Relatório Trabalho 1 Sistemas Operacionais

1) Identificação

Nome: Dante Navaza **Matrícula:** 2321406 **Turma:** 3WA

Nome: Marcela Issa **Matrícula:** 2310746 **Turma:** 3WA

2) Objetivo

O objetivo do trabalho é implementar um interpretador e um escalonador que simulem o gerenciamento de processos com diferentes políticas de escalonamento: REAL-TIME, PRIORIDADE e ROUND-ROBIN. O interpretador lê comandos de um arquivo e os envia ao escalonador, que cria e controla os processos com base na política indicada, utilizando sinais como SIGSTOP, SIGCONT e SIGKILL. O sistema deve respeitar a hierarquia entre as políticas (RT > PRIO > RR), evitar conflitos entre processos REAL-TIME e exibir claramente a ordem de execução e preempções ao longo de 120 unidades de tempo.

3) Estrutura do programa

O trabalho está dividido em arquivos separados:

- **interpretador.c**
    - Implementa o interpretador de comandos.
    - Lê, linha por linha, os comandos de escalonamento presentes no arquivo exec.txt e os envia, com intervalo de 1 unidade de tempo (UT), para o escalonador via stdout, utilizando um pipe.
- **escalonador.c**
    - Implementa o escalonador de processos.
    - Recebe os comandos do interpretador, cria os processos filhos correspondentes e os controla com base nas políticas de escalonamento REAL-TIME, PRIORIDADE e ROUND-ROBIN. Usa sinais do sistema (SIGSTOP, SIGCONT, SIGKILL e SIGINT) para suspender, continuar e

finalizar os processos. Exibe mensagens informando qual processo está sendo executado e o conteúdo das filas.

- **main.c**
  - Responsável por iniciar o sistema.
  - Cria um pipe e dois processos filhos: um para o interpretador e outro para o escalonador. Redireciona a saída do interpretador para a entrada do escalonador. Após iniciar ambos, encerra o processo principal.
- **P1.c, P2.c, P3.c, P4.c, P5.c, P6.c:**
  - Cada um desses arquivos representa um processo simulado do sistema de abastecimento de água. Para o proposito desse trabalho, simulamos eles usando execução contínua com laço infinito e sleep(1) de CPU-bound.

exec.txt

- Arquivo de entrada com os comandos de escalonamento que devem ser enviados do interpretador para o escalonador.
- Cada linha define um processo e sua política, por exemplo:
- Run P1 I=5 D=20 → Processo P1 do tipo REAL-TIME, iniciando na 5ª UT e executando por 20 UTs.
- Run P3 P=1 → Processo P3 com prioridade 1.
- Run P5 → Processo P5 com escalonamento do tipo ROUND-ROBIN.

4) Solução

Arquivo: interpretador.c

-Bibliotecas e definição:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define UT 1
```

- stdio.h: Para podermos utilizar operações de entrada e saída, como printf() e fgetc().
- stdlib.h: Utilizada para funções auxiliares como exit().
- string.h: Utilizada para manipular strings como em strcspn().
- UT: Define a unidade de tempo como 1 segundo

-Função main:

```c
//Interpretador que lê comandos de exec.txt e os envia para o escalonador e
//envia um por um, a cada 1 UT, para o escalonador via stdout (pipe)
int main(void) {
    fprintf(stderr, "[Interpretador] Iniciando leitura de comandos.\n");
    // Abre o arquivo exec.txt para leitura
    FILE *arquivo = fopen("exec.txt", "r");
    if (!arquivo) {
        perror("[Interpretador] Erro ao abrir exec.txt");
        exit(1);}
    char linha[256]; // Cria um buffer para armazenar as linhas lidas
    while (fgets(linha, sizeof(linha), arquivo)) {
        linha[strcspn(linha, "\n")] = '\0';  // Substitui o \n por \0 no final
        fprintf(stderr, "[Interpretador] Enviando: %s\n", linha);
         // Envia o comando para o escalonador via pipe (stdout)
        // Enviamos strlen(linha) + 1 para incluir o '\0' no final
        write(STDOUT_FILENO, linha, strlen(linha) + 1);
        // Aguarda 1 unidade de tempo antes de enviar o próximo comando
        sleep(UT);}
    fclose(arquivo);
    fprintf(stderr, "[Interpretador] Fim da execução.\n");
    return 0;}
```

- **Parâmetros:**
    - O programa não recebe parâmetros por linha de comando. Ele lê automaticamente o arquivo exec.txt, localizado no mesmo diretório de execução.
- **Objetivo:**
    - Ler os comandos de escalonamento especificados no arquivo exec.txt e enviá-los um a um, com intervalo de 1 unidade de tempo, para o escalonador, utilizando comunicação via pipe através da saída padrão.

- **Como foi feito:**
    - O arquivo exec.txt é aberto com fopen. Cada linha é lida com fgets, tem o caractere de nova linha removido, e é enviada ao escalonador usando a função write. Entre cada envio, é feita uma pausa de 1 segundo com sleep, simulando uma unidade de tempo. As mensagens de status são impressas na saída de erro padrão (stderr) para fins de depuração.
- **Retorno:**
    - Ao final da leitura de todas as linhas, o programa fecha o arquivo, imprime uma mensagem indicando o término da execução e encerra com retorno 0. Caso ocorra erro ao abrir o arquivo, o programa imprime uma mensagem de erro e encerra com retorno diferente de zero.

Arquivo: P1.c, P2.c, P3.c, P4.c, P5.c e P6.c

-Biblioteca:

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

- stdio.h: Para entrada e saída padrão
- unistd.h: Para criação de processos e controle.

-Função main (exemplo P1.c) :

Os arquivos P1.c, P2.c, P3.c, P4.c, P5.c e P6.c seguem a mesma estrutura básica, pois todos têm o mesmo comportamento: simulam processos CPU-bound que permanecem em execução contínua. A única diferença entre eles está no nome do programa impresso na tela, o que permite identificá-los durante a execução. Por isso, basta apresentar um dos arquivos como exemplo para ilustrar o funcionamento de todos.

```
int main() {
    printf("Executando programa P1 (pid=%d)…\n", getpid());

    while (1) {
        sleep(1); // Simula trabalho infinito }
    return 0;}
```

- **Parâmetros:**
    - Não há entrada de parâmetros. O programa apenas inicia e permanece em execução simulando carga de CPU.
- **Objetivo:**
    - Simular um processo do tipo CPU-bound, que permanece em execução contínua e controlável pelo escalonador.
- **Como foi feito:**
    - Cada programa imprime seu próprio PID ao iniciar e entra em um laço infinito com sleep(1), representando um processo ativo. O nome do executável (P1, P2 etc.) representa diferentes funções em um sistema de monitoramento.
- **Retorno:**
    - O programa não retorna normalmente, pois é interrompido ou finalizado pelo escalonador. Se fosse encerrado manualmente, retornaria 0.

Arquivo: escalonador.c

-Biblioteca:

```c
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

#include <signal.h>
```

- stdio.h: Para entrada e saída padrão, como o printf().
- stdlib.h: Para funções de controle de processo (exit()).
- unistd.h: Para funções como fork(), execlp(), sleep().
- signal.h: Para envio de sinais (kill()).
- sys/wait.h: Para esperar o término dos processos filhos (waitpid()).

-Variaveis:

```c
#define MAX_PROCESSOS 50

#define UT 1

typedef enum { ROUND_ROBIN, PRIORIDADE, REAL_TIME } Tipo;

typedef struct {

   pid_t pid;

   char  nome[20];

   Tipo  tipo;

   int   prioridade;

   int   inicio;

   int duracao;

   int   tempo_executado;

   int   ativo;

} Processo;

Processo processos[MAX_PROCESSOS];

int num_processos = 0;

int tempo_global = 0;

int round_robin_ultimo = -1;
```

- Definimos o numero maximo de processos em nosso sistema como 50, além de definir o intervalo de tempo da linha do tempo da execucao dos processos como 1 UT. Criamos uma lista enumerada dos tipos diferentes dos processos para acesso facil e definimos a struct Processo, que consiste de todos os valores necessarios de um processo como seu nome, status de ativo ou inativo, tipo, inicio e fim de duracao e tempo executado. Por fim, criamos um array capaz de armazenar 50 processos (max_processos) e inicializamos o numero de processos registrados como 0, igual como o tempo global. Inicializamos o index do ultimo round robin executado como -1 (representando o primeiro).

-void exibir_filas(Processo *atual):

```c
void exibir_filas(Processo *atual)
```

```
{
    printf("[Tempo %d] Filas:\n", tempo_global);

    printf("  REAL_TIME:   ");
    for (int i = 0; i < num_processos; i++)
    {
        if (atual && processos[i].pid == atual->pid) { // se o processo for o atual, ele nao estara na fila de espera
            continue;
        }
        if (processos[i].ativo && processos[i].tipo == REAL_TIME) // pega os processos real time
        {
            printf("%s ", processos[i].nome);
        }
    }

    printf("\n  PRIORIDADE: ");
    for (int i = 0; i < num_processos; i++)
    {
        if (atual && processos[i].pid == atual->pid) {
            continue;
        }
        if (processos[i].ativo && processos[i].tipo == PRIORIDADE)
        {
            printf("%s(P=%d) ", processos[i].nome, processos[i].prioridade);
        }
    }

    printf("\n  ROUND_ROBIN:   ");
    for (int i = 0; i < num_processos; i++)
```

```
    {
        if (atual && processos[i].pid == atual->pid)
        {
            continue;
        }
        if (processos[i].ativo && processos[i].tipo == ROUND_ROBIN)
        {
            printf("%s ", processos[i].nome);
        }
    }

    printf("\n");
}
```

- **Parâmetros:**
  - Recebe o processo atual como parametro
- **Objetivo:**
  - Exibir a fila de processos em espera no formato <TIPO PROCESSO> <PROCESSO>
- **Como foi feito:**
  - Tres loops for (invés de 1 unico for para garantir que cada tipo de processo apareca separado invés de misturado com outros processos)
  - Cada loop percorre todos os processos ativos e verifica seu tipo, se ambas as condicoes forem verdadeiras ele o exibe usando seu index.
  - Cada loop tambem verifica se o processo do index atual, é o processo que esta sendo executado. Caso ele seja, ele nao aparecerá na lista (o continue ira ignorar essa iteração)
- **Retorno:**
  - Nenhum, a função é tipo void e serve apenas para exibir informações.

-int conflitoRT(int inicio, int duracao):

```
int conflitoRT(int inicio, int duracao){
```

```
  for (int i = 0; i < num_processos; i++){

      if (processos[i].tipo != REAL_TIME){  // ignora os que nao sao real time

          continue;}

      // calcula o intervalo de tempo deles

      int inicio_processo_existente = processos[i].inicio;

      int fim_processo_existente = inicio_processo_existente + processos[i].duracao;

      // Calcula o intervalo de tempo do novo processo

      int inicio_novo_processo = inicio;

      int fim_novo_processo = inicio_novo_processo + duracao;

      // tem conflito se o início do novo for antes do fim do existente e o início do
existente for antes do fim do novo

      if (inicio_novo_processo < fim_processo_existente && inicio_processo_existente
< fim_novo_processo){

          return 1;}}

  return 0;
```

- **Parâmetros:**
  - Recebe um int de inicio de tempo e outro de duracao de execucao.
- **Objetivo:**
  - A função verifica se um novo processo REAL_TIME entra em conflito de tempo com algum outro já existente. Ela retorna 1 se houver sobreposição entre os intervalos de execução, e 0 caso contrário.
- **Como foi feito:**
  - A função percorre todos os processos já carregados e ignora os que não são REAL_TIME. Para cada processo REAL_TIME, ela calcula o intervalo de tempo ocupado (início até início + duração) e compara com o intervalo do novo processo. Se houver sobreposição entre os intervalos — ou seja, se o novo processo começar antes de o atual terminar e o atual começar antes de o novo terminar a função retorna 1, (conflito). Caso nenhum conflito seja encontrado, retorna 0.

- **Retorno:**
    - Retorna 1 se deu conflito e 0 se não houve

-Função main:

```c
int lerNumero(char **p){

 // transforma inteiro positivo de string para numero

   int valor = 0;

   while (**p >= '0' && **p <= '9'){

       valor = valor * 10 + (**p - '0');

       (*p)++;}

   return valor;}
```

- **Parâmetros:**
    - Recebe uma string de numeros positivos
- **Objetivo:**
    - A funcao recebe uma string de numero positivo e converte em int. Sera usada com frequência na leitura do txt.
- **Como foi feito:**
    - A função recebe um ponteiro para ponteiro (char **p), que aponta para a posição atual em uma string. Ela percorre caractere por caractere enquanto o caractere for um dígito ('0' a '9'), convertendo esse caractere para número (**p - '0') e acumulando em valor (multiplicando o valor anterior por 10 para cada dígito.
- **Retorno:**
    - Retorna o int convertido

-Função main:

```c
int main(void){

   char linha[256];

   printf("[Escalonador] Iniciando...\n");
```

```c
// simula 120 UT
while (tempo_global < 120) {
    // Lê comandos enviados pelo STDIN (pipe do interpretador)
    int n = read(STDIN_FILENO, linha, sizeof(linha) - 1);
    if (n > 0){
        linha[n] = '\0';  // adicionando final a uma string
        // variaveis para inicializar os processos
        char nome[20];
        int prioridade = -1;
        int inicio = -1;
        int duracao = -1;
        Tipo tipo = ROUND_ROBIN;
        // Parsing do comando
        char *p = linha;
        while (*p == ' '){
            p++;}  // pula espaços
        if (strncmp(p, "Run", 3) == 0) {
            p += 3;
            while (*p == ' ') p++;
            // Lê o nome do processo
            int i = 0;
            while (*p && *p != ' ') {
                nome[i++] = *p++;}
            nome[i] = '\0';
            while (*p == ' '){
```

```
                    p++;}

                // Verifica se é PRIORIDADE ou REAL_TIME

                if (*p == 'P' && *(p+1) == '=') {

                    p += 2;

                    prioridade = lerNumero(&p);

                    tipo = PRIORIDADE;} // porque tem P =

                else if (*p == 'I' && *(p+1) == '=') {

                    p += 2;

                    inicio = lerNumero(&p);

                    while (*p == ' '){

                        p++;}

                    if (*p == 'D' && *(p+1) == '=') {

                        p += 2;

                        duracao = lerNumero(&p);

                        tipo = REAL_TIME; // pq tem I =

                        // Verifica se há conflito de tempo

                        if (inicio + duracao > 60 || conflitoRT(inicio, duracao)) {

                            printf("[Escalonador] Conflito no processo %s (REAL_TIME).
Ignorado.\n", nome);

                            continue;}}}}

            // Verifica se o processo já foi criado

            int duplicado = 0;

            for (int i = 0; i < num_processos; i++) {

                if (strcmp(processos[i].nome, nome) == 0 && processos[i].ativo) {

                    printf("[Escalonador] Processo %s ja existe. Ignorado.\n", nome);
```

```
                duplicado = 1;
                break;}}
        if (duplicado){
            continue;}
        // Cria processo filho e interrompe (aguarda escalonamento)
        pid_t pid = fork();
        if (pid == 0) {
            execl(nome, nome, NULL);
            perror("[Escalonador] execl");
            exit(1);}
        kill(pid, SIGSTOP);  // pausa imediatamente após criação
        // Registra o processo na lista
        Processo pnovo = { pid, "", tipo, prioridade, inicio, duracao, 0, 1 };
        strncpy(pnovo.nome, nome, sizeof(pnovo.nome) - 1);
        processos[num_processos++] = pnovo;
        printf("[Escalonador] %s carregado (PID %d)\n", nome, pid);}
    else if (n == -1){
        perror("[Escalonador] Erro de leitura");}
    // escolhendo o processo atual
    Processo *atual = NULL;
    int menor_prioridade = 100; // menor prioridade
    int segundos = tempo_global % 60;
    // Prioridade para processos REAL_TIME dentro da janela de execução
    for (int i = 0; i < num_processos; i++) {
        Processo *p = &processos[i];
```

```
        if (!p->ativo || p->tipo != REAL_TIME){
            continue;}
        if (segundos >= p->inicio && segundos < p->inicio + p->duracao) {
            atual = p;
            break;}}
    // Depois, PRIORIDADE com menos prioridade (menor valor) e menos de 3 UTs
    if (!atual) {
        for (int i = 0; i < num_processos; i++) {
            Processo *p = &processos[i];
            if (!p->ativo || p->tipo != PRIORIDADE){
                continue;}
            if (p->tempo_executado < 3 && p->prioridade < menor_prioridade) {
                menor_prioridade = p->prioridade;
                atual = p;}}}
    // ultimo, ROUND_ROBIN em ordem circular
    if (!atual) {
        for (int deslocamento = 1; deslocamento <= num_processos; deslocamento++){
            int index = (round_robin_ultimo + deslocamento) % num_processos;
            Processo *p = &processos[index];
            if (p->ativo && p->tipo == ROUND_ROBIN) {
                atual = p;
                round_robin_ultimo = index;
                break; }}}
    // executando os processos
    if (atual) {
```

```
            int tempo_restante = -1;

            // Calcula tempo restante com base no tipo

            if (atual->tipo == PRIORIDADE){

                tempo_restante = 3 - atual->tempo_executado;}

            else if (atual->tipo == REAL_TIME){

                tempo_restante = (atual->inicio + atual->duracao) - segundos;}

            // Mostra mensagem com tempo restante

            if (tempo_restante >= 0){

                printf("\n\n[Tempo %d] Executando %s (restam %d UTs)\n", tempo_global,
atual->nome, tempo_restante);}

            else{

                printf("\n\n[Tempo %d] Executando %s\n", tempo_global, atual->nome);}

            // Executa 1 UT

            kill(atual->pid, SIGCONT);

            sleep(UT);

            kill(atual->pid, SIGSTOP);

            atual->tempo_executado++;

            // Finaliza PRIORIDADE após 3 UTs

            if (atual->tipo == PRIORIDADE && atual->tempo_executado == 3){

                atual->ativo = 0;

                kill(atual->pid, SIGKILL);

                printf("[Tempo %d] %s finalizado\n", tempo_global, atual->nome);}}

        else{

            // Nenhum processo pronto para execução

            printf("[Tempo %d] Nenhum processo para executar\n", tempo_global);
```

```
            sleep(UT);}

        exibir_filas();

        tempo_global++;}

    printf("[Escalonador] Tempo maximo atingido.\n");

    for (int i = 0; i < num_processos; i++){

        if (processos[i].ativo){

            printf("[Escalonador] Finalizando %s (pid %d)\n", processos[i].nome,
processos[i].pid);

            kill(processos[i].pid, SIGKILL);

            waitpid(processos[i].pid, NULL, 0);}}

    return 0;}
```

- **Parâmetros:**
  - Nao recebe parametros
- **Objetivo:**
  - ê os comandos enviados pelo interpretador via stdin (pipe).
  - Interpreta os comandos, identificando se o processo é ROUND_ROBIN, PRIORIDADE ou REAL_TIME.
  - Cria e registra processos filhos conforme os comandos recebidos.
  - Em cada unidade de tempo (UT), decide qual processo deve executar com base na política de escalonamento:
  - Prioridade para REAL_TIME,
  - Depois PRIORIDADE (até 3 UTs),
  - Depois ROUND_ROBIN circular.
  - Executa o processo selecionado por 1 UT, o preempra (pausa), e repete até 120 UTs.
  - No final, encerra todos os processos ainda ativos.
- **Como foi feito:**
  - Percorre um while ate o tempo global chegar a 120 UT

```
    int n = read(STDIN_FILENO, linha, sizeof(linha) - 1);
```

- Lê uma linha enviada pelo interpretador.
- Cada linha representa um comando Run <nome> P=n I=n D=n.
- Após isso ele inicializa as variaveis para criar os processos

```
            if (strncmp(p, "Run", 3) == 0) {
```

- Após remover os espaços iniciais, verifica se o comando começa com Run.
- Em seguida, lê o nome do processo.
- Se houver P=, marca como PRIORIDADE e armazena a prioridade.
- Se houver I= e D=, marca como REAL_TIME e define o intervalo de tempo.
- Também chama conflitoRT para verificar se há sobreposição com outros REAL_TIME e verifica se há conflito com I + D > 60.

```
                if (strcmp(processos[i].nome, nome) == 0 && processos[i].ativo)
{
```

- Depois, a funcao verifica se ja há um processo duplicado registrado, caso tenha, ele ira ignorar essa iteracao do loop

```
            pid_t pid = fork();

            if (pid == 0) {

                execl(nome, nome, NULL);

                perror("[Escalonador] execl");

                exit(1);

            }

            kill(pid, SIGSTOP);  // pausa imediatamente após criação

            // Registra o processo na lista

            Processo pnovo = { pid, "", tipo, prioridade, inicio, duracao, 0, 1
};

            strncpy(pnovo.nome, nome, sizeof(pnovo.nome) - 1);

            processos[num_processos++] = pnovo;
```

- o processo é criado usando pid e é imediatamente interrompido (para ser executado apenas quando for sua vez na linha do tempo). O processo entao é registrado na lista de processos com seus valores (tempo, duracao, tipo, etc).
- Após isso, ele seleciona a ordem em que os processos serao executados, seguindo as metricas abaixo (a hierarquia da execucao dos diferentes tipos é definida pela ordem em que os loops sao executados, o loop do real_time executa primeiro, em seguida o da prioridade, e por ultimo o round robin.
- REAL_TIME ativo dentro do seu intervalo e com I + duracao > segundos (como processos real time apenas executam em janelas de 60 segundos, a variável segundos é calculada com tempo_global % 60)
  - if (segundos >= p->inicio && segundos < p->inicio + p->duracao)
  - Isso garante que cada processo round robin é iniciado no segundo I de cada minuto
- PRIORIDADE com menor valor e menos de 3 UTs executadas
  - if (p->tempo_executado < 3 && p->prioridade < menor_prioridade)
    - Aqui é feita a lógica de escolha do melhor processo de prioridade. Um processo PRIORIDADE só pode executar até 3 unidades de tempo (UTs) no total. Se já executou as 3, ele será ignorado. Menor valor significa maior prioridade
  - Se esse processo satisfaz ambas as condições, ele é um bom candidato para execução.
  - Atualiza menor_prioridade com a prioridade desse processo (para comparar com os próximos).
  - Salva esse processo como o atual a ser executado.
- ROUND_ROBIN circular com índice rotativo (round_robin_ultimo)

```
if (!atual) {
            for (int deslocamento = 1; deslocamento <=
num_processos; deslocamento++)
            {
                int index = (round_robin_ultimo +
```

```
deslocamento) % num_processos;
                Processo *p = &processos[index];
                if (p->ativo && p->tipo == ROUND_ROBIN) {
                    atual = p;
                    round_robin_ultimo = index;
                    break;
                }
            }
        }

```

- Verifica se nenhum processo foi selecionado ainda (if (!atual)).
- Percorre a lista de processos a partir do próximo após o último processo ROUND_ROBIN executado.
- Calcula o índice de forma circular com (round_robin_ultimo + deslocamento) % num_processos (se a soma for maior que num_processos, ele retorna ao inicio da lista).
- Verifica se o processo nesse índice está ativo e é do tipo ROUND_ROBIN.
- Se for válido, seleciona o processo para execução.
- Atualiza round_robin_ultimo com o índice do processo escolhido.
- Interrompe o loop assim que encontra um processo válido.

- Após tudo isso, a funcao calcula o UT restante dos processos real time e prioridade (o round robin nao precisa pois rodara infinitamente) e os exibe junto com a fila.
- Por fim, utilizando a chamada sleep(UT), avancamos 1 UT na linha do tempo da execucao dos processos e tambem incrementamos a variavel tempo_global, para avancar no while de 120 UT.
- Se o tempo executado do processo de prioridade chegar a 3, ele é finalizado e caso a linha do tempo chegue a 120 UT todos os processos em fila sao finalizados, encerrando a execucao do programa

- **Retorno:**
  - Durante a execucao do programa sera exibida uma linha do tempo com a fila de todos os processos em espera e tambem com o processo atual em

execucao (junto com o tempo UT que falta para aquele processo finalizar). Após a linha do tempo passar de 120 UT, mensagens para cada processador ainda na fila de espera serao exibidos anunciando que eles serao finalizados. Caso todo o programa rode corretamente, ele ira retornar 0.

Arquivo: main.c

-Biblioteca:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

- stdio.h: Para entrada e saída padrão printf().
- stdlib.h: Usada para funções utilitárias como exit().
- unistd.h: Para funções de controle de processo (sleep(), getpid()).

-Função main:

```c
int main(void) {
    int pipefd[2]; // pipefd[0] = leitura, pipefd[1] = escrita
    // Cria o pipe para comunicação entre interpretador e escalonador
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(1);}
    pid_t pid_int = fork();
    if (pid_int == 0) {
        close(pipefd[0]);                       // Fecha a extremidade de leitura
        dup2(pipefd[1], STDOUT_FILENO);         // Redireciona stdout para o pipe
        close(pipefd[1]);                       // Fecha a extremidade original após duplicar
        execl("./interpretador", "interpretador", NULL); // Executa o programa interpretador
        perror("execl interpretador");                   // Só chega aqui se execl falhar
        exit(1);}
    pid_t pid_esc = fork();
    if (pid_esc == 0) {
        close(pipefd[1]);                      // Fecha a extremidade de escrita
        dup2(pipefd[0], STDIN_FILENO);         // Redireciona stdin para o pipe
```

```
    close(pipefd[0]);                       // Fecha a extremidade original após duplicar
    execl("./escalonador", "escalonador", NULL); // Executa o programa escalonador
    perror("execl escalonador");                 // Só chega aqui se execl falhar
    exit(1);}
// Fecha os lados do pipe
close(pipefd[0]);
close(pipefd[1]);
printf("[Main] Interpretador (PID %d) e Escalonador (PID %d) iniciados.\n",
       pid_int, pid_esc);
printf("[Main] Processo principal encerrado.\n");

return 0;}
```

- **Parâmetros:**
    - A função main não recebe argumentos. O funcionamento depende apenas da existência dos executáveis "interpretador" e "escalonador" no mesmo diretório.
- **Objetivo:**
    - Inicializar o sistema de escalonamento criando dois processos distintos, um para o interpretador e outro para o escalonador, conectando-os por um pipe para comunicação interprocesso.
- **Como foi feito:**
    - Foi criado um pipe usando a função pipe(). Em seguida, foram feitos dois fork()s: o primeiro cria o processo do interpretador e redireciona sua saída padrão para o pipe; o segundo cria o escalonador, redirecionando sua entrada padrão para o pipe. A função execl() é usada para iniciar os executáveis respectivos. O processo pai fecha ambos os lados do pipe e finaliza.
- **Retorno:**
    - Se tudo ocorre corretamente, o programa imprime uma mensagem indicando os PIDs dos processos criados e retorna 0. Em caso de erro na criação do pipe ou ao executar os processos, imprime uma mensagem de erro e encerra com código diferente de zero.

5) Observações e conclusões

Linha do tempo:

exec.txt testado

Run P1 I=5 D=20

Run P2 I=30 D=5

Run P3 P=5

Run P4 P=3

Run P5

Run P6

Processos Carregados

- P1: PID 72667, Tipo: REAL_TIME, Duração: 40 (executado duas vezes)
- P2: PID 72668, Tipo: REAL_TIME, Duração: 10 (executado duas vezes)
- P3: PID 72671, Tipo: PRIORIDADE, Prioridade: 5, Duração: 3
- P4: PID 72672, Tipo: PRIORIDADE, Prioridade: 3, Duração: 3
- P5: PID 72673, Tipo: ROUND_ROBIN
- P6: PID 72674, Tipo: ROUND_ROBIN

Resumo de Execução por Tempo

- **Tempo 0 a 1**: Nenhum processo foi executado.
- **Tempo 2**: Executa P3 (PRIORIDADE 5), restam 3 UTs
- **Tempo 3-4**: Executa P4 (PRIORIDADE 3), restam 2 UTs
- **Tempo 5 a 24**: Executa P1 (REAL_TIME), de 20 a 1 UT restante
- **Tempo 25**: Finaliza P4 (PRIORIDADE 3)
- **Tempo 26-27**: Executa e finaliza P3 (PRIORIDADE 5)

- **Tempo 28-29**: Executa P5 e P6 (ROUND_ROBIN)
- **Tempo 30-34**: Executa e finaliza P2 (REAL_TIME)
- **Tempo 35-64**: Executa alternadamente P5 e P6
- **Tempo 65-84**: Executa P1 (REAL_TIME), de 20 a 1 UT restante
- **Tempo 85-89**: Alterna entre P5 e P6
- **Tempo 90-94**: Executa e finaliza P2 (REAL_TIME)
- **Tempo 95-119**: Alterna entre P5 e P6

Finalização dos Processos no Tempo 119

- P1: PID 72667
- P2: PID 72668
- P5: PID 72673
- P6: PID 72674

Fim da Execução: Tempo Global = 120

Observações:

A linha do tempo acima comprova que a ordem de execução dos programas foi a esperada, com todos os eventos de preempção e hierarquia de processos (real time > prioridade > round robin) funcionando corretamente. Os processos de real time ocorrem a partir do segundo ‘I’ de cada minuto e executam por duração ‘D’, enquanto os de prioridade executam por 3 UT de maneira finita conforme solicitado (com os processos com número de prioridade menor sendo executados primeiro). Por fim, os Round Robin, executam infinitamente (nesse caso até o UT 120 limite do programa) com a menor prioridade dentre os três processos.

Nossas principais dificuldades foram em pensar na lógica de calcular o conflito de processos realtime e em decidir se era necessário ou não contar o tempo de execução UT quando não tinha nenhum processo sendo executado (como visto no começo da linha do tempo acima), porém após revisar o conteúdo, decidimos manter a progressão do UT mesmo quando não há processos em execução, pois ainda é considerado parte da execução do sistema.

Como compilar:

Para compilar, use o comando gcc no terminal com os arquivos:

gcc main.c -o main

gcc interpretador.c -o interpretador

gcc escalonador.c -o escalonador

gcc P1.c -o P1

gcc P2.c -o P2

gcc P3.c -o P3

gcc P4.c -o P4

gcc P5.c -o P5

gcc P6.c -o P6

./main