In this assignment, you will implement a program with Unix semaphores and shared memory to simulate the work of several airline reservation agents connected to the Real-Time airline's central computer.

Each travel agent will be represented by a process and the input of that process will list all the reservations/ticketing made by that agent. An agent can make reservations, ticketing (selling of seats) and cancellation of reservations/ticketing. Two or more agents may be doing the same transactions at around the same time. Obviously, reservations/ticketing to the same flight must be performed atomically (mutual exclusively). If an agent tries to reserve/ticket a seat that has already been taken, then disallow this action and print a "seat taken" message. If a transaction tries to operate on a non-existent flight, print an error message. To ensure these concurrent operations yield correct results, you are to use Unix semaphores to control access to flights, which are stored in shared variables. Each individual flight should be controlled by at least one semaphore. Also, the information database for each flight is stored in shared memory.

To simulate (1) the transmission delay between the airline's central computer and an agent's computer terminal and (2) the processing of each transaction, the first four lines in the body of each agent specify the required total execution time (in seconds) for each of the four operations performed at that agent. In your implementation, each specified time is the length of the critical section for the corresponding transaction.

Valid transactions are:

- `reserve flight_number seat_number passenger_name`
- `ticket flight_number seat_number passenger_name`
- `cancel flight_number seat_number passenger_name`
- `check_reservations passenger_name`  // show seats reserved or ticketed

Agents don't need to reserve a seat before they ticket it but they cannot cancel a seat that was not reserved or ticketed. Seat numbers consist of one or two digits followed by a single letter starting with 1A, 1B and so forth. Passenger names may contain spaces and will always be terminated by a return.

**INPUT**

The input is as follows:

```
begin flights  // list of all valid flights
RT flight_number number_of_rows_of_seats number_of_seats_per_row
RT flight_number number_of_rows_of_seats number_of_seats_per_row
...
end flights
begin agent  // all agent inputs follow the same pattern
reserve reserve_time  // in seconds
ticket ticket_time  // in seconds
cancel cancel_time  // in seconds
check_passenger check_time  // in seconds
reserve flight_number seat_number passenger_name
other agent operations
...
end agent
begin agent
...
end agent
...
```
OUTPUT
Your output should consist of:
1. a one-line summary of the outcome of each transaction
2. a final report listing the seat assignments of each flight with passenger names.

HINTS
1. Don’t forget the following includes:
   #include <sys/types.h>
   #include <sys/ipc.h>
   #include <sys/shm.h>
   #include <sys/sem.h>
2. To get nbytes bytes of shared memory and attach them to the address space of your process use:
   int shmid; // will contain the shard memory id
   long key; // use your telephone number
   int nbytes; // the size of your shared memory segment
   char *pmem; // will point to your shared memory segment
   shmid = shmget(key, nbytes, 0666 | IPC_CREAT);
   pmem = shmat(shmid, 0, 0);
   To test for error, if (pmem == (char *)(-1)) ...
3. To destroy a shared memory segment, use:
   shmdt(pmem); // must detach before destroying
   semctl(shmid, 0, IPC_RMID, 0);
4. To create an array containing nsems semaphores, use:
   int semid; // will contain the semaphore id
   long key; // use your telephone number
   int nsems; // should be no greater than 6 or 7
   semid= semget(key, 1, 0666 | IPC_CREAT);
   You can use the same key for your shared memory segment and your semaphore array. The initial value of your semaphores will always be zero.
5. To do a P() operation on semaphore i of array sid, use:
   struct sembuf sb;
   sb.sem_num = i; // identifies which semaphore you want to wait on
   sb.sem_op = -1; // because we want to decrement the semaphore
   sb.sem_flg = 0; // because it works
   semop(semid, &sb, 1); // do not forget the ampersand
6. To do a V() operation on semaphore i of array sid, do as above with 1 instead of -1
7. To destroy a semaphore array, use:
   semctl(sid,0, IPC_RMID, 0);
8. Anytime your program has crashed, do $ipcs to see which semaphore arrays and shared memory segments you have on the system. Delete them immediately using
   ipcrm -m shmid
   ipcrm -s semid

People who leave semaphore arrays and/or shared memory segments on the system when they log out risk (a) a lowered grade for their third assignment and (b) the loss of their Bayou account.

This document was updated last on Friday, November 22, 2002.