

Sistemas Operativos 25/26

Trabalho Prático - Programação em C para UNIX

Relatório - Simulação de uma Plataforma de Gestão de Táxis Autónomos

Guilherme Ferreira Reis Eça – 2024143849

Miguel Zara – 2024143873

Índice

Índice	1
1. Introdução	2
2.Arquitetura do Sistema	2
2.1 Componentes do Sistema	3
Controlador (Gestor Central)	3
Cliente (Interface de Utilizador)	3
Veículo (Simulação)	3
2.2 Mecanismos de Comunicação entre Processos e Sincronização	3
Named Pipes (FIFOs):	3
Anonymous Pipes:	3
Sinais (Signals):	4
3. Estratégia de Implementação e Opções Tomadas	4
3.1 Gestão de Memória e Estruturas de Dados	4
3.2 Modelo de Concorrência e Sincronização	4
3.3 Gestão de Sinais e I/O Bloqueante	5
Leitura de Comandos (Standard I/O)	5
Interrupção Forçada de Bloqueios	5
Comunicação Enriquecida por Sinais	5
Protocolo de Encerramento Condicional (Client-Side)	5
3.4 Criação de Processos e Redirecionamento de I/O	5
4. Estruturas de Dados	6
4.1 Definição de Limites e Constantes	6
4.2 Entidades Principais	6
4.3 Estrutura de Controlo Central	6
5.Tabela de Funcionalidades	6
Geral	7
Controlador	7
Cliente	7
Veículo	7
6. Instruções de Utilização	8
6.1 Compilação	8
6.2 Execução	8
7.Conclusão	8

1. Introdução

O presente relatório descreve o trabalho prático desenvolvido no âmbito da unidade curricular de Sistemas Operativos, que consiste na conceção e implementação de uma plataforma de simulação para a gestão de uma frota de táxis autónomos em ambiente UNIX.

O objetivo central do projeto é aplicar os conhecimentos adquiridos sobre a API do sistema operativo, focando-se na manipulação de processos, *threads*, mecanismos de comunicação entre processos e sincronização. O sistema desenvolvido permite que múltiplos utilizadores solicitem serviços de transporte, que são geridos por um controlador central e executados por processos que simulam veículos autónomos.

A arquitetura da solução baseia-se em três componentes principais, conforme especificado no enunciado, controlador, cliente e veículo. Para a concretização deste sistema foram usados os conteúdos lecionados nas aulas recorrendo a chamadas de sistema (*syscalls*) para a criação de processos (**fork**, **exec**), gestão de multithreading(**pthread**s), comunicação (**named pipes** e **pipes anónimos**) e gestão de eventos assíncronos através de sinais (**SIGINT**, **SIGUSR1**, **SIGALRM**).

Um aspecto crítico que foi considerado no desenvolvimento foi a eficiência na utilização dos recursos do sistema. A implementação privilegia mecanismos que evitam a "espera ativa", optando por abordagens reativas e bloqueantes, como a multiplexagem de I/O (**select**) na aplicação Cliente e o uso de **mutexes** no Controlador para garantir a integridade dos dados sem desperdício de processamento.

O presente documento encontra-se estruturado de forma a detalhar a Arquitetura desenhada para responder aos requisitos, descrevendo as Estruturas de Dados utilizadas, as Estratégias de Implementação adotadas e os Mecanismos de Segurança.

Por fim, é apresentada uma tabela com as funcionalidades pedidas e cumpridas e conclusão.

2. Arquitetura do Sistema

A solução desenvolvida baseia-se numa arquitetura distribuída multiprocesso, desenhada para operar em ambiente UNIX. O sistema foi decomposto em três componentes fundamentais — **Controlador**, **Cliente** e **Veículo**.

A arquitetura privilegia a separação de responsabilidades e a não-bloqueabilidade dos processos, garantindo que a interface com o utilizador permanece responsiva mesmo enquanto ocorrem operações de gestão de frota ou simulação de deslocações.

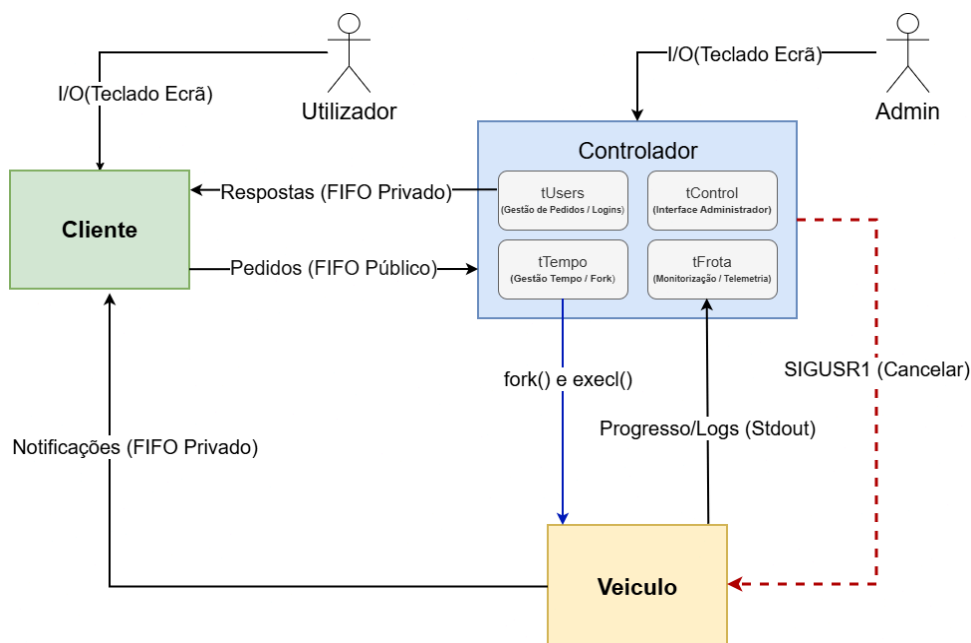


Figura 1: Diagrama da Arquitetura do Sistema e Fluxos de Comunicação

2.1 Componentes do Sistema

Controlador (Gestor Central)

O Controlador atua como o núcleo do sistema. A sua implementação segue um modelo **Multithreaded** para garantir uma resposta eficaz aos múltiplos eventos simultâneos (pedidos de clientes, comandos de administração, gestão do tempo e monitorização de veículos). O processo é dividido em quatro *threads* distintas, sincronizadas através de *mutexes* para garantir a consistência dos dados partilhados (listas de utilizadores, serviços e frota):

—**Thread Clientes** (`tClientes`): Responsável por ler pedidos do *named pipe* principal e processar logins, agendamentos e cancelamentos.

—**Thread Controlo** (`tControl`): Gere a interface de administração local, processando comandos como **listar**, **frota** ou **terminar**.

—**Thread Tempo** (`tTempo`): Gere o relógio da simulação e verifica periodicamente se existem serviços agendados prontos a iniciar, respeitando a disponibilidade da frota e dos utilizadores. É responsável pela criação dos processos Veículo (**fork/exec**).

—**Thread Frota** (`tFrota`): Monitoriza o estado dos veículos ativos, lendo o progresso enviado por estes através de *pipes* *anónimos*.

Cliente (Interface de Utilizador)

O Cliente fornece a interface para os utilizadores interagirem com a plataforma. Ele lida com as comunicações assíncronas do utilizador. O utilizador pode estar a escrever um comando ao mesmo tempo que recebe uma notificação de chegada de um veículo. Para evitar o bloqueio do processo numa leitura (esperar pelo teclado ou esperar por uma mensagem), foi utilizado o mecanismo de multiplexagem de I/O **select()**. Isto permite ao Cliente monitorizar simultaneamente o **STDIN** e o seu *Named Pipe* de receção.

Veículo (Simulação)

Cada serviço em execução corresponde a um processo Veículo independente, criado pelo Controlador. Este processo simula o comportamento físico do táxi, calculando o progresso da viagem com base no tempo simulado. O Veículo opera de forma autónoma, reportando o seu estado tanto ao Controlador (para gestão) como diretamente ao Cliente (para notificações de serviço).

2.2 Mecanismos de Comunicação entre Processos e Sincronização

A interação entre os processos é assegurada através de uma combinação de mecanismos de IPC (*Inter-Process Communication*) nativos do UNIX:

Named Pipes (FIFOs):

No que respeita aos **Named Pipes (FIFOs)**, estabelecemos dois tipos de canais de comunicação distintos. O primeiro é o **Canal Público (PIPE_CONTROLADOR)**, um recurso partilhado utilizado por todos os processos Cliente para submeter pedidos de *login* e enviar comandos iniciais ao Controlador. Depois ainda criamos **Canais Privados (PIPE_CLIENTE_<nome>)** para cada Cliente. Estes canais foram criados para assegurar a receção de respostas por parte do Controlador e para permitir a entrega direta de notificações de serviço provenientes dos processos Veículo, sem interferência entre utilizadores.

Anonymous Pipes:

Utilizados para a comunicação unidirecional **Veículo -> Controlador**. O descritor de escrita do *pipe* é mapeado para o **STDOUT** do processo Veículo, permitindo que a *thread* **tFrota** do Controlador leia o progresso da viagem (mensagens de "10%" e conclusão) de forma transparente.

Sinais (Signals):

O sistema ainda recorre ao mecanismo de **Sinais (Signals)** para três funções críticas. O sinal **SIGINT** (interrupção) utilizamos para desencadear o encerramento seguro de todos os componentes. Depois para a gestão temporal, configuramos o sinal **SIGALRM** para gerar interrupções periódicas de um segundo, servindo para o avanço do tempo simulado e o escalonamento de serviços. Por fim, o sinal **SIGUSR1** possui um duplo propósito na arquitetura, **Controlador \rightarrow Veículo: Força o cancelamento imediato do serviço.** **Controlador \rightarrow Cliente: Notifica o Cliente de que o sistema vai encerrar. Neste caso, o Cliente termina silenciosamente (sem enviar "LOGOUT"), evitando tráfego redundante.**

3. Estratégia de Implementação e Opções Tomadas

Na implementação do sistema priorizamos a robustez, a modularidade e a segurança na gestão de memória. Em detrimento de soluções simplistas baseadas em variáveis globais, optamos por uma abordagem estruturada que facilita a manutenção e evita efeitos colaterais indesejados entre as diferentes *threads*.

3.1 Gestão de Memória e Estruturas de Dados

Uma das principais decisões de arquitetura foi a **abolição de variáveis globais** para o armazenamento do estado do sistema. Em sua substituição, foi definida uma estrutura central, **SistemaControlador**, que encapsula todas as variáveis: os *arrays* de utilizadores, serviços e veículos, bem como as variáveis de controlo de tempo e *mutexes*.

Esta estrutura é alocada dinamicamente na *Heap* no início da execução do Controlador. O acesso a estes dados por parte das várias *threads* é feito através da passagem de um ponteiro (**SistemaControlador *sys**) como argumento na função **pthread_create**. Esta abordagem garante que todas as *threads* operam sobre o mesmo espaço de memória partilhado, mantendo o encapsulamento e permite um controlo rigoroso sobre o ciclo de vida dos dados. A única exceção é um ponteiro global estritamente necessário para o funcionamento dos *handlers* de sinais, devido às limitações da assinatura da função **sigaction**.

3.2 Modelo de Concorrência e Sincronização

Pelo facto do sistema ser assíncrono, foi preciso implementar com controlo rigoroso de concorrência. Para evitar **erros de sincronização** que poderiam ocorrer quando múltiplas *threads* tentam acessar ou modificar os dados compartilhados de forma concorrente, foi implementada uma estratégia de **Exclusão Mútua** baseada em *Mutexes*. Desta forma, conseguimos garantir que operações, como login/logout de utilizadores, manipulação de serviços e controlo do estado dos veículos, sejam realizadas de maneira **atômica e sem interferência de outras threads**.

→users_mutex: Protege o acesso ao *array* de utilizadores (login/logout)

→servicos_mutex: Garante a integridade na manipulação da lista de serviços.

→frota_mutex: Controla a leitura e escrita no estado da frota de veículos

Esta separação, ilustrada na **Figura 2**, evita o uso de um bloqueio global único, maximizando o paralelismo do sistema. Por exemplo, permite que a *thread* do Tempo verifique a disponibilidade da frota sem bloquear a capacidade da *thread* de Clientes de processar novos pedidos.

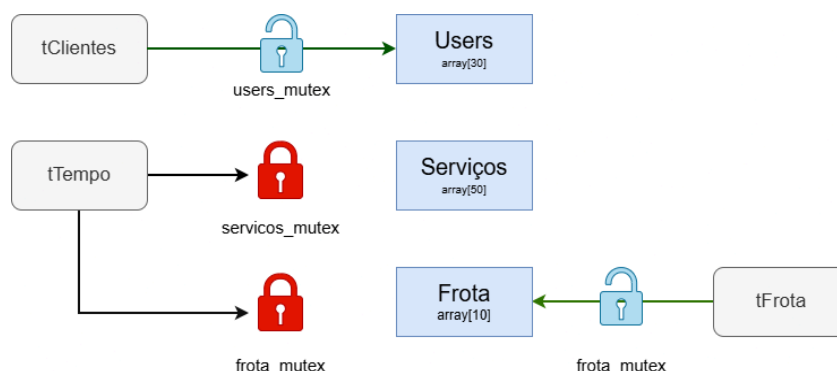


Figura 2: Estratégia de Sincronização dos *Mutexes*.

3.3 Gestão de Sinais e I/O Bloqueante

A interação entre a entrada de dados do utilizador e a receção de sinais assíncronos (como interrupções ou temporizadores) constituiu um dos principais desafios de implementação. A estratégia adotada baseou-se em três pilares fundamentais para garantir a reatividade e a robustez do sistema:

Leitura de Comandos (Standard I/O)

Para a interface de administração, optou-se pela utilização da função de biblioteca **fgets()**. Esta escolha permite uma leitura de *strings* segura e simplificada a partir do STDIN, garantindo o tratamento adequado de *buffers* de entrada sem a complexidade de gestão manual de caracteres exigida pelas *syscalls* de baixo nível.

Interrupção Forçada de Bloqueios

Dado que a função **fgets()** é bloqueante, o sistema poderia ficar "preso" à espera de input durante um pedido de encerramento (**SIGINT**). Para resolver este problema sem abdicar do **fgets**, implementamos uma estratégia de cancelamento ativo. No *handler* do sinal **SIGINT**, o processo principal não só altera as variáveis de controlo (**loop = 0**), como também invoca explicitamente **pthread_cancel()** direcionado à *thread* de controlo. Esta ação força a interrupção imediata da operação de I/O pendente, garantindo que o programa encerra instantaneamente sem obrigar o administrador a pressionar "Enter".

Comunicação Enriquecida por Sinais

Para o protocolo de encerramento e comunicação de emergência com os clientes, usamos o **sigqueue** (extensão *Real-time Signals*). Ao contrário do **kill** tradicional, esta abordagem permite acoplar dados auxiliares ao sinal enviado e é a maneira mais moderna de enviar sinais, preparando a arquitetura para cenários onde seja necessário comunicar códigos de estado específicos aquando da notificação de eventos críticos.

Protocolo de Encerramento Condicional (Client-Side)

Para garantir a integridade dos dados no encerramento, implementou-se uma lógica de terminação dependente do estado no processo Cliente. A rotina de tratamento de sinais (**handleSinal**) distingue a origem da ordem de paragem:

- **Origem Local (SIGINT):** Se o utilizador interromper o Cliente, a variável de estado **logout** é ativada. O Cliente envia formalmente o comando "LOGOUT" ao Controlador antes de sair, permitindo a atualização da lista de utilizadores ativos.
- **Origem Remota (SIGUSR1):** Se o Controlador for desligado, este envia um sinal SIGUSR1 (via **sigqueue**) a todos os clientes. Neste cenário, a *flag* **logout** é desativada, instruindo o Cliente a terminar a execução imediatamente sem tentar contactar o Controlador (que já se encontra em processo de fecho), prevenindo assim erros de comunicação (*Broken Pipe*) e redundância de mensagens.

3.4 Criação de Processos e Redirecionamento de I/O

A simulação dos veículos exige que cada unidade opere como um processo independente, garantindo que uma falha ou atraso num veículo não afete o Controlador. Para tal, foi implementado um mecanismo de criação e configuração de processos:

Criação do Canal (pipe)

Antes de criar o processo filho, a *thread* **tTempo** cria um *Anonymous Pipe*. Este canal é unidirecional e serve exclusivamente para o veículo enviar dados de telemetria.

Duplicação do Processo (fork)

O Controlador invoca **fork()**, criando uma cópia exata de si mesmo.

Redirecionamento do STDOUT (dup)

No processo filho (Veículo), antes de carregar o novo programa, manipulam-se os descritores de ficheiro. O descritor de saída padrão (STDOUT_FILENO) é fechado e substituído pela extremidade de escrita do *pipe* (usando a *syscall* **dup**).

Substituição da Imagem (exec1):

Finalmente, a função `exec1` substitui a memória do processo filho pelo binário do programa `veiculo`, passando os argumentos necessários (ID, Origem, Distância e Pipe do Cliente) via linha de comandos. Do lado do Controlador (Pai), a extremidade de leitura do *pipe* é configurada como **Não Bloqueante** (`O_NONBLOCK`). Isto serve para que a *thread* de monitorização (`tFrota`) possa verificar o estado de múltiplos veículos num único ciclo sem ficar "presa" à espera que um veículo específico escreva algo.

4. Estruturas de Dados

A organização da informação no sistema foi centralizada no ficheiro de cabeçalho **`comum.h`**, garantindo que tanto o Controlador como os processos Veículo e Cliente partilham as mesmas definições de tipos e constantes. As estruturas de dados foram desenhadas para serem compactas e conterem apenas a informação necessária para a gestão de estado e comunicação entre processos.

4.1 Definição de Limites e Constantes

Para cumprir os requisitos de alocação estática e limites do sistema definidos no enunciado, foram estabelecidas constantes que dimensionam os *arrays* principais:

MAX_USERS (30): Define o limite máximo de utilizadores simultâneos na plataforma.

MAX_VEICULOS (10): Estabelece o teto para a frota, embora o número real seja configurável via variável de ambiente `NVEICULOS`.

MAX_SERVICOS (50): Limita o histórico e fila de serviços ativos/agendados.

4.2 Entidades Principais

O modelo de dados baseia-se em três estruturas fundamentais que representam os atores do sistema:

struct User: Armazena a sessão de um cliente. Guarda o **nome**, o **pid_cliente**, o caminho do seu **fifo_privado** (para envio de respostas dedicadas) e ainda uma flag **ativo** para gestão de logins/logouts.

struct Servico: Representa um pedido de transporte. Contém os detalhes do agendamento (**id**, **hora_agendada**, **origem**, **distancia**) e o **estado** atual (`SERV_AGENDADO`, `SERV_EM_CURSO`, `SERV_CONCLUIDO`). Faz a ligação entre as entidades guardando o **pid_cliente** e o **pid_veiculo** atribuído.

struct Veiculo: Mantém o estado da frota. Para além do **pid_veiculo** e do estado (`LIVRE/OCUPADO`), guarda o descritor de ficheiro **fd_leitura** do *pipe* anónimo, permitindo à *thread* de monitorização (`tFrota`) ler o progresso da viagem em tempo real.

4.3 Estrutura de Controlo Central

Para suportar a arquitetura sem variáveis globais dispersas, foi criada a estrutura **SistemaControlador**. Esta estrutura agrega todos os *arrays* mencionados acima (`users`, `servicos`, `frota`), as variáveis de sincronização (*mutexes*) e os dados de controlo de execução (*flags* de loop e tempo). Esta abordagem permite passar todo o contexto do sistema às diversas *threads* através de um único ponteiro.

5. Tabela de Funcionalidades

Geral

Funcionalidade	Estado	Obs / Justificação
Arquitetura Multiprocesso	Cumprido	Sistema dividido em Cliente, Controlador e Veículo.
Comunicação entre os processos (IPC)	Cumprido	Uso de <i>Named Pipes</i> , <i>Anonymous Pipes</i> e Sinais conforme exigido.

Controlador

Funcionalidade	Estado	Obs / Justificação
Comando listar	Cumprido	Lista todos os serviços (agendados e em curso).
Comando utiliz	Cumprido	Lista utilizadores conectados.
Comando frota	Cumprido	Mostra o progresso (%) de cada veículo ativo.
Comando cancelar <id>	Cumprido	Cancela serviços específicos ou todos (id=0).
Comandos km e hora	Cumprido	Contabilização correta de km totais e tempo simulado.
Comando terminar	Cumprido	Encerramento gracioso com notificação (sigqueue) aos clientes.
Gestão de Frota	Cumprido	Respeita o limite NVEICULOS (fila de espera implementada).

Cliente

Funcionalidade	Estado	Obs / Justificação
Login / Identificação	Cumprido	Validação de nomes únicos no arranque.
Comando agendar	Cumprido	Envio de pedidos ao controlador via <i>Named Pipe</i> .
Comando consultar	Cumprido	Visualização dos serviços próprios.
Comando cancelar	Cumprido	Cancelamento de serviços pendentes.
Receção Assíncrona	Cumprido	Implementado com select() .

Veículo

Funcionalidade	Estado	Obs / Justificação
Simulação de Viagem	Cumprido	Avanço de 1km/s (tempo simulado).
Telemetria (10%)	Cumprido	Reporta progresso ao Controlador via <i>stdout</i> .
Notificações	Cumprido	Informa o Cliente diretamente via <i>Named Pipe</i> privado.

Interrupção	Cumprido	Reage ao sinal SIGUSR1 para cancelamento imediato.
-------------	----------	--

6. Instruções de Utilização

O projeto inclui um `Makefile` configurado com as regras exigidas (`all`, `controlador`, `cliente`, `veiculo`, `clean`) para facilitar a compilação e execução.

6.1 Compilação

Para compilar todo o sistema, executar o comando na raiz do projeto

```
make all
```

6.2 Execução

A ordem de execução deve ser respeitada para garantir a correta criação dos canais de comunicação (FIFOs).

Iniciar o Controlador

```
./bin/controlador
```

(Nota: Pode definir o limite de veículos via variável de ambiente: `export NVEICULOS=5`)

Iniciar um ou mais Clientes

```
./bin/cliente <nome_utilizador>
```

7. Conclusão

O trabalho prático permitiu consolidar a aplicação dos conceitos de programação de sistemas em UNIX, nomeadamente a gestão de processos, a concorrência e a comunicação IPC. A solução desenvolvida destaca-se pela abolição de variáveis globais em favor de uma estrutura de controlo centralizada (`SistemaControlador`) e pela implementação de uma estratégia de sincronização fina com *mutexes*, prevenindo conflitos de acesso concorrente e assegurando a integridade dos dados. A robustez do sistema foi assegurada através de mecanismos de I/O e de um protocolo de encerramento seguro com sinais em tempo real (*sigqueue*). Em suma, a plataforma cumpre integralmente os requisitos propostos, demonstrando uma arquitetura modular, estável e eficiente na gestão de recursos do sistema.