CS-451 Operating System
The Gaming Parlor Problem

Summary of the Problem

Write a program that simulates the front desk and customers of the "Gaming Parlor" using semaphores and threads. The problem is as described below:

There are eight group of customers come in to a "gaming parlor" to play games. Each group comes into the parlor wanting to play their favorite game. Games involve different number of dice. Each group has a representative, group representative (a thread) that goes to the front desk (a thread) to obtain one or more dice depending on which game they are playing. The dice are used by the corresponding group while they are playing their game, and then returned to the front desk. The front desk is in charge of lending out the dice and collecting them after each game is finished.

The gaming parlor owns only 8 dice, which are available at the front desk before the first group comes in. No group *knows* how many dice there are and is *not allowed to change* the total number of dice at the front desk. Only front desk can assign dice and reclaim dice and thus change the number of dice.

Each group is assigned to play one of the following games: Backgammon, Risk, Monopoly or Pictionary. Listed below after each game in parentheses is the number of dice required to play that game.

Backgammon (4 dice)
Risk (5 dice)
Monopoly (2 dice)
Pictionary (1 die)

Here are the list of games of played by groups 1 to 8. Note that each game is played by two groups.

Group Number	Game Played
1	Backgammon
2	Backgammon
3	Risk
4	Risk
5	Monopoly
6	Monopoly
7	Pictionary
8	Pictionary

Each group plays its game 5 times (rounds). Once the game starts, the time to taken to play any of the game is between 0 to 10 seconds, generated randomly. After each game however, the

group MUST return its dice and ask for them it again, after each game. The groups leave the game parlor after playing their game 5 times.

For example, group 7 that plays Pictionary will play Pictionary 5 times. Pictionary needs 1 die. Therefore group 7 will ask for 1 die and play one game of Pictionary for anywhere between 0 to 10 seconds, return the die, and ask for it again. This request-return occurs 5 times by group 7 and then group 7 leaves the parlor

Each representative will *request* the corresponding number of dice, and, after using the dice for the specified period of time, will *return* the number of dice that they borrowed.

Synchronization Requirement:

Here are the synchronization requirements:

1. Regardless of the order in which the groups make their requests and return their dice, each die must never be allocated to more than one group at a time.
2. Groups must be allowed make requests even if not enough dice is available. However, it should never be the case that groups are allowed to proceed when there are too few dice.
3. *The game parlor must make the group wait until enough dice are available for that group.* This will completely avoid deadlocks.
4. Likewise, if a group has returned its dice, other groups which are waiting must be allowed to proceed once enough dice have become available.

With each group limited to playing only 5 games, all the outstanding dice will always get returned eventually and starvation can never occur.

Synchronization Coding Requirements

1. Any global variable read/writes must be surrounded by semaphore waits and signals so the variable is protected.
2. Do not make the number of dice a global variable. *Only front desk knows* the number of dice that exist.
3. The groups *do not know* how many dice the front desk currently has, they only make the request, and, when request is granted, they play game.
4. The groups cannot modify the number of dice since they do not know how many dice there are at the front desk.

What does my program do?

Simulate the front desk thread and the eight group representative threads. Use semaphores as the only synchronization primitives in your implementation (no locks or pause of sleep to be used in synchronization. Sleep should be used when playing the game however.).

If you use global data manipulated by multiple threads, their read/write must be surrounded by semaphore wait/signal.

What should the output look like?

Your program should print out when a group requests dice, obtains dice, returns dice and leaves the system. After each group's printout, Front Desk thread should print the number of dice available whenever number of dice it holds changes.

For example, a sample run might look like the following:

```
Front Desk: I have 8 dice available

Group 1: Requesting 4 dice for Backgammon.

Group 1: We are playing our first game of Backgammon

Front Desk: I have 4 dice available

Group 3: Requesting 3 dice for Risk.

Group 3: We are playing our first game of Risk

Front Desk: I have 1 dice available

Group 2: Requesting 4 dice for Backgammon

Group 1: We are done playing our first game of Backgammon.
Returning 4 dice.

Front Desk: I have 5 dice available

Group 2: We are playing our first game of Backgammon

Front Desk: I have 1 dice available

Etc. etc.
```

Your output order may vary from the above depending on the random game time, and, order of execution of the threads. Any correct output will ensure that Sychronization Requirement stated above is met.

Testing your the code

Once your program works as expected, to stress test the code, try the following

Add sleep(for random time) in different locations in the code to simulates threads running at different speeds.

With the restriction above, make sure that Sychronization Requirement stated above is met.

How do I get started with Threads?

Use the library pthread which contains all the thread related functions. Make sure you have #include <pthread.h> in your include list. While compiling the source file use **`-lpthread`** at the end of compilation to include the thread library as follows

```
%gcc main.c -o main.out -lpthread
```

`pthread_create` routine will create a thread. Once a thread is created it starts running the code specified. The order of execution or speed of each thread is unknown. The following code creates 5 threads.

```
int main()
{
    pthread_t thread_id[5];

   // The following code creates 5 threads.
   for (int i =0; i <5; i++)
        pthread_create(&thread_id [i], NULL, thread_task, (void *) i);

  // The following code makes sure the main program waits until all threads
have finished execution

  for (int i =0; i <5; i++)
     pthread_join(thread_id[i], NULL);
}

void thread_task(int i) {
    //Code for the thread task goes here

    pthread_exit(0); // this code returns to the corresponding pthread_join
issued in main()
}
```

How do I get started with Semaphores?

You will be using the library <semaphore.h> for the semaphore functions. You do not need to do anything special to compile the code with semaphore.h functions. The following functions would be most useful.

`sem_t mutex;`
`sem_init(&mutex, 0, 1);` // will initialize the value of the mutex to 1. You can initialize the mutex to any value of your choice.
`sem_wait(&mutex);` // If the mutex value is 1 or more the thread does not block, and it reduces the mutex value by 1; If the mutex value is 0 or below the thread executing this code will block until another thread wakes it up executing sem_post
`sem_post(&mutex)` // Increase the mutex value by 1 and unblock a thread that is blocked at sem_wait.

Do I have to create a demo files?

Yes. Make a demo of the program with no delays in the code except for playing the game. To stress test the code, add sleep at random points in the code and make a demo of the program. While submitting the code however the only sleep should be for playing the game.