Sistemas Operativos

Licenciatura em Engenharia Informática

Trabalho Prático 2

Simulador Gestor de Memória

e

Simulador de Escalonamento e Memória

UNIVERSIDADE DE ÉVORA

Miguel Pombeiro, 57829 | Miguel Rocha, 58501

Departamento de Informática
Universidade de Évora
Junho 2025

Índice

1 Introdução **2**

2 Estrutura do Programa **2**

2.1 Parte 1: Simulador Gestor de Memória 2

2.1.1 Estruturas 2

2.1.2 Estados 3

2.1.3 Funções 3

2.1.4 Execução 3

2.2 Parte 2: Simulador de Escalonamento e Memória 4

2.2.1 Estruturas 4

2.2.2 Estados 5

2.2.3 Funções 5

2.2.4 Execução 6

3 Comentários **7**

1 Introdução

Este relatório descreve a implementação, em linguagem C, de dois programas, um que simula a gestão de memória com paginação e memória virtual e outro que simula o modelo de cinco estados do primeiro trabalho, com gestão de memória.

O primeiro simulador de memória implementa dois algoritmos de substituição de páginas, sendo estes, *First-in-first-out* (FIFO) e *Least Recently Used* (LRU). O algoritmo FIFO, utiliza uma fila para guardar as páginas e, quando é necessário substituí-las, a que está no topo da fila é a primeira a sair. Já no algoritmo LRU, é utilizado um sistema com tempos de acesso, e quando é necessário substituir uma página, a página com o tempo de acesso mais antigo é então substituída. Este simulador recebe como *input* dois *arrays*, um que contém o espaço de endereçamento de cada programa, e outro que contém pares com o identificador do programa e endereço de memória a aceder.

O segundo simulador implementa um escalonador com algoritmo *Round Robin*, com um *time quantum* de 3 instantes, bem como um sistema de gestão de memória virtual com paginação com LRU. Esta simulação corre no máximo 100 instantes de tempo, guardando tanto o estado de cada processo ativo bem como de cada *frame* de memória por cada instante, de forma a seguir o seu desenvolvimento. O simulador recebe como *input* uma matriz de valores inteiros, onde a primeira linha contém o espaço de endereçamento de cada programa e as restantes linhas representam instruções, também organizadas em diferentes programas.

2 Estrutura do Programa

2.1 Parte 1: Simulador Gestor de Memória

2.1.1 Estruturas

As *frames* de memória são representadas pela seguinte estrutura:

- **processID**: O ID do processo ao qual a *Frame* está associada;
- **pageNumber**: Número da página do processo que está associado à *frame*.
- **timestamp**: Último instante de tempo em que a *frame* foi acedida.

Cada processo é representado por uma estrutura que inclui:

- **processID**: Identificador único, atribuído incrementalmente a partir de 1;
- **addressSpace**: Espaço de endereçamento do programa associado a este processo;
- **pageTable**: *Array* de inteiros que mapeia cada uma das páginas do processo ao índice de uma *frame* de memória. Se a página não se encontra, atualmente, em memória, irá conter o valor -1;
- **maxPages**: Número máximo de páginas que o espaço de endereçamento do processo pode conter;
- **state**: O estado atual do processo.

2.1.2 Estados

Como se trata de uma implementação simplificada, foram apenas considerados 3 estados diferentes, representados por um inteiro de 0-2, sendo estes respetivamente:

- **ACTIVE**, o processo está ativo e vai executar acessos à memória;
- **SEGFAULT**, ocorreu um *Segmentation Fault* e no próximo instante o processo vai deixar de estar ativo;
- **INACTIVE**, o processo já não está ativo;

2.1.3 Funções

De forma a simular os dois algoritmos de substituição utilizados para selecionar as páginas a substituir em memória - FIFO e LRU -, foram implementadas várias funções.

Ao nível das *frames* de memória, foram implementas duas funções principais:

- `findUpdateEmptyFrame`, que irá procurar por uma *frame* vazia e, caso encontre, irá associá-la a uma página de um processo.
- `updateFrameRemove`, que irá dissociar uma *frame* já preenchida com uma página de um processo, e associá-la a uma nova página.

Para a simulação da substituição por FIFO, foi implementada a função `simulateFIFO`, que faz a simulação deste algoritmo de substituição. Este algoritmo recorre a uma fila para selecionar qual a *frame* a substituir, não sendo necessárias novas implementações para esta estrutura de dados.

Para a simulação da substituição por LRU, para além da função `simulateLRU`, que faz a simulação deste algoritmo de substituição, foi necessário implementar a forma de selecionar das *frames* a substituir. Assim, foi implementada a função `removeFrameLRU`, que permite selecionar a *frame* em memória que foi acedida à mais tempo, para que a página nela contida possa ser substituída.

2.1.4 Execução

O programa começa por ler um *input* a partir dos argumentos da linha de comandos, que indica qual dos *inputs* do ficheiro de testes fornecido deve ser utilizado. Após escolhido, são inicializados todos os processos com os respetivos espaços de endereçamento, bem como todas as sete *frames* de memória disponíveis.

Ambas as simulações irão correr até que sejam esgotados todos os acessos em memória representados no *array* do *input*. As simulações foram implementas de forma semelhante, nomeadamente, ao nível das verificações necessárias, sendo que apenas as partes que dizem respeito ao algoritmo de substituição utilizado variam. Assim, durante a simulação, e antes de ser executado o acesso à memória, é verificado se o processo em questão está ativo, ou seja, se nenhum acesso anterior provocou um *segmentation fault*. Caso esta situação não se verifique, há ainda, a necessidade de verificar se o endereço a aceder se encontra dentro do espaço de endereçamento do processo.

Estando cumpridos todos os requisitos para um acesso seguro e, caso a página que corresponde a esse endereço já se encontre em memória, é feito o acesso. Caso contrário, é necessário carregar essa página para memória, quer seja numa *frame* vazia, quer seja pela substituição de uma página usando o algoritmo de substituição.

Por fim, quando todos os acessos à memória dados no *array* de execução forem simulados por ambos os algoritmos, tanto as *frames* como os processos e respetivas estruturas são destruídos.

2.2 Parte 2: Simulador de Escalonamento e Memória

2.2.1 Estruturas

As *Frames* de memória são representadas pela seguinte estrutura:

- **processID**: O ID do processo ao qual a *frame* está associada;
- **pageNumber**: Número da página do processo que está associado à *frame*.
- **timestamp**: Último instante de tempo em que a *frame* foi acedida.

Cada processo é representado por uma estrutura que inclui:

- **id**: Identificador único, atribuído incrementalmente a partir de 1;
- **state**: O estado atual do processo;
- **currentInstruction**: índice da instrução atual;
- **instructions**: *Array* de instruções do programa a que corresponde o processo;
- **nInstructions**: Número de instruções do programa;
- **time**: Contador de instantes, para gerir a permanência nos estados `NEW`, `RUN`, `BLOCKED` e `EXIT`;
- **addressSpace**: Espaço de endereçamento do programa associado a este;
- **pageTable**: *Array* de inteiros que mapeia cada uma das páginas do processo ao índice de uma *frame* de memória. Se a página não se encontra, atualmente, em memória, irá conter o valor -1;
- **maxPages**: Número máximo de páginas que o espaço de endereçamento do processo pode conter;

O escalonador (*scheduler*) é representado pela seguinte estrutura:

- **Lista NEW**: Representa o estado `NEW`. Quando os processos são criados devem estar nesta lista dois instantes de CPU;
- **Fila READY**: Representa o estado `READY`. Contém os processos que estão à espera para serem executados;
- **runningProcess**: Processo que está no estado `RUN`;
- **Lista BLOCKED**: Representa o estado `BLOCKED`. O tempo de permanência de processo nesta lista corresponde ao módulo do valor da instrução I/O executada;
- **Lista EXIT**: Lista na qual os processos passam três instantes de CPU antes de serem terminados.

- **numProcesses**: Número de processos atuais do *scheduler*;
- **processes**: *Array* que contém todos os processos;
- **instructions**: Matriz que contém todas as instruções por programa, dados no *input*;
- **nInstructionsProgram**: Número de instruções por programa;
- **nPrograms**: Número de programas dados no *input*;
- **nActiveProcesses**: Número de processos ativos no escalonador. Quando este valor chega a 0, a simulação termina;
- **instantTime**: Instante atual do simulador, usado para aplicar o algoritmo de substituição LRU. Quando este valor chega a 100, a simulação também termina;
- **programsAddressSpace**: *Array* que contém todos os espaços de endereçamentos de cada programa;
- **framesArray**: *Array* de *Frames* (a estrutura *Frame* está definida na acima). Este *array* representa o estado atual de todas as *frames* do simulador.

2.2.2 Estados

Foram implementados nove estados diferentes, onde os estados NEW - TERMINATED foram representados como um inteiro de 0-5 e os estados SIGILL - SIGEOF foram representados por um inteiro de 100-102, respetivamente:

- **NEW**, corresponde ao estado com o mesmo nome no modelo de cinco estados;
- **READY**, corresponde ao estado com o mesmo nome no modelo de cinco estados;
- **RUN**, corresponde ao estado com o mesmo nome no modelo de cinco estados;
- **BLOCKED**, corresponde ao estado com o mesmo nome no modelo de cinco estados;
- **EXIT**, corresponde ao estado com o mesmo nome no modelo de cinco estados;
- **TERMINATED**, o processo fica neste estado depois de sair do estado EXIT;
- **SIGILL**, acontece quando o processo tenta aceder a instruções fora do programa;
- **SIGSEGV**, acontece quando existe uma tentativa de aceder a um endereço de memória que não está reservado ao processo;
- **SIGEOF**, acontece quando o programa não contêm a instrução `HALT` e tenta executar uma instrução depois da última.

2.2.3 Funções

Tendo o simulador sido dividido em três componentes principais, foram implementadas funções que as permitem manipular.

Ao nível do escalonador, as funções podem ser divididas em dois tipos principais diferentes:

- Funções que permitem manipular os estados dos processos em execução e alterar a estrutura de dados do escalonador, em que estão representados. Os nomes destas

funções são representativos das várias operações do modelo de cinco estados, *e.g.* `schedulerNew`, `schedulerBlock`, `schedulerRelease`.
- Funções que permitem decrementar, para todos os processos num determinado estado, o tempo que estes ainda lá devem permanecer, *e.g.* `timeDecrementRunning`, `timeDecrementNew`.

Ao nível dos processos, podem ser destacadas duas funções principais:

- `fetchNextInstruction`, que permite ler a próxima instrução a executar.
- `changeCurrentInstruction`, que permite alterar o *program counter* do processo (neste caso, o índice da instrução atual no vetor de instruções), para um valor válido, de forma a implementar as instruções do tipo `JUMP`.

Ao nível de memória, podem ser destacadas três funções principais:

- `findEmptyFrame`, que irá procurar por uma *frame* vazia.
- `updateFrameRemove`, que irá dissociar uma *frame* já preenchida com uma página de um processo, e associá-la a uma nova página.
- `LRUPageReplacement`, que faz a simulação do algoritmo de substituição por LRU.

Foram ainda implementadas duas funções auxiliares para tratar as instruções de memória, `LOAD/STORE` e `SWAP/MEMCPY`, que fazem as verificações e chamam as funções necessárias para o bom funcionamento das mesmas.

Por fim, foram implementadas algumas funções que simulam a interpretação e execução das instruções dos programas, de acordo com as orientações do enunciado, respetivamente `getInstructionType` e `executeInstruction`.

2.2.4 Execução

O programa começa por ler um *input* a partir dos argumentos da linha de comandos (`argv[1]`), que indica qual das matrizes de programas, presentes no ficheiro de *inputs* fornecido, deverá ser utilizada. Após escolhida a matriz de *input*, é separada num *array* com os espaços de endereçamento de cada programa e numa matriz com as instruções de cada programa. Esta última foi processada de modo a que cada programa fique representado numa linha (e não numa coluna) da matriz, de forma a tornar os posteriores acessos mais rápidos.

Cada tipo de instrução tem uma função diferente, podendo esta fazer um salto de `n` instruções para trás (`JUMPB`), fazer um salto de `n` instruções para a frente (`JUMPF`), bloquear um processo por `n` instantes (`IO`), lançar um novo processo que executa um certo programa (`EXEC`) ou até terminar a execução do processo (`HALT`). Também podem ser instruções de acesso à memória, como a instrução (`LOAD/STORE`) que tenta aceder a um endereço de memória, ou até a instrução (`SWAP/MEMCPY`) que pode aceder a dois endereços de memória.

Antes de se iniciar a simulação da execução, é inicializado o *scheduler* e todas as estruturas de dados a ele associadas. Esta simulação irá correr no máximo 100 instantes.

Por cada instante de tempo, são realizados os seguintes passos:

- ***Dispatch*** - Se não existe nenhum programa no estado `RUN` e a fila `READY` não está vazia, então um processo é movido do topo da fila e começa a correr.
- **Execução da Instrução** - Se um processo está a correr, a próxima instrução é lida do vetor de instruções e executada de acordo com o seu tipo (*e.g.* `JUMP`, `EXEC`).
    - Se a instrução implica um acesso à memória, `SWAP/MEMCPY` ou `LOAD/STORE`, é necessário verificar se o acesso é válido. Caso a página a aceder já exista em memória, é feito um acesso. Caso a página ainda não exista em memória, é necessário carregá-la para uma *frame* vazia, ou fazer substituição por LRU se todas as *frames* estiverem ocupadas.
- **Decrementar os tempos** - Uma vez que os processos devem passar um número de instantes especifico no estado em que se encontram, com a exceção do estado `READY`, então, após a execução da instrução, todos os processos nos estados `NEW`, `RUN`, `BLOCKED` e `EXIT` têm os seus tempos de permanência decrementados de um instante. Quando o tempo de um processo chega a 0, este passa para o respetivo próximo estado, de acordo com o modelo de cinco estados.
    - No caso de processos que passam a estados de erro durante o estado `RUN`, estes são terminados e passam para o estado `EXIT`.
- **Impressão do estado** - Todos os processos em execução têm o seu estado impresso para o *output*.

A simulação da execução do programa é realizada pela função `simulate`, que começa pela atribuição do programa 1 ao primeiro processo, no instante 1. Nessa atribuição é feita a entrada desse processo na lista `NEW`, onde deverá passar dois instantes. Depois de passar dois instantes, é então possível a leitura das instruções. Essas instruções são executadas de acordo com o seu tipo, durante um instante.

A simulação termina quando transcorridos o número máximo de 100 instantes ou quando todos os processos terminam, sendo tanto o *scheduler* e respetivas estruturas de dados, bem como a lista de programas gerada, destruídos.

3 Comentários

O programa deve ser executado dentro da diretoria de cada parte, de acordo com o `README` que acompanha o código fonte, nomeadamente através do `makefile` fornecido, de forma a que os ficheiros de *output* tenham a denominação e o formato pedidos.