CS39002 Operating Systems Laboratory
Spring 2025

Lab Assignment: 6
Date: 05–March–2025

Shared memory and semaphores

Barrackpore Food Bar (BarFooBar) is an upscale dine-in restaurant that accepts lunch customers from 11:00am to 3:00pm. It has two *cooks* *C* and *D* who prepare food in parallel. The restaurant has only ten tables, each capable of accommodating a customer party (henceforth abbreviated just as a *customer*) consisting of one to four individuals. No table is shared by multiple customers at any time. Only after one customer finishes and leaves, another new customer can occupy that table. The restaurant has five *waiters* *U*, *V*, *W*, *X*, and *Y* who serve the customers in a circular fashion, that is, *U* serves the first customer, *V* the second customer, *W* the third customer, *X* the fourth customer, *Y* the fifth customer, and then again *U* serves the sixth customer, *V* serves the seventh customer, and so on. Each customer, upon arrival to BarFooBar, checks whether there is any empty table. If not, it leaves. Otherwise, it uses an empty table, seeks help from the next serving waiter, gives order, and waits until food is served. A waiter, upon receiving the order from a customer, queues a cooking request. The two cooks *C* and *D* read requests from the queue, prepare food in the orders, and notify the waiters from whom the orders came. The waiters serve the prepared food to the corresponding customers. After a customer eats, it leaves, releasing the table it occupies.

In this assignment, you simulate a lunch session using multiple processes that share information using a shared array of integers, and synchronize using semaphores. All times are given in minutes relative to the beginning of the session at 11:00am. However, unlike the discrete-event system simulation of Assignment 3, you make a timed simulation. Your simulation must not run for four-plus hours (11:00am to 3:00pm+). Scale down each minute to a small time like 100 ms, so that the simulation finishes within a minute. For today's computers, 100 ms is a lot of time, and should be sufficient for a few operations to be done in a *minute* by the processes in this assignment.

The processes

Each entity in the simulation is to be implemented by a process. The behaviors of these processes are now elaborated.

Cooks *C* and *D*

Each cook is a process. These processes are launched first (before any waiter or customer process). A cook runs the following loop.

> Repeat until no more cooking requests can come on the day (see *End of session* below):
> {
>   Wait until woken up by a waiter submitting a cooking request.
>   Read a cooking request (waiter no, customer ID, count of persons for that customer).
>   Prepare food for the order (five minutes needed for each person).
>   Notify the waiter in the order that food is ready.
> }

Write a program *cook.c* to implement this behavior. The program launches a wrapper process. This is the first program that you run from a terminal. The wrapper creates two cook processes *C* and *D*, each of which jumps to the function *cmain*() that implements the above pseudocode for each cook. Note that *cmain*() does not return to the *main*() function of the wrapper process (the parent of cooks). Do not use *exec*. The parent should wait for *C* and *D* to terminate.

The cook program must be the first to run, and it is the duty of the wrapper (parent) to create the IPC resources (shared memory and semaphores) explained later. All other processes (including the cooks *C* and *D*) will only use these resources. Use *ftok*() with appropriate parameters so that unrelated processes can share the *shmid* and the *semid*'s.

Waiters *U*, *V*, *W*, *X*, and *Y*

The pseudocode for each waiter is given below.

```
Repeat until there are no more customers to serve on the day (see End of session below):
{
    Wait until woken up by a cook or a new customer.
    If a signal from a cook is pending, then do: {
        Serve food to the customer of that order (wake up the waiting customer).
    } else if a signal from a new customer is pending, then do: {
        Read details of the customer (number and count) from the waiter's queue.
        Take order from the customer:
            This involves no real operations.
            Assume that each order collection takes one minute.
            The waiter simulates order collection by waiting for one minute.
        Write the order (waiter_id, customer_id, customer_cnt) to the cooks' queue.
    }
}
```

Write a program *waiter.c* for the waiters. This program again launches a wrapper process which creates five child processes *U*, *V*, *W*, *X*, and *Y*, each of which jumps to a function *wmain*() that implements the above pseudocode for a waiter. The function *wmain*() does not return to the *main*() function of the wrapper process (parent of the five waiters). Do not use *exec*. The parent waits until all the five child processes terminate.

This is the second program to run. Recall that *cook* runs in one terminal. Launch *waiter* in a second terminal.

The customers

Each customer runs the following pseudocode.

```
If the time is after 3:00pm, leave.
If no table is empty, leave.
Otherwise, do the following: {
    Use an empty table.
    Find the waiter to serve.
    Write (customer_ID, customer_cnt) to that waiter's queue.
    Signal the waiter to take the order.
```

Wait for the waiter to attend (signal from the waiter).
Place order (no real operation).
Wait for food to be served (signal from that waiter).
Eat food (this takes 30 minutes irrespective of the count of individuals).
Free the table for future customers (if any).
Leave.
}

Write a file *customer.c*. This program launches a wrapper process which is the parent of all customer processes. The parent reads the information of all customers from a text file *customers.txt*. This file lists all customers chronologically with respect to their arrival times. Each line consists of the following information about a customer (party).

- Customer_ID (a positive integer in the sequence 1, 2, 3, . . . in the order of arrival time)
- Arrival time (a non-negative integer, minutes after 11:00am)
- Count (an integer in the range 1 – 4 standing for the number of individuals in the party)

The file is terminated by a line starting with –1 (an invalid Customer_ID). A random customer generator will be provided to you in the file *gencustomers.c*.

The parent of all customer processes reads the file line by line, and forks a child process for each customer. Each child (a customer) jumps to the function *cmain*() which implements the above pseudocode for a customer, and does not return to the *main*() function of the parent. The three items about the customer (ID, arrival time, and count) as read from the input file are passed as arguments of *cmain*(). The parent process must maintain the time (in the simulated sense with one minute scaled down to 100 ms). That is, if the arrival times of two consecutive customers are different, then the parent should wait for the interval standing for the difference of these two arrival times.

This is the last program to start running (cooks and waiters must be ready beforehand to serve customers). Do it in a third terminal. Note also that the last process to terminate is a customer that is the last to finish eating. The parent of the customer processes must wait until all the forked child processes terminate. The parent then removes the IPC resources (shared memory and semaphores) from the system, before it exits.

End of session

For customers, the input file *customers.txt* is an indication of when the customers arrive and leave (with or without the service of the restaurant). But the cooks and the waiters must know when to stop. The first criterion is that the (simulated) time must be at least 3:00pm.

A cook can leave when it is after 3:00pm and the cook queue is empty. Additionally, the last cook sees that there are no orders to be served. It then wakes up all the five waiters.

For a waiter, the termination criterion consists of time later than 3:00pm and no new-customer requests pending in that waiter’s queue. The waiter can check that either after serving food to the last of its customers or when it is woken up by the last terminating cook.

The above termination criterion does not handle the situation when the restaurant has no customers at 3:00pm. Assume that BarFooBar is a very busy restaurant, and this situation will not happen in practice.

Simulation of time

In this assignment, time is simulated by time (not by an event queue). Each minute is to be scaled down to a small interval like 100 ms. Every event should happen using that scaled-down definition of a minute. The simulated time (an integer, number of minutes after 11:00am) is maintained in a shared variable *time*. This variable is initialized to 0 (by the parent of the cooks), indicating that the restaurant opens for customers at 11:00am. A new customer sets *time* to its arrival time. Waits on mutexes and semaphores do not change *time*. The waits that simulate some delay changes *time*. This includes the following three delays.

- The parent of customers waits between two consecutive arrivals as given in the input file.
- A waiter takes order from a customer. This takes one minute.
- A cook prepares food for c customers in an order. This takes $5c$ minutes.
- A customer eats after food is served to it. This takes 30 minutes.

Each such delay is implemented by using *usleep*() with an appropriate argument. The process calling this should record the current *time* (call it *curr_time*) before it goes to the sleep. When it wakes up (end of sleep, not a signal), it adds the delay time (in minutes) to *curr_time*, and stores that sum in the shared variable *time*. Notice that *time* may have been changed (by some other process) during the interval [*curr_time*, *curr_time* + *delay*]. So *time* should be read before going to sleep (not after waking up from the sleep).

Keep a check that no process attempts to change *time* to a strictly smaller value (changing *time* to the same value is possible). If so, issue a warning message that setting *time* fails (and do not set *time*). In a proper simulation (assuming that the scaled-down interval for one minute is sufficient for all the operations done by the processes in that *minute*), this should never happen. If your program issues this warning, address the problem. If your computer (perhaps virtual) is very slow, try by changing 100 ms to one second (or slightly longer). If the problem persists, the source must be elsewhere. After all, one second is electronically a huge time for a few calculations and a few queue and semaphore operations, and there are not too many processes doing those in any minute.

IPC resources

Shared-memory segment

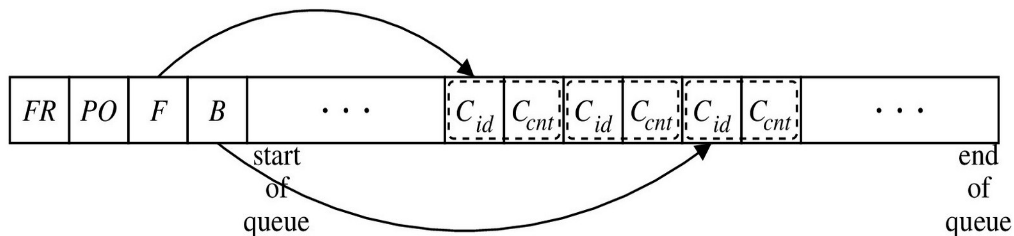
All the processes in this assignment share information by writing to and reading from shared memory. Use a single shared-memory segment (call it *M*) that should be spacious enough to store all the shared data structures needed. This includes some shared int variables along with some queues, as explained below. Assume that we can have no more than 200 customers per day, so each single waiter would handle at most 40 customers. Some customers leave owing to unavailability of tables without involving any waiter at all. In any case, plan for each waiter capable of handling about 100 customers. Organize the shared-memory segment *M* as follows.

M[0]	time	to be initialized to 0
M[1]	number of empty tables	to be initialized to 10
M[2]	waiter number to serve the next customer	to be initialized to 0
M[3]	number of orders pending for the cooks	to be initialized to 0
. . .		

You may store, as per your need, other individual int variables at the beginning of *M*. Let us reserve the first 100 int cells of *M* for that purpose.

Additionally, *M* should store six queues: one for the cooks, and five others for the waiters. Each such queue can grow to a limited size known beforehand. You may implement each queue as a circular queue at certain location on *M* (known beforehand to all the relevant processes). However, you may go for a simpler implementation. Again because of the limited total size, the queue may be allowed to move forward in *M* as time passes.

Each element in a waiter's queue consists of a customer ID (positive integer) and the count of individuals in that customer party (an integer in the range 1 – 4). Upon arrival, a new customer puts these two integers at the back of the queue of its waiter (read from *M*[2]). We also need to maintain the indices of the front and back of the queue. In short, a waiter queue will look as follows.



In addition to this queue, a waiter needs to maintain two int values. A waiter is woken up by a cook (when food is ready) or by a new customer willing to place order. Upon waking up, the waiter needs to know the reason why it is woken up. The cook writes the customer ID (for which food is just prepared) in the cell marked as *FR*. The cell *PO* may be used by the customers to indicate how many customers are waiting to place order. After waking up, a waiter first attends *FR* (if available). If not, a single *PO* request is served. Since a waiter handles less than 100 customers, use a 200-int contiguous area of *M* for storing all these items, for each waiter.

A cooking request consists of three items: the waiter who supplies the order, and the customer ID and the customer count in that order. The cooking-request queue can be implemented by storing the requests in consecutive triples. Again, this can be a moving array (circular arrays may be avoided for simplicity). Since there are at most 200 customers, 600 int cells suffice. You need two additional cells for storing the front and the back indices of the queue.

To sum up, the shared-memory segment is organized as follows.

First 100 cells	Global int variables (you don't have to use all the cells)
Cells 100 – 299	For waiter *U*
Cells 300 – 499	For waiter *V*
Cells 500 – 699	For waiter *W*
Cells 700 – 899	For waiter *X*
Cells 900 – 1099	For waiter *Y*
Cells 1100 – 2000	For cooking queue

Decide the size of the shared-memory segment (for *shmget*) accordingly.

Semaphores

mutex	a single semaphore for protecting *M*	to be initialized to 1
cook	a single semaphore for *both* the cooks	to be initialized to 0
waiter	five semaphores, one for each waiter	to be initialized to 0
customer	a set of semaphores, one for each customer	to be initialized to 0

The wait and signal operations on these semaphores are explained in connection with the processes.

Do not use POSIX semaphores (*sem_open*, *sem_wait*, *sem_post*). Use legacy System V semaphores (*semget*, *semctl*, *semop*), because we (the teachers) want you to grow familiarity with these calls.

Sample makefile

```
all:
	gcc -Wall -o cook cook.c
	gcc -Wall -o waiter waiter.c
	gcc -Wall -o customer customer.c

db:
	gcc -Wall -o gencustomers gencustomers.c
	./gencustomers > customers.txt

clean:
	-rm -f cook waiter customer gencustomers
```

Sample Output

A sample input file *customers.txt* and the transcripts in the three windows (cook, waiter, and customer) will be provided to you as a zip archive. You may also combine the three transcripts into a single one by running the three programs in the same terminal. First, run *cook* in the background. Then, run *waiter*, again in the background. Finally, run *customer* (in foreground or background). The combined transcript is also supplied in the zip archive.

Submit a single tar/tgz/zip archive containing all your c/cpp sources.