Sistemas Operativos

Licenciatura em Engenharia Informática

2º Trabalho

Parte 1: Simulador Gestor de Memória
Parte 2: Simulador de escalonamento e memória

André Gonçalves, 58392 | André Zhan, 58762 | Gonçalo Carvalho, 51817

Departamento de Informática
Universidade de Évora
26 de Junho de 2025

Conteúdo

1 Introdução

Este relatório descreve a implementação das duas partes solicitadas no enunciado. A primeira parte consiste na simulação da gestão de memória, incluindo a implementação dos algoritmos de substituição FIFO (*First-In-First-Out*) e LRU (*Least Recently Used*). Na segunda parte, procedeu-se à integração do simulador de escalonamento desenvolvido no primeiro trabalho de Sistemas Operativos com o gestor de memória implementado.

2 Parte 1

2.1 Biblioteca `memoria.h`

Foi criada uma biblioteca para gerir a memória principal, usando a seguinte estrutura:

```
typedef struct {
    int id;
    int ocupada;
    int proc_id;
    int pag_num;
    int timestamp;
} Frame;
```

Cada *frame* tem um ID, indicador de ocupação, ID do processo, número da página e *timestamp* para LRU.

Funções implementadas:

- `inicializar_memoria`: Inicializa todas as frames como livres.
- `carregar_pagina`: Carrega uma página na primeira *frame* livre (por ordem crescente). Retorna o *frame* em caso de sucesso caso contrário -1, porque a memória está cheia.
- `libertar_frames_processo`: Liberta todas as *frames* associadas a um determinado processo.
- `pagina_esta_na_memoria`: Verifica se a página está na memória, atualizando *timestamp* se LRU.
- `substituir_pagina_fifo`: Substitui a página segundo o algoritmo FIFO. Remove da fila o primeiro *frame* e substitui o seu processo pelo atual e respetivo nº da página acedida.
- `substituir_pagina_lru`: Substitui a página menos usada recentemente (LRU). Encontra o *frame* com o menor *timestamp*.

- `substituir_pagina_lru_excluindo`: Tem a mesma finalidade que a função anterior, mas com a opção de ignorar uma página. Esta função foi usada apenas na Parte 2.

2.2 Execução do simulador do Gestor de Memória

Foi implementada a função `imprimir_estado`, responsável por apresentar o estado da memória e dos processos num dado instante. Para cada processo, são exibidas as *frames* ocupadas, ordenadas pelo número da página.

A função `main` gere a execução do simulador, determinando através do argumento de entrada se o algoritmo de substituição utilizado será FIFO ou LRU. O ciclo principal de execução percorre todos os pares de ID do processo e endereço de memória definidos no array de execução.

A cada iteração, é avaliado o par (ID do processo, endereço) a ser acedido. Quando o endereço excede o espaço de endereçamento atribuído ao processo, ocorre um *Segmentation Fault*, resultando no término do processo e libertação das suas *frames*:

```
if (endereco >= limite) {
    terminado[proc_id] = instante;
    libertar_frames_processo(memoria, proc_id);
    imprimir_estado(instante++, memoria, terminado, num_proc);
    continue;
}
```

Caso o endereço seja válido, calcula-se o número da página correspondente:

```
int pag_num = endereco / TAM_FRAME;
```

Se a página não se encontrar carregada em memória, procede-se ao seu carregamento:

```
if (!pagina_esta_na_memoria(memoria, proc_id, pag_num, instante,
   usar_lru)) {
    int frame = carregar_pagina(memoria, proc_id, pag_num,
       instante);

    if (frame != -1) {
        if (usar_fifo) {
            int *frame_id = malloc(sizeof(int));
            *frame_id = frame;
            enqueue(fifo_queue, frame_id);
        }
    } else {
        if (usar_fifo) {
            substituir_pagina_fifo(memoria, proc_id, pag_num,
               fifo_queue);
```

```
13          } else if (usar_lru) {
14              substituir_pagina_lru(memoria, proc_id, pag_num,
                    instante);
15          }
16      }
17  }
```

Caso não existam *frames* livres, o simulador verifica o algoritmo de substituição selecionado e procede à respetiva substituição de página.

3 Parte 2

Nesta segunda parte, procedeu-se à integração do simulador de escalonamento desenvolvido no primeiro trabalho de Sistemas Operativos com o gestor de memória implementado na parte 1, acrescentando diversas funcionalidades ao simulador.

3.1 Estados adicionais

Foram adicionados três novos estados: SIGSEGV, SIGILL e SIGEOF.

3.2 Estrutura dos processos

Foi adicionado o campo `address_space` à estrutura dos processos, responsável por armazenar o espaço de endereçamento atribuído a cada processo.

3.3 Funções de apoio

- `listar_frames_processo`: lista, no ficheiro de saída, as *frames* ocupadas por um processo, ordenadas pelo número da página.
- `print_output`: imprime o estado de todos os processos e as *frames* associadas a cada um, num dado instante.

3.4 State SIGEOF

Sempre que um processo tentar aceder a instruções para além do final do programa sem `HALT`, será atribuído o estado `SIGEOF` e o processo será terminado.

```
1  if (runningProcess->pc >= runningProcess->program_length) {
2      runningProcess->state = STATE_SIGEOF;
3      runningProcess->exit_time = EXIT_TIME;
4      runningProcess = NULL;
5  }
```

3.5 Instruções JUMPB e JUMPF

A instrução `JUMP` foi dividida em duas variantes:

- **JUMPB (Jump Back)**: faz com que o `Program Counter` recue `n` instruções (sendo `n` os dois dígitos menos significativos da instrução). Se o `Program Counter` ficar negativo, é gerado o erro `SIGILL` e o processo termina.

```
else if (instr >= 101 && instr <= 199) {
    int jump = instr % 100;
    runningProcess->pc -= jump;
    if (runningProcess->pc < 0) {
        runningProcess->state = STATE_SIGILL;
        runningProcess->exit_time = EXIT_TIME;
        runningProcess = NULL;
    }
}
```

- **JUMPF (Jump Front)**: faz com que o `Program Counter` avance `n` instruções (sendo `n` os dois dígitos menos significativos da instrução). Se o `Program Counter` ficar fora dos limites do programa, é gerado o erro `SIGILL` e o processo termina.

```
else if (instr >= 1 && instr <= 100) {
    int jump = instr % 100;
    runningProcess->pc += jump;
    if (runningProcess->pc < 0 || runningProcess->pc >=
       runningProcess->program_length) {
        runningProcess->state = STATE_SIGILL;
        runningProcess->exit_time = EXIT_TIME;
        runningProcess = NULL;
    }
}
```

3.6 Instruções LOAD/STORE

Foi adicionada a instrução `LOAD/STORE` (intervalo de 1000 a 15999), que indica o acesso a um endereço de memória calculado como:

$$\text{endereço} = \text{instrução} - 1000$$

Segue a lógica de paginação implementada na Parte 1, em que:

- Se o endereço for superior ao `address_space` do processo, é gerado um `SIGSEGV` e o processo termina.

- Caso contrário, é verificado se a página está em memória. Se não estiver, é carregada ou substituída utilizando o algoritmo LRU, conforme necessário.

```
else if (instr >= 1000 && instr <= 15999) {
    int endereco = instr - 1000;

    if (endereco >= runningProcess->address_space) {
        runningProcess->state = STATE_SIGSEGV;
        runningProcess->exit_time = EXIT_TIME;
        runningProcess = NULL;
    } else {
        int pagina = endereco / TAM_FRAME;
        if (!pagina_esta_na_memoria(memoria, runningProcess->id,
           pagina, t, 1)) {
            int frame = carregar_pagina(memoria,
               runningProcess->id, pagina, t);
            if (frame == -1) {
                substituir_pagina_lru(memoria,
                   runningProcess->id, pagina, t);
            }
        }
        runningProcess->pc++;
    }
}
```

3.7 Instruções SWAP/MEMCPY

Foi adicionada a instrução `SWAP/MEMCPY` (intervalo de 1 000 000 000 a 2 109 999 999), que indica o acesso a dois endereços de memória, de forma simultânea, seguindo as regras definidas:

- O **primeiro endereço** é determinado pelos cinco dígitos mais significativos (posições 1 a 5) da instrução, subtraindo-se 10 000 ao valor obtido.
- O **segundo endereço** é dado pelos cinco dígitos seguintes (posições 6 a 10) da instrução.

```
int primeiro_end = (instr / 100000) % 100000 - 10000;
int segundo_end = instr % 100000;
```

Em caso de sucesso, não há *Segmentation Fault* (SIGSEGV), e calcula as respetivas páginas (pag1 para o endereço1, pag2 para o endereço2).

- Se `pag1` e `pag2` são diferentes:
  - Se ambas estão em memória, atualizam-se os *timestamps* usando `pagina_esta_na_memoria`.
  - Se nenhuma está em memória, tenta-se `carregar_pagina`, e em caso de falha usa-se `substituir_pagina_lru`.
  - Se apenas uma está em memória:
    * Se `pag1` estiver em memória, atualiza-se o seu *timestamp*, tenta-se carregar `pag2` e, se necessário, usa-se `substituir_pagina_lru_excluindo` ignorando `pag1`.
    * Se `pag2` estiver em memória, tenta-se carregar `pag1`, usando `substituir_pagina_lru_excluindo` ignorando `pag2` se necessário, e atualiza-se o *timestamp* de `pag2`.
- Se `pag1` e `pag2` são iguais:
  - Apenas `pag1` é consultada ou carregada caso não esteja em memória, evitando processamento desnecessário.

3.8 Melhoria

Por fim, conforme solicitado no enunciado e indicado como melhoria ao Trabalho 1 durante a apresentação, o simulador passa a terminar automaticamente caso todos os processos concluam a sua execução antes de serem atingidos os 100 instantes de tempo.

```
int todos_terminados = 1;
for (int i = 0; i < totalProcesses; i++) {
    if (processList[i] != NULL && !processList[i]->terminated) {
        todos_terminados = 0;
        break;
    }
}
if (todos_terminados) break;
```