Trabalho Prático Nº 2

Simulação de um sistema de home banking

Implementação de uma arquitetura cliente/servidor baseada em FIFOs

---

Estrutura das mensagens

Cliente → Servidor:

Os pedidos enviados pelo cliente estão no formato *TLV* (usando as estruturas fornecidas). Existem 4 tipos de pedido, um para cada operação possível – **criar conta**, **ver saldo**, **fazer transferência**, **encerrar servidor**. Todos os pedidos incluem um *header* no *value* que inclui `ID de processo`, `ID de conta`, `password` e `atraso`. O pedido só é processado caso seja válido (por exemplo se o utilizador existe ou se *password* corresponde ao seu ID). Caso seja um pedido de criação de conta ou de transferência, inclui ainda no value uma estrutura de criação de conta ou de transferência com os dados necessários para efetuar o que é pedido.

Servidor → Cliente:

As respostas enviadas pelo servidor estão no formato *TLV* (usando as estruturas fornecidas). Existem 4 tipos de resposta, um para cada operação possível – **criar conta**, **ver saldo**, **fazer transferência**, **encerrar servidor**. Todas as respostas incluem um *header* no *value* que inclui `ID de conta` e `código de retorno`. O código de retorno indica se a operação foi bem sucedida ou, se não, qual o erro. Caso seja um pedido de verificação de saldo, de transferência ou de encerramento do servidor, inclui ainda no value uma estrutura verificação de saldo, de transferência ou de encerramento do servidor com informação sobre o estado da conta ou do servidor depois do processamento do pedido (saldo no caso de verificação do saldo ou tranferência e número de caixas ativas no caso de encerramento do servidor).

Mecanismos de Sincronização

De forma a facilitar a comunicação entre a *thread main* e os balcões (restantes *threads*) foi implementado uma fila (*queue*) de pedidos enviados pelos clientes. Visto que o pedido de encerramento tem prioridade em relação aos restantes, esse mesmo pedido, quando verificado que foi enviado pelo *admin*, é processado pela *thread main* e a mesma notifica as restantes *threads*. Caso não se trate de um pedido de encerramento, o pedido é colocado na fila de pedidos e posteriormente retirado pelas outras *threads*. De forma a não haver conflitos, apenas uma *thread* (*thread main* inclusivé) consegue aceder à fila de pedidos, esta sincronização é feita recorrendo a um `mutex` (`queue_mut`). O mesmo se passa para o ficheiro **slog.txt** e a varivavél global `num_active_threads` que mantém informação sobre o número de *threads* ativas, recorrendo aos `mutexes` `log_mut` e `counter_mut` respetivamente.

De forma a evitar *busy waiting* por parte das *threads* balcões, foi utilizado a **variavél de condição** `cond_queued_red` cujo sinal é enviado pela *thread main* após colocar um pedido na fila de pedidos. Assim uma das *threads* bloqueadas à espera desse mesmo sinal pode começar a processar o pedido sem estar constantemente a verificar se a fila de pedidos não está vazia. Esta mesma variavel é também usada para o encerramento do servidor, como será explicado no próximo tópico.

Source Code

Trabalho feito pela *thread main*

```
while (1) {
    // ---- getting request from user
    tlv_request_t req;
    while(!read_request(fifo_request, &req));

    // ---- main thread should take care of shutdown requests
    if (req.type == OP_SHUTDOWN) {
        tlv_reply_t shutdown_reply;
        int fifo_reply;
        if (validate_request(&req, &shutdown_reply)) {
            shutdown_req_pid = req.value.header.account_id;
            // ---- Log the delay introduced (server shutdown only)
            logDelay(logfile, MAIN_THREAD_ID, req.value.header.op_delay_ms);
            // ---- command
            shutdown_server(&shutdown_reply.value, &fifo_request);
            // ---- send signal to unlock all threads waiting for new requests
            pthread_cond_broadcast(&cond_queued_req);
            logSyncMech(logfile, MAIN_THREAD_ID, SYNC_OP_COND_SIGNAL, SYNC_ROLE_PRODUCER, shutdown_req_pid);
            // ---- reply type
            shutdown_reply.type = req.type;
            // ---- reply length
            shutdown_reply.length = sizeof(shutdown_reply.value);
            // ---- create user fifo
```

```
            user_fifo_create(&fifo_reply, req.value.header.pid);
            // ---- write reply
            write(fifo_reply, &shutdown_reply, sizeof(tlv_reply_t));
            // ---- break cycle
            break;
        }
        // ---- reply type
        shutdown_reply.type = req.type;
        // ---- reply length
        shutdown_reply.length = sizeof(shutdown_reply.value);
        // ---- create user fifo
        user_fifo_create(&fifo_reply, req.value.header.pid);
        // ---- write reply
        write(fifo_reply, &shutdown_reply, sizeof(tlv_reply_t));
    }
    else {
        logSyncMech(logfile, MAIN_THREAD_ID, SYNC_OP_MUTEX_LOCK, SYNC_ROLE_PRODUCER, req.value.header.account_id);
        pthread_mutex_lock(&queue_mut);
        // ---- enqueue the request
        push(&request_queue, req);
        // ---- send queued_req signal
        pthread_cond_signal(&cond_queued_req);
        logSyncMech(logfile, MAIN_THREAD_ID, SYNC_OP_COND_SIGNAL, SYNC_ROLE_PRODUCER, req.value.header.account_id);

        pthread_mutex_unlock(&queue_mut);
        logSyncMech(logfile, MAIN_THREAD_ID, SYNC_OP_MUTEX_UNLOCK, SYNC_ROLE_PRODUCER, req.value.header.account_id);
    }
}
```

Trabalho feito pelas *threads* balcão

```
while (1) {
    // ---- lock threads if not shutdown
    if (!shutdown) {
        logSyncMech(logfile, *(int *) thread_id, SYNC_OP_MUTEX_LOCK, SYNC_ROLE_CONSUMER, getpid());
        pthread_mutex_lock(&queue_mut);
    }
    else {
        pthread_mutex_unlock(&queue_mut);
        logSyncMech(logfile, *(int *) thread_id, SYNC_OP_MUTEX_UNLOCK,SYNC_ROLE_CONSUMER, shutdown_req_pid);
    }

    // ---- if request queue is empty and shutodwn is set, break the loop and close offices
    if (empty(&request_queue)) {
        if (shutdown)
            break;
        else {
            logSyncMech(logfile, *(int *) thread_id, SYNC_OP_COND_WAIT, SYNC_ROLE_CONSUMER, getpid());
            pthread_cond_wait(&cond_queued_req, &queue_mut);
        }
    }
    if (!empty(&request_queue)) {
        // ---- get next request
        next_request = front(&request_queue);
        // ---- Log the delay introduced immediately after entering the critical section of an account
        logSyncDelay(logfile, *(int *) thread_id, next_request.value.header.account_id, next_request.value.header.op_del
        // ---- increment active threads
        logSyncMech(logfile, *(int *) thread_id, SYNC_OP_MUTEX_LOCK, SYNC_ROLE_CONSUMER, next_request.value.header.accou
        pthread_mutex_lock(&counter_mut);
        num_active_threads++;
        pthread_mutex_unlock(&counter_mut);
        logSyncMech(logfile, *(int *) thread_id, SYNC_OP_MUTEX_UNLOCK,SYNC_ROLE_CONSUMER, next_request.value.header.acco
        // ---- dequeue the request
        pop(&request_queue);
        // ---- unlock threads (queue access is done)
        pthread_mutex_unlock(&queue_mut);
        // ---- process request
        acknowledge_request(&next_request, &reply, *(int *) office_id);
        // ---- create user fifo
        user_fifo_create(&fifo_reply, next_request.value.header.pid);
        // ---- write reply
```

```
        write(fifo_reply, &reply, sizeof(tlv_reply_t));
        // ---- close user fifo
        close(fifo_reply);
        // ---- decrement active threads
        logSyncMech(logfile, *(int *) thread_id, SYNC_OP_MUTEX_LOCK, SYNC_ROLE_CONSUMER, next_request.value.header.accou
        pthread_mutex_lock(&counter_mut);
        num_active_threads--;
        pthread_mutex_unlock(&counter_mut);
        logSyncMech(logfile, *(int *) thread_id, SYNC_OP_MUTEX_UNLOCK,SYNC_ROLE_CONSUMER, next_request.value.header.acco

        printf("Waiting for user request...");
    }
}
```

Encerramento do servidor

Quando o servidor recebe um pedido do *admin* para encerrar, altera imediatamente as permissões do seu `fifo` para **leitura apenas**, de forma a que não receba mais pedidos dos utilizadores. Faz, também, *set* a uma variável que indica se o encerramento está iminente (`shutdown`) e atualiza o valor de balcões ativos no `reply value`. De seguida, desbloqueia todas as threads que estão à espera de novos pedidos (com `pthread_cond_broadcast`), atualiza o `reply type` e o `reply length` e envia o `reply` para o utilizador através de um `fifo` criado para isso. Finalmente, espera que todas as *threads* terminem o seu trabalho (incluindo todos os pedidos que estavam em fila de espera quando o pedido de encerramento foi feito) e fecha e remove o fifo do servidor, terminando o programa.