SOPE

Estrutura das Mensagens

Depois de compilar o programa corretamente e o programa *user* ser inicializado, os seus argumentos são verificados para confirmar que respeitam todas as normas especificadas. Posteriormente, estes são extraídos para uma struct criada, *st_instruction*, que os separa e guarda em variáveis fáceis de aceder.

```
struct st_instruction{
    int pid;
    int id;
    char* password;
    int latency;
    int operation;
    bool has_args;
    char* arg1;
    char* arg2;
    char* arg3;
};
```

Para os enviar para o servidor é necessário passar a informação para a struct *tlv_log_request*. Este formato torna mais fácil a leitura do pedido para o servidor e, através da função *logReply(),* passa para todas as informações necessárias para o terminal e para o ficheiro *ulog.txt*.

```
/**
 * @brief Full request message in TLV format.
 */
typedef struct tlv_request {
  enum op_type type;
  uint32_t length;
  req_value_t value;
} __attribute__((packed)) tlv_request_t;
```

Relativamente ao programa *server*, os seus argumentos são igualmente verificados, tornando o código mais robusto. O servidor pode receber vários *requests* do *user* e, sendo assim, tem uma fila de pedidos que têm de ser processados. Cada balcão vai buscar pedidos a essa *queue*. No fim de cada pedido retorna para o cliente um elemento da struct *tlv_reply_t* e imprime após processamento todas as informações importantes no ficheiro *slog.txt*. O *user*, posteriormente, utiliza a informação da *reply* para imprimir no terminal e no seu ficheiro correspondente a resposta do servidor.

```
typedef struct tlv_reply {
  enum op_type type;
  uint32_t length;
  rep_value_t value;
} __attribute__((packed)) tlv_reply_t;
```

A comunicação entre os dois programas é realizada usando FIFO's. Ambos criam um FIFO de leitura e enviam a sua informação para o correspondente programa.

Os vários pedidos de clientes são processados pelo servidor através de *threads*. O número de threads é passado como argumento no início do programa.

Encerramento do servidor

Para encerrar o servidor é necessário que um utilizador envie um pedido para o mesmo a requisitar esta acção. Quando o servidor recebe o pedido, este é colocado na queue de pedidos a processar pelos balcões. Assim que um balcão estiver disponível para processar o pedido, o pedido vai ser retirado da queue e inicia-se o processamento.

O balcão identifica o pedido através do seu tipo, no caso do shutdown OP_SHUTDOWN, e verifica se este é válido (se a conta que fez o pedido foi a do administrador e se o login foi bem sucedido). A flag shutdown, uma variável partilhada pelo servidor e todos os balcões, é colocada a 1, valor que corresponde ao encerramento. É preenchida uma reposta que será enviada ao utilizador e é aplicado o delay presente no pedido.

A partir do momento que a flag shutdown é ativada, o servidor entra em modo de encerramento, querendo isto dizer que as permissões do FIFO para o qual os utilizadores enviavam os pedidos são alteradas de modo a permitirem apenas leituras, o que garante que não serão recebidos novos pedidos. No entanto, todos os pedidos que ainda se encontram por processar, quer estes se encontrem no FIFO quer na queue, ainda terão de ser processados. Para tal acontecer, o servidor continua a adicionar os pedidos que se encontram no FIFO à queue e os balcões continuam a sua operação normal. Quando o FIFO estiver vazio, este é fechado e removido do sistema. O servidor fica à espera que cada um dos balcões termine as suas operações para os encerrar e, de seguida, fecha também o seu logfile e termina.

Mecanismos de Sincronização Utilizados

Para a realização deste projeto utilizamos como mecanismos de sincronização mutexes e variáveis de condição.

Um dos primeiros problemas com o qual nos deparámos foi o acesso à fila de pedidos, pois diversos threads teriam de operar sobre a mesma fila e para que cada thread tivesse a informação acerca da fila atualizada, decidimos que seria uma boa opção utilizar um mutex para que apenas um thread tenha acesso à fila de cada vez.

No entanto isto não seria suficiente, pois no caso de a fila se encontrar vazia e um thread a bloqueasse, o thread principal não conseguiria adicionar novos pedidos à fila, acabaríamos portanto num deadlock. Para evitar esta situação introduzimos duas variáveis de condição, empty_queue e full_queue, que se referem a dois estados da fila, vazia e cheia, respectivamente. No caso da fila estar vazia, cada thread espera que o servidor adicione algo à fila e dê sinal de que esta já não se encontra vazia para continuar o seu processamento. No caso de a fila se encontrar cheia, o servidor espera que um dos balcões retire um pedido da fila e que este dê sinal de que a mesma já não se encontra cheia para adicionar um novo pedido.

Foram também criados mutexes para cada uma das contas, para que apenas fosse permitido um thread alterar os dados dessa conta. Sempre que uma conta é criada e adicionada ao array de contas é criado um mutex que é armazenado num array no índice correspondente ao número identificador da conta, account_id. Tal como referido antes o mutex apenas é bloqueado quando são alterados dados da conta ou quando se acede a informação que pode ser alterada, como o saldo da mesma. Portanto as contas não são bloqueadas quando para se verificar se a informação de login está correta.

Identificámos também a possibilidade de ocorrer um deadlock quando se tentassem efetuar duas transferências simultâneas em que numa o identificador da conta de origem fosse, por exemplo, 1 e o da conta de destino fosse, por exemplo, 2 e na outra transferência fosse a ordem inversa. Para lidar com isto os nossos balcões bloqueiam sempre em primeiro lugar a conta cujo identificador seja de menor ordem.