COSC 3360/6310 - Operating Systems Fall 2020

Programming Assignment 2: Unix/Linux Semaphores and Shared Memory; EDF and LLF Scheduling

Due Date: Friday, November 6, 2020, 11:59pm CST

In this assignment, you will implement several programs with Unix/Linux semaphores and shared memory to handle airline reservations taken by different agents connected to the "Fall-OS" airline's central computer.

Suppose each travel agent is represented by a process and the body of the process consists of all the reservations/ticketing made by that agent. An agent can make reservations, ticketing (selling of seats), 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 or seat, print an appropriate 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 five lines in the body of each agent specify the required total execution time (in milliseconds) for each of the five 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 name_of_passenger deadline d1
wait flight_number seat_number name_of_passenger deadline d2
ticket flight_number seat_number name_of_passenger deadline d3
cancel flight_number seat_number name_of_passenger deadline d4
check_passenger passenger_name deadline d5 /* show seats reserved or ticketed */
```

You don't need to reserve before you ticket a seat. The 'wait' transaction is like 'reserve' except that 'wait' will wait-list the passenger for the selected seat if this seat is currently not available, and is like 'ticket' if this seat is currently available. The agent with this 'wait' transaction continues to perform the next transaction regardless of the availablity of the selected seat. Note that if this seat is currently unavailable and later becomes available as a result of a 'cancel' transaction by another passenger, then this seat will be sold to the passenger who first executed the 'wait' transaction in case there are two or more passengers waiting for this seat. You can cancel a seat only after you have reserved/ticketed it.

Each transaction is followed by the keyword deadline and its numerical value (in milliseconds, relative to the start time of the process or process creation time) for completing this transaction.

The input is as follows:

```
n     /*  number of flights */
flight_number number_of_rows_of_seats number_of_seats_in_each_row
```

```
flight_number number_of_rows_of_seats number_of_seats_in_each_row
   :
   :
flight_number number_of_rows_of_seats number_of_seats_in_each_row
      /* there are n flights, a seat is labeled by number-letter
         starting with 1, like 1A, 5C, 18F   */

m     /* number of agents */
agent_1:
reserve reserve_time (in milliseconds)
wait reserve_time    (in milliseconds)
ticket ticket_time   (in milliseconds)
cancel cancel_time   (in milliseconds)
check_passenger check_time (in milliseconds)
  :
valid operations
  :
end.
  :
  :
atm_m:
reserve reserve_time (in milliseconds)
wait reserve_time    (in milliseconds)
ticket ticket_time   (in milliseconds)
cancel cancel_time   (in milliseconds)
check_passenger check_time (in milliseconds)
  :
valid operations
  :
end.
```

Implement three versions: (1) the first without considering passengers' deadlines; (2) the second with EDF scheduling of the transactions; and (3) the third with LLF scheduling of the transactions, where the remaining computation time of a process is the sum of the computation times of this process' remaining operations.

The output for each implementation after completing all transactions is: a report showing the transactions and resulting flight seat assignments with passenger names, names of passengers (if any) and the selected seats being wait-listed, and whether each transaction's deadline is met for every customer (if not, show the lateness in milliseconds).

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, use:

```
int shmid;
long key;
int nbytes;
shmid = shmget(key, nbytes, 0666 | IPC_CREAT);
```

To avoid conflicts, use your Student ID as key.

(3) To attach the shared memory segment to your address space, use:

```
char *pmem;
pmem = shmat(shmid, 0, 0);
```

To test for error, if (pmem == (char *)(-1)) ...

(4) To detach the shared memory segment from your address space before destroying the segment, use:

```
shmdt(pmem)
```

(5) To destroy a shared memory segment, use:

```
semctl(shmid, 0, IPC_RMID, 0);
```

(6) To create a single semaphore, use:

```
int sid;
sid = semget(key, 1, 0666 | IPC_CREAT);
```

Initial value is ZERO

(7) To do a DOWN (wait) operation, use:

```
struct sembuf sb;
sb.sem_num = 0;
sb.sem_op = -1;
sb.sem_flg = 0;
semop(sid, &sb, 1);
```

(8) To do an UP (signal) operation, do as above with 1 instead of -1

(9) To destroy a semaphore, use:

```
semctl(sid,0, IPC_RMID, 0);
```