UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
CURSO DE ESPECIALIZAÇÃO EM DESENVOLVIMENTO DE SOFTWARE PARA O SETOR AUTOMOTIVO

VICTOR AUGUSTO MEDEIROS BALBINO

SIMULADOR DE CONTROLE VEICULAR

Recife-PE
2025

VICTOR AUGUSTO MEDEIROS BALBINO

SIMULADOR DE CONTROLE VEICULAR

Projeto de pesquisa apresentado como requisito à obtenção de aprovação na disciplina de Padrão Posix do Curso de Especialização em Desenvolvimento de Software para o Setor Automotivo da Universidade Federal de Pernambuco.

Prof. Dr. Renato Coral Sampaio .

Recife-PE
2025

Sumário

1 Introdução

Este relatório detalha as decisões de design e a implementação de um sistema distribuído composto por dois módulos principais: Controlador Principal, Simulador de Sensores e Painel de Comandos. O sistema foi projetado com base em conceitos de programação paralela, sincronização de processos e comunicação interprocessual, utilizando as APIs e recursos fornecidos pelo padrão POSIX (Portable Operating System Interface).

O objetivo do projeto é simular e monitorar dados de sensores em tempo real, permitindo a comunicação segura entre os módulos e o controle dinâmico de estados internos.

2 Estrutura do Projeto

1. Painel de Controle (painel.c): Interface de usuário para enviar comandos ao simulador.
2. Controlador (controlador.c): Processa os comandos recebidos do painel e atualiza a memória compartilhada.
3. Simulador (simulador.c): Simula o comportamento do veículo com base nos comandos recebidos e atualiza os sensores.

3 Decisões de design

3.1 Modularização

O projeto foi modularizado em três componentes principais para separar as responsabilidades e facilitar a manutenção:

- Painel de Controle: Responsável por capturar os comandos do usuário e enviá-los ao controlador.
- Controlador: Processa os comandos e atualiza a memória compartilhada.
- Simulador: Simula o comportamento do veículo e atualiza os sensores.

3.2 Comunicação entre Processos

Para a comunicação entre os componentes, foram utilizadas duas técnicas principais:

- Memória Compartilhada: Utilizada para compartilhar os dados dos sensores entre o controlador e o simulador.
- Filas de Mensagens: Utilizadas para enviar comandos do painel de controle para o controlador.

3.3 Conceitos POSIX Aplicados

3.4 Memória Compartilhada

A memória compartilhada foi utilizada para armazenar os dados dos sensores, permitindo que o controlador e o simulador acessem e atualizem esses dados de forma eficiente.

- Criação e Mapeamento:

```
int fd = shm_open(SHARED_MEMORY_NAME, O_CREAT | O_RDWR, 0666);
ftruncate(fd, SHARED_MEMORY_SIZE);
DadosSensores *shared_mem = mmap(NULL, SHARED_MEMORY_SIZE,
    PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

- Desmapeamento e Unlink:

```
munmap(shared_mem, SHARED_MEMORY_SIZE);
close(fd);
shm_unlink(SHARED_MEMORY_NAME);
```

3.5 Filas de Mensagens

As filas de mensagens foram utilizadas para enviar comandos do painel de controle para o controlador, garantindo a comunicação assíncrona entre os processos.

- Criação e Envio de Mensagens:

```
mqd_t mq = mq_open(FILA_COMANDOS, O_WRONLY | O_CREAT, 0644, &attr);
mq_send(mq, (const char *)&msg, sizeof(msg), 0);
```

- Recebimento de Mensagens:

```
mqd_t mq = mq_open(MESSAGE_QUEUE_NAME, O_RDONLY | O_CREAT, 0666,
    NULL);
mq_receive(mq, (char *)&comando, sizeof(comando), NULL);
```

3.6 Threads e Sincronização

O projeto utiliza threads para simular os sensores e processar comandos de forma concorrente. A sincronização entre threads é garantida utilizando mutexes.

- Criação de Threads:

```
pthread_create(&thread_velocidade, NULL, sensor_velocidade, (void
    *)shared_mem);
```

- Uso de Mutexes:

```
pthread_mutex_lock(&sensores->mutex);
// Acesso aos dados compartilhados
pthread_mutex_unlock(&sensores->mutex);
```

3.7 Sinais

Os sinais POSIX foram utilizados para controlar a execução do controlador, permitindo pausar, retomar e encerrar o programa de forma controlada.

- Configuração dos Manipuladores de Sinal:

```
signal(SIGUSR1, signal_handler);
signal(SIGCONT, signal_handler);
signal(SIGUSR2, signal_handler);
```

- Manipulador de Sinais:

```
void signal_handler(int signo) {
    if (signo == SIGUSR1) {
        executar = 0; // Pausar
    } else if (signo == SIGCONT) {
        executar = 1; // Retomar
    } else if (signo == SIGUSR2) {
        executar = -1; // Encerrar
    }
}
```

4 Detalhamento dos Componentes

4.1 Painel de Controle (painel.c)

O painel de controle é responsável por capturar os comandos do usuário e enviá-los ao controlador através de uma fila de mensagens.

- Configuração da Fila de Mensagens:

```
struct mq_attr attr;
attr.mq_flags = 0;
attr.mq_maxmsg = 10;
attr.mq_msgsize = sizeof(msg);
attr.mq_curmsgs = 0;
```

- Envio de Comandos:

```
mqd_t mq = mq_open(FILA_COMANDOS, O_WRONLY | O_CREAT, 0644, &attr);
mq_send(mq, (const char *)&msg, sizeof(msg), 0);
```

4.2 Controlador (controlador.c)

O controlador processa os comandos recebidos do painel e atualiza a memória compartilhada com os dados dos sensores.

- Recebimento de Comandos:

```
mqd_t mq = mq_open(MESSAGE_QUEUE_NAME, O_RDONLY | O_CREAT, 0666,
    NULL);
mq_receive(mq, (char *)&comando, sizeof(comando), NULL);
```

- Atualização da Memória Compartilhada:

```
pthread_mutex_lock(&sensores->mutex);
sensores->comando = comando.comando;
pthread_mutex_unlock(&sensores->mutex);
```

4.3 Simulador (simulador.c)

O simulador lê os comandos da memória compartilhada e ajusta os estados internos (velocidade, RPM, temperatura) com base nos comandos recebidos.

- Leitura de Comandos:

```
pthread_mutex_lock(&sensores->mutex);
int comando = sensores->comando;
sensores->comando = 0;
pthread_mutex_unlock(&sensores->mutex);
```

- Atualização dos Estados Internos:

```
void atualizar_estados(int comando) {
    float aceleracao = 0.0;
    if (comando == 1) {
        aceleracao = 5.0;
    } else if (comando == 2) {
        aceleracao = -7.0;
    } else {
        aceleracao = -0.5;
    }
    estados.estado_velocidade += aceleracao;
    if (estados.estado_velocidade < 0) {
```

```
            estados.estado_velocidade = 0;
        }
        estados.estado_rpm = estados.estado_velocidade * 40;
        estados.estado_temperatura += (estados.estado_rpm / 1000.0) *
            0.2;
        estados.estado_temperatura -= 0.1;
        if (estados.estado_temperatura < 20.0) {
            estados.estado_temperatura = 20.0;
        }
}
```

5 Conceitos posix aplicados

5.1 Memória Compartilhada

A memória compartilhada é uma técnica de IPC (Inter-Process Communication) que permite que múltiplos processos acessem uma região de memória comum. Isso é útil para compartilhar grandes volumes de dados de forma eficiente, evitando a necessidade de copiar dados entre processos. **Criação e Mapeamento:**

- *shm_open*: Cria ou abre um objeto de memória compartilhada.
- ftruncate: Define o tamanho do objeto de memória compartilhada.
- mmap: Mapeia o objeto de memória compartilhada no espaço de endereço do processo.

Desmapeamento e Unlink:

- munmap: Desmapeia a região de memória compartilhada.
- close: Fecha o descritor de arquivo associado ao objeto de memória compartilhada.
- *shm_unlink*: Remove o objeto de memória compartilhada.

5.2 Filas de Mensagens

As filas de mensagens são uma técnica de IPC que permite que processos enviem e recebam mensagens de forma assíncrona. Cada mensagem é colocada em uma fila e pode ser lida por outro processo.

Criação e Envio de Mensagens:

- mq_open: Cria ou abre uma fila de mensagens.
- mq_send: Envia uma mensagem para a fila de mensagens.

Recebimento de Mensagens:

- mq_receive: Recebe uma mensagem da fila de mensagens.
- mq_close: Fecha a fila de mensagens.
- mq_unlink: Remove a fila de mensagens.

5.3 Threads e Sincronização

As threads permitem a execução concorrente de múltiplas tarefas dentro do mesmo processo. A sincronização entre threads é crucial para evitar condições de corrida e garantir a consistência dos dados compartilhados.

Criação de Threads:

- pthread_create: Cria uma nova thread.

Sincronização com Mutexes:

- pthread_mutex_lock: Adquire um mutex, bloqueando outras threads de acessar a região crítica.
- pthread_mutex_unlock: Libera o mutex, permitindo que outras threads acessem a região crítica.
- pthread_mutex_init: Inicializa um mutex.
- pthread_mutex_destroy: Destrói um mutex.

5.4 Sinais

Os sinais são uma técnica de IPC que permite que processos enviem notificações assíncronas uns aos outros. Eles são frequentemente usados para controlar a execução de processos.

Configuração dos Manipuladores de Sinal:

- signal: Define um manipulador de sinal para um sinal específico.

Manipulador de Sinais:

- Um manipulador de sinal é uma função que é chamada quando um processo recebe um sinal específico.

6 Uso do Makefile

Para facilitar a compilação e execução dos programas, foi criado um Makefile que define alvos para compilar e executar cada componente do projeto.

6.1 Estrutura do Makefile

```
# Makefile

CC = gcc
CFLAGS = -Wall -pthread

# Nomes dos execut veis
CONTROLADOR = controlador
SIMULADOR = simulador
PAINEL = painel

# Arquivos fonte
CONTROLADOR_SRC = controlador.c
SIMULADOR_SRC = simulador.c
PAINEL_SRC = painel.c

# Alvo padr o
all: $(CONTROLADOR) $(SIMULADOR) $(PAINEL)

# Compilar o controlador
$(CONTROLADOR): $(CONTROLADOR_SRC)
    $(CC) $(CFLAGS) -o $(CONTROLADOR) $(CONTROLADOR_SRC) -lrt

# Compilar o simulador
$(SIMULADOR): $(SIMULADOR_SRC)
    $(CC) $(CFLAGS) -o $(SIMULADOR) $(SIMULADOR_SRC) -lrt

# Compilar o painel
$(PAINEL): $(PAINEL_SRC)
    $(CC) $(CFLAGS) -o $(PAINEL) $(PAINEL_SRC) -lrt

# Limpar os arquivos compilados
clean:
    rm -f $(CONTROLADOR) $(SIMULADOR) $(PAINEL)

# Executar o controlador e o simulador ao mesmo tempo
run: $(CONTROLADOR) $(SIMULADOR)
    ./$(SIMULADOR) &
    ./$(CONTROLADOR) &
    wait

# Executar o painel
run_painel: $(PAINEL)
    ./$(PAINEL)

# Parar a execu  o do controlador e do simulador
```

```
stop:
    @pkill -f $(CONTROLADOR)
    @pkill -f $(SIMULADOR)
```

7 Conclusões

Este projeto demonstra a aplicação de conceitos POSIX para a criação de um simulador de controle de veículo. A modularização do código, o uso de memória compartilhada, filas de mensagens, threads, mutexes e sinais garantem a eficiência e a sincronização necessária para o funcionamento do sistema. O controlador processa os comandos recebidos do painel e atualiza a memória compartilhada, enquanto o simulador ajusta os estados internos com base nesses comandos, proporcionando uma simulação realista. O uso do Makefile facilita a compilação e execução dos diferentes componentes do projeto, permitindo uma integração eficiente e um fluxo de trabalho simplificado.

Referências