UNIVERSIDADE
BEIRA INTERIOR

2º Relatório

PROJETO 2020

Pedro Carrasco, Afonso Simões, Pedro Lopes, Luís do Espírito Santo| Sistemas Operativos | 28/04/2020

Introdução

Neste relatório vamos mencionar o que foi e como foi implementado: gestão de memoria e um escalonador preemptivo no trabalho até a data.

Também se encontra implementado um gerador de 10000 pedidos de memória como foi indicado.

O nosso programa está a ler os programas através de um ficheiro que indica todos os nomes de ficheiros e os seus tempos a serem executados e, no fim da leitura, este reporta a informação dessa mesma execução no ecrã.

Existem algumas falhas na implementação de “threads” que podem levar a que exista uma probabilidade de gerar um “segmentation fault” (SIGSEV) que não foram resolvidas e a que não se encontrou solução.

A implementação de memória segue o princípio de “FIRST FIT” onde se vai tentar colocar sempre o programa por ordem de chegada.

Estruturas de dados “PROCESSO”

```
typedef struct processo{
    char *nome;
    int pid;
    int ppid;
    int prioridade;
    int tempoVida;
    int PC;
    int processValue;
    int quantidadeDeIntrucoes;
} processo;
```

Esta estrutura contém os elementos em baixo referidos:

- NOME que representa o nome do processo;
- PID que representa o identificador de processo gestor PID = 0;
- PPID que representa o pai da PID, se este for filho;
- PRIORIDADE que representa a prioridade de execução 1 > 2 > 3;
- TEMPOVIDA que representa o tempo a ser executado;
- PC que representa o número de instruções;
- PROCESSVALUE que representa o valor do processo;
- QUANTIDADEDEINTRUCOES que representa o número de instruções do programa.

Esta estrutura permite de forma simples guardar toda a informação que define um programa, no sentido em que se for preciso consultar alguma informação por exemplo o PID ou PPID de um programa apenas será preciso procurar pelo nome do programa à estrutura associada à informação.

Estruturas de dados “PROGRAMA”

```
typedef struct programa{
    char *nomeProg;
    processo infoProcesso;
    char ** listaDeIntrucoes;
    int estado;
}programa;
```

Esta estrutura contém os elementos em baixo referidos:

- NOMEPROG que contém o nome do programa;
- PROCESSO que contém a estrutura com informações do programa;
- LISTADEINTRUCOES que contém a lista com as operações do programa;
- ESTADO que contém todos os estados do programa RUNNING 0 / PARADO 1 / MORTO 2.

Desta forma fica fácil manipular quando um programa deve ou não ser executado, por exemplo se um programa de maior prioridade aparecer, será simplesmente necessário trocar o programa em execução sem haver perda da posição da execução do programa anterior.

Optou-se por usar um “array de strings” (“array de array de caracteres” em C) para a organização das instruções por este simbolizar o método mais fácil de implementar e provavelmente manipular no tipo de estruturas possíveis, como por exemplo filas, pilhas, árvores, listas que requerem uma maior capacidade de abstração na sua implementação.

Estruturas de dados “MEM_SPACE”

```
typedef struct mem_Space {
    int num;
    struct mem_Space *nextptr;
} LL_proc;
```

A estrutura de dados MEM_SPACE permite simular um bloco de memoria do tipo lista, isto é, criar uma estrutura que corresponde a um par (ID, TAMANHO) que correspondem ao PID e ao número de elementos alocados, e o NEXTPTR que corresponde à ligação ao próximo elemento da lista.

Desta forma foram criadas operações para criar a memória, alocar memória, de-alocar memória e contar o número de fragmentos que existem entre duas alocações diferentes.

A nossa interpretação e logica do projeto

Como o enunciado do projeto é ambíguo em relação a como este deve ser implementado e qual o caminho a se tomar, abaixo encontra-se explicada a metodologia que terá sido usada.

Na atual situação do projeto encontra-se disponível um sistema de escalonamento não preemptivo, ou seja, não flexível na execução de programas, isto é cada programa é lido e executado, ignorando a existência de outros programas que possam ou não ter maior prioridade que ele. Para que tal assim fosse implementado, através da função "PROGRAMARUNNERFIFO" é executada uma sequência de instruções, após a leitura na função "ATRIBUIDORDEINTRUCOES" e após ser unificada pela função "JUNTOR" definidas abaixo, é executada a função "PERCORRERINTRUCOES", o que leva a que seja impossível um processo impedir a execução de outro, visto que o segundo apenas será definido após a conclusão do primeiro.

```
void atribuidorDeInstrucoes(char *nomeFich,char **arrayFinalStrings, processo *processoAtual);
```

```
programa juntor(processo info,char ** listaDeIntrucoesInfo);
```

```
void percorrerIntrucoes(programa *progAPercorrer);
```

De forma a poder-se simular o FORK, no caso instrução C, utilizou-se uma função auxiliar "filho" de forma a que se possam evitar confusões na execução de instruções após o FORK, numa possível implementação esta função irá retornar o PID do filho para esta seja o mais próximo da implementação do FORK em C.

Para simular a instrução EXEC do C, temos o caso da instrução L que chama a função "atribuidor de Instruções".

Na atual situação do projeto encontra-se disponível um sistema de escalonamento preemptivo, ou seja, flexível na execução de programas. Cada programa é lido e executado podendo ou não ser interrompido tendo em conta a existência de outros programas que possam ou não ter maior prioridade que ele.

O pograma começa por ler o ficheiro com os programas através da função "PRIORITY" onde eles são lidos e guardados. Então a função PROGRAMARUNNERPRIORITY vai definir o comportamento para a implementação do algoritmo de escalonamento "PRIORITY" em que, após tê-lo feito, os mesmos vão chamar a função "PERCORRERINSTRUCOESPRIORITITY" que vai percorrer uma instrução apenas, visto que esta irá ser chamada até ao fim do programa, alternando entre si e a função "PROGRAMMARUNNERPRIORITY" para comparar o tempo percorrido e verificar se temos algum programa mais prioritário que irá tomar a vez do programa atual.

O "PERCORRERINSTRUCOESPRIORITITY" difere do "PROGRAMARUNNERFIFO" num só aspeto que é: percorre uma instrução de cada vez em vez de percorrer tudo de uma vez. Isto tem de ser feito pois a qualquer momento pode surgir um programa com maior prioridade.

```
//Precorre em SJF
void priority(char *listaDeProgramas);
```

```
void programaRunnerPriority(programa *listaDeProgramas,int counter);
```

```
//Executar em SJF
void percorrerIntrucoesPriority(programa *progAPercorrer);
```

```
//Precorre todo o programa Serve para o FIFO
void programaRunnerFifo(char *nomeDoPrograma);
```

Na segunda parte do nosso projeto acrescentámos também a função "PERCORRERINTRUCOESTHREAD" sendo o seu papel executar instruções em simultâneo, precisamos desta mesma para executar o pai e filho como acontece na instrução C. Esta função tem uma chance de gerar um SIGSEV devido ao facto que esta pode ser chamada recursivamente dentro de um “thread” levando à corrupção do segundo “thread”.

Na componente de memória criou-se uma lista de 100 unidades alocáveis aos programas que pode ser acedido através das funções "ALOCATE_MEM" que vai alocar o processo que lhe passámos com o seu devido tamanho no bloco de memoria, “DEALLOCATE_MEM” que vai libertar o espaço guardado pelo “ALOCATE_MEM” e a função “FRAGMENT_COUNT” responsável por contar as falhas de tamanho 1 ou 2 na memória.

```
//aloc
int alocate_mem (int process_id, int num_units,struct mem_Space *mem);
```

```
// contador de fragmentos
int fragment_count (struct mem_Space *mem);
```

```
//dealoc
int deallocate_mem(int process_id,struct mem_Space *mem);
```

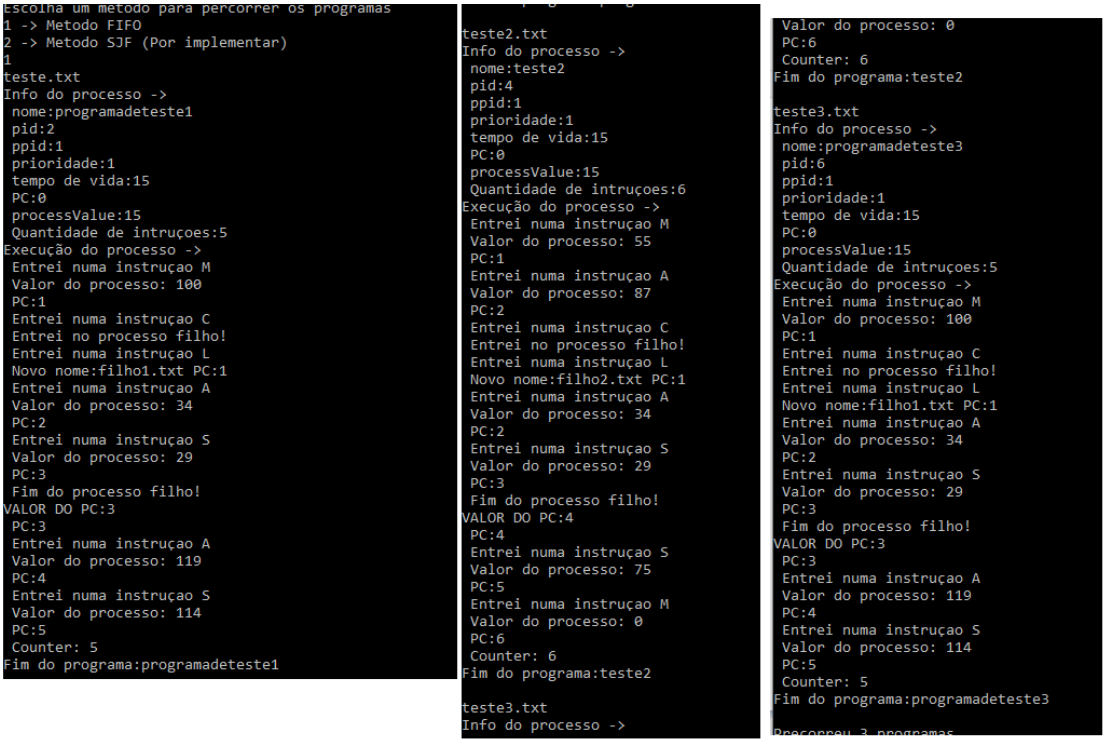
Para que fosse estudada a implementação da memória e das suas funções foi implementado um teste com uma memória de 1000 unidades para 10000 requisições de memória, onde cada requisição teria a oportunidade de pedir entre um a três espaços na memória, usando uma SEED constante. Obtiveram-se os seguintes resultados:

```
-----------------TABELA--------------------

Tamanho da memoria:1000
Sucessos a alocar memoria:4117  Falhas a alocar memoria:5883
Sucessos a dealocar memoria:434 Falhas a dealocar memoria:3683
Numero de fragmentos:27388

-----------------FIMTABELA-----------------
```

Execução do método de escalonamento FIFO (não preemptivo)



Execução do método de escalonamento PRIORITY (preemptivo)