



Simulação de um sistema de reserva de lugares

Implementação de uma arquitetura cliente/servidor baseada em FIFOs

Bruno Carvalho
up201606517@fe.up.pt

João Alves
up201605236@fe.up.pt

João Agulha
up201607930@fe.up.pt

23 de Maio de 2018

Conteúdo

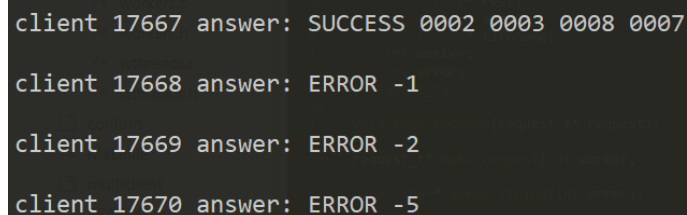
1	Estrutura das Mensagens	2
2	Mecanismos de Sincronização	2
3	Encerramento das Bilheteiras	3

1 Estrutura das Mensagens

Como não foi especificado o formato das mensagens a partilhar na *fifo ansXXXXX*, decidimos usar dois formatos, um para reservas bem sucedidas e outro para reservas mal sucedidas.

Se o *client* for compilado em modo *debug* (ver *debug.h*) então as mensagens recebidas no *fifo ansXXXXX* são escritas em *stdout*.

1. Para reservas de *N* lugares bem sucedidas:
SUCCESS *seat1 seat2 ... seatN*
2. Para reservas mal sucedidas:
ERROR *code*



```
client 17667 answer: SUCCESS 0002 0003 0008 0007
client 17668 answer: ERROR -1
client 17669 answer: ERROR -2
client 17670 answer: ERROR -5
```

Figura 1: Mensagens exemplo, ecoadas pelo *client* no terminal

2 Mecanismos de Sincronização

Usamos mecanismos de sincronização no acesso aos *seats* pelas bilheteiras e no acesso à *queue* de mensagens entre o *thread* principal e as bilheteiras.

Para salvaguardar do acesso simultâneo aos *seats*, cada *seat* tem um *mutex* associado. O *mutex i* do *seat i* serve de fila de acesso unitário a todas as bilheteiras a tentar aceder a esse mesmo *seat*. Como são usados *mutex* POSIX, não há garantia de espera limitada.

Para gerir a *queue* de mensagens, é usado um esquema à semelhança do *Reader-Writer* estudado nas aulas. São usados dois *mutexes* para gerir a permissão de escrita e de leitura, cada um com o seu semáforo auxiliar: o primeiro – *not_full_sem* – que conta o número de espaços disponíveis para escrita, e outro – *not_empty_sem* – que conta o número de mensagens por ler. Um escritor (o *thread* principal) que queira escrever tem de tomar posse do *mutex* de escrita e, se necessário, esperar no primeiro semáforo por um espaço livre; já um leitor (uma bilheteira) tem de tomar posse do *mutex* de leitura e, se necessário, esperar no segundo semáforo por uma mensagem nova.

A especificação requer que a *queue* de mensagens tenha tamanho unitário, mas a implementação suporta um *queue* de qualquer tamanho, vários leitores e escritores.

```
int is_seat_free(int seat_num) {
    assert(is_valid_seat(seat_num));

    lock_seat(seat_num);

    int ret = isSeatFree(seats, seat_num);

    unlock_seat(seat_num);

    return ret;
}
```

Figura 2: Acesso aos *seats*

```
static void _read_message(const char** message_ptr) {
    lock_read_access();
    wait_for_message();

    *message_ptr = message_queue[queue_read_p];
    queue_read_p = (queue_read_p + 1) % queue_size;

    post_new_space();
    unlock_read_access();

    lock_unread();
    --queue_unread;
    unlock_unread();
}
```

Figura 3: Acesso à *queue*

3 Encerramento das Bilheteiras

O único escritor do *queue* é o *thread* principal, e este escreve as mensagens lidas do *fifo requests* diretamente para esse *queue* sem qualquer processamento prévio. Para terminar as bilheteiras, o *thread* principal escreve no *queue* uma mensagem específica definida pela implementação, de momento `WORKER_EXIT`, por cada bilheteira.

Por sua vez, as bilheteiras que lêem esta mensagem identificam-na e terminam a sua execução de forma natural.

A terminação natural do servidor por *timeout* ou por cancelamento no terminal (Ctrl-C, ...) são equivalentes, e chamam `exit` diretamente. Existem vários **atexit handlers** instalados, o primeiro a ser executado fecha e destrói imediatamente o *fifo requests*, o segundo envia as mensagens de terminação às bilheteiras e espera que elas terminem a execução normalmente (o que pode demorar tendo em consideração `DELAY()`). Os restantes libertam toda a memória reservada para as estruturas de dados usadas, *strings* guardadas, *mutexes*, etc.

```
static int worker(int id) {
    slog_worker_open(id);

    while (true) {
        const char* message;
        read_message(&message);

        if (is_exit_command(message)) break;

        request_t* request = make_request(id, message);

        process_request(request);
        answer_client(request);
        slog_request(request);
        free_request(request);
    }

    slog_worker_exit(id);
    return 0;
}
```

Figura 4: Bilheteiras identificam mensagens

```
// Handler for SIGHUP, SIGQUIT, SIGTERM, SIGINT
static void sighandler_kill(int signum) {
    //if (PDEBUG) write(STDOUT_FILENO, str_kill,
    exit(EXIT_FAILURE);
}

// Handler for SIGALRM
static void sighandler_alarm(int signum) {
    //if (PDEBUG) write(STDOUT_FILENO, str_alarm,
    exit(EXIT_SUCCESS);
}
```

Figura 5: Signal Handlers