UNIVERSIDADE DE ÉVORA

Sistemas Operativos I

Relatório

António Peixe – 34164

Évora, 03/06/2016

Índice

Lista de Figuras

Introdução

No âmbito da unidade curricular de Sistemas Operativos I inserida na licenciatura de Engenharia Informática, foi proposto a elaboração e desenvolvimento de um programa que simulasse uma arquitetura sobre o modelo de cinco estados que consome programas constituídos por operações definidas por dois dígitos.

Numa fase inicial, a criação do programa passou pela implementação de estruturas de dados capazes de realizar o pretendido. Desta forma, foram criadas as estruturas para os estados, para os processos e ainda uma estrutura que possibilita a gerência dos espaços livres em memória. Todos os estados são considerados *queues*, ou seja, têm protocolo FIFO (*First In First Out*), o mesmo acontece para o estado RUN embora esteja limitado apenas a um processo.

Relativamente à leitura do ficheiro de *input,* esta é realizada toda de uma vez para um BUFFER. É de notar que este BUFFER é também uma *queue* e, como tal, pode ser utilizada a mesma estrutura dos estados do escalonador.

Assim, para além do ficheiro Makefile que a facilita a compilação e execução, o programa contém ainda cinco ficheiros de código. A divisão destes ficheiros está feita de acordo com as estruturas utilizadas de modo a ter um programa mais modular. O ficheiro Makefile tem as seguintes regras:

- make all – compila o trabalho
- make run – corre o trabalho
- make clean – remove todos os ficheiros .o e executáveis

Por fim, é importante ter em consideração que algumas das funções implementadas prejudicam a performance do programa, no entanto, são necessárias para que exista um *output* legível. Estas poderiam ser removidas (ou não chamadas) pois a sua execução não interfere com o normal procedimento de um escalonador.

Ficheiros e Estruturas de Dados

I – estados_pcb.h / c

Nestes ficheiros encontram-se as estruturas de dados e funções necessárias para implementar os estados do escalonador, bem como os processos. Deste modo, as funções existentes são apenas funções de *get* e *set* de variáveis ou construtores.

Quando um processo chega ao estado NEW é criado o seu PCB. Desta forma, é necessária uma estrutura para guardar esta informação. Sendo que as suas variáveis simbolizam, respetivamente, o identificador do processo, o *program counter* (que aponta para a próxima instrução a executar) e os índices iniciais e finais da memória física onde se encontra guardada a informação do processo.

```
7  typedef struct pcb {
8      int id, pc;
9      int mem_init, mem_end;
10 } pcb;
11
```

Figura 1 - Estrutura PCB

Como referido anteriormente, os estados são *queues* e, como tal, os processos vão estar em nós (elementos) dessas mesmas *queues*. Assim, teremos:

```
12 typedef struct node {
13     pcb *process;
14     int entrada;
15     int *inst;
16     int n_inst;
17     int block_counter;
18     struct node *next;
19 } node;
20
```

Figura 3 - Estrutura NODE

```
23 typedef struct estado {
24     node *head;
25     node *tail;
26     int size;
27 } estado;
28
```

Figura 2 - Estrutura ESTADO

Um NODE inclui além de um apontador para um PCB, inúmeras variáveis extra que permitem controlar os processos nos diferentes estados, dito isto, temos:

- **entrada –** esta variável tem duas funções. Quando é criado o BUFFER serve para guardar o tempo de chegada, ou seja, quando o processo passa para o estado NEW. Em qualquer um dos restantes estados vai tomar o valor 1 ou 0 que significam, respetivamente, que o processo já transitou de estado no ciclo atual ou que ainda pode avançar.

- **inst –** apenas usado enquanto os processos se encontram no BUFFER. É um apontador para um *array* que contém as instruções do processo, este será desalocado quando o processo for para o NEW e as instruções guardadas na memória. Nos restantes estados toma o valor NULL.

- **n_inst –** apenas usado enquanto os processos se encontram no BUFFER. Necessário para saber o tamanho do *array,* apontado pela variável anterior. Sendo que toma o valor -1 nos restantes estados.

- **block_counter –** apenas usado enquanto os processos se encontram no BLOCK. Permite contabilizar o número de ciclos que o processo ainda tem que fazer no estado. Toma o valor -1 em todos os restantes estados.

- **next –** um apontador para o próximo processo do estado, visto ser uma *queue* esta é uma variável indispensável.

II – espaços_mem.h / c

Nestes ficheiros temos as estruturas de dados e funções, necessárias para implementar uma forma que permita gerir o espaço livre na memória. Tendo em conta que a memória pode ficar muito fragmentada e, posteriormente, retomar um estado com espaços mais compactos, apresentando-se bastante flutuante a abordagem utilizada para guardar estes fragmentos, foi uma lista ligada.

```
 7  typedef struct fragment {
 8    int inicio;
 9    int fim;
10    int tamanho;
11    int delete_flag;
12    struct fragment *next;
13  } fragment;
14
```

Figura 5 - Estrutura FRAGMENT

```
15  typedef struct memSpace {
16    fragment *head;
17    fragment *tail;
18    int size;
19  } memSpace;
20
```

Figura 4 - Estrutura MEMSPACE

A estrutura FRAGMENT não é mais do que um nó da lista ligada MEMSPACE e as suas variáveis indicam, respetivamente, o início e fim do fragmento na memória, o seu tamanho, o *delete_flag* que toma valores 1 ou 0 caso o fragmento deva ou não ser removido da estrutura e ainda o apontador para o próximo FRAGMENT.

As seguintes funções são usadas para retornar, respetivamente, o melhor fragmento para um processo, o que mais se ajusta ao tamanho e o pior fragmento, ou seja, o que deixa mais espaço livre.

```
26  fragment *memSpace_getBest(memSpace *ms, int process_size);
27  fragment *memSpace_getWorst(memSpace *ms, int process_size);
```

Figura 6 - Funções para obteção de fragmentos

Estas funções não removem o fragmento da lista ligada, este é retornado e depois de usado atualizado. Esta atualização passa por modificar a variável de início e recalcular o tamanho.

Quando se insere um fragmento há a possibilidade deste ser contíguo com um fragmento já existente. Assim, sempre que existe uma inserção é necessário correr a lista ligada para confirmar esta situação. Este processo funciona do seguinte modo:

- Marcar um nó como atual;
- [Ciclo] olhar para os seguintes;
  - [se] for contíguo, modificar o nó atual, ativar *delete_flag* deste nó e reiniciar ciclo;

- [se] chegou ao fim dos seguintes;
    - [se] atual não for o último avançar o atual e recomeçar ciclo;

No final desta função é então necessário chamar uma função para limpar todos os fragmentos que estejam marcados para serem removidos.

III – 34164main.c

Ficheiro que contém a função main(). No qual estão também presentes as seguintes funções:

```
22  // assinatura de funcoes existentes
23  // funcao para ler o ficheiro para um bufer
24  void read_input(estado *buffer);
25
26  // funcao para imprimir a memoria bem como os 5 estados do escalonador
27  void print_output(int *mem, int n_pcb, estado *new, estado *ready, estado *run,
28      estado *block, estado *end);
29
30  // funcoes para manipulacao da memoria "fisica"
31  void memory_clean(int *mem);
32  void memory_clean_portion(int *mem, int ini, int end);
33  void memory_insert(int *mem, int *inst, int quant, int ini);
34  void memory_print(int *mem, int n_pcb, estado *new, estado *rea, estado *run, estado *bl);
35
36  // funcoes de transferencia de processoes de um estado para outro
37  void trans_buffer_new(estado *buf, estado *new, int *mem, memSpace *spc, int contPCB);
38  void trans_new_ready(estado *new, estado *rea);
39  void trans_ready_run(estado *rea, estado *run);
40  void trans_run_ready(estado *run, estado *rea);
41  void trans_run_blocked(estado *run, estado *bl);
42  void trans_run_end(estado *run, estado *e, int *mem, memSpace *spc);
43  void trans_blocked_ready(estado *bl, estado *rea);
44
45  // funcao para executar a instrucao - retorna a instrucao executada
46  int executar_inst(int *mem, pcb *process);
47
48  // funcao para executar o fork de um processo
49  void fork_process(int *mem, pcb *process, memSpace *spc, estado *new, int contPCB);
50
```

Figura 7 - Funções existentes em 34164main.c

Linha 24 – Esta função recebe o "estado" BUFFER, irá abrir o ficheiro de *input* e colocar cada processo no BUFFER.

Linha 27 – Recebe o apontador para *array* usado como memória, a quantidade de processos e todos os estados. Tem como objetivo fazer o *print* para a consola.

Linhas 31 a 34 – Funções para alterar a memória.

Linha 31 – Limpa toda a memória.

Linha 32 – Limpa apenas uma zona da memória, com início em INI e fim em END. Usado quando um processo termina.

Linha 33 – Insere na memória a partir de INI toda a informação que está no *array* INST. Usado quando um processo entra no estado NEW.

Linha 34 – Esta função imprime a memória. De modo a ter uma apresentação legível esta função corre todos os estados guardando o início dos processos existentes para poder

apresentar uma marca. Esta função prejudica a performance do programa mas sendo apenas de apresentação da informação poderia não ser chamada.

Linhas 37 a 43 – Funções para fazer a transferência entre estados de processos.

Linha 37 – Para passar do BUFFER para o estado NEW esta função procura um fragmento de acordo com o modo do gestor de memória, atualiza o fragmento usado, copia para a memória todas as instruções e por fim muda o processo de estado.

Linha 46 – Esta função executa todas as operações matemáticas e incrementa o *program counter*, isto é, as operações 0, 1, 2, 3, 4, 5 e 8. Nas restantes não faz nada. No final a função retorna a instrução.

Linha 49 – Realiza o FORK de um processo. Começa por procurar um fragmento de memória para colocar o processo clone, atualiza o fragmento. Coloca o novo processo no estado NEW e copia as variáveis e instruções restantes do processo original para a posição de memória do processo clonado.

Usando estas funções e as estruturas definidas torna-se bastante simples criar a função main(). Esta segue a seguinte abordagem:

- Criar todos os estados, ler o *input* e criar BUFFER e criar o *array* para a memória;
- Iniciar um ciclo que corre enquanto houver processos nos estados BUFFER, NEW, READY, RUN ou BLOCK;
    - Verificar se há algum processo no BUFFER que deva passar para NEW e executar;
    - Verificar se há algum processo em BLOCK que deva sair e passa-lo para READY;
    - Verificar se há processos em NEW que possam passar para READY e executar;
    - Se houver um processo em READY que possa passar para RUN transferir;
    - Decrementar o contador de todos os processos que estão em BLOCK;
    - Havendo um processo em RUN, executar instrução, incrementar quantum;
        - Se quantum == limite quantum, *reset* quantum e passar processo para READY;
    - Fazer print se for esse o caso;
    - Limpar a variável de ENTRADA de todos os NODES de todos os ESTADOS, para permitir que os processos possam transitar no próximo ciclo.
    - Incrementar contador de tempo geral.

Conclusão

A realização deste trabalho permitiu complementar a matéria lecionada durante o semestre, na medida em que, tornou mais percetível a forma de execução de um processo numa arquitetura de cinco estados.

Para desenvolver este trabalho foi necessário utilizar a linguagem C, o que permitiu ter uma primeira interação com esta e perceber quais as suas vantagens e desvantagens. Uma vez que, esta linguagem não é na prática ensinada, a sua utilização inicial torna-se complexa, no entanto, e depois de perceber o funcionamento dos apontadores não senti grandes dificuldades na sua aplicação, passando mesmo achar que tem bastantes vantagens.

Quanto às estruturas criadas, concluo que são a forma mais simples de representar fidedignamente o pretendido. Isto porque a complexidade das operações nos estados é bastante reduzida, pois inserções e remoções são feitas diretamente. Pelo contrário, na estrutura que gere a memória as funções que verificam se há fragmentos contíguos têm uma complexidade mais elevada, no entanto, o varrimento é sempre necessário e a estrutura adequa-se devido à irregularidade da quantidade de fragmentos ao longo da execução do programa.