New York University Abu Dhabi
CS-UH 3010 (001) - Spring 2018
Programming Assignment 3
Due: April 14th, 2018

Preamble:

In this assignment, you will write independent programs that run concurrently, get synchronized at various points, and their cooperative work yields the operation of WestEnd Dinner restaurant. “Individuals” (processes) enter the restaurant, wait in line to give their order and pay, remain in line until their order is ready, and finally, they pick up their meal, sit down to eat for some time and ultimately, exit the dinner. There is a number of individuals –realized as processes– who are involved in the operation of the dinner: clients, cashiers, and a server; all these are coordinated by a dinner manager process.

The main objective of the assignment is to create independent processes that essentially “carry out” (i.e., *simulate*) the work of clients, cashiers and the server. These 3 types of processes have to synchronize among themselves using semaphores. Moreover, the various programs that cooperatively make up the operation of the WestEnd Dinner have to comply with a number of synchronization properties at all times. At the end of the day or when there are no more incoming clients, the dinner manager process compiles various statistics about the business of the day.

You will have to launch the various programs involved through possibly different `ttys` or have the manager spawn different processes using `fork()` and `exec()` calls. It is also your responsibility to show correct execution.

In the above context, you will be:

- using *(POSIX) Semaphores* to implement the required synchronization conditions and have clients, cashiers and server coordinate their work invoking only appropriate P() and V() primitives,
- creating pertinent structure(s) on a *shared memory segment* so that statistics for the orders placed by clients can be maintained,
- having all client/cashier/server processes *attach* the above shared segment so that they can conveniently access and possibly manipulate the content of the main-memory resident structure(s).

Procedural Matters:

◇ Your programs are to be written in `C/C++`
◇ You will have to first submit your project to newclasses.nyu.edu and then, demonstrate your work.
◇ Nabil Rahiman (`nr83-AT+nyu.edu`) will be responsible for answering questions as well as reviewing and marking the assignment.

Our WestEnd Dinner Problem:

Figure 1 depicts the operation of the dinner.

Clients arrive at random times and *may* line up in a *FIFO* queue to give their orders. An order consists of a *single item* (for simplicity) available on the menu as provided in *Appendix A*. There is a limit of how many client processes can wait in queue for service (`maxPeople`. If there are more clients arriving, they simply turn arround and leave.

There are at most k cashiers taking orders from people. When a client works with a cashier, it takes $t_{service}$ time for the order to be placed (client talks to the cashier, places the order, pays and gets a receipt). Then, each client proceeds to the area in front of the server's table and awaits for her food to arrive. Any time there is a cashier available, she/he calls a person up from the queue for service. If there are no more people, the cashier takes a small break for a period of time t_{break}. When she comes back, she continues along the same line (calls up for service people waiting and if no people are in the queue, she takes another break, etc.)

Clients who have already given their order wait in the area in front of the server's table while their food is being prepared. This waiting time depends on the type of meal they have asked; *min* and *max* times for preparing specific items on the menu are provided in *Appendix A*. There is only 1 server who coordinates matters with the (invisible) back-end preparation room of WestEnd Dinner and puts items on the table. When an order arrives for final dispatch, the server calls up the respective client and hands over the food in *FIFO*; this interaction lasts for up to t_{server} time.

Once served, a client goes to the dinning area and stays there for a random period of time (up to t_{eat}) while eating and talking to others on her table. Finally, she clears her space, disposes the trash and exits the shop.

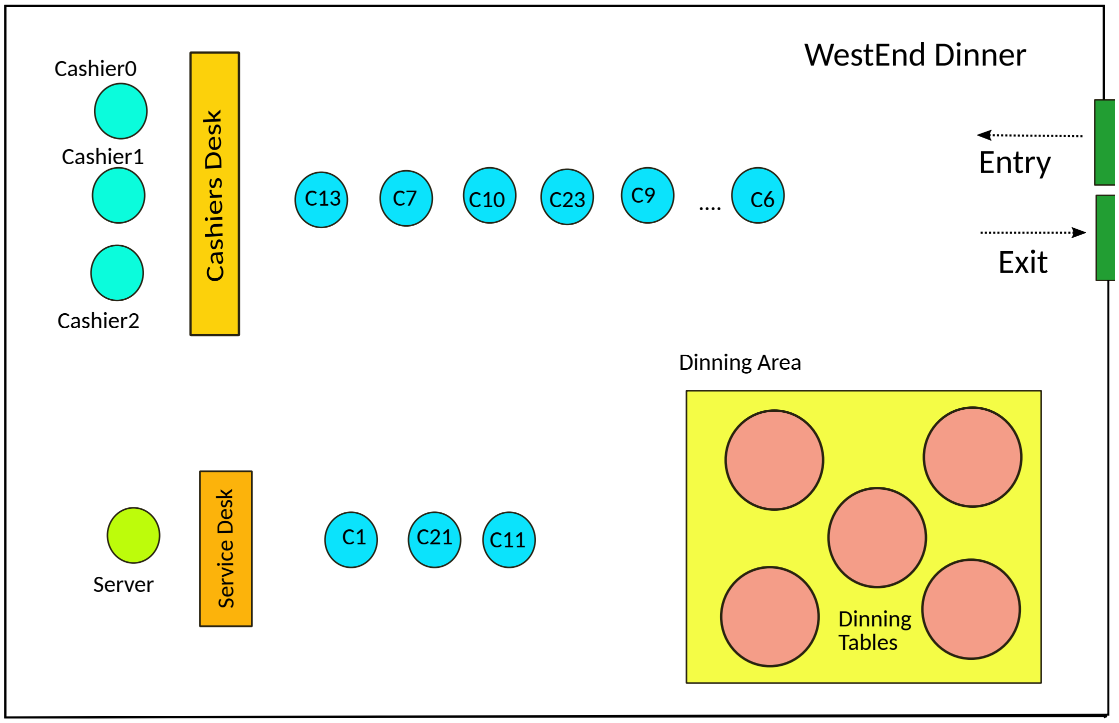

Figure 1: WestEnd Dinner Operation (Coordinator process not visible)

While working in the above lay-out, your processes must follow the restrictions below at all times:

- Clients line-up to give order in a *FIFO* fashion.
- If upon arrival, a client finds `maxPeople` waiting, the client simply departs.
- There are maximum k cashiers available to take orders. Once a cashier is free, he calls up the next-in-line customer.
- If no clients are in the queue, a cashier who is idle make take a break for up to t_{break} seconds. When she returns and still finds nobody in the queue, she can repeat the break.
- The interaction between a cashier and a client may take up to $t_{service}$ seconds.
- The cashier records the type of meal a client wants and enters the respective record in the dinner's "database"; this record consists of the item ordered, its value and the client *id*. Portions of this database may be kept in the shared segment of the memory (such as statistics); the full listing of orders may be preserved in a file.
- Each client awaits her food for a time period that lasts between the *min* and *max* of its preparation (in seconds see *Appendix A*).

- The time required for a server to dispense the order to a specific client is randomly selected for up to t_{server} seconds. During this time both client and server are engaged and cannot do anything else.
- A client remains in the dinning area for up to t_{eat} seconds and then exits.
- After the last client exits the shop, both the server and the cashiers may depart (i.e., cease to exist). The dinner manager process then has to generate the following information:
  - For each client, her id, the time she spent in the shop, the time in waiting and the money she spent.
  - Average waiting time for all clients after entering the dinner (or joining the cashiers queue) and until they leave the restaurant.
  - What are the top-5 most popular menu items and how much revenue each has generated.
  - Total number of clients visiting the dinner and revenue generated in the day.

Designing your Programs:

You are free to adopt any structure you wish for your client/cashier/server programs. Also you may introduce a (limited) number of semaphores, auxiliary structures and possibly other executables (if deemed appropriate). For instance, it would be a good idea to develop a program that up-front creates a shared segment, initializes the structure of statistics to be compiled, and helps maintain content (for the client transactions recorded). The above program could also introduce the required semaphores as well as any auxiliary data structures you have to put in place in order to achieve the operation of WestEnd Dinner. The program could also dump all information about individual orders in a designated file (your simplified "database"). At the very end, the program could help the last cashier (i.e., manager) produce all statistics and summary figures required.

Also, it would be good to create a program that at the very end *cleans up* and *purges* shared memory segment and semaphores so that system resources get properly released. As a matter of fact, this purging of shared memory segment is imperative; otherwise, the system will be depleted and it will be ultimately unable to furnish additional segments.

Invocation of your programs:

Your `coordinator` program could be invoked as follows:

```
./coordinator -n MaxNumOfCashiers
```

where

`-n MaxNumOfCashiers` indicates the *maximum* number of cashiers that can be working in WestEnd Dinner. In actuality, you may have fewer cashiers involved; a typical such number for your runs could be 3.

This `coordinator` program is also responsible for creating shared memory and pther global resources before clients start coming in to the restaurant, compiles the statistics of the business day, and finally releases all acquired system resources in a graceful manner before the WestEnd Dinner operation ceases.

Your `cashier` program could be invoked as follows:

```
./cashier -s serviceTime -b breakTime -m shmid
```

where

`./cashier` is the name of your (executable) program to be executed by a cashier process,

`-s serviceTime` indicates the maximum time that this specific cashier takes to provide service for a client (time randomly selected in the interval [1 .. `serviceTime`] seconds),

`-b breakTime` designates the maximum time period that the process spends in "break" (time randomly selected in the interval [1 .. `breakTime`] seconds), and finally,

`-m shmid` offers the identifier of the shared memory in which any required structures reside in.

Your `client` program could be invoked as follows:
`./client -i itemId -e eatTime -m shmid`
where

`./client` is the name of your (executable) program,

`-i itemId` designates the menu ID of the meal,

`-e eatTime` indicates the maximum time that a client spends eating her food in the dinning area before heading out (randomly selected in the interval [1 .. `eatTime`] seconds),

`-m shmid` offers the identifier of the shared memory in which any required structures reside in.

Similarly the `server` program could be invoked as follows:
`./server -m shmid`
where

`./server` is the name of the respective executable,

`-m shmid` offers the identifier of the shared memory in which any required structures reside in.

There is only 1 server in the restaurant.

You may introduce additional (or eliminate) flags of your choice in the invocation of the above programs.

What you Need to Submit:

1. A directory that contains all your work including source, header, make, a `README` file, etc.
2. A short write-up about the design choices you have taken in order to design your program(s); 1-2 pages in `ASCII`-text would be more than enough.
3. All the above should be submitted in the form of `tar` or `zip` file bearing your name (for instance `AlexDelis-Proj3.tar`).
4. Submit the above tar/zip-ball using *NYUclasses*.

Grading Scheme:

Aspect of Programming Assignment Marked	Percentage of Grade (0–100)
Quality in Code Organization & Modularity	20%
Correct Execution for Queries	30%
Addressing All Requirements	30%
Use of Makefile & Separate Compilation	7%
Well Commented Code	8%
Use of Linux	5%

Noteworthy Points:

1. The project is to be done ***individually***.
2. You **have to use *separate compilation*** in the development of your program.
3. Although it is understood that you may exchange ideas on how to make things work and seek advice from fellow students, **sharing of code is *not allowed***.
4. If you use code that is not your own, you will have to provide ***appropriate citation*** (i.e., explicitly state where you found the code). Otherwise, plagiarism questions may ensue. Regardless, you have to fully understand what and how such pieces of code do.

Appendix A:

The menu of WestEnd Dinner consists of the following items:

ItemID	*Description*	*Price*	*Min Time (seconds)*	*Max Time (seconds)*
1	BBQ-Chicken-Salad	8.95	18	24
2	Spinach-Power	9.15	12	16
3	Garden-Salad	4.75	10	13
4	Steak-Blue-Cheese	7.25	12	15
5	Ceasars-Salad	6.75	13	15
6	Chicken-Salad	9.15	15	21
7	Mongolian-BBQ-Plate	9.75	21	31
8	Club-Sandwich	6.35	13	18
9	Belgian-Cheese-Sub	10.25	15	19
10	Rio-Grande-Beef-Sub	9.35	18	20
11	Argentine-Asado-Club	11.75	23	30
12	Sierra-Sub	10.38	12	15
13	Avocado-BLT	8.05	12	13
14	Soup-de-Egion	3.20	11	15
15	Soup-de-Sur	2.75	6	9
16	Coffee	1.25	2	4
17	Hot-Tea	1.05	1	4
18	Hot-Chocolate	2.15	1	2
19	Mocha	3.25	2	3
20	Cafe-Late	3.75	5	7