SOPE- Relatório do Projeto 2- T1G01

Protocolo de comunicação:

A estrutura das mensagens trocadas entre o servidor e os clientes, foi a descrita no enunciado do problema. A mensagem possui as três componentes, *type* que identifica o tipo de pedido, *length* que informa o tamanho(em bytes) da secção seguinte, o *value* que contém a informação necessária para a autenticação e realização da operação bem como quaisquer argumentos complementares do tipo de pedido.

O utilizador trata de verificar se os argumentos são válidos e constrói a mensagem, colocando no FIFO de pedido, apenas a quantidade correta de informação (neste caso 4 bytes do type + 4 bytes do header + o valor do *length*).

O servidor, lê primeiro os primeiros 8 bytes do pedido, de modo a saber o *length,* e, consequentemente o número de bytes que restam ler.

user_util.c - sendRequest:

```
int sendRequest(tlv_request_t *request)
{
   write(request_fifo_fd, request, sizeof(op_type_t) + sizeof(uint32_t) + request->length);

   return 0;
}
```

server_bank_office.c - waitRequests:

```
nRead = read(request_fifo_fd, &received_request.type, sizeof(op_type_t));
        ...
nRead = read(request_fifo_fd, &received_request.length, sizeof(uint32_t));
        ...
nRead = read(request_fifo_fd, &received_request.value, received_request. length);
```

De forma semelhante, após processamento de um pedido, o servidor (mais concretamente, uma das threads que o constituem), envia para o FIFO de resposta ao pedido a quantidade correta de informação conforme a mensagem que pretende enviar.

server_bank_office.c - sendReply:

```
...
write(reply_fifo_fd, &reply, sizeof(op_type_t) + sizeof(uint32_t) + reply.length;
...
```

user_util.c - readReply:

```
read(response_fifo_fd, &(reply->type), sizeof(op_type_t));
...
```

```
read(response_fifo_fd, &(reply->length), sizeof(uint32_t));
...
read(response_fifo_fd, &(reply->value), reply->length);
```

É de notar que a escrita no FIFO de pedidos do servidor deve ser atómica para que não exista qualquer entrelaçamento de informação de outros pedidos. No entanto, a escrita no FIFO de resposta do utilizador pode ser feita em vários passos.

Mecanismos de sincronização:

Os mecanismos utilizados neste projeto foram os semáforos e os mutexes.

Os semáforos foram utilizados para a implementação da comunicação entre o thread principal do servidor que contém a fila de pedidos e o fifo correspondente, e os vários threads que pretendem tratar os pedidos da mesma fila. Para isso foi utilizada uma solução semelhante ao problema dos produtores consumidores em que dois semáforos (full e empty) controlam o acesso aos pedidos por parte dos threads. O semáforo full indica que existem pedidos para ler na fila, e o empty que existe espaço para a leitura de um novo pedido.

server_bank_office.c - bank_office_service_routine (simplificada):

```
...
while(true){
        // Consumer
        sem_wait(&full);
        tlv_request_t currentRequest = queue_pop(&requests);
        sem_post(&empty);
        handleRequest(currentRequest, office->id);

}
...
```

server_bank_office.c - waitForRequests (simplificada):

```
...
while (!shutdown || !isEOF){

        // read request
        ...

        // Producer
        sem_wait(&empty);
        pthread_mutex_lock(&request_queue_mutex);

        queue_push(&requests, received_request);

        pthread_mutex_unlock(&request_queue_mutex);
        sem_post(&full);

}
...
```

Os mutexes são utilizados essencialmente em dois aspetos. O primeiro pode ser visto no exemplo anterior para permitir o acesso mutuamente exclusivo à fila de pedidos por parte dos diferentes threads (offices). O outro uso passa pelas operações referentes às contas de utilizador que utilizam os mutexes para garantir que quando uma operação está a utilizar uma conta nenhuma outra operação irá alterá-la.

server_operations.c - op_checkBalance (simplificada):

```
...
   pthread_mutex_lock(&account_mutex[reply->value.header.account_id]);

   reply->value.balance.balance = account.balance;

   pthread_mutex_unlock(&account_mutex[reply->value.header.account_id]);
...
```

Encerramento do servidor:

Para o encerramento o *office* que recebeu o pedido, altera as permissões do FIFO de pedidos para apenas de leitura e fecha o fifo de modo a permitir que as as leituras que sejam feitas posteriormente a ser lido as informações que ainda estavam no FIFO resultem em EOF. Os offices continuam a executar a sua rotina até que a fila de pedidos estava vazia.

A função *closeOffices* executa um número de posts igual ao número de threads de modo a desbloquear os que estejam bloqueados numa espera pelo semáforo full. De seguida efetua um *pthread_join* para terminar todos os threads.

server_office.c - shutdown_server (simplificada):

```
int shutdown_server()
{
   shutdown = true;
   fchmod(request_fifo_fd, S_IRUSR | S_IRGRP | S_IROTH);
   close(request_fifo_fd_DUMMY;

   return 0;
}
```

server_bank_office.c - waitForRequests (simplificada):

```
...
while (!shutdown || !isEOF){

       nRead = read(request_fifo_fd, &received_request.type, sizeof(op_type_t));

       if (nRead == 0)
           break;
       else if (nRead == -1)
           continue;

       nRead = read(request_fifo_fd, &received_request.length, sizeof(uint32_t));
```

```
        if (nRead == 0)
            break;
        else if (nRead == -1)
            continue;

        nRead = read(request_fifo_fd, &received_request.value, received_request.
length);

        if (nRead == 0)
            break;
        else if (nRead == -1)
            continue;


}
...
```

server_bank_office.c - bank_office_service_routine (simplificada):

```
...
sem_wait(&full);

pthread_mutex_lock(&request_queue_mutex);

if(shutdown && queue_is_empty(&requests)){

        pthread_mutex_unlock(&request_queue_mutex);
        break;
}
...
```

server_bank_office.c - closeOffices :

```
int closeOffices()
{

   for(int i = 1; i <= total_thread_cnt;i++)
       sem_post(&full);

   for (int i = 1; i <= total_thread_cnt; i++){

       pthread_join(offices[i].tid,NULL);
       logBankOfficeClose(getLogfile(), offices[i].id, offices[i].tid);
   }

   return 0;
}
```