUST** compile and run on the course VM.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).
