Relatório 2º Trabalho Sistemas Operativos 2018/19

Trabalho realizado por:

Diogo Machado —> up201706832@fe.up.pt

Eduardo Ribeiro —> up201705421@fe.up.pt

Eduardo Macedo —> up201703658@fe.up.pt

Neste relatório irão ser discutidos os seguintes tópicos:

- A estrutura das mensagens trocadas entre cliente e servidor (e vice-versa);
- Os mecanismos de sincronização utilizados;
- A forma como é feito o encerramento do servidor.

Estrutura das mensagens trocadas

Relativamente à estrutura das mensagens em formato TLV trocadas entre o cliente e o servidor (e vice-versa), foram utilizadas as estruturas que nos foram dadas no início do projeto, ou seja, as structs tlv_request_t e tlv_reply_t, para definir as mensagens de pedidos de utilizador e respostas do servidor, respetivamente.

Para cada programa do utilizador que é executado, é retirada a informação dos argumentos da linha de comandos, de modo a preencher os diferentes campos das structs que depois irão constituir a struct tlv_request_t. À medida que a informação é extraída e recolhida, é descoberto o tipo de operação característico do pedido, sendo que os campos **type** e **length** são atualizados de modo a, quando o pedido for enviado para o servidor, este ocupe o nº mínimo de bytes para um certo tipo de pedido.

Ex: envio de pedidos por parte do utilizador, para o servidor.

```
int totalLength = sizeof(op_type_t) + sizeof(uint32_t) + tlvRequestMsg.length;
int n;

// send request to server program
if((n = write(fdFifoServer, &tlvRequestMsg, totalLength)) < 0) {
    replyMsg.value.header.ret_code = RC_SRV_DOWN;
}
```

A mesma estratégia é adotada no servidor, quando se constrói as mensagens de resposta aos pedidos dos utilizadores, com a struct tlv_reply_t.

Relativamente à leitura, tanto dos pedidos por parte do servidor, como das respostas por parte do utilizador, são feitas em 3 partes separadas, de modo a ler estritamente os bytes necessários: primeiro é lido o **type** (que é enviado primeiro), seguidamente a **length**, e por fim o **value**. Os dois primeiros campos têm tamanho fixo; o tamanho do campo **value** é indicado pelo campo anterior, **length**; por isso é que se lê os três campos separadamente.

Ex: leitura dos pedidos no servidor.

```c
int readRequest(tlv_request_t* request) {

    int n;

    // reads request type
    if((n = read(fdFifoServer, &(request->type), sizeof(op_type_t))) < 0) {
        perror("Read request type");
        exit(EXIT_FAILURE);
    }

    if (n == 0) return 1;

    // reads request length
    if((n = read(fdFifoServer, &(request->length), sizeof(uint32_t))) < 0) {
        perror("Read request length");
        exit(EXIT_FAILURE);
    }

    if (n == 0) return 1;

    // reads request value
    if((n = read(fdFifoServer, &(request->value), request->length)) < 0) {
        perror("Read request value");
        exit(EXIT_FAILURE);
    }

    if (n == 0) return 1;

    return 0;
}
```

Mecanismos de sincronização

Relativamente aos mecanismos de sincronização, estes dividem-se em dois diferentes tipos/propósitos, tal como era indicado no enunciado do trabalho: os mecanismos de sincronização da **fila de pedidos**, e os de sincronização do **array de contas**.

Em relação à **fila de pedidos**, era pedido no enunciado para ser utilizada a estratégia do produtor-consumidor. Como tal, foi implementada uma solução semelhante à apresentada nos slides das aulas teóricas, na qual são utilizados **mutexes** e **variáveis de condição**. Estas são as estruturas utilizadas:

Fig: array de pedidos, bem como índices de acesso e outros valores.

```
tlv_request_t requests[MAX_REQUESTS];
int bufferIn = 0, bufferOut = 0, items = 0, slots = MAX_REQUESTS;
```

Fig: mutexes e variáveis de condição relacionadas com a fila de pedidos.

```
// sync mechanisms - producer/consumer
pthread_mutex_t buffer_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t slots_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t slots_cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t items_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t items_cond = PTHREAD_COND_INITIALIZER;
```

Relativamente ao **array de contas**, era dito no enunciado que esta parte da sincronização do programa poderia ser feita como achássemos melhor, sendo dada liberdade para implementar a nossa própria solução. A solução que decidimos implementar foi a utilização de um **array de mutexes**, um para cada conta, de modo a termos acesso exclusivo a uma conta (ou duas, no caso da transferência) quando for necessário fazer uma operação sobre ela(s).

Fig: array de mutexes para ter acesso exclusivo às contas.

```
// sync mechanisms - bank account array - one mutex per account
pthread_mutex_t accountMutexes[MAX_BANK_ACCOUNTS];
```

Um aspeto importante da nossa implementação, a nosso ver, é a forma como são evitados os deadlocks, quando são feitas operações de transferências entre contas. Como, no nosso programa, é necessário requisitar uma conta e depois a outra, isto poderia levar a uma situação de deadlock se, ao mesmo tempo, ocorressem duas operações inversas (ou seja, uma que transfere dinheiro da conta A para a conta B, e outra que transfere da B para a A).

Para evitar a espera circular que levaria ao deadlock, optamos por requisitar sempre primeiro a conta com menor ID. Assim, duas requests que utilizem as mesmas duas contas irão sempre requisitar essas contas pela mesma ordem, evitando assim a possibilidade da ocorrência de deadlock.

Encerramento do servidor

Relativamente à maneira como se faz o encerramento do servidor no nosso programa, podemos concluir que há duas condições que se têm de verificar para tal acontecer, após ser recebido o pedido de encerramento: a **fila de pedidos tem de estar vazia**, e **o FIFO do servidor tem de também estar vazio**. Isto indica que todas as requests já foram processadas e que o buffer do FIFO se encontra vazio. Tal como nos era sugerido, quando se recebe um pedido de encerramento as permissões do FIFO do servidor são alteradas para “apenas leitura”, de modo a não receber mais requests de utilizadores.

A thread main, cujo propósito é receber os pedidos dos utilizadores e colocá-los na fila de pedidos, termina o seu ciclo de execução quando o buffer do FIFO estiver vazio (ou seja, quando uma leitura do mesmo retornar 0, admitindo que ele não está aberto para escrita).

Fig: Condição necessária para a main thread terminar o seu ciclo de execução.

```
// reads the request from a user, and sends it to the request queue
fifoClosed = readRequest(&currentRequest);
int requestPid = currentRequest.value.header.pid;

// unlocks all the waiting threads, so they can exit
if (fifoClosed) {
    pthread_cond_broadcast(&items_cond);
    break;
}
```

A main thread executa **pthread_cond_broadcast** antes de sair do seu ciclo, de modo a "acordar" todas as threads que possam estar "presas" num **pthread_cond_wait**, para que estas possam terminar também.

Em relação às threads consumidoras, estas saem do seu ciclo de execução quando o FIFO estiver fechado e quando a fila de pedidos estiver vazia.

Fig: Condição necessária para as threads consumidoras terminarem o seu ciclo.

```
// in order to close the thread
if (fifoClosed && (items == 0)) {
    pthread_mutex_unlock(&items_lock);

    if(logSyncMech(fdServerLogFile, bankOfficeId, SYNC_OP_MUTEX_UNLOCK, SYNC_ROLE_CONSUMER, 0) < 0) {
        perror("Write in server log file");
        exit(EXIT_FAILURE);
    }

    return 1;
}
```