Licenciatura de Engenharia Informática

Sistemas Operativos

Relatório do Trabalho Prático

Plataforma de gestão de serviços de Táxi

2025/2026

Diogo Antunes - 2018016615
Leandro Lopes - 2022122058

Coimbra, 10 de Dezembro de 2025

Conteúdos

1. Introdução

O presente trabalho consistiu na implementação de um sistema de gestão de serviços com frota de veículos, onde múltiplos clientes podem agendar serviços em horários específicos, cancelar serviço agendados e acompanhar a sua execução. O sistema central é o Controlador, que gere a frota e os pedidos, enquanto cada cliente interage com o controlador ao enviar comandos e receber respostas. Cada veículo é um processo separado que envia telemetria sobre as viagens.

O objetivo principal foi criar uma aplicação coordenada, utilizando threads, pipes e sinais, com integridade dos dados compartilhados e comunicação eficiente entre processos que fosse capaz de simular de forma realista a operação de uma frota de veículos atendendo pedidos de clientes, garantindo que cada operação fosse tratada de maneira ordenada, segura e responsiva.

O sistema foi projetado para lidar com múltiplos clientes que podem, simultaneamente, agendar serviços, consultar o estado de seus pedidos, cancelar serviços ou encerrar as suas sessões. Cada cliente interage com o Controlador, que atua como núcleo central do sistema, gerindo a frota de veículos, o estado dos serviços e a comunicação com os clientes. Cada veículo, por sua vez, é representado por um processo independente, que simula o deslocamento até o local do serviço e envia eventos de telemetria, como chegada ao destino, progresso percentual e conclusão do serviço.

O desafio principal do sistema era a coordenação entre múltiplos processos e threads, de forma a garantir que os dados compartilhados (listas de utilizadores, serviços e frota) permaneçam consistentes, evitando condições de corrida e garantindo a integridade das operações.

Para alcançar estes objetivos, o sistema utiliza uma combinação de mecanismos, incluindo Named Pipes para comunicação entre clientes e controlador, pipes anônimos para telemetria dos veículos, e sinais para notificação assíncrona de eventos críticos, como cancelamentos de serviços em execução. Paralelamente, o uso de threads permite que o controlador execute de forma simultânea o processamento de pedidos, a simulação do tempo e o recebimento de telemetria, mantendo o sistema responsivo e evitando bloqueios.

O relatório que se segue detalha a estratégia de implementação, a arquitetura do sistema, a comunicação entre os componentes e as decisões de design adotadas, justificando cada escolha e descrevendo como cada módulo contribui para o funcionamento correto e eficiente do sistema.

2. Objetivos dos Programas

O sistema segue o modelo cliente-servidor:

Controlador:
O controlador representa o núcleo central da plataforma, tendo os seguintes objetivos:

- Processa pedidos de clientes (login, agendamento, consulta, cancelamento e encerramento);
- Informa os clientes do sucesso ou insucesso acerca dos seus pedidos;
- Gere a frota de veículos, lançando os processos de cada veículo, e libertando-os no final da viagem;
- Gere o estado de cada serviço, marcando-os se estão agendados, em execução, permitindo também o cancelamento dos mesmos;
- Recebe telemetria de veículos em execução e passa mensagens informativas aos clientes;
- Controla o tempo interno do sistema e simula o avanço das horas.
- Guarda os km percorridos por todos os veículos;
- Permite listar os estados da frota, utilizadores e dos serviços agendados e em execução;
- Lidar simultaneamente com comandos inseridos pelo administrador, pedidos do cliente e telemetria de cada um dos veículos;
- Gere múltiplas conexões de utilizadores;
- Tratar de situações concorrentes de forma segura;
- Encerramento da plataforma de forma controlada.

Cliente:
O programa Cliente funciona como interface do utilizador, tendo os seguintes objetivos:

- Permite os utilizadores identificarem-se através de um username;
- Enviam comandos via FIFO do controlador;
- Recebem respostas via FIFO próprio;
- Suportam múltiplos comandos como agendar, consulta ou cancelamento de serviços;
- Verificam se existe um Controlador ativo antes de iniciar;
- Verificam se existe algum cliente com o mesmo username antes de iniciar;
- Terminam de o Controlador for encerrado;
- Lidar simultaneamente com comandos inseridos pelo utilizador e mensagens recebidas.

Veículos:
O programa Veículo simula um veículo, prestando um serviço de viagem a um cliente.

- Cada veículo é um processo independente, que simula o deslocamento até ao local do serviço;
- Envia eventos de telemetria (chegada, progresso, conclusão, cancelamento) através do stdout;
- Avisa o cliente do estado da viagem, sendo o controlador a reencaminhar essa comunicação.

3. Estruturas de Dados

UTILIZADOR

Estrutura utilizada para armazenar os dados de um cliente ligado no sistema. Guarda username e PID para permitir enviar respostas ao FIFO correspondente.

```
typedef struct{
    char username[MAX_USERNAME];
    int pid;
} UTILIZADOR;
```

SERVICO

Esta estrutura armazena os dados necessários de um serviço agendado por um cliente..

```
typedef struct {
    int id;
    int hora;
    int distancia;
    char username[MAX_USERNAME];
    char local[MAX_LOCAL];
    int ativo;
    int execucao;
    int pid_veiculo;
    int pid_cliente;
} SERVICO;
```

VEICULO

Esta estrutura é usada para guardar o estado de cada veículo da frota (livre ou em serviço), qual serviço está a executar e a percentagem da viagem já feita.

```
typedef struct{
    int ativo;
    int servico_id;
    int pid;
    int percentagem;
} VEICULO;
```

PEDIDO

Esta é a estrutura utilizada para representar um pedido emitido pelo cliente ao controlador. Contém a informação necessária para o controlador interpretar o pedido do cliente.

```
typedef struct {
    int pid_cliente;
    char username[MAX_USERNAME];
    int cmd;
    int  hora;
    int distancia;
    char local[MAX_LOCAL];
    int id_servico;
} PEDIDO;
```

RESPOSTA

Estrutura usada para representar a resposta enviada pelo controlador ao cliente indicando o sucesso ou insucesso do seu pedido. Além disso, é utilizada para reencaminhar as mensagens emitidas pelo veículo ao cliente, assim como quando o controlador avisa que vai encerrar.

```
typedef struct {
    int res;
    int cmd;
    char msg[MAX_MSG];
} RESPOSTA;
```

CM_CODE

Esta enumeração foi utilizada para facilitar a interpretação das mensagens tanto no cliente como no controlador, indicando o tipo de comando que estava a ser enviado.

```
typedef enum {
    CMD_LOGIN = 1,
    CMD_AGENDAR = 2,
    CMD_CONSULTAR = 3,
    CMD_CANCELAR = 4,
    CMD_TERMINAR = 5,
    CMD_INFO = 6
} CMD_CODE;
```

TDATA

Estrutura utilizada para armazenar os dados partilhados entre as threads_pedidos e a thread_tempo. Contém ponteiros para todas as estruturas do sistema e um mutex para sincronização.

```
typedef struct {
    int continua;
    pthread_mutex_t *ptrinco;

    SERVICO *servicos;
    UTILIZADOR *utilizadores;
    VEICULO *frota;

    int *total_servicos;
    int *total_veiculos_ativos;
    int *tempo;
    int *nveic;
    int *km_total;
} TDATA;
```

TELEM_DATA

Estrutura utilizada para armazenar os dados utilizados por cada Thread Telemetria, que acompanha um processo veículo. Contém apenas os dados relacionados com o veículo em execução e o pipe de leitura, evitando interferência entre as threads.

```
typedef struct {
    int canal_fd;
    SERVICO *servico;
    VEICULO *veiculo;
    int id_veiculo;
    pthread_mutex_t *trinco;
    int *total_veiculos_ativos;
    int *km_total;
} TELEM_DATA;
```

4. Funções

Controlador

Funções de Utilidade

- **void enviar_resposta(int pid, RESPOSTA *r)**

  Permite enviar uma resposta ao cliente através do FIFO privado dele.

- **void remover_utilizador(TDATA *td, int pid)**

  Remove um utilizador da lista de utilizadores ativos, quando este termina.

- **void notificar_todos_clientes_terminar(TDATA *td)**

Envia mensagem de encerramento a todos os clientes, quando o controlador vai encerrar**.**

- **int encontrar_utilizador(TDATA *td, const char *username)**

  Procura um username na lista de utilizadores e retorna o índice da sua posição na tabela, se não encontrar retorna -1. Utilizada para impedir logins duplicados.

- **int encontrar_slot_livre(TDATA *td)**

  Procura a primeira posição livre na tabela de utilizadores. Utilizada para registar novos utilizadores.

Funções de Comandos do Cliente

- **void comando_login(PEDIDO *p, TDATA *td)**

  Comando utilizado para registar um novo utilizador no sistema, verificando se o username já é utilizado, se existem slots disponíveis**.** Atualiza a tabela de clientes se o login for bem sucedido e informa o cliente do sucesso ou falha do login.

- **void comando_agendar(PEDIDO *p, TDATA *td)**

  Cria um novo serviço a pedido do cliente, verificando se a hora já passou e se a tabela de serviços está cheia. Preenche a estrutura SERVICO e envia a resposta ao cliente.

- **void comando_consultar(PEDIDO *p, TDATA *td)**

  Envia ao Cliente a lista de todos os serviços ativos do mesmo.

- **void comando_cancelar(PEDIDO *p, TDATA *td)**

Cancela um ou todos os serviços agendados do utilizador.

- **void comando_terminar(PEDIDO *p, TDATA *td)**

  Função que permite verificar se o Cliente pode encerrar, quando faz esse pedido. Se tiver serviços em execução não permite. Caso contrário, cancela todos os serviços do mesmo, se existirem e envia confirmação.

Funções de Processamento

- **void tratar_pedido_cliente(PEDIDO *p, TDATA *td)**

  Interpretador geral dos pedidos do Cliente. Chama as funções correspondentes ao pedido feito pelo Cliente. É chamada pela thread_pedidos.

- **void *thread_pedidos(void *data)**

  Thread que lê e trata pedidos do FIFO do controlador. Chama a função que trata os pedidos e mantem a sincronização com mutex.

- **void *thread_tempo(void *data)**

  Thread que incrementa o tempo simulado e verifica serviços a iniciar.

- **void *thread_telemetria(void *data)**

  Thread que lê telemetria do processo veículo. É responsável por encaminhar as mensagens enviadas pelo veículo ao cliente. Atualiza os dados da frota e liberta os veículos quando a viagem é concluída ou cancelada. É também responsável por fechar o ciclo do serviço.

- **void naofaznada(int s, siginfo_t *si, void *uc)**

  É responsável por acordar as threads bloqueadas.

Funções de Lista

- **void listar_utilizadores(TDATA *td)**

  Mostra utilizadores ativos e estado dos seus serviços.

- **void listar_frota(TDATA *td)**

Mostra o estado da frota, se estão livres, ou se estiverem em viagem, mostra a percentagem do percurso da mesma.

- **void listar_servicos(TDATA *td)**

  Mostra todos os serviços ativos e em execução.

Funções de Execução de Veículo

- **void lancar_processo_veiculo(int idx, int idv, TDATA *td)**

  Função responsável por lançar um processo veículo, redirecionamento do stdout, preparar a estrutura TELEM_DATA e criar a thread_telemetria para o veículo.

- **void verifica_lancar_veiculo(TDATA *td)**

  Verifica a cada segundo se deve iniciar algum serviço. Se existir algum veículo livre, lança a execução do veículo. Se não houver, cancela o serviço e avisa o cliente.

Funções de Cancelamento

- **void cancelar_todos_os_servicos(TDATA *td)**

  Cancela todos os serviços ativos. Utilizada quando o controlador termina.

- **void cancelar_servico(int id, TDATA *td)**

  Cancela um serviço específico ou todos os serviços. Se for um serviço em execução, envia sinal SIGUSR1 ao processo veículo para terminar.

Funções de Comando do Controlador

- **void processar_comando(char *linha, TDATA *td)**

  Interpreta e processa os comandos do utilizador do controlador.

Função Auxiliar

- **int ler_nveiculos()**

  Obtém o valor da variável de ambiente NVEICULOS, caso esteja entre os limites válidos e esta esteja criada e exportada guarda esse valor. Caso contrário termina o controlador. Garante que a simulação arranque com um número de veículos válido.

Cliente

- **int processar_comando(char *buffer, PEDIDO *p)**

  Interpreta comandos do utilizador, validando a sua sintaxe e prepara a estrutura PEDIDO para enviar ao controlador.

Veículo

- **void trata_cancelamento(int sig, siginfo_t *info, void *ucontext)**

  Handler para sinal SIGUSR1 indicando o cancelamento da viagem.

5. Como funciona a comunicação

Para comunicação entre o controlador e os clientes foram usados named pipes, enquanto para a comunicação entre o controlador e o veículo foram usados pipes anônimos com redirecionamento para o stdout.

Criação dos FIFOs

- Quando o controlador inicia cria o FIFO principal crtl_fifo que fica à escuta dos pedidos dos clientes.
- Cada cliente cria o seu FIFO privado com base no seu PID, cli_<pid>, que vai permitir que o controlador envie respostas para o cliente correto.

Fluxo Comunicação (Cliente->Servidor)

- Sempre que um cliente executa um comando, por exemplo, agendar, consultar, cancelar, o cliente preenche a estrutura PEDIDO com o tipo de Comando, e o resto dos dados referentes a esse comando, por exemplo o agendar, preenche os campos hora, distancia e local, assim como o seu PID, para depois o controlador saber para que fifo há-de enviar a resposta.
- A seguir o cliente depois de ter a estrutura PEDIDO preenchida, envia uma mensagem pelo FIFO principal do Controlador.
- A thread_pedidos lê o FIFO crtl_fifo e encaminha o comando para tratamento.

Fluxo Comunicação (Servidor->Cliente)

- Após processar o comando, envia uma estrutura RESPOSTA para o fifo cli_<PID>, tendo a estrutura resposta o campo res, que indica o sucesso ou insucesso do pedido, assim como uma mensagem a detalhar o mesmo.
- O cliente, a partir do select(), é avisado que chegou a resposta vinda do controlador a partir do seu FIFO privado, processa se o pedido foi aceito ou não e mostra a resposta ao utilizador.

Controlador -> Veículo (Arranque do Veículo)

- Quando chega a hora de iniciar um serviço agendado (verificado pela thread_tempo()), o controlador lança um processo veículo para executar a viagem.
- Para isso, primeiro cria o pipe anónimo pipe(canal), para permitir que o veículo envie mensagens para o controlador através do stdout, que será redirecionado para o canal[1], descritor de escrita.
- Cria um novo processo com o fork(), faz o redirecionamento do stdout, executando depois o processo veiculo. No processo pai, o descritor de escrita é fechado e agora o controlador apenas lê a informação vinda do pipe.
- Seguidamente, preenchem-se os dados para a thread de telemetria do veículo e cria-se a mesma.

Veículo -> Controlador (Telemetria)

- O veículo envia mensagens através de printf, que são capturadas pela thread de telemetria, que interpreta o conteúdo das mensagens.
- As mensagens de início de viagem, cancelamento a meio e final da viagem, são redirecionadas pelo controlador ao cliente através do FIFO privado do cliente.
- Além de reenviar estas mensagens o controlador também as imprime para saber o estado do serviço, assim como a percentagem do progresso (a cada 10%), onde este atualiza a estrutura veículo, para depois poder consultar os percursos das viagens de cada veículo.
- Quando a viagem termina ou é cancelada a thread telemetria depois incrementa os km totais, liberta o veículo e dá por finalizado o serviço, atualizando a tabela.

Cancelamento de Viagens (Controlador → Veículo com SIGUSR1)

- Quando o controlador pede para cancelar um serviço que está em execução, o controlador, vai à procura do PID do veículo, com o id do serviço a cancelar e envia um sinal SIGUSR1 ao veículo. O veículo a partir do handler trata_cancelamento, sinaliza a viagem como cancela e envia a informação ao controlador.

Gestão de Múltiplas Conexões e sincronizações

- O controlador tem várias threads, uma para ler pedidos dos clientes e fazer o processamento dos mesmo, a thread tempo e uma thread telemetria para cada veiculo. Todas elas compartilham as estruturas de dados, para serviços, veículos, contadores e tempo. Por isso, esses dados devem ser protegidos, para que as threads não mexam nos dados ao mesmo tempo. Para isso, quando uma thread vai acessar a esses dados compartilhados utilizamos os mutex, para que as outras esperem, evitando assim race conditions, prevenindo inconsistência nos dados.

Encerramento Controlado

- No caso do cliente, quando emite o comando terminar, este envia o pedido de encerramento ao controlador via FIFO, se este tiver algum serviço em execução, o controlador envia a resposta ao Cliente pelo FIFO privado do mesmo, indicando que o pedido foi negado.
- No caso, de não ter nenhum serviço em execução, o controlador cancela as viagens agendados do cliente (se tiver), atualiza a tabela de utilizadores e serviços e envia a resposta ao cliente que pode encerrar. A seguir remove o seu FIFO privado e fecha o mesmo e o do controlador.
- No caso do controlador, quando o administrador do controlador emite o comando terminar, o controlador cancela todos os serviços ativos e atualiza as tabelas de dados, notifica todos os clientes que vai encerrar para procederem ao encerramento descrito no último ponto. A seguir acorda as threads bloqueadas utilizando o handler naofaznada(), com os phtread_join espera pelo encerramento das threads fechando lá os fifos, destroi os mutex e remove o FIFO principal.

6. Tabela das funcionalidades requeridas

Cliente

Funcionalidade	**Estado**
Criar FIFO privado (cli_pid)	Cumprido
Verificar controlador ativo	Cumprido
Utilizador com username único	Cumprido
Enviar comandos ao controlador	Cumprido
Ler respostas do controlador	Cumprido
Leitura de stdin e pipe simultaneamente	Cumprido
Comando agendar	Cumprido
Comando consultar	Cumprido
Comando Cancelar <id>	Cumprido
Comando terminar	Cumprido
Receber informações do Veículo	Cumprido
Encerramento limpo	Cumprido

Veículo

Funcionalidade	**Estado**
Recebe argumentos corretos	Cumprido
Mensagem inicial (Chegar ao local)	Cumprido
Simulação da viagem (1 km/seg)	Cumprido
Enviar percentagens a cada 10%	Cumprido
Cancelamento via SIGUSR1	Cumprido
Imprimir “viagem cancelada a meio”	Cumprido
Imprimir “Viagem concluída”	Cumprido
Terminação limpa	Cumprido

Controlador

Funcionalidade	**Estado**
Um único Controlador	Cumprido
Criar FIFO principal (crtl_fifo)	Cumprido
Gerir Utilizadores (login / terminar)	Cumprido
Leitura da variável ambiente NVEICULOS	Cumprido
Agendar Serviços	Cumprido
Iniciar serviço no tempo certo	Cumprido
Cancelar Serviços agendados	Cumprido
Cancelar Serviços em execução	Cumprido
Contar km totais	Cumprido
Incrementar tempo simulado	Cumprido
Listar serviços	Cumprido
Listar utilizadores	Cumprido
Listar frota	Cumprido
Leitura de pipes e stdin simultaneamente	Cumprido
Gerir simultaneamente tempo, pedidos e telemetria	Cumprido
Sincronização com mutex	Cumprido
Encerramento seguro	Cumprido

Comunicação

Funcionalidade	**Estado**
Comunicação Cliente -> Controlador via Named Pipe	Cumprido
Comunicação Controlador -> Cliente via Named Pipe	Cumprido
Comunicação veículo -> Controlador via	Cumprido

pipe anônimo	
Comunicação Veiculo -> Cliente	Cumprido
Leitura de telemetria	Cumprido
Cancelamento via sinal (SIGUSR1)	Cumprido
Sincronização de Threads	Cumprido

7. Conclusão

A realização deste trabalho permitiu-nos aplicar na prática vários conceitos fundamentais de Sistemas Operativos, entre eles, criação e gestão de processos, gestão da comunicação entre os mesmos, sincronização de threads para proteção de dados com mutex.

A implementação do programa permitiu-nos entender a importância de garantir consistência em ambientes concorrentes, assegurar um encerramento seguro dos processos, assim como garantir a integridade e consistência de dados.

No global, o projeto deu-nos a possibilidade de consolidar de forma clara e prática os conhecimentos adquiridos ao longo do semestre.