Trabalho Prático Nº2
Simulação de um sistema de reserva de lugares

Sistemas Operativos 2017/2018

Francisco Friande - up201508213
João Reis - up201203562
Pedro Silva – up201604470

Comunicação entre Client e Server

O servidor lê informação de todos os clientes através de um único fifo, de nome "requests" criado em "/tmp/" dado o seu carater temporário.

Tendo de enviar um pedido com vários elementos, passa cada um destes em escritas ao fifo diferentes, o programa ficaria vulnerável a escritas intercaladas de clientes diferentes e, por tal, corrupção de informação. Para evitar isso, optámos por criar um buffer que cada client cria para guardar toda a informação relativa ao pedido para apenas realizar uma escrita por pedido.

A estrutura deste buffer segue a orientação dos pedidos no enunciado:

"<PID_Client> <Num_Seats> <List_Of_Wanted_Seats>"

Onde:

- "PID_Client" é o PID do processo client que executa o pedido.
- "Num_Seats" é o número de lugares que o cliente deseja reservar.
- "List_Of_Wanted_Seats" é a lista com os números dos lugares das quais o cliente pretende reservar "Num_Seats lugares". Cada lugar está separado por um espaço.

Por exemplo:

"16413 2 12 13 64 100" significaria que o cliente com PID 16413 desejar reservar 2 dos lugares com identificador referido a seguir (12, 13, 64 ou 100).

```
// Request Goal: "<PID_Client> <Num_Seats> <List_Of_Wanted_Seats>"
char buffer[BUFFER_SIZE];
int n = 0;
// Create a default Request: "<PID_Client> <Num_Seats> 0"
snprintf(buffer, BUFFER_SIZE, "%d %d %n", getpid(), num_wanted_seats, &n);
// Add Seats: "<PID_Client> <Num_Seats> s_list[0] s_list[1] (...) s_list[num_pref_seat]"
for (int i = 0, c; i < num_pref_seat; ++i, n += c)
    snprintf(buffer + n, BUFFER_SIZE - n, "%d %n", s_list[i], &c);
//Request separator since the fifo itself works as a buffer to server
strcpy(&buffer[n - 1], "\n");
```

Comunicação entre Server e Client

A comunicação de resposta do servidor para o cliente é realizado através de fifos especificos de cada cliente, de nome ansXXXXX, onde XXXXX representa o PID do cliente.

Caso a reserva ocorra com sucesso, o servidor escreverá no fifo correspondente ao cliente em questão, o número de lugares que reservou e quais os seus identificadores.

"<Num_Seats> <Seat(0)> <Seat(1)> (…) <Seat(Num_Seats -1)"

Em caso de erro, o servidor escreverá nesse mesmo fifo a razão do erro:

"-1"	Quantidade de lugares pretendidos é superior ao permitido.
"-2"	Número de lugares pretendidos menor do que o número de identificadores dados.
"-3"	Identificadores dos lugadores pretendidos não são válidos
"-4"	Outros erros nos parâmetros
"-5"	Não conseguiu reservar o número de lugares pretendidos
"-6"	Sala cheia (não implementado)

Mecanismos de sincronização

Para evitar problemas de sincronização recorremos ao uso de mutexes.

Ao inicializar a estrutura de dados para os lugares (Array de structs Seat), é atribuído a cada lugar um valor de um mutex previamente criado.

A rodear as secções críticas (quando o estado do array "seats" é verificado ou editado), existe uma chamada para bloquear o acesso ao Seat **"pthread_mutex_lock(&seats[seatNum].mut);"** e uma para desbloquear o lugar para poder ser usado por outras threads **"pthread_mutex_unlock(&seats[seatNum].mut);"**.

No entanto, como a reserva (bookSeat) apenas é feita depois de verificar se o lugar está vazio (isSeatFree), a função isSeatFree não desbloqueia o Seat caso este esteja livre para evitar que outra bilheteira reserve esse lugar.

```
void bookSeat(Seat *seats, int seatNum, int clientId) {
    pthread_mutex_lock(&seats[seatNum].mut);
    if (!seats || !seats[seatNum].free) {
        pthread_mutex_unlock(&seats[seatNum].mut);
        return;
    }
    seats[seatNum].free = false;
    seats[seatNum].clientId = clientId;
    DELAY();
    pthread_mutex_unlock(&seats[seatNum].mut);
    return;
}
```

```
int isSeatFree(Seat *seats, int seatNum) {
    if (!seats)
        return -1;
    pthread_mutex_lock(&seats[seatNum].mut);
    DELAY();
    if (seats[seatNum].free)
        return 1;
    else
        pthread_mutex_unlock(&seats[seatNum].mut);
    return 0;
}
```

Não existirá problema em fazer lock duas vezes (que normalmente dá undefined behavior), visto que os mutexes foram criados usando o atributo "PTHREAD_MUTEX_ERRORCHECK", que garante que lock's posteriores ao primeiro não terão qualquer efeito.

```
pthread_mutexattr_t mut_attr;
pthread_mutexattr_init(&mut_attr);
pthread_mutexattr_settype(&mut_attr, PTHREAD_MUTEX_ERRORCHECK);
for (int i = 0; i < num_seats; ++i) {
    seats[i].free = true;
    seats[i].clientId = 0;
    pthread_mutex_init(&seats[i].mut, &mut_attr);
}
```

Encerramento do servidor

Ao encerrar o servidor o main thread termina todas as threads em execução.

```
// sinaliza todos os threads que estes devem terminar as operações correntes,
// libertar os seus recursos e terminar a execução
for (int i = 0; i < num_ticket_offices; ++i)
    pthread_cancel(ticket_office_thr[i]);

// esperar pelo término da execução de cada thread
for (int i = 0; i < num_ticket_offices; ++i) {
    if (pthread_join(ticket_office_thr[i], NULL) < 0)
        perror("pthread_join");
} printf("threads joined\n");
```

Para um thread bilheteira não ser interrompido enquanto está a processar um request, estes só permitem o seu próprio cancelamento quando estão à espera de novos requests, utilizando “pthread_setcancelstate()”.

A função “queue_take” faz com que o thread bloqueie à espera de adquirir um mutex para receber um request novo. De maneira a poder ser interrompido enquanto está bloqueado foi usado um cancelamento assíncrono, recorrendo a “PTHREAD_CANCEL_ASYNCHRONOUS”.

```
(;;) {
if (pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL))
    fprintf(stderr, "to%0*d: Unable to set cancel state. Thread may not be "
            "canceled and run in an infinite loop.\n", thr_id_width, args->ticket_office_id);
pthread_testcancel();

if (pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL))
    fprintf(stderr, "to%0*d: Unable to set cancel type to aynchrnous. Thread may not be "
            "canceled and run in an infinite loop.\n", thr_id_width, args->ticket_office_id);

printf("to%0*d waiting for msg in queue...\n", thr_id_width, args->ticket_office_id);
// get request from requests_buffer, this operation blocks until there is something on it
if (queue_take(args->requests_buffer, &request) < 0) {
    fprintf(stderr, "to%0*d: Error taking request from queue\n", thr_id_width, args->ticket_office_id);
    pthread_exit(NULL);
}
printf("to%0*d received message from queue: %s\n", thr_id_width, args->ticket_office_id, request);

if (pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL))
    fprintf(stderr, "to%0*d: Unable to set cancel state. Thread may be canceled "
            "sooner than expected, unable to free some resources.\n", thr_id_width, args->ticket_office_id);
```

Ao ser interrompido desta maneira o thread necessita de efectuar o bloqueio do mutex, por isso, foi utilizado a função demonstrada a seguir, para garantir que todos os threads libertam o mutex quando terminam.

```
pthread_cleanup_push(ticket_office_cleanup, args);
```

```
void ticket_office_cleanup(void *params) {
   struct ticket_office_thr_params *args = params;
   fprintf(args->slog, "%02d-CLOSED\n", args->ticket_office_id);
   pthread_mutex_unlock(&args->requests_buffer->mut);
   free(args);
   return;
}
```

Para finalizar, o main thread faz pthread join a todos as threads criadas e liberta todos os recursos utilizados durante a execução.