RELATÓRIO TP

Sistemas Operativos 25/26

Diogo Estimado – 2023130197

Lucas Golobovante - 2023140728

Índice

Introdução

O presente relatório descreve a conceção e implementação de uma solução de software para a simulação e gestão de uma frota de táxis autónomos, desenvolvida no âmbito da unidade curricular de Sistemas Operativos.

Objetivo

O objetivo principal deste projeto é a aplicação prática dos conceitos fundamentais da programação de sistemas em ambiente Unix/Linux. O sistema desenvolvido demonstra a utilização de mecanismos avançados de comunicação entre processos, gestão de concorrência com threads, sincronização de recursos partilhados e manipulação de sinais.

Âmbito da solução

A solução proposta, desenvolvida na linguagem C, consiste numa arquitetura distribuída composta por três módulos principais:

- Controlador: O núcleo do sistema, responsável pela gestão centralizada dos recursos, agendamento de viagens e monitorização da frota.
- Cliente: A interface que permite aos utilizadores interagirem com o sistema, efetuarem pedidos de transporte e consultarem o estado das suas viagens.
- Veículo: Um processo independente, instanciado dinamicamente pelo Controlador, que simula o transporte físico e reporta o seu progresso em tempo real.

Arquitetura do sistema

Visão geral

A solução desenvolvida segue uma arquitetura distribuída do tipo Cliente-Servidor, centrada num processo gestor (Controlador) que coordena múltiplos clientes e processos de simulação (Veículos). O sistema foi desenhado para funcionar em ambiente Linux, tirando partido de mecanismos nativos de comunicação entre processos (IPC) para garantir a troca de dados em tempo real.

O sistema divide-se em três componentes lógicos principais que operam de forma concorrente:

1. Controlador: O núcleo central de processamento.
2. Cliente: A interface de interação com os utilizadores.
3. Veículo: Entidades dinâmicas que realizam os serviços.

Descrição dos componentes

Controlador (Servidor)

É o componente mais complexo do sistema. Funciona como um daemon (processo de fundo) que deve estar sempre ativo.

- Responsabilidades: Gerir a base de dados em memória de utilizadores e viagens, controlar o relógio de simulação, validar o limite da frota (definido pela variável de ambiente NVEICULOS) e lançar processos filhos.
- Modelo de Concorrência: Para evitar o bloqueio do sistema enquanto espera por input, o Controlador é multithreaded. Possui threads dedicadas para leitura de comandos, gestão de clientes, relógio e monitorização individual de cada veículo ativo.

Cliente

Representa o ponto de acesso dos utilizadores. Podem existir múltiplas instâncias do cliente a correr simultaneamente.

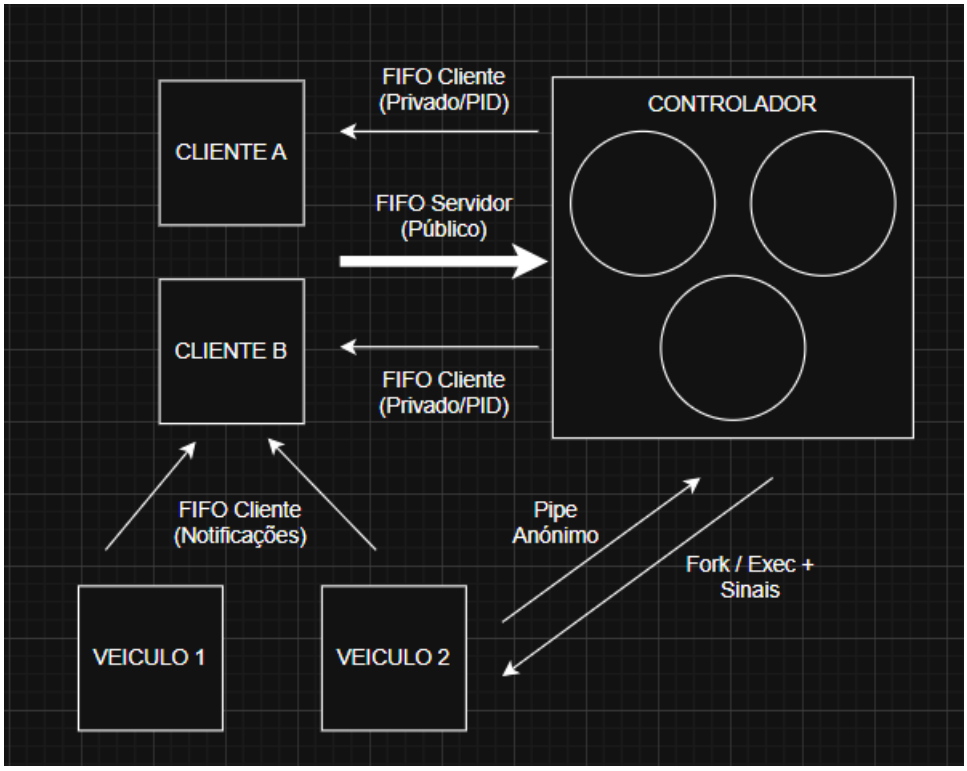
- Responsabilidades: Autenticar o utilizador, enviar pedidos de agendamento/cancelamento e apresentar notificações.
- Arquitetura: Utiliza duas threads. A thread principal gere a leitura do teclado (interação síncrona), enquanto uma thread secundária escuta permanentemente o canal de comunicação privado para receber avisos assíncronos do servidor ("Viagem cancelada", "Servidor encerrou").

Veículo

É um processo temporário criado pelo Controlador através das chamadas ao sistema fork() e exec().

- Responsabilidades: Simular o decorrer de uma viagem (consumindo tempo proporcional à distância) e enviar telemetria.
- Simplificação: Conforme permitido pelo enunciado, o veículo assume que o cliente entra imediatamente após a chegada, iniciando a viagem sem necessidade de comandos adicionais de interação manual.

Topologia e Comunicação



Fluxo de Informação

O fluxo de dados no sistema percorre o seguinte ciclo de vida típico de uma viagem:

1. Pedido: O Cliente envia uma estrutura PedidoCliente via Named Pipe público.
2. Processamento: A thread de clientes do Controlador recebe o pedido. Se for um agendamento, guarda-o na lista interna protegida por Mutex.
3. Despacho: Quando o relógio virtual atinge a hora agendada, o Controlador verifica se há vagas na frota. Se sim, cria um Pipe Anónimo, executa um fork() e lança o processo Veículo.
4. Execução:
    - O Veículo envia mensagens de "Início" e "Percentagem" para o stdout, que o Controlador lê através do pipe anónimo.
    - O Veículo envia mensagens de "Chegada" diretamente para o Named Pipe do Cliente.
5. Conclusão: Quando o Veículo termina, o Controlador deteta o fim da comunicação, atualiza as estatísticas globais (KMs) e liberta a vaga na frota.

Protocolo de Comunicação e Estrutura de Dados

Para garantir a interoperabilidade entre os diferentes processos (que possuem espaços de memória distintos), foi definido um protocolo de comunicação estrito no ficheiro de cabeçalho common.h. Este ficheiro é partilhado por todos os módulos do sistema.

Tipos de mensagens

Para que o Controlador saiba distinguir um pedido de login de um pedido de agendamento, foi utilizada uma enumeração (enum) que codifica o tipo de operação. Isto permite utilizar uma estrutura switch-case eficiente no tratamento das mensagens.

```
typedef enum {
    MSG_LOGIN,
    MSG_PEDIDO_VIAGEM,
    MSG_CANCELAR_PEDIDO,
    MSG_PEDIDO_CONSULTA,
    MSG_RESPOSTA_SERVIDOR,
    MSG_VEICULO_CHEGOU,
    MSG_LOGOUT
} TipoMsg;
```

Descrição dos Tipos:

- MSG_LOGIN: Registo inicial do utilizador.
- MSG_PEDIDO_VIAGEM: Solicitação de agendamento com parâmetros (tempo, origem, distância).
- MSG_CANCELAR_PEDIDO: Ordem para abortar um serviço.
- MSG_PEDIDO_CONSULTA: Solicitação do histórico de viagens.
- MSG_SHUTDOWN / MSG_LOGOUT: Protocolos de encerramento.

Estrutura de dados

A troca de dados é feita através do envio de blocos de memória estruturados (structs), em vez de texto livre, o que facilita o parsing e evita erros de formatação.

Pedido do Cliente (PedidoCliente)

Esta estrutura é enviada do Cliente para o Servidor via Named Pipe. (Insira aqui um print screen da struct PedidoCliente)

Destaca-se o campo pid_cliente, fundamental para que o servidor saiba para onde enviar a resposta (construindo o nome do pipe /tmp/taxi_cli_[PID]_fifo), e o campo dados, uma string genérica que transporta argumentos variáveis (como o nome do utilizador ou os dados da viagem).

Resposta do Servidor (RespostaServidor)

Esta estrutura é enviada do Servidor (ou Veículo) para o Cliente. (Insira aqui um print screen da struct RespostaServidor)

O campo mensagem foi dimensionado com 2048 bytes. Esta decisão de design permite que o servidor envie listagens completas (como o histórico de viagens obtido pelo comando consultar) numa única operação de escrita, garantindo que o texto aparece formatado e sem interrupções no terminal do cliente.

Implementação

A implementação do sistema obedeceu rigorosamente aos requisitos de concorrência e robustez. Nesta secção, detalham-se as estratégias utilizadas para garantir que o Controlador processa múltiplos eventos simultaneamente sem corromper dados ou bloquear a execução.

Concorrência e Multithreading

O Controlador não pode bloquear à espera de input de um cliente ou de um veículo, sob pena de congelar toda a simulação. Para resolver isto, adotou-se uma arquitetura fortemente baseada em threads POSIX (pthread), dividindo as responsabilidades da seguinte forma:

1. Thread Principal (main): Dedicada exclusivamente à interação com o administrador (leitura do teclado para comandos como terminar, frota, listar).
2. Thread de Gestão de Clientes (trataClientes): Executa um ciclo infinito de leitura bloqueante (read) no Named Pipe do servidor. Assim que chega um pedido, processa-o imediatamente (Login, Agendamento, etc.) e volta a esperar.
3. Thread de Temporização (trataRelogio): Atua como o "pacemaker" do sistema. A cada segundo (sleep(1)), incrementa o tempo de simulação e verifica se existem viagens agendadas para aquele instante, despachando veículos se houver frota disponível.
4. Threads de Monitorização de Veículos (trataVeiculo):
    - O Problema: O controlador precisa de ler o stdout de múltiplos veículos. Se usasse um único ciclo de leitura, poderia bloquear num veículo lento enquanto outros tentavam enviar dados.
    - A Solução: Por cada veículo lançado, o Controlador cria uma nova thread dinâmica dedicada exclusivamente a ler o pipe anónimo desse veículo específico. Isto garante paralelismo real na receção de telemetria.

```
pthread_create(&t_cli, NULL, trataClientes, &dados);
pthread_create(&t_rel, NULL, trataRelogio, &dados);
```

Sincronização e Exclusão Mútua

A existência de múltiplas threads a aceder às mesmas variáveis globais (arrays de viagens, utilizadores e contadores) introduz o risco de Race Conditions.

Para mitigar este risco, foi implementado um mecanismo de Exclusão Mútua utilizando um Mutex (pthread_mutex_t).

- Qualquer acesso de leitura ou escrita às estruturas partilhadas (dados->viagens, dados->utilizadores, dados->veiculos_ativos) é precedido por pthread_mutex_lock e seguido por pthread_mutex_unlock.
- Exemplo Prático: Quando a thread do relógio decide lançar um veículo, ela bloqueia o mutex. Se, no mesmo milissegundo, um administrador tentar cancelar essa viagem, o comando ficará em espera até o relógio libertar o recurso, garantindo a integridade do estado do sistema.

```
void *trataRelogio(void *arg) {
    ControladorData *dados = (ControladorData*)arg;
    while(dados->continua) {
        sleep(1);
        pthread_mutex_lock(&dados->mutex);
        dados->tempo_sistema++;
        for(int i=0; i<dados->num_viagens; i++) {
            if(dados->viagens[i].status == 0 && dados->viagens[i].hora_inicio <= dados->tempo_sistema) {
                if (dados->veiculos_ativos < dados->max_veiculos) {
                    lancaVeiculo(dados, i);
                } else {
                    dados->viagens[i].status = -1;
                    pthread_mutex_unlock(&dados->mutex);
                    enviaResposta(dados->viagens[i].pid_cliente, MSG_RESPOSTA_SERVIDOR, "Cancelada: Frota cheia", 0);
                    pthread_mutex_lock(&dados->mutex);
                }
            }
        }
        pthread_mutex_unlock(&dados->mutex);
    }
    return NULL;
}
```

Criação de Processos e Redirecionamento de I/O

A criação dos veículos envolve um pipeline complexo de chamadas ao sistema para permitir que o processo pai (Controlador) capture o output do filho (Veículo):

1. Pipe Anónimo: Criação de um canal unidirecional com pipe().
2. Fork: Duplicação do processo.
3. No Processo Filho (Veículo):
    - Fecha o lado de leitura do pipe.
    - Usa dup2(fd[1], STDOUT_FILENO) para substituir a saída padrão pelo pipe.
    - Executa o código do veículo com execl("./veiculo", ...).

4. No Processo Pai (Controlador):
    - Fecha o lado de escrita do pipe.
    - Passa o descritor de leitura para a nova thread trataVeiculo, que utiliza a chamada ao sistema read() para processar as mensagens diretamente do descritor de ficheiro, byte a byte, garantindo a conformidade com os requisitos de utilização da API do sistema

Gestão de Sinais e Encerramento

Um dos maiores desafios foi garantir que o Cliente termina a sua execução quando o Servidor encerra, mesmo que o utilizador não toque no teclado.

- No Controlador: O comando terminar envia uma mensagem especial com a palavra-chave "SHUTDOWN" para todos os clientes conectados.
- No Cliente: A thread de escuta analisa todas as mensagens recebidas. Ao detetar a keyword "SHUTDOWN", executa uma rotina de auto-destruição:

    kill(getpid(), SIGKILL) -> O uso do sinal SIGKILL enviado ao próprio PID (getpid()) garante que o processo cliente é terminado instantaneamente pelo Sistema Operativo, interrompendo qualquer chamada bloqueante (como o fgets do menu principal) que estivesse a impedir o encerramento normal.

Comandos

Os comandos são impressos na inicialização de cada ficheiro a partir da função mostraAjuda().

```
COMANDOS DE ADMINISTRADOR
utiliz -> Listar utilizadores conectados
listar -> Listar todas as viagens
frota -> Ver estado dos veículos ativos
km -> Total de KMs percorridos pela frota
hora -> Mostra valor atual do tempo simulado
cancelar <id> -> Cancelar uma viagem
terminar -> Encerrar o sistema e fechar clientes
```

```
MENU DO UTILIZADOR
agendar <t> <loc> <km> -> Agendar nova viagem
consultar -> Ver estado das minhas viagens
cancelar <id> -> Cancelar uma viagem específica
terminar -> Sair da aplicação
```

Conclusão

O trabalho prático desenvolvido permitiu alcançar com sucesso todos os objetivos propostos para a unidade curricular de Sistemas Operativos. Foi implementada uma plataforma funcional de gestão de frota de táxis, capaz de gerir múltiplos clientes e veículos em simultâneo sem conflitos de dados.

Robustez: O sistema demonstrou estabilidade sob carga, gerindo corretamente o acesso concorrente através de mutexes.

Interatividade: A utilização de Named Pipes e Sinais permitiu uma experiência de utilização fluida, com notificações em tempo real.

Segurança: O protocolo de encerramento foi um dos pontos fortes da implementação, garantindo que não restam processos "zombies" nem clientes bloqueados após o fim da execução do servidor.

A principal dificuldade encontrada residiu na sincronização do encerramento do sistema. Inicialmente, ao terminar o Controlador, os veículos continuavam a escrever no stdout, poluindo o terminal, e os clientes ficavam bloqueados à espera de input. Esta questão foi resolvida através da implementação de um protocolo de handshake de encerramento: o controlador envia um sinal de paragem aos veículos e uma mensagem de "SHUTDOWN" aos clientes, aguardando (via wait loop) que os recursos sejam libertados antes de terminar a sua própria execução.

Em suma, o projeto consolidou os conhecimentos teóricos sobre a API do sistema operativo Unix, demonstrando a importância de uma gestão cuidada de processos e recursos em ambientes multitarefa.