Felipe Cancella - 2210487
Diego Miranda - 2210996

Trabalho 1 SO

Objetivo;

Desenvolver um sistema de gerenciamento de processos que inclui um interpretador de comandos e um escalonador de programas. O sistema deve ser capaz de lidar com dois tipos de políticas de escalonamento de processos: ROUND-ROBIN e REAL-TIME.

O interpretador lê comandos de um arquivo (exec.txt) que especifica a execução de programas, e o escalonador coordena a execução desses programas, garantindo as prioridades estabelecidas, e tempo de execução de cada um.

Além de considerar as prioridades, o sistema precisa garantir que os processos não entrem em conflito com os processos já em execução, e que a execução dos programas não ultrapasse o início do próximo minuto (além do limite total de 2 minutos).

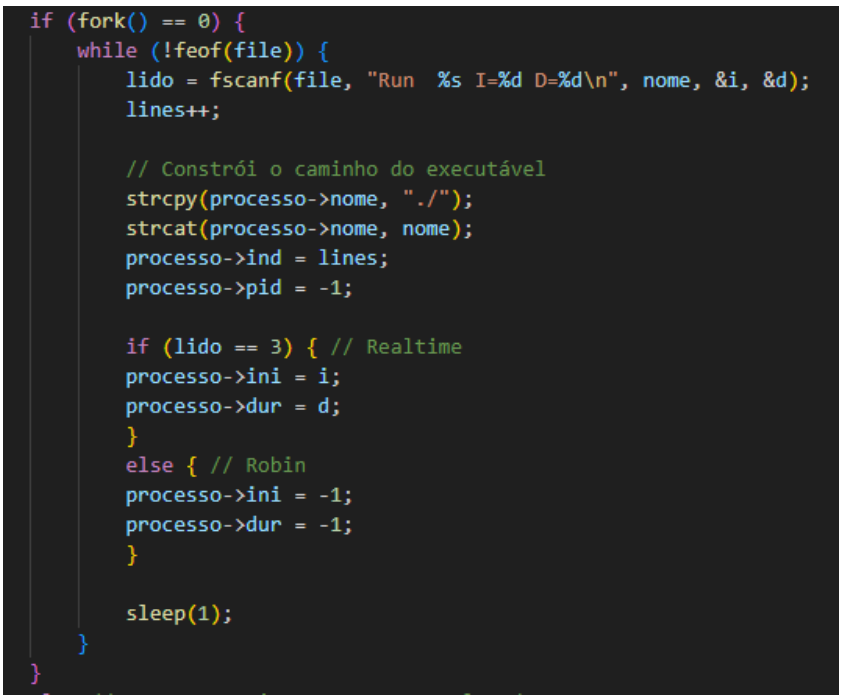
Estrutura do programa;

Interpretador -

O código interpretador.c é responsável por ler cada linha do exec.txt e coletar as informações de cada linha de comando. Ou seja, é nesse .c que nós vamos identificar o tipo (Real-Time ou Robin) do comando a ser executado, e identificar também seu início e tempo de duração.

Ele realiza essa tarefa iterando sobre cada linha, interpretando ela, e enviando isso de volta para o escalonador por meio de memória compartilhada (Guarda todas as informações necessárias para o escalonador em um struct compartilhado).

Depois de ler a linha em questão, o interpretador identifica cada informação dela e guarda no struct compartilhado.



```c
if (fork() == 0) {
    while (!feof(file)) {
        lido = fscanf(file, "Run  %s I=%d D=%d\n", nome, &i, &d);
        lines++;

        // Constrói o caminho do executável
        strcpy(processo->nome, "./");
        strcat(processo->nome, nome);
        processo->ind = lines;
        processo->pid = -1;

        if (lido == 3) { // Realtime
        processo->ini = i;
        processo->dur = d;
        }
        else { // Robin
        processo->ini = -1;
        processo->dur = -1;
        }

        sleep(1);
    }
}
```

Escalonador -

O escalonador é responsável por monitorar o tempo e coordenar a execução de cada processo/arquivo com base nas políticas de escalonamento ROUND-ROBIN e REAL-TIME.

Utilizando memória compartilhada para obter informações sobre os processos a serem executados e uma bandeira que indica a disponibilidade de novos processos, o escalonador toma decisões de escalonamento, interrompendo e retomando a execução dos processos conforme necessário.

Exemplos de quando o arquivo atual já foi lido e iniciado. Nesse caso ele continua o processo “novo” e para o atual dependendo da prioridade. Ou simplesmente para/guarda o processo novo para começar depois.
Obs.: segundos é um vetor de inteiros que guarda basicamente uma booleana em cada índice indicando se aquele segundo já está tomado por uma tarefa. (Run arq2 i=20 d=7. Vai ocupar os índices 20,21,22,23,24,25,26)

```
while (ticks < 120) {
    if (rt_cont + rr_cont != *num) { // se falta processo a ser inserido
        qsort(rt_vetor, rt_cont, sizeof(Processo), comparaProcesso);
        if (processo->ini != -1) {//Realtime
            if (validaProcessosRT(rt_cont, processo, rt_vetor) == 0) { //verifica o processo
                alocaProcesso(&rt_cont, &rt_vetor[rt_cont], processo);
                printf("Alocou RT: \t%s\n", processo->nome);
            }
            else
            printf("Nao vai alocar RT: \t%s (colide com %s)\n",processo->nome, rt_vetor[rt_cont-1].nome);
        }
        else {//Robin
            alocaProcesso(&rr_cont, &rr_vetor[rr_cont], processo);
            printf("Alocou RR: \t%s\n", processo->nome);
        }
    }
```

Os processos REAL-TIME são executados de acordo com seus momentos de início, enquanto os processos ROUND-ROBIN são alocados em uma fila com base em sua disponibilidade.

Na imagem acima podemos ver as duas primeiras condições para real-time, e a última para round robin.

Solução;

Interpretador -

Struct que guarda as informações que serão compartilhadas com o escalonador:

```
struct processo {
    char nome[30];
    int ini;
    int dur;
    int tipo;
    int ind;
    pid_t pid;
} typedef Processo;
```

Começamos o loop de iteração definindo “padrões” antes de ler a linha em questão, definindo nome, posição no vetor e pid, e depois partimos para pegar o resto das informações específicas por tipo, e por fim botamos tudo isso dentro do struct compartilhado.

```
if (fork() == 0) {
    while (!feof(file)) {
        lido = fscanf(file, "Run  %s I=%d D=%d\n", nome, &i, &d);
        lines++;

        // Constrói o caminho do executável
        strcpy(processo->nome, "./");
        strcat(processo->nome, nome);
        processo->ind = lines;
        processo->pid = -1;

        if (lido == 3) { // Realtime
        processo->ini = i;
        processo->dur = d;
        }
        else { // Robin
        processo->ini = -1;
        processo->dur = -1;
        }

        sleep(1);
    }
}
```

Chamada do escalonador, além de passar os últimos dados para a memória compartilhada

```
else // Processo pai: executa o escalonador
    execvp(command, args);
printf("--%d--",lines);
// Escreve o número de linhas no segmento de memória compartilhada
*num = lines;

// Fecha o arquivo
fclose(file);

// Desanexa os segmentos de memória compartilhada
shmdt(processo);
shmdt(num);

// Aguarda indefinidamente
while (1);

return 0;
```

Escalonador -

Struct que armazena os dados de cada processo

```c
struct processo {
    char nome[30];
    int ini;
    int dur;
    int tipo;
    int ind;
    pid_t pid;
} typedef Processo;
```

Função e trecho que vão manipular e ordenar as filas de execução por tipo de processo

```c
while (ticks < 120) {
    if (rt_cont + rr_cont != *num) { // se falta processo a ser inserido
        qsort(rt_vetor, rt_cont, sizeof(Processo), comparaProcesso);
        if (processo->ini != -1) {//Realtime
            if (validaProcessosRT(rt_cont, processo, rt_vetor) == 0) { //verifica o processo
                alocaProcesso(&rt_cont, &rt_vetor[rt_cont], processo);
                printf("Alocou RT: \t%s\n", processo->nome);
            }
            else
            printf("Nao vai alocar RT: \t%s (colide com %s)\n",processo->nome, rt_vetor[rt_cont-1].nome);
        }
        else {//Robin
            alocaProcesso(&rr_cont, &rr_vetor[rr_cont], processo);
            printf("Alocou RR: \t%s\n", processo->nome);
        }
    }
```

Após receber um processo, organizamos por ordem de início, então verificamos de qual tipo de execução ele pertence, para que seja alocado corretamente e se dê a mensagem de inserção.

```c
//Realiza a divisao para a execucao do processo
void alocaProcesso(int* ind, Processo* comando, Processo* processo) {
    strcpy(comando->nome, processo->nome);
    comando->ind = processo->ind;
    comando->ini = processo->ini;
    comando->dur = processo->dur;
    comando->pid = fork();

    if (comando->pid == 0) {
        if (execvp(comando->nome, args) == -1) {
        perror("execvp");
        exit(EXIT_FAILURE);
        }
    }
    else if (comando->pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    else {
        kill(comando->pid, SIGSTOP);
        (*ind)++;
    }
}
```

Dentro do loop com limite de tempo, após alocar devidamente os processos, registra o tempo decorrido e faz o tratamento de execução

```c
gettimeofday(&t, NULL);
tempo = t.tv_sec % 60;
printf("\nSegundos: \t%ds\n", tempo);

// Tratando Processo REAL-TIME
if (rt_cont > 0){
    for (int i = 0; i < rt_cont; i++) { // Checa cada Processoo RT existente
        // Condicao de inicio
        if (tempo == rt_vetor[i].ini) {
            kill(rt_vetor[i].pid, SIGCONT);
            rt_rodando = 1;
            printf("Rodando RT \t%s\n", rt_vetor[i].nome);
        }
        // Condicao de parada
        if (tempo == (rt_vetor[i].ini + rt_vetor[i].dur) - 1) {
            rt_rodando = 0;
            kill(rt_vetor[i].pid, SIGSTOP);
        }
    }
    if(rt_rodando!=0){
        printf("Rodando RT\n");
    }
}
// Tratando Processoo ROUND-ROBIN
if (rt_rodando == 0 && rr_cont > 0) {
    int rr_anterior = (rr_rodando - 1 + rr_cont) % rr_cont; // Circularmente anterior
    kill(rr_vetor[rr_anterior].pid, SIGSTOP);
    if (rr_anterior != rr_rodando){ // Se tem apenas um RR na lista, apenas para
        kill(rr_vetor[rr_rodando].pid, SIGCONT);
        printf("Rodando RR \t%s\n", rr_vetor[rr_rodando].nome);
    }
    rr_rodando = (rr_rodando + 1) % rr_cont; // Próximo Processoo RR
}
sleep(1);
ticks++;
```

Verifica se há RT para ser executado, e para cada um dos processos compara o tempo atual com o de início e de fim, para respectivamente iniciar ou parar a execução. Já para os RR, verifica que não há RT rodando e que o RR recém encerrado não é o próximo a ser executado, para garantir que ele será executado por apenas 1 segundo

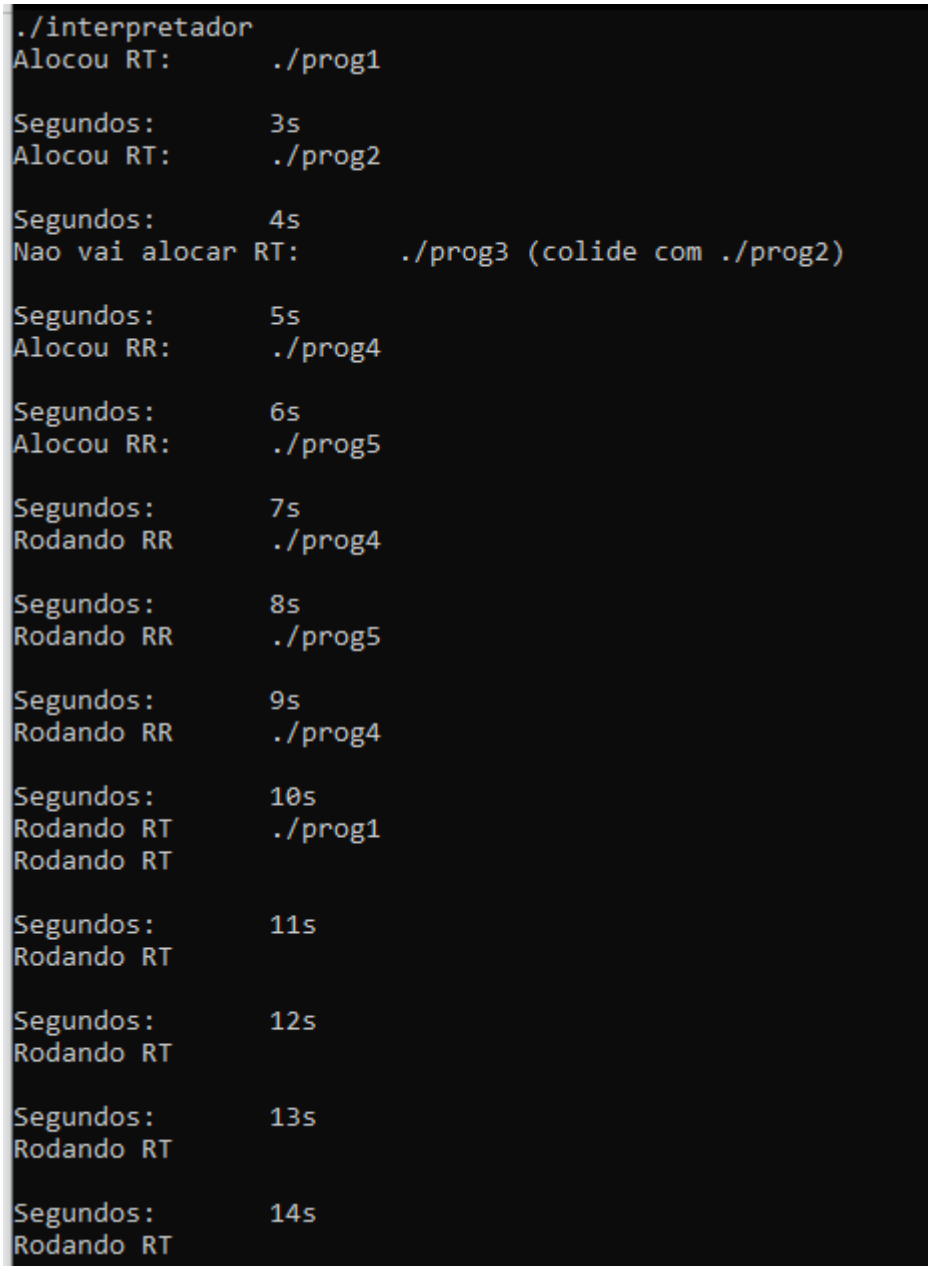
Por fim, liberamos as memórias compartilhadas ao fim do ciclo de execução

```c
shmdt(processo);
shmctl(segP, IPC_RMID, 0);

shmdt(num);
shmctl(segN, IPC_RMID, 0);

return 0;
```

Um exemplo da saída do programa:



Observações e conclusões;

Este trabalho demandou bastante luta nossa, pois passamos por sérios problemas com a manipulação e acesso da memória compartilhada, que ainda precisamos executar mais de uma vez o arquivo para que ele tenha toda sua eficácia.