Relatório 2º projeto de Sistemas Operativos

.Mecanismos de sincronização utilizados

Para sincronização de threads foram somente usados semáforos em três momentos diferentes, o acesso ao buffer unitário, o acesso à secção crítica de lock dos semáforos e o acesso à secção crítica de reserva de lugares.

O acesso ao buffer unitário para retirar as informções de um request é uma secção crítica, só um thread pode estar a executar essa secção de código num determinado momento, pois o acesso simultâneo de vários threads a esse buffer levaria a que várias bilheteiras processassem o mesmo request o que não é pretendido. Logo, para que tal não aconteça, o mecanismo de sincronização utilizado consiste em dois semáforos (sem_buffer_empty e sem_buffer_full) que permitem o acesso ao buffer só a um thread de cada vez. Este lock/unlock destes semáforos é feito na função enableOfficeTicket(void * arg).

```
// is there any pendent request ?
if (sem_wait(info->sem_buffer_full)){
  perror(NULL);
}
printf("Thread %d got access to request buffer!\n", myOfficeTicketID);
// the server may indicate there are pendent requests to unlock threads so that they can terminate
if (*(info->isTimeOut))
  break;

// copy the request from the buffer. releasing the buffer as fast as possible
Request myRequest = *req;
printf("Thread %d got the request from client %d\n", myOfficeTicketID, myRequest.clientID);
// set buffer as empty
if (sem_post(info->sem_buffer_empty)){
  perror(NULL);
```

Para a reserva de lugares pelos threads foi necessário um semáforo principal(sem_access) para evitar a ocorrência de race condition. Para cada seat é criado um semáforo. O semáforo principal(sem_access) restringe o acesso à condição crítica onde é feito o lock dos semáforos associados aos lugares/seats presentes na lista de preferências. Depois de todos os semáforos pretendidos serem bloqueados, o thread liberta esta secção crítica e começa a reserva do lugares.

```
// there can only be one thread locking seats at each time
// for that reason, there's the sem_access semaphore
sem_wait(info->sem_access);

// try to lock the seats mentioned on preference list
for (int i = 0; i < req->numSeatsPreferences; i++) {
  int seat_id = req->seatsPreferences[i];
  sem_wait(room->seats[seat_id - 1].sem_unlocked);
}

printf("Could lock all seats on pref list!\n");

// unlock access to seats
sem_post(info->sem_access);
```

Cada semáforo bloqueado permite o acesso à secção crítica de leitura e escrita de informação na struct Seat(lugar) associada. Todas as structs Seat e os semáforos sem_access, sem_buffer_full e sem_buffer_empty estão presentes numa outra struct Room(sala de espetáculos). Quando o request for

acabado, bem sucedido ou não, ocorre o unlock dos semáforos associados as seats/lugares que se tentou reservar.

.**Encerramento do servidor**

Quando é atingido o limite de tempo estabelecido o servidor indica aos threads a ocorrência de timeout através de uma flag. Os threads, que não estiverem bloqueados, após terem recebido tal ocorrência acabam de processar o seu request, fecham a FIFO associada ao cliente e terminam. Os threads que estiverem bloqueados num semáforo são desbloqueados e de seguida terminam imediatamente, fecham a FIFO associada ao cliente e não fazem o processamento de mais nenhum request.

```
printf("server: So timeout... let's wait for threads!\n");
// time is out
// because threads are likely stucked at sem_wait(full), we will increment it to unlock all threads
for(int i = 0; i < s.numOfficeTickets; i++) {
    sem_post(&sem_full);
    sem_post(&sem_access);
}
// now that threads unlocked, they will detect the time is out and return. now we call pthread_join
for(int i = 0; i < s.numOfficeTickets; i++) {
    pthread_join(officeTickets[i], NULL);
    printf("Thread %d returned!\n", i+1);
}
```

Após todas as bilheteiras terem fechado o servidor encarrega-se de destruir todas as estruturas de dados e comunicação utilizadas que ainda não tenham sido destruidas, ou seja, a FIFO requests, os semáforos sem_buffer_full, sem_buffer_empty e sem_access e as structs.

.**Estrutura das mensagens trocadas entre clientes e servidor**

As informações trocadas entre clientes e servidor são feitas através dos dois tipos de estruturas. As mensagens enviadas para o buffer unitário (cliente → servidor) são uma estrutura binária com dois packets, o primeiro packet tem as informações número de preferências de lugares, id do cliente e números de lugares pretendidos, enquanto o segundo tem os números das preferências de lugares do cliente. Estas informações são convertidas numa struct Request para serem processadas pelo servidor. A resposta do servidor para o cliente é feita escrevendo para a FIFO o código de erro, se o request for mal sucedido ou o número de lugares reservados juntamente com o número dos lugares reservados, se o request tiver sucesso.

```
// write answer on fifo
if (list_booked_seats == NULL)
{
  // the request was not successful
  // write a negative number
  printf("Failed request, writing a single number!\n");

  write(fd, &errorCode, sizeof(int));
  serverLogFailure(threadID, req->clientID, req->numSeats, req->seatsPreferences, errorCode);
}
else
{
  printf("Successful request, writing chunck of data\n");

  if(write(fd, &req->numSeats, sizeof(int)) <= 0) {
    if(errno)
      perror(NULL);
  }
  if(write(fd, list_booked_seats, sizeof(int)*req->numSeats) <= 0) {
    if(errno)
      perror(NULL);
  }
```