Universidade do Porto

FEUP Faculdade de Engenharia

Mestrado Integrado de Engenharia Informática e Computação

Sistemas Operativos

2nd Project - Report

Paulo Moutinho up201704710
Pedro Mendes up201704219
Pedro Azevedo up201603816

We have two FIFO's which our client and server use in order to communicate back and forth with each other:

A server FIFO, through which the client will send its requests to the server after having checked the inputted arguments and formulating the request from them,

```c
int SendRequest(tlv_request_t* req){
    if((fd_server = open(SERVER_FIFO_PATH, O_WRONLY | O_NONBLOCK | O_APPEND)) == -1){
        //Send SHUT_DOWN reply
        //perror("FIFO_server");
        tlv_reply_t rep;
        rep.type = req->type;
        rep.value.header.account_id = req->value.header.account_id;
        rep.length = sizeof(rep.value.header);
        rep.value.header.ret_code = RC_SRV_DOWN;
        rep.value.balance.balance = 0;
        logReply(fd_ulog, getpid(), &rep);
        return -1;
    }

    //char aux[sizeof(tlv_request_t)];
    //memcpy(aux, req, sizeof(tlv_request_t));

    if(write(fd_server, req, sizeof(tlv_request_t))==-1){
        fprintf(stderr, "Cannot write into fd_server");
        return -1;
    }
    printf("%d\n", req->value.header.account_id);
    return 0;
}
```

(the section in blue makes it so that in case the server FIFO can't be opened, a reply is automatically made in place of the server by the user, attributing the error to the server being down, and logging it accordingly)

and a user FIFO that will allow the client to receive the server's response to its request.

```c
if((fd_user = open(userFifo, O_RDONLY | O_NONBLOCK)) == -1){
```

```c
time_t start = time(NULL);
time_t cur_time = start;
ssize_t reading = 0;

// If the server takes more than FIFO_TIMEOUT_SECS, TIMEOUT reply will be send
while((cur_time - start) <= FIFO_TIMEOUT_SECS)
{
    cur_time = time(NULL);
    reading = read(fd_user, &rep, sizeof(tlv_reply_t));
    if (reading < 0)
    {
        if (errno != EAGAIN)
            perror("Cannot read");
    }
    // If it reads something, it means a reply was received
    else if (reading > 0){
        if (logReply(fd_ulog, getpid(), &rep) < 0)
            fprintf(stderr, "Cannot log\n");
        return;
    }

    else
    {
        //fprintf(stderr, "Waiting for a reply\n");
    }
}
rep.type = req->type;
rep.value.header.account_id = req->value.header.account_id;
rep.length = sizeof(rep.value.header);
rep.value.header.ret_code = RC_SRV_TIMEOUT;
rep.value.balance.balance = 0;
logReply(fd_ulog, getpid(), &rep);
```

(In case a response is not issued within the allotted time frame, the user will again issue the reply itself and classify the error as a timeout before logging it)

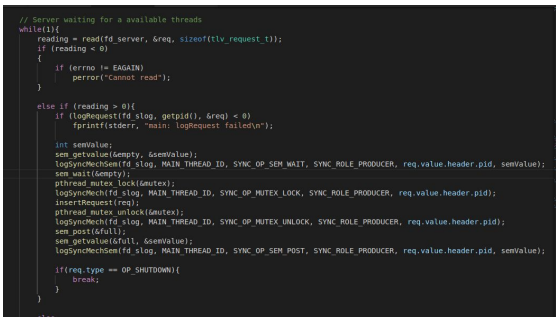
As for the synchronization mechanisms used, we have two semaphores, empty and full, as well as a mutex.

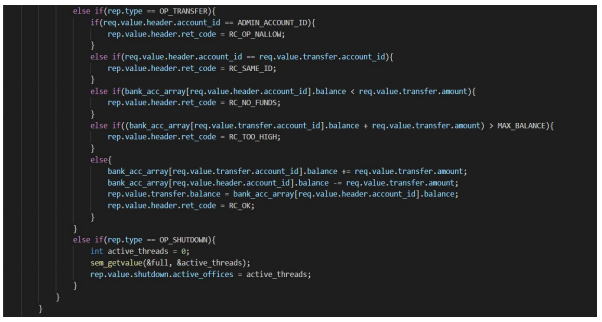
```c
//Iniciar os semáforos
logSyncMechSem(fd_slog, 0, SYNC_OP_SEM_INIT, SYNC_ROLE_PRODUCER, 0, 0);
sem_init(&full, 0, 0);
logSyncMechSem(fd_slog, 0, SYNC_OP_SEM_INIT, SYNC_ROLE_PRODUCER, 0, num_threads);
sem_init(&empty, 0, num_threads);

//Iniciar as threads = balcões
unsigned ids[num_threads];
for(int i = 0; i < num_threads; i++){
    ids[i]=i+1;
    pthread_create(&thread_ar[i], NULL, processRequest, &(ids[i]));
}
```

Full will make sure that the threads don't try to access the request queue (through popRequest) when there is none to be processed,

```c
void * processRequest(void * id){

    unsigned i = *(unsigned*)id;
    logBankOfficeOpen(fd_slog, i, pthread_self());
    int semValue;
    pthread_mutex_t mutex;
    tlv_request_t req;
    char userFifo [USER_FIFO_PATH_LEN];
    //Processing Request -> threads waiting for requests
    while(1){
        sem_getvalue(&full, &semValue);
        logSyncMechSem(fd_slog, i, SYNC_OP_SEM_WAIT, SYNC_ROLE_CONSUMER, 0, semValue);
        sem_wait(&full);
        logSyncMech(fd_slog, i, SYNC_OP_MUTEX_LOCK, SYNC_ROLE_CONSUMER, 0);
        pthread_mutex_lock(&mutex);
        popRequest(&req);
        pthread_mutex_unlock(&mutex);
        logSyncMech(fd_slog, i, SYNC_OP_MUTEX_UNLOCK, SYNC_ROLE_CONSUMER, req.value.header.pid);
        sem_post(&empty);
        sem_getvalue(&empty, &semValue);
        logSyncMechSem(fd_slog, i, SYNC_OP_SEM_POST, SYNC_ROLE_CONSUMER, req.value.header.pid, semValue);

        usleep(req.value.header.op_delay_ms * 1000);
        logSyncDelay(fd_slog, i, req.value.header.account_id, req.value.header.op_delay_ms);



        tlv_reply_t rep;
        rep.type = req.type;
        rep.value.header.account_id = req.value.header.account_id;
        rep.length = sizeof(rep.value.header);
        rep.value.header.ret_code = RC_OK;
```

and empty that new requests aren't inserted into the queue (through insertRequest) when there are no threads available to process them.





*

As for the mutex, its purpose is to provide mutual exclusion between threads when accessing the queue and the accounts.
Finally, the server shuts down if after a thread gets a request from the queue, as you can see in the last picture from the line marked by the asterisk(*), its type is that of "OP_SHUTDOWN", by breaking out of the cycle.