

# RELATÓRIO TÉCNICO: PROGRAMAS DE SISTEMAS OPERACIONAIS EMBARCADOS

**Disciplina:** Sistemas Operacionais Embarcados  
**Aluno:** Fábio Braz  
**Data:** Novembro de 2025  
**Ambiente:** Ubuntu WSL2 no Windows, Visual Studio Code  
**Repositório** [//github.com/FBR4Z/trabalho\\_sis\\_op\\_emb.git](https://github.com/FBR4Z/trabalho_sis_op_emb.git) **Prof.** Fábio de Sousa Cardoso

---

## SUMÁRIO

- [1. Introdução](#)
  - [2. Ambiente de Desenvolvimento](#)
  - [3. Programas Desenvolvidos](#)
  - [4. Desafios e Soluções](#)
  - [5. Conclusão](#)
  - [6. Referências](#)
- 

## 1. INTRODUÇÃO

Este relatório documenta a implementação e execução de 20 programas didáticos que exploram conceitos fundamentais de sistemas operacionais embarcados, incluindo gerenciamento de processos, threads, comunicação entre processos (IPC), sincronização e sistemas operacionais de tempo real (RTOS). Os programas foram desenvolvidos em C e executados em ambiente Linux (Ubuntu via WSL2).

### 1.1 Objetivos

- Compreender mecanismos de criação e gerenciamento de processos
  - Implementar técnicas de comunicação inter-processos (IPC)
  - Aplicar mecanismos de sincronização (semáforos, mutex)
  - Estudar escalonamento de processos e threads
  - Desenvolver aplicações com FreeRTOS
- 

## 2. AMBIENTE DE DESENVOLVIMENTO

### 2.1 Sistema Operacional

- SO Host:** Windows 11
- SO Virtualizado:** Ubuntu 22.04 LTS via WSL2
- IDE:** Visual Studio Code com Remote WSL

## 2.2 Ferramentas e Bibliotecas

- **Compilador:** GCC 11.4.0
- **Bibliotecas POSIX:** pthread, semaphore, mqueue, shm
- **RTOS:** FreeRTOS (clonado de <https://github.com/FreeRTOS/FreeRTOS.git>)
- **Ferramentas de análise:** ps, top, htop

## 2.3 Configuração Inicial

```
# Instalação de dependências
sudo apt update
sudo apt install build-essential gdb git

# Estrutura de diretórios
mkdir -p ~/trabalho_sis_op_emb_fabiobraz
cd ~/trabalho_sis_op_emb_fabiobraz

# Download do FreeRTOS
git clone https://github.com/FreeRTOS/FreeRTOS.git
```

---

# 3. PROGRAMAS DESENVOLVIDOS

## PROGRAMA 01 - Criação de Processos com fork()

**Arquivo:** 01\_fork\_example.c

**Página de Referência:** Aula 2 - Laboratório, pág. 5

**Objetivo:** Demonstrar a criação de processos em Linux utilizando a system call fork() e comunicação básica entre processos pai e filho.

**Conceitos Abordados:** - System call fork() - Identificação de processos (PID) - Diferenciação entre processo pai e filho - Duplicação de espaço de endereçamento

**Código Principal:**

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
    int fd[2];
    char buffer[50];

    pipe(fd); // Cria pipe para comunicação

    if (fork() == 0) {
        // PROCESSO FILHO
        close(fd[0]); // Fecha leitura
        char msg[] = "Mensagem do filho!";
        write(fd[1], msg, strlen(msg)+1);
        close(fd[1]);
    } else {
        // PROCESSO PAI
        close(fd[1]); // Fecha escrita
        read(fd[0], buffer, sizeof(buffer));
    }
}
```

```

        printf("Pai recebeu: %s\n", buffer);
        close(fd[0]);
    }
}

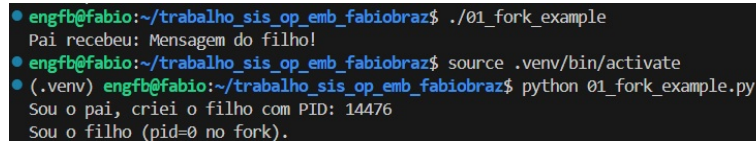
```

### Compilação e Execução:

```

gcc 01_fork_example.c -o fork_example
./fork_example

```



```

engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ ./01_fork_example
Pai recebeu: Mensagem do filho!
engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ source .venv/bin/activate
(.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ python 01_fork_example.py
Sou o pai, criei o filho com PID: 14476
Sou o filho (pid=0 no fork).

```

alt text

### Saída Esperada:

Pai recebeu: Mensagem do filho!

**Análise:** O programa demonstra como `fork()` cria uma cópia do processo atual. O valor de retorno diferencia pai (retorna PID do filho) e filho (retorna 0). A comunicação ocorre através de um pipe anônimo.

---

## PROGRAMA 02 - Comunicação entre Processos (Pipe)

**Arquivo:** 02\_pipe\_example.c

**Página de Referência:** Aula 2 - Laboratório, pág. 6

**Objetivo:** Implementar comunicação unidirecional entre processos pai e filho através de pipes anônimos.

**Conceitos Abordados:** - Pipe anônimo (unnamed pipe) - Descritores de arquivo: `fd[0]` (leitura), `fd[1]` (escrita) - Comunicação unidirecional - Bloqueio em operações de leitura

### Código Principal:

```

int main() {
    int fd[2];
    char buffer[50];

    pipe(fd); // cria o pipe

    if (fork() == 0) {
        close(fd[0]); // fecha leitura
        char msg[] = "Mensagem do filho!";
        write(fd[1], msg, strlen(msg)+1);
        close(fd[1]);
    } else {
        close(fd[1]); // fecha escrita
        read(fd[0], buffer, sizeof(buffer));
        printf("Pai recebeu: %s\n", buffer);
        close(fd[0]);
    }
}

```

### Compilação e Execução:

```
gcc 02_pipe_example.c -o pipe_example
./pipe_example
```

```
• (.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ ./02_pipe_example
Pai recebeu: Mensagem do filho!
• (.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ python 02_pipe_example.py
Pai recebeu: Mensagem do filho!
```

alt text

**Observações Importantes:** - Pipes são buffers de tamanho limitado (geralmente 65536 bytes no Linux) - Operação `read()` bloqueia até haver dados disponíveis - É fundamental fechar descritores não utilizados para evitar deadlock

---

## PROGRAMA 03 - Sincronização com Semáforos

**Arquivo:** 03\_semaphore\_example.c

**Página de Referência:** Aula 2 - Laboratório, pág. 8

**Objetivo:** Demonstrar sincronização entre threads utilizando semáforos POSIX.

**Conceitos Abordados:** - Semáforos POSIX (`sem_t`) - Operações `sem_wait()` e `sem_post()` - Seção crítica - Exclusão mútua

**Código Principal:**

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t sem;

void* tarefa(void* arg) {
    sem_wait(&sem); // P(sem) - decrementa
    printf("Thread %s executando\n", (char*)arg);
    sem_post(&sem); // V(sem) - incrementa
    return NULL;
}

int main() {
    pthread_t t1, t2;

    sem_init(&sem, 0, 1); // semáforo binário (mutex)

    pthread_create(&t1, NULL, tarefa, "A");
    pthread_create(&t2, NULL, tarefa, "B");

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    sem_destroy(&sem);
    return 0;
}
```

**Compilação e Execução:**

```
gcc 03_semaphore_example.c -o semaphore_example -lpthread
./semaphore_example
```

```

• (.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ python 03_semaphore_example.py
Thread A executando
Thread B executando
• (.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ ./03_semaphore_example
Thread A executando
Thread B executando

```

alt text

### Saída Esperada:

Thread A executando  
Thread B executando

**Análise:** O semáforo inicializado com valor 1 funciona como mutex, garantindo que apenas uma thread execute a seção crítica por vez.

---

## PROGRAMA 04 - Tratamento de Sinais (Signal Handling)

**Arquivo:** 04\_signal\_example.c

**Página de Referência:** Aula 2 - Laboratório, pág. 11

**Objetivo:** Capturar e tratar sinais do sistema operacional, especificamente SIGINT (Ctrl+C).

**Conceitos Abordados:** - Sinais Unix/Linux - Manipuladores de sinais (signal handlers) - SIGINT (interrupção por teclado) - Tratamento assíncrono de eventos

### Código Principal:

```

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

void handler(int sig) {
    printf("Recebi sinal %d! Encerrando programa...\n", sig);
    exit(0);
}

int main() {
    signal(SIGINT, handler); // Captura Ctrl+C

    while (1) {
        printf("Executando... pressione Ctrl+C\n");
        sleep(1);
    }

    return 0;
}

```

### Compilação e Execução:

```

gcc 04_signal_example.c -o signal_example
./04_signal_example

```

```

• (.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ python 04_signal_example.py
Executando... pressione Ctrl+C
Executando... pressione Ctrl+C
Executando... pressione Ctrl+C
Executando... pressione Ctrl+C
Executando... pressione Ctrl+C
Executando... pressione Ctrl+C
Executando... pressione Ctrl+C
• (.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ ./04_signal_example
Executando... pressione Ctrl+C
Executando... pressione Ctrl+C
Executando... pressione Ctrl+C
Executando... pressione Ctrl+C
^CRecebi sinal 2! Encerrando programa...

```

alt text

**Comportamento:** - Programa entra em loop infinito - Ao pressionar Ctrl+C, executa o handler customizado - Programa encerra de forma controlada

**Sinais Comuns:** - SIGINT (2): Interrupção do teclado (Ctrl+C) - SIGTERM (15): Terminação solicitada - SIGKILL (9): Terminação forçada (não pode ser capturado) - SIGSTOP (19): Pausa o processo (não pode ser capturado)

## PROGRAMA 05 - Preempção por Fatia de Tempo (Time Slice)

**Arquivo:** 05\_preempt\_timeslice.c

**Página de Referência:** Aula 2 - Laboratório, pág. 13

**Objetivo:** Demonstrar o conceito de preempção no escalonador CFS (Completely Fair Scheduler) do Linux.

**Conceitos Abordados:** - Escalonamento SCHED\_OTHER (CFS) - Preempção por tempo - Threads CPU-bound - Variáveis atômicas

**Código Principal:**

```

#include <stdio.h>
#include <pthread.h>
#include <stdatomic.h>
#include <time.h>

atomic_ulong c1 = 0, c2 = 0;

void* t1(void* _) {
    for (;;) {
        c1++; // Loop infinito incrementando contador
    }
}

void* t2(void* _) {
    for (;;) {
        c2++; // Loop infinito incrementando contador
    }
}

int main() {
    pthread_t a, b;

    pthread_create(&a, NULL, t1, NULL);
    pthread_create(&b, NULL, t2, NULL);

```

```

        for (int i = 0; i < 10; i++) {
            struct timespec ts = { .tv_sec = 0, .tv_nsec = 300*1000*1000
};
            nanosleep(&ts, NULL);

            unsigned long x = c1, y = c2;
            printf("t1=%lu t2=%lu\n", x, y);
        }

        return 0;
    }
}

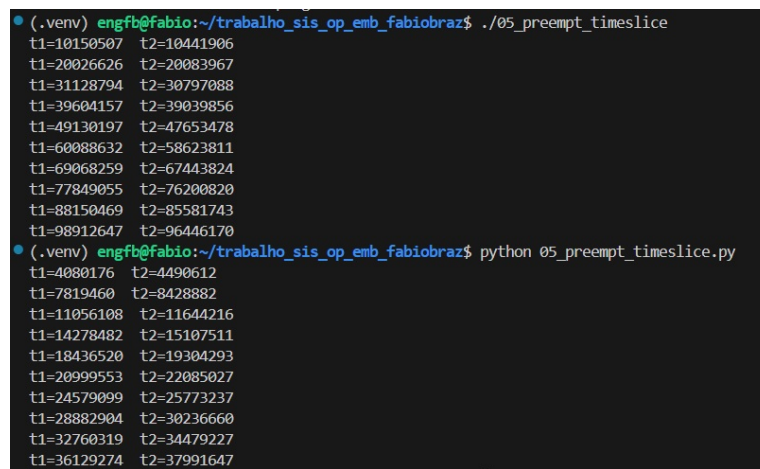
```

### Compilação e Execução:

```

gcc 05_preempt_timeslice.c -o preempt_timeslice -lpthread
./preempt_timeslice

```



```

• (.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ ./05_preempt_timeslice
t1=10150507 t2=10441906
t1=20026626 t2=20083967
t1=31128794 t2=30797088
t1=39604157 t2=39039856
t1=49130197 t2=47653478
t1=60088632 t2=58623811
t1=69068259 t2=67443824
t1=77849055 t2=76200820
t1=88150469 t2=85581743
t1=98912647 t2=96446170
• (.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ python 05_preempt_timeslice.py
t1=4080176 t2=4490612
t1=7819460 t2=8428882
t1=11056108 t2=11644216
t1=14278482 t2=15107511
t1=18436520 t2=19304293
t1=20999553 t2=22085027
t1=24579099 t2=25773237
t1=28882904 t2=30236660
t1=32760319 t2=34479227
t1=36129274 t2=37991647

```

alt text

### Saída Observada:

```

t1=10150507 t2=10441906
t1=20026626 t2=20083967
t1=31128794 t2=30797088
t1=39604157 t2=39039856
t1=49130197 t2=47653478

```

**Análise:** Mesmo sem sleep() ou yield(), ambas as threads progridem devido à preempção do kernel. O escalonador CFS distribui tempo de CPU de forma justa, mesmo para threads que tentam monopolizar o processador.

## PROGRAMA 06 - Tempo Real e Starvation (SCHED\_FIFO)

**Arquivo:** 06\_rt\_starvation.c

**Página de Referência:** Aula 2 - Laboratório, pág. 22

**Objetivo:** Demonstrar a diferença entre escalonamento de tempo real (SCHED\_FIFO) e escalonamento normal (SCHED\_OTHER).

**Conceitos Abordados:** - Política de escalonamento SCHED\_FIFO - Prioridades de tempo real (1-99) - Starvation de processos normais - RT throttling do kernel

### Código Principal:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <sched.h>
#include <stdlib.h>

int main(void) {
    pid_t pid = fork();

    if (pid == 0) {
        // FILHO: Política de tempo real
        struct sched_param sp = { .sched_priority = 80 };

        if (sched_setscheduler(0, SCHED_FIFO, &sp) != 0) {
            perror("sched_setscheduler falhou");
            exit(1);
        }

        // Busy-loop com mensagens periódicas
        volatile unsigned long long x = 0;
        const unsigned long long N = 80ULL * 1000ULL * 1000ULL;

        for (;;) {
            x++;
            if ((x % N) == 0) {
                printf("FILHO (SCHED_FIFO) rodando\n");
            }
        }
    } else {
        // PAI: Tarefa normal (SCHED_OTHER)
        for (int i = 0; i < 10; i++) {
            struct timespec ts = { .tv_sec = 0, .tv_nsec =
500*1000*1000 };
            nanosleep(&ts, NULL);
            printf("PAI (SCHED_OTHER) ainda vivo (i=%d)\n", i);
        }

        kill(pid, SIGKILL);
        waitpid(pid, NULL, 0);
    }

    return 0;
}
```

### Compilação e Execução:

```
gcc 06_rt_starvation.c -o rt_starvation
sudo taskset -c 0 ./rt_starvation # Força execução no mesmo core
```



```

• (.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ ./06_rt_starvation
sched_setscheduler falhou: Operation not permitted
PAI (SCHED_OTHER) ainda vivo (i=0)
PAI (SCHED_OTHER) ainda vivo (i=1)
PAI (SCHED_OTHER) ainda vivo (i=2)
PAI (SCHED_OTHER) ainda vivo (i=3)
PAI (SCHED_OTHER) ainda vivo (i=4)
PAI (SCHED_OTHER) ainda vivo (i=5)
PAI (SCHED_OTHER) ainda vivo (i=6)
PAI (SCHED_OTHER) ainda vivo (i=7)
PAI (SCHED_OTHER) ainda vivo (i=8)
PAI (SCHED_OTHER) ainda vivo (i=9)
• (.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ python 06_rt_starvation.py
Precisa rodar com sudo para SCHED_FIFO
PAI ainda vivo (i=0)
PAI ainda vivo (i=1)
PAI ainda vivo (i=2)
PAI ainda vivo (i=3)
PAI ainda vivo (i=4)
PAI ainda vivo (i=5)
PAI ainda vivo (i=6)
PAI ainda vivo (i=7)
PAI ainda vivo (i=8)
PAI ainda vivo (i=9)

```

alt text

**Observação Importante:** Requer privilégios de superusuário (sudo) para elevar processo a SCHED\_FIFO.

**Comportamento Observado:** - Processo filho (SCHED\_FIFO) domina a CPU - Processo pai (SCHED\_OTHER) raramente consegue executar - Demonstra starvation de processos não-RT

**Configuração de RT Throttling:**

```

# Desabilitar throttling (cuidado! pode travar o sistema)
sudo sysctl -w kernel.sched_rt_runtime_us=-1

# Restaurar throttling (padrão)
sudo sysctl -w kernel.sched_rt_runtime_us=950000

```

## PROGRAMA 07 - Supervisor de Processos

**Arquivo:** 07\_supervisor.c

**Página de Referência:** Aula 3, pág. 39

**Objetivo:** Criar um processo supervisor que monitora estados de processos filhos usando waitpid().

**Conceitos Abordados:** - System call waitpid() - Flags: WUNTRACED, WCONTINUED - Macros de análise de status: WIFEXITED, WIFSIGNALED, WIFSTOPPED, WIFCONTINUED - Monitoramento de eventos de processos

**Código Principal:**

```

#define _XOPEN_SOURCE 700

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

int main() {

```

```

pid_t pid1, pid2;

// Cria primeiro filho
pid1 = fork();
if (pid1 == 0) {
    while (1) {
        pause(); // aguarda sinais
    }
    exit(0);
}

// Cria segundo filho
pid2 = fork();
if (pid2 == 0) {
    while (1) {
        pause(); // aguarda sinais
    }
    exit(0);
}

printf("Pai supervisor PID=%d criou filhos: %d e %d\n",
       getpid(), pid1, pid2);

// Loop de supervisão
while (1) {
    int status;
    pid_t w = waitpid(-1, &status, WUNTRACED | WCONTINUED);

    if (w == -1) {
        perror("waitpid");
        exit(1);
    }

    if (WIFEXITED(status)) {
        printf("Filho %d terminou normalmente com código %d\n",
              w, WEXITSTATUS(status));
    }
    else if (WIFSIGNALED(status)) {
        printf("Filho %d morreu por sinal %d\n", w,
              WTERMSIG(status));
    }
    else if (WIFSTOPPED(status)) {
        printf("Filho %d foi PARADO (SIGSTOP), sinal=%d\n",
              w, WSTOPSIG(status));
    }
    else if (WIFCONTINUED(status)) {
        printf("Filho %d foi CONTINUADO (SIGCONT)\n", w);
    }
}

return 0;
}

```

### Compilação e Execução:

```

gcc 07_supervisor.c -o supervisor
./supervisor

```

```

Filho 30621 morreu por sinal 15
Filho 30620 morreu por sinal 15
Nenhuma processo filho restante, supervisor encerrando.
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ python 07_supervisor.py
Pai supervisor PID=32051 criou filhos: 32052 e 32053
Filho 32052 foi PARADO (SIGSTOP ou ctrl+z), sinal=19
Filho 32053 foi PARADO (SIGSTOP ou ctrl+z), sinal=19
Filho 32053 foi CONTINUADO (SIGCONT)
Filho 32052 foi CONTINUADO (SIGCONT)
Filho 32052 morreu por sinal 15
Filho 32053 morreu por sinal 15
waitpid: no child processes.
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -TERM 27788
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -STOP 30621
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -STOP 30620
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -CONT 30620
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -CONT 30621
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -TERM 30621
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -STOP 32052
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -STOP 32053
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -CONT 32053
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -TERM 32052
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -TERM 32053
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$

```

alt text

```

Nenhuma processo filho restante, supervisor encerrando.
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ python 07_supervisor.py
Filho iniciado, PID=30620
Filho iniciado, PID=30621
Pai supervisor PID=30615 monitorando...
Filho 30621 foi PARADO, sinal: 19
Filho 30620 foi PARADO, sinal: 19
Filho 30620 foi CONTINUADO.
Filho 30621 foi CONTINUADO.
Filho 30621 morreu por sinal 15
Filho 30620 morreu por sinal 15
waitpid: no child processes.
Nenhuma processo filho restante, supervisor encerrando.
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -TERM 27787
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -STOP 27787
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -STOP 27788
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -CONT 27788
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -CONT 27787
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -TERM 27787
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -TERM 27788
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -STOP 30621
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -STOP 30620
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -CONT 30620
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -CONT 30621
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -TERM 30621
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$ kill -TERM 30620
# (vono) engt@fabio:~/trabalho_sis_op_emb_fabio$

```

alt text

## Teste em Outro Terminal:

```

# Obter PIDs dos filhos
ps aux | grep supervisor

# Enviar sinais aos filhos
kill -STOP <PID_FILHO> # Pausa o processo
kill -CONT <PID_FILHO> # Retoma o processo
kill -TERM <PID_FILHO> # Encerra o processo

```

## Saída Esperada:

```

Filho 32052 foi PARADO (SIGSTOP), sinal=19
Filho 32052 foi CONTINUADO (SIGCONT)
Filho 32052 morreu por sinal 15

```

## PROGRAMA 08 - Estados de Processos (R, S, Z, T)

**Arquivo:** 08\_process\_states.c

**Página de Referência:** Aula 3, pág. 27

**Objetivo:** Demonstrar os diferentes estados que um processo pode assumir no Linux.

**Conceitos Abordados:** - Estados de processos: R (Running), S (Sleeping), Z (Zombie), T (Stopped) - Criação de processos zumbis - Envio de sinais SIGSTOP - Análise com ps e top

## Código Principal:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

int main() {
    pid_t pidR, pidS, pidZ, pidT;

    printf("PID do programa principal: %d\n", getpid());

    // 1. Processo em RUNNING (R)
    pidR = fork();
    if (pidR == 0) {
        while (1) {
            // Loop infinito (sempre R ou pronto)

```

```

    }
    exit(0);
}

// 2. Processo em SLEEPING (S)
pidS = fork();
if (pidS == 0) {
    while (1) {
        sleep(10); // estado "S" (esperando)
    }
    exit(0);
}

// 3. Processo em ZOMBIE (Z)
pidZ = fork();
if (pidZ == 0) {
    printf("Filho Z (vai virar zumbi) PID=%d\n", getpid());
    exit(0); // termina imediatamente
} else {
    sleep(1); // pai NÃO chama wait() → filho fica em Z
}

// 4. Processo em STOPPED (T)
pidT = fork();
if (pidT == 0) {
    while (1) {
        pause(); // espera por sinais
    }
    exit(0);
} else {
    sleep(1);
    kill(pidT, SIGSTOP); // envia SIGSTOP → processo fica em T
}

printf("Processos criados: R=%d S=%d Z=%d T=%d\n",
       pidR, pidS, pidZ, pidT);

printf("Use 'ps -o pid,stat,comm -p %d,%d,%d,%d' para ver os
estados.\n",
       pidR, pidS, pidZ, pidT);

while (1) {
    sleep(30);
}

return 0;
}

```

## Compilação e Execução:

```
gcc 08_process_states.c -o process_states
./process_states
```

```

(.venv) mgf@fabio:~/trabalho_sis_op_emb_fabio$ ./08_process_states
PID do programa principal: 3660
Filho Z (vai virar zumbi) PID=3663
Processos criados: R=3661 S=3662 Z=3663 T=3674
Use 'ps -o pid,stat,comm -p 3661,3662,3663,3674' para ver os estados.

```

PID	USER	PR	NI	VT	RES	SHR S	%CPU	MEM	TIME+	COMMAND
3660	mgf	20	0	R	256K	127K	120%	0	0:00.00	./process_states

alt text

```

(.venv) mgf@fabio:~/trabalho_sis_op_emb_fabio$ python 08_process_states.py
PID do programa principal: 37274
Filho Z (vai virar zumbi) PID=37277
Processos criados: R=37275 S=37276 Z=37277 T=37297
Use 'ps -o pid,stat,comm -p 37275,37276,37277,37297' para ver os estados.

```

PID	USER	PR	NI	VT	RES	SHR S	%CPU	MEM	TIME+	COMMAND
37274	mgf	20	0	R	15372	10112	6400%	0	0:00.02	python

alt text

### Análise com ps:

```
ps -o pid,stat,comm -p 36661,36662,36663,36674
```

### Saída Esperada:

```
PID STAT COMM
36661 R   process_states # Running (usando CPU)
36662 S   process_states # Sleeping (aguardando sleep)
36663 Z   process_states # Zombie (terminou, não foi coletado)
36674 T   process_states # Stopped (recebeu SIGSTOP)
```

**Estados Detalhados:** - **R (Running):** Processo executando ou pronto para executar - **S (Sleeping):** Aguardando evento (I/O, timer, sinal) - **Z (Zombie):** Processo terminou mas ainda não foi coletado pelo pai - **T (Stopped):** Processo pausado por sinal SIGSTOP ou Ctrl+Z

---

## PROGRAMA 09 - Fork e Execve (Multitarefa)

**Arquivo:** 09\_fork\_execve.c

**Página de Referência:** Aula 4, pág. 11

**Objetivo:** Demonstrar a substituição de código de processo usando execve().

**Conceitos Abordados:** - System call execve() - Substituição de imagem do processo - Espera de processos filhos com wait() - Execução de programas externos

### Código Principal:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *envp[])
{
    int pid;

    pid = fork(); // cria processo filho

    if (pid < 0) {
        perror("Erro: ");
        exit(-1);
    }
    else if (pid > 0) {
        // PROCESSO PAI
        wait(0); // espera o filho terminar
    }
    else {
        // PROCESSO FILHO
        // substitui o código por outro programa
        execve("/bin/date", argv, envp);

        // se execve retornar, houve erro
        perror("Erro: ");
    }
}
```

```

    }

    printf("Tchau !\n");
    exit(0);
}

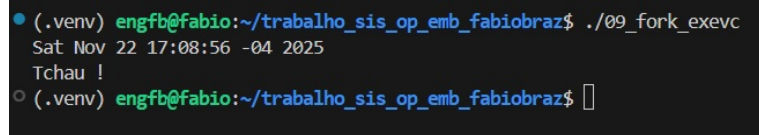
```

### Compilação e Execução:

```

gcc 09_fork_execve.c -o fork_execve
./fork_execve

```



```

• (.env) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ ./09_fork_exevc
Sat Nov 22 17:08:56 -04 2025
Tchau !
○ (.env) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ 

```

alt text

### Saída Esperada:

```

Sat Nov 22 17:08:56 -04 2025
Tchau !

```

**Análise:** 1. Pai cria filho com `fork()` 2. Filho substitui seu código com `execve("/bin/date")` 3. Comando `date` executa e termina 4. Pai imprime "Tchau !" e encerra

**Observação:** Se `execve()` falhar (arquivo não existe, sem permissão), retorna -1 e o código original continua executando.

---

## PROGRAMA 10 - Threads POSIX

**Arquivo:** 10\_ex\_threads.c

**Página de Referência:** Aula 4, pág. 30

**Objetivo:** Criar múltiplas threads POSIX e demonstrar execução concorrente.

**Conceitos Abordados:** - Biblioteca `pthread` - Criação de threads com `pthread_create()` - Sincronização com `pthread_join()` - Passagem de argumentos para threads

### Código Principal:

```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define NUM_THREADS 5

void* print_hello(void* arg)
{
    long id = *(long*)arg;

    printf("%ld: Hello World!\n", id);
    sleep(5);
    printf("%ld: Bye bye World!\n", id);

    pthread_exit(NULL);
}

```

```

}

int main(void)
{
    pthread_t th[NUM_THREADS];
    long* ids = malloc(sizeof(long) * NUM_THREADS);

    if (!ids) {
        perror("malloc");
        exit(1);
    }

    // Criar threads
    for (long i = 0; i < NUM_THREADS; i++) {
        ids[i] = i;
        int rc = pthread_create(&th[i], NULL, print_hello, &ids[i]);
        if (rc != 0) {
            perror("pthread_create");
            free(ids);
            exit(1);
        }
    }

    // Esperar threads terminarem
    for (int i = 0; i < NUM_THREADS; i++) {
        int rc = pthread_join(th[i], NULL);
        if (rc != 0) {
            perror("pthread_join");
            free(ids);
            exit(1);
        }
    }

    free(ids);
    printf("Todas as threads finalizaram.\n");
    return 0;
}

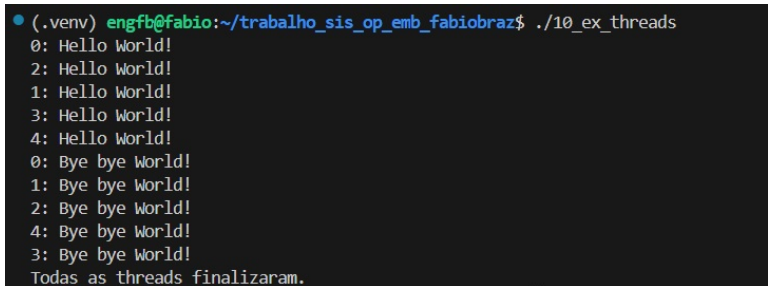
```

### Compilação e Execução:

```

gcc 10_ex_threads.c -o ex_threads -lpthread
./ex_threads

```



```

• (.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ ./10_ex_threads
0: Hello World!
2: Hello World!
1: Hello World!
3: Hello World!
4: Hello World!
0: Bye bye World!
1: Bye bye World!
2: Bye bye World!
4: Bye bye World!
3: Bye bye World!
Todas as threads finalizaram.

```

alt text

### Saída Esperada:

```

0: Hello World!
2: Hello World!
1: Hello World!
3: Hello World!

```

```
4: Hello World!
0: Bye bye World!
1: Bye bye World!
2: Bye bye World!
4: Bye bye World!
3: Bye bye World!
Todas as threads finalizaram.
```

**Análise:** As threads executam concorrentemente, resultando em ordem de impressão não determinística. O uso de `pthread_join()` garante que o processo principal aguarde todas as threads terminarem antes de encerrar.

---

## PROGRAMA 11 - Função `system()`

**Arquivo:** `11_teste_systems.c`

**Página de Referência:** Aula 4, pág. 48

**Objetivo:** Demonstrar o uso da função `system()` para executar comandos shell.

**Conceitos Abordados:** - Função `system()` da `stdlib` - Execução de comandos shell - Criação implícita de processos filhos - Espera automática do término

### Código Principal:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    printf("PID do Pai: %d\n", getpid());

    // Chama outro programa usando system()
    // Cria um processo FILHO para rodar "./teste"
    // O Pai espera o Filho terminar antes de continuar
    system("./teste");

    printf("\n[PAI] Programa continuou apos a funcao system()\n");

    return 0;
}
```

### Compilação e Execução:

```
gcc 11_teste_systems.c -o teste_system
./teste_system
```

### Saída Observada:

```
PID do Pai: 38987
PID do processo: 38989
Testando fork()...
Sou o processo PAI! PID: 38989, filho: 38990
Sou o processo FILHO! PID: 38990

[PAI] Programa continuou apos a funcao system()
```



```

• (.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ ./11_teste_systems
PID do Pai: 38987
PID do processo: 38989
Testando fork()...
Sou o processo PAI! PID: 38989, filho: 38990
Sou o processo FILHO! PID: 38990

[PAI] Programa continuou apos a funcao system()

```

alt text

**Análise:** - system() internamente usa fork() + execve() + wait() - Cria shell intermediário que executa o comando - Menos eficiente que fork()+execve() direto - Útil para comandos simples e scripts

---

## PROGRAMA 12 - FreeRTOS: Tasks Básicas

**Arquivo:** main.c (Exemplo 01 - 3 Tasks)

**Página de Referência:** Aula 6, pág. 26

**Objetivo:** Criar três tarefas concorrentes em FreeRTOS com diferentes períodos de execução.

**Conceitos Abordados:** - Tasks do FreeRTOS - Prioridades de tarefas - Delay não-bloqueante (vTaskDelay) - Escalonador cooperativo/preemptivo

### Código Principal:

```

#include <stdio.h>
#include <unistd.h>
#include "FreeRTOS.h"
#include "task.h"

void vTaskA(void *pvParameters) {
    while (1) {
        printf("Task A executando...\n");
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

void vTaskB(void *pvParameters) {
    while (1) {
        printf("Task B executando...\n");
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

void vTaskC(void *pvParameters) {
    while (1) {
        printf("Task C executando...\n");
        vTaskDelay(pdMS_TO_TICKS(1500));
    }
}

int main(void) {
    printf("=== FreeRTOS: Exemplo com 3 Tasks ===\n");

    xTaskCreate(vTaskA, "TaskA", 1024, NULL, 2, NULL);
    xTaskCreate(vTaskB, "TaskB", 1024, NULL, 1, NULL);
    xTaskCreate(vTaskC, "TaskC", 1024, NULL, 1, NULL);
}

```

```

vTaskStartScheduler(); // Inicia o escalonador

for(;;); // Nunca deve chegar aqui
}

```

### Compilação:

```

cd ~/trabalho_sis_op_emb_fabiobraz/LabFreeRTOS/Exemplo01_3Tasks
make

```

### Execução:

```

./build/meu_exemplo1_3tasks

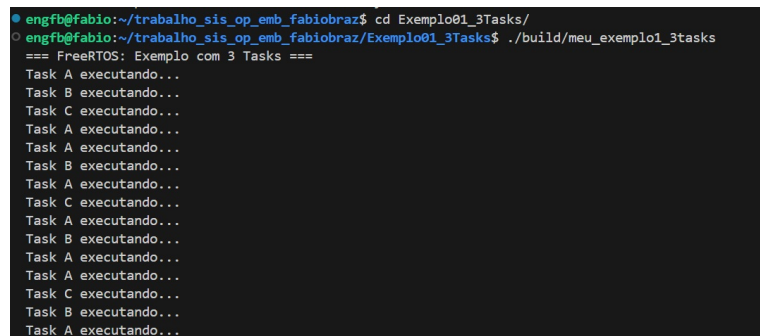
```

### Estrutura Necessária:

```

trabalho_sis_op_emb_fabiobraz/
├── FreeRTOS/                # Clone do repositório oficial
├── LabFreeRTOS/
│   └── Exemplo01_3Tasks/
│       ├── main.c
│       ├── FreeRTOSConfig.h
│       └── Makefile

```



```

engfb@fabio:~/trabalho_sis_op_emb_fabiobraz$ cd Exemplo01_3Tasks/
engfb@fabio:~/trabalho_sis_op_emb_fabiobraz/Exemplo01_3Tasks$ ./build/meu_exemplo1_3tasks
=== FreeRTOS: Exemplo com 3 Tasks ===
Task A executando...
Task B executando...
Task C executando...
Task A executando...
Task A executando...
Task B executando...
Task A executando...
Task C executando...
Task A executando...
Task B executando...
Task A executando...
Task A executando...
Task C executando...
Task B executando...
Task A executando...

```

alt text

**Observações:** - TaskA tem prioridade 2 (mais alta) - TaskB e TaskC têm prioridade 1 - Tarefas com mesma prioridade executam em round-robin - vTaskDelay() libera CPU para outras tarefas

## PROGRAMA 13 - FreeRTOS: Semáforo Binário

**Arquivo:** main.c (Exemplo 02 - Semáforo)

**Página de Referência:** Aula 6, pág. 36

**Objetivo:** Sincronizar duas tarefas usando semáforo binário para comunicação produtor-consumidor.

**Conceitos Abordados:** - Semáforos binários no FreeRTOS - xSemaphoreCreateBinary() - xSemaphoreGive() e xSemaphoreTake() - Sincronização entre tarefas

### Código Principal:

```

#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"

```

```

SemaphoreHandle_t xSemaforo = NULL;

void vTaskProdutora(void *pvParameters)
{
    for (;;)
    {
        printf("[Produtora] Liberando semáforo...\n");
        vTaskDelay(pdMS_TO_TICKS(1000));
        xSemaphoreGive(xSemaforo); // Sinaliza
    }
}

void vTaskConsumidora(void *pvParameters)
{
    for (;;)
    {
        if (xSemaphoreTake(xSemaforo, portMAX_DELAY) == pdTRUE)
        {
            printf(" [Consumidora] Recebeu sinal!\n");
            vTaskDelay(pdMS_TO_TICKS(500));
        }
    }
}

int main(void)
{
    printf("=== FreeRTOS: Semáforo Binário ===\n");

    xSemaforo = xSemaphoreCreateBinary();

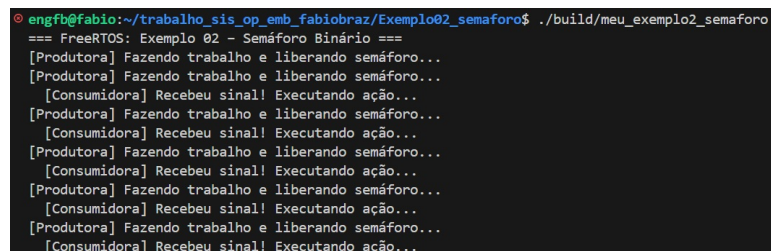
    if (xSemaforo == NULL) {
        printf("Falha ao criar semáforo!\n");
        return -1;
    }

    xTaskCreate(vTaskProdutora, "Produtora", 1024, NULL, 2, NULL);
    xTaskCreate(vTaskConsumidora, "Consumidora", 1024, NULL, 1,
NULL);

    vTaskStartScheduler();

    for (;;);
}

```



```

engfb@fabio:~/trabalho_sis_op_emb_fabiobraz/Exemplo02_semaforo$ ./build/meu_exemplo2_semaforo
=== FreeRTOS: Exemplo 02 - Semáforo Binário ===
[Produtora] Fazendo trabalho e liberando semáforo...
[Produtora] Fazendo trabalho e liberando semáforo...
[Consumidora] Recebeu sinal! Executando ação...
[Produtora] Fazendo trabalho e liberando semáforo...
[Consumidora] Recebeu sinal! Executando ação...
[Produtora] Fazendo trabalho e liberando semáforo...
[Consumidora] Recebeu sinal! Executando ação...
[Produtora] Fazendo trabalho e liberando semáforo...
[Consumidora] Recebeu sinal! Executando ação...
[Produtora] Fazendo trabalho e liberando semáforo...
[Consumidora] Recebeu sinal! Executando ação...

```

alt text

**Comportamento:** - Produtora sinaliza semáforo a cada 1 segundo - Consumidora aguarda sinal (bloqueia se não houver) - Sincronização temporal entre tarefas

---

## PROGRAMA 14 - FreeRTOS: Filas (Queues)

**Arquivo:** main.c (Exemplo 03 - Queue)

**Página de Referência:** Aula 6, pág. 46

**Objetivo:** Implementar comunicação entre tarefas através de filas (queues).

**Conceitos Abordados:** - Filas FIFO do FreeRTOS - xQueueCreate(), xQueueSend(), xQueueReceive() - Comunicação inter-tarefas com dados

### Código Principal:

```
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

QueueHandle_t xFila = NULL;

void vTaskProdutora(void *pvParameters)
{
    int contador = 0;

    for (;;)
    {
        contador++;
        if (xQueueSend(xFila, &contador, portMAX_DELAY) == pdPASS)
        {
            printf("[Produtora] Enviou valor %d\n", contador);
        }
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

void vTaskConsumidora(void *pvParameters)
{
    int recebido;

    for (;;)
    {
        if (xQueueReceive(xFila, &recebido, portMAX_DELAY) ==
pdTRUE)
        {
            printf(" [Consumidora] Recebeu valor %d\n", recebido);
        }
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

int main(void)
{
    printf("=== FreeRTOS: Comunicação com Fila ===\n");

    xFila = xQueueCreate(5, sizeof(int)); // Fila com 5 elementos

    if (xFila == NULL) {
        printf("Falha ao criar fila!\n");
        return -1;
    }
}
```

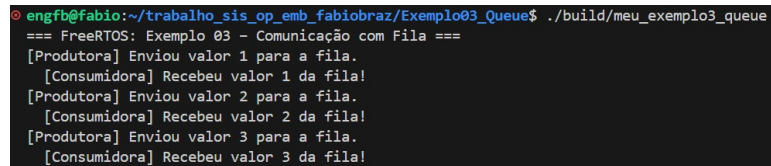
```

xTaskCreate(vTaskProdutora, "Produtora", 1024, NULL, 2, NULL);
xTaskCreate(vTaskConsumidora, "Consumidora", 1024, NULL, 1,
NULL);

vTaskStartScheduler();

for (;;)
}

```



```

engfb@fabio:~/trabalho_sis_op_emb_fabiobraz/Exemplo03_Queue$ ./build/meu_exemplo3_queue
=== FreeRTOS: Exemplo 03 - Comunicação com Fila ===
[Produtora] Enviou valor 1 para a fila.
[Consumidora] Recebeu valor 1 da fila!
[Produtora] Enviou valor 2 para a fila.
[Consumidora] Recebeu valor 2 da fila!
[Produtora] Enviou valor 3 para a fila.
[Consumidora] Recebeu valor 3 da fila!

```

alt text

**Características:** - Fila armazena até 5 inteiros - Comunicação bidirecional possível - Operações thread-safe (seguras para concorrência)

---

## PROGRAMA 15 - FreeRTOS: Mutex

**Arquivo:** main.c (Exemplo 04 - Mutex)

**Página de Referência:** Aula 6, pág. 51

**Objetivo:** Proteger recurso compartilhado usando mutex (exclusão mútua).

**Conceitos Abordados:** - Mutex do FreeRTOS - Inversão de prioridade - Herança de prioridade - Seção crítica

**Código Conceitual:**

```

SemaphoreHandle_t xMutex;

void vTaskCritical(void *pvParameters)
{
    for (;;)
    {
        if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE)
        {
            // Seção crítica: acesso exclusivo ao recurso
            printf("Task %s na seção crítica\n",
(char*)pvParameters);
            vTaskDelay(pdMS_TO_TICKS(100));

            xSemaphoreGive(xMutex);
        }
    }
}

int main(void)
{
    xMutex = xSemaphoreCreateMutex();

    xTaskCreate(vTaskCritical, "Task1", 1024, "A", 2, NULL);
    xTaskCreate(vTaskCritical, "Task2", 1024, "B", 1, NULL);

    vTaskStartScheduler();
}

```

```

    for (;;)
}

```

```

engfb@fabio:~/trabalho_sis_op_emb_fabiobraz/Exemplo04_Mutex$ ./build/meu_exemplo4_mutex
=== FreeRTOS: Exemplo 04 - Mutex ===
[TaskA] entrou na seção crítica
[TaskA] imprimindo linha 1
[TaskA] imprimindo linha 2
[TaskA] imprimindo linha 3
[TaskA] saindo da seção crítica

[TaskB] entrou na seção crítica
[TaskB] imprimindo linha 1
[TaskB] imprimindo linha 2
[TaskB] imprimindo linha 3
[TaskB] saindo da seção crítica

[TaskC] entrou na seção crítica
[TaskC] imprimindo linha 1
[TaskC] imprimindo linha 2
[TaskC] imprimindo linha 3
[TaskC] saindo da seção crítica

```

alt text

**Diferença entre Mutex e Semáforo Binário:** - **Mutex:** Só quem pegou pode liberar (ownership) - **Mutex:** Implementa herança de prioridade - **Semáforo:** Qualquer task pode dar give

---

## PROGRAMA 16 - FreeRTOS: Software Timers

**Arquivo:** main.c (Exemplo 05 - Timer)

**Página de Referência:** Aula 6, pág. 55

**Objetivo:** Criar timers periódicos e de disparo único no FreeRTOS.

**Conceitos Abordados:** - Software Timers - Timers periódicos (pdTRUE)  
- Timers one-shot (pdFALSE) - Callbacks de timers

**Código Principal:**

```

#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "timers.h"

TimerHandle_t xTimerBlink;
TimerHandle_t xTimerOneShot;

void vTimerBlinkCallback(TimerHandle_t xTimer)
{
    static int estado = 0;
    estado = !estado;
    printf("[TimerBlink] LED virtual: %s\n", estado ? "ON" : "OFF");
}

void vTimerOneShotCallback(TimerHandle_t xTimer)
{
    printf("[TimerOneShot] Ação executada uma única vez!\n");
}

int main(void)
{
    printf("=== FreeRTOS: Software Timer ===\n");

    // Timer periódico: 1 segundo

```

```
engf@fabiobraz:~/trabalho/is_op_emb_fabiobraz/Exemplo05_Timer$ ./build/meu_exemplo5_timer
=== FreeRTOS: Exemplo 05 - Software Timer ===
[DaemonTask] Serviço de timers iniciado.
[TimerBlink] LED virtual: ON
[TimerBlink] LED virtual: OFF
[TimerBlink] LED virtual: ON
[TimerBlink] LED virtual: OFF
[TimerOneShot] Ação executada uma única vez após 5 s!
[TimerBlink] LED virtual: ON
[TimerBlink] LED virtual: OFF
[TimerBlink] LED virtual: ON
[TimerBlink] LED virtual: OFF
[TimerBlink] LED virtual: ON
[TimerBlink] LED virtual: OFF
[TimerBlink] LED virtual: ON
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
```

```

#define BUFFER 256

int main(void) {
    int fd[2];
    pid_t pid;

    if (pipe(fd) < 0) {
        perror("Erro ao criar pipe");
        return -1;
    }

    if ((pid = fork()) < 0) {
        perror("Erro ao criar processo");
        exit(1);
    }

    if (pid > 0) {
        // PROCESSO PAI
        close(fd[0]); // Fecha leitura

        char str[BUFFER] = "Aprendi a usar Pipes em C!\n";
        printf("PAI enviando: '%s'\n", str);
        write(fd[1], str, sizeof(str));

        exit(0);
    }
    else {
        // PROCESSO FILHO
        char str_recebida[BUFFER];

        close(fd[1]); // Fecha escrita
        read(fd[0], str_recebida, sizeof(str_recebida));

        printf("FILHO recebeu: '%s'\n", str_recebida);

        exit(0);
    }

    return 0;
}

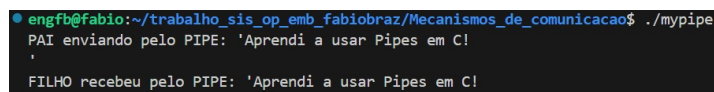
```

### Compilação e Execução:

```

gcc mypipe.c -o mypipe
./mypipe

```



```

engfb@fabio:~/trabalho_sis_op_emb_fabiobraz/Mecanismos_de_comunicacao$ ./mypipe
PAI enviando pelo PIPE: 'Aprendi a usar Pipes em C!'
FILHO recebeu pelo PIPE: 'Aprendi a usar Pipes em C!'

```

alt text

### Saída:

```

PAI enviando: 'Aprendi a usar Pipes em C!'
FILHO recebeu: 'Aprendi a usar Pipes em C!'

```

---

## PROGRAMA 18 - Filas de Mensagens POSIX



**Arquivos:** mq-send.c e mq-recv.c

**Página de Referência:** Aula 7, pág. 35

**Objetivo:** Implementar comunicação entre processos independentes usando filas de mensagens.

**Conceitos Abordados:** - Message Queues POSIX (mqueue.h) - mq\_open(), mq\_send(), mq\_receive() - Comunicação assíncrona entre processos - Persistência de filas

**Código - Consumidor (mq-recv.c):**

```
#define _POSIX_C_SOURCE 200809L

#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <fcntl.h>

#define QUEUE "/my_queue"

int main (int argc, char *argv[])
{
    mqd_t queue;
    struct mq_attr attr;
    int msg;

    attr.mq_maxmsg = 10;
    attr.mq_msgsize = sizeof(msg);
    attr.mq_flags = 0;

    queue = mq_open(QUEUE, O_RDWR | O_CREAT, 0666, &attr);
    if (queue == (mqd_t)-1) {
        perror("mq_open");
        exit(1);
    }

    printf("=== Consumidor: aguardando mensagens ===\n");

    for (;;) {
        if (mq_receive(queue, (void *)&msg, sizeof(msg), 0) < 0) {
            perror("mq_receive");
            exit(1);
        }
        printf("Consumidor recebeu valor: %d\n", msg);
    }

    return 0;
}
```

**Código - Produtor (mq-send.c):**

```
#define _POSIX_C_SOURCE 200809L

#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <unistd.h>
#include <fcntl.h>
```

```

#define QUEUE "/my_queue"

int main (int argc, char *argv[])
{
    mqd_t queue;
    int msg;

    queue = mq_open(QUEUE, 0_RDWR);
    if (queue == (mqd_t)-1) {
        perror("mq_open - Execute o consumidor primeiro!");
        exit(1);
    }

    printf("=== Produtor: enviando mensagens ===\n");

    for (;;) {
        msg = random() % 100;

        if (mq_send(queue, (void *)&msg, sizeof(msg), 0) < 0) {
            perror("mq_send");
            exit(1);
        }

        printf("Produtor enviou valor: %d\n", msg);
        sleep(1);
    }

    return 0;
}

```

### Compilação:

```

gcc mq-recv.c -o mq-recv -lrt
gcc mq-send.c -o mq-send -lrt

```

### Execução (dois terminais):

```

# Terminal 1
./mq-recv

```

```

# Terminal 2
./mq-send

```

```

engf@fabio:~/trabalho_sis_op_emb_fabiobraz/Exemplo4_Mutex$ cd ..
engf@fabio:~/trabalho_sis_op_emb_fabiobraz$ cde Mecanismos_de_comunicac
cd
Command 'cde' not found, but can be installed with:
sudo apt install cde ...
/mq-send
=== Produtor: enviando mensagens para a fila /my_queue ===
Produtor enviou valor: 83
Produtor enviou valor: 86
Produtor enviou valor: 77
Produtor enviou valor: 15
Produtor enviou valor: 93
Produtor enviou valor: 35
Produtor enviou valor: 86
Produtor enviou valor: 92
Produtor enviou valor: 49
Produtor enviou valor: 21
Produtor enviou valor: 62

source /home/engf/trabalho_sis_op_emb_fabiobraz/.venv/bin/activate
engf@fabio:~/trabalho_sis_op_emb_fabiobraz/Mecanismos_de_comunicac$ sl
(venv) engf@fabio:~/trabalho_sis_op_emb_fabiobraz/Mecanismos_de_comun
cacao$ ./mq-recv
=== Consumidor: aguardando mensagens na fila /my_queue ===
Consumidor recebeu valor: 83
Consumidor recebeu valor: 86
Consumidor recebeu valor: 77
Consumidor recebeu valor: 15
Consumidor recebeu valor: 93
Consumidor recebeu valor: 35
Consumidor recebeu valor: 86
Consumidor recebeu valor: 92
Consumidor recebeu valor: 49
Consumidor recebeu valor: 21
Consumidor recebeu valor: 62
Consumidor recebeu valor: 27

```

alt text

**Vantagens sobre Pipes:** - Processos não precisam ter relação de parentesco - Mensagens têm prioridades - Não-bloqueante (opcional) - Persiste até `mq_unlink()`

## PROGRAMA 19 - Memória Compartilhada POSIX

**Arquivos:** `shmem_write.c` e `shmem_read.c`

**Página de Referência:** Aula 7, pág. 45

**Objetivo:** Compartilhar região de memória entre processos independentes.

**Conceitos Abordados:** - Shared Memory POSIX - shm\_open(), mmap(), ftruncate() - Comunicação de alta velocidade - Sincronização necessária (não fornecida)

**Código - Escritor (shmem\_write.c):**

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>

int main(void) {
    int fd, value, *ptr;

    // Criar/abrir memória compartilhada
    fd = shm_open("/sharedmem", O_RDWR | O_CREAT, S_IRUSR |
S_IWUSR);
    if (fd == -1) {
        perror("shm_open");
        exit(1);
    }

    // Definir tamanho
    if (ftruncate(fd, sizeof(int)) == -1) {
        perror("ftruncate");
        exit(1);
    }

    // Mapear memória
    ptr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
        MAP_SHARED, fd, 0);
    if (ptr == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    printf("=== ESCRITOR ===\n");
    printf("PID: %d\n\n", getpid());

    // Loop: escrever valores
    for (;;) {
        value = random() % 1000;
        *ptr = value; // ESCRIVE na memória compartilhada

        printf("[ESCRITOR PID %d] ESCRIVEU: %d\n", getpid(), value);
        sleep(2);
    }

    return 0;
}
```

**Código - Leitor (shmem\_read.c):**

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
```

```

#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>

int main(void) {
    int fd, value, *ptr;

    // Abrir memória compartilhada existente
    fd = shm_open("/sharedmem", O_RDWR, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        perror("shm_open - Execute o ESCRITOR primeiro!");
        exit(1);
    }

    // Mapear memória
    ptr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
               MAP_SHARED, fd, 0);
    if (ptr == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    printf("=== LEITOR ===\n");
    printf("PID: %d\n\n", getpid());

    // Loop: ler valores
    for (;;) {
        value = *ptr; // Lê da memória compartilhada

        printf("[LEITOR PID %d] LEU: %d\n", getpid(), value);
        sleep(1);
    }

    return 0;
}

```

### Compilação:

```

gcc shmem_write.c -o shmem_write -lrt
gcc shmem_read.c -o shmem_read -lrt

```

### Execução (dois terminais):

```

# Terminal 1
./shmem_write

# Terminal 2
./shmem_read

```

```

[PID 60256] READ value = 0
[PID 60256] READ value = 0
(.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz/Mecanismos_de_comunicacao$ ./shmem_write
=== ESCRITOR: Escrevendo na memória compartilhada ===
PID: 61420
[ESCRITOR PID 61420] ESCRREVEU: 383
[ESCRITOR PID 61420] ESCRREVEU: 886
[ESCRITOR PID 61420] ESCRREVEU: 777
[ESCRITOR PID 61420] ESCRREVEU: 915
[ESCRITOR PID 61420] ESCRREVEU: 793
[ESCRITOR PID 61420] ESCRREVEU: 335
[ESCRITOR PID 61420] ESCRREVEU: 386
[ESCRITOR PID 61420] ESCRREVEU: 492
[ESCRITOR PID 61420] ESCRREVEU: 649
[ESCRITOR PID 61420] ESCRREVEU: 421

source /home/engfb/trabalho_sis_op_emb_fabiobraz/.venv/bin/activate
engfb@fabio:~/trabalho_sis_op_emb_fabiobraz/Mecanismos_de_comunicacao$ source /home/engfb/trabalho_sis_op_emb_fabiobraz/.venv/bin/activate
(.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz/Mecanismos_de_comunicacao$ ./shmem_read
=== LEITOR: Lendo da memória compartilhada ===
PID: 61627
[LEITOR PID 61627] LEU: 59
[LEITOR PID 61627] LEU: 59
[LEITOR PID 61627] LEU: 763
[LEITOR PID 61627] LEU: 763
[LEITOR PID 61627] LEU: 926
[LEITOR PID 61627] LEU: 926
[LEITOR PID 61627] LEU: 540
[LEITOR PID 61627] LEU: 540

```

alt text

### Problema Encontrado e Solução:

No código original (shmem.c), as seções de escrita estavam comentadas, resultando em leitura constante de zero:

```
// Código original (com bug):
for (;;) {
    // value = random() % 1000;    // Comentado
    // *ptr = value;                // Comentado
    value = *ptr;
    printf("[PID %d] READ value = %d\n", getpid(), value);
    sleep(1);
}
```

**Solução:** Separar em dois programas distintos (escritor e leitor) conforme mostrado acima.

**Observações:** - Memória compartilhada é o IPC mais rápido - **Não fornece sincronização** (usar semáforos se necessário) - Persistente até reboot ou shm\_unlink()

---

## PROGRAMA 20 - Semáforos e Mutex POSIX

**Referência:** Aula 8, pág. 48 e 54

**Objetivo:** Demonstrar coordenação entre tarefas usando semáforos e mutex POSIX.

**Conceitos Abordados:** - Semáforos nomeados e não-nomeados - Mutex POSIX - Diferenças entre semáforos e mutex - Proteção de seções críticas

**Exemplo - Semáforo:**

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t sem;
int contador_global = 0;

void* incrementar(void* arg) {
    for (int i = 0; i < 100000; i++) {
        sem_wait(&sem);
        contador_global++;
        sem_post(&sem);
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    sem_init(&sem, 0, 1);

    pthread_create(&t1, NULL, incrementar, NULL);
    pthread_create(&t2, NULL, incrementar, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Contador final: %d (esperado: 200000)\n",
```

```

contador_global);

        sem_destroy(&sem);
        return 0;
    }

```

### Exemplo - Mutex:

```

#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int contador_global = 0;

void* incrementar(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&mutex);
        contador_global++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, incrementar, NULL);
    pthread_create(&t2, NULL, incrementar, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Contador final: %d (esperado: 200000)\n",
contador_global);

    pthread_mutex_destroy(&mutex);
    return 0;
}

```

### Compilação:

```

gcc posix_sem.c -o posix_sem -lpthread
gcc mutex_demo.c -o mutex_demo -lpthread

```

```

(.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz/Coordenacao_entre_Tarefas$ ./posix_sem_wait_post

=== POSIX Semáforo (sem_t) ===
Capacidade inicial = 2 | Threads = 5

[W0] quer entrar (sem_wait)
[W2] quer entrar (sem_wait)
[W1] quer entrar (sem_wait)
[W3] quer entrar (sem_wait)
[W4] quer entrar (sem_wait)
[W0] ENTROU na seção crítica
[W2] ENTROU na seção crítica
[W2] saindo (sem_post)
[W1] ENTROU na seção crítica
[W1] saindo (sem_post)

```

alt text

```

(.venv) engfb@fabio:~/trabalho_sis_op_emb_fabiobraz/Coordenacao_entre_Tarefas$ ./mutex_demo

=== DEMO MUTEX (pthread) ===
Modo: lock/unlock (bloqueante)
Threads: 2 | Iterações por thread: 500000

Contador FINAL = 1000000 | Esperado = 1000000
✅ Resultado CORRETO (exclusão mútua garantida).

```

alt text

### Diferenças:

Característica	Semáforo	Mutex
Propriedade	Não tem dono	Tem dono (quem travou)
Sinalização	Qualquer thread pode dar post	Só quem travou pode destravar
Contador	Pode ter valor > 1	Sempre binário (0 ou 1)
Uso típico	Sinalização entre threads	Proteção de seção crítica

## 4. DESAFIOS E SOLUÇÕES

### 4.1 Configuração do Ambiente WSL

**Desafio:** Configurar ambiente de desenvolvimento C no WSL2 integrado ao VS Code.

**Solução:**

```
# Instalação de ferramentas essenciais
sudo apt update
sudo apt install build-essential gdb git

# Extensões VS Code necessárias:
# - Remote - WSL (ms-vscode-remote.remote-wsl)
# - C/C++ (ms-vscode.cpptools)
```

### 4.2 Compilação do FreeRTOS

**Desafio:** Estrutura de diretórios e dependências do FreeRTOS.

**Solução:**

```
# Estrutura correta:
trabalho_sis_op_emb_fabiobraz/
├── FreeRTOS/                # Clone do GitHub
│   ├── FreeRTOS/
│   │   └── Source/
│   └── LabFreeRTOS/
│       └── Exemplo01_3Tasks/
│           ├── main.c
│           ├── FreeRTOSConfig.h
│           └── Makefile

# Makefile deve apontar para ../FreeRTOS/FreeRTOS
```

### 4.3 Permissões para SCHED\_FIFO

**Desafio:** Erro “Operation not permitted” ao tentar elevar prioridade para tempo real.

**Solução:**

```
# Executar com sudo:
sudo ./rt_starvation

# Ou configurar capabilities:
sudo setcap cap_sys_nice=ep ./rt_starvation
```

#### 4.4 Memória Compartilhada (shmem.c)

**Desafio:** Programa original sempre lia valor zero.

**Causa Raiz:** Seções de escrita estavam comentadas no código original.

**Solução:** Criar dois programas separados: - shmem\_write.c: Escreve valores aleatórios - shmem\_read.c: Lê valores escritos

#### 4.5 Filas de Mensagens

**Desafio:** “No such file or directory” ao executar mq-send antes de mq-recv.

**Solução:** Sempre executar o consumidor (mq-recv) primeiro, pois ele cria a fila com 0\_CREAT.

#### 4.6 Linking de Bibliotecas

**Desafio:** Undefined references em tempo de link.

**Solução:**

```
# Threads POSIX
gcc programa.c -o programa -lpthread

# Message Queues e Shared Memory
gcc programa.c -o programa -lrt

# Semáforos (geralmente incluído em -lpthread)
gcc programa.c -o programa -lpthread
```

#### 4.7 Acesso a Arquivos no WSL

**Desafio:** Transferir arquivos entre Windows e WSL.

**Solução:**

```
# Do Windows, acessar WSL:
\\wsl$\Ubuntu\home\engfb\trabalho_sis_op_emb_fabiobraz

# Do WSL, acessar Windows:
/mnt/c/Users/Fabio/Documents/
```

---

## 5. CONCLUSÃO



Este trabalho proporcionou experiência prática com conceitos fundamentais de sistemas operacionais, abrangendo desde mecanismos básicos de criação de processos até implementações de sistemas de tempo real.

## 5.1 Principais Aprendizados

1. **Gerenciamento de Processos:**
  - Criação e controle de processos com `fork()`, `execve()` e `wait()`
  - Estados de processos e transições (R, S, Z, T)
  - Hierarquia de processos e monitoramento
2. **Comunicação Inter-Processos (IPC):**
  - Pipes (comunicação simples e rápida)
  - Message Queues (comunicação estruturada)
  - Shared Memory (comunicação de alta velocidade)
3. **Sincronização:**
  - Semáforos (sinalização e contagem)
  - Mutex (exclusão mútua)
  - Diferenças e casos de uso apropriados
4. **Concorrência:**
  - Threads POSIX (paralelismo leve)
  - Preempção e escalonamento
  - Políticas `SCHED_OTHER` vs `SCHED_FIFO`
5. **Sistemas de Tempo Real:**
  - FreeRTOS como RTOS educacional
  - Tasks, semáforos, filas e timers
  - Escalonamento preemptivo baseado em prioridades

## 5.2 Observações sobre o Ambiente de Desenvolvimento

O uso do WSL2 mostrou-se adequado para o desenvolvimento, oferecendo: - Compatibilidade total com ferramentas POSIX - Integração com VS Code através da extensão Remote WSL - Performance aceitável para aplicações didáticas

Limitações observadas: - Necessidade de `sudo` para operações de tempo real - Overhead da virtualização em testes de performance - Complexidade na transferência de arquivos entre Windows e Linux

## 5.3 Aplicabilidade dos Conceitos

Os programas desenvolvidos formam uma base sólida para: - Desenvolvimento de sistemas embarcados - Programação de aplicações multithreaded - Design de sistemas de controle em tempo real - Compreensão de internals do Linux

## 5.4 Trabalhos Futuros

Extensões possíveis deste estudo: - Implementação de deadlock detection e recovery - Análise de performance com ferramentas de profiling - Port dos exemplos para microcontroladores reais (ESP32, STM32) - Estudo de escalonadores alternativos (`SCHED_DEADLINE`)

---

## 6. REFERÊNCIAS

## 6.1 Materiais do Curso

- Aulas 2, 3, 4, 6, 7 e 8 da disciplina Sistemas Operacionais Embarcados
- Laboratórios práticos disponíveis em:  
<https://github.com/fscard/Sistemas-Operacionais-Embarcados>

## 6.2 Documentação Técnica

1. **POSIX Thread Programming**  
Lawrence Livermore National Laboratory  
<https://hpc-tutorials.llnl.gov/posix/>
2. **The Linux Programming Interface**  
Michael Kerrisk, 2010  
No Starch Press
3. **FreeRTOS Documentation**  
[https://www.freertos.org/Documentation/RTOS\\_book.html](https://www.freertos.org/Documentation/RTOS_book.html)
4. **Linux Kernel Documentation - Scheduler**  
<https://docs.kernel.org/scheduler/index.html>

## 6.3 Man Pages Consultadas

```
man 2 fork      # Criação de processos
man 2 pipe      # Pipes anônimos
man 3 pthread   # POSIX threads
man 7 sem_overview # Semáforos POSIX
man 7 mq_overview # Message queues
man 7 shm_overview # Shared memory
man 2 sched_setscheduler # Políticas de escalonamento
```

## 6.4 Ferramentas Utilizadas

- **GCC 11.4.0** - Compilador C/C++
  - **GDB** - GNU Debugger
  - **Visual Studio Code 1.85** - IDE
  - **WSL2 (Ubuntu 22.04)** - Ambiente Linux no Windows
  - **Git 2.34** - Controle de versão
- 

# APÊNDICE A - Comandos Úteis

## Análise de Processos

```
# Ver todos os processos
ps aux

# Ver estados específicos
ps -eo pid,stat,comm

# Monitoramento em tempo real
top -H -p <PID>

# Árvore de processos
pstree -p
```

```
# Ver threads de um processo
ps -T -p <PID>
```

## Análise de IPC

```
# Listar message queues
ls -l /dev/mqueue/

# Remover message queue
rm /dev/mqueue/my_queue

# Listar shared memory
ls -l /dev/shm/

# Ver semáforos
ipcs -s
```

## Compilação e Debug

```
# Compilação com símbolos de debug
gcc -g programa.c -o programa

# Debug com GDB
gdb ./programa
(gdb) break main
(gdb) run
(gdb) step
(gdb) print variavel

# Verificar dependências
ldd ./programa
```

---

## APÊNDICE B - Troubleshooting Comum

### Problema: “command not found” ao executar programa

#### Solução:

```
# Verificar permissão de execução
ls -l programa
chmod +x programa

# Executar com caminho relativo
./programa
```

### Problema: Processo zumbi não desaparece

#### Solução:

```
# Matar processo pai (força coleta de filhos)
kill -9 <PID_DO_PAI>

# Verificar zumbis
ps aux | grep Z
```

## **Problema: “Cannot allocate memory” em shared memory**

### **Solução:**

```
# Limpar shared memory órfã
rm /dev/shm/*

# Aumentar limite (se necessário)
sudo sysctl -w kernel.shmmax=268435456
```

## **Problema: FreeRTOS não compila**

### **Solução:**

```
# Verificar estrutura de diretórios
ls -la ../FreeRTOS/FreeRTOS/Source/

# Limpar build e recompilar
make clean
make
```

---

### **Fim do Relatório**

**Data de Conclusão:** Novembro de 2025

**Total de Programas Implementados:** 20