Universidade Federal de Lavras
PPGCC
PCC508 – Sistemas Operacionais

Tópico 5 Lista Avaliativa

Douglas Aquino T. Mendes
3 de janeiro de 2025

Sumário

1 Introdução

Este documento tem como objetivo apresentar o desenvolvimento das atividades avaliativas para o tópico 5 da disciplina de Sistemas Operacionais, focando na implementação de códigos em linguagem C. Serão apresentadas as questões, a resolução, os códigos desenvolvidos, seguidos de uma explicação sobre sua lógica de funcionamento.

2 Questões

2.1 Algoritmo de substituição de páginas ótimo

Pergunta: 1) Poderíamos criar uma ferramenta para analisar cada um dos programas para prever quando cada página seria utilizada, de modo a implementarmos o algoritmo de substituição de páginas ótimo? Explique.

Resposta: Tanenbaum e Bos [1, p. 144] dizem que a criação de uma ferramenta para analisar programas e prever o uso futuro de páginas, com o objetivo de implementar o algoritmo ótimo de substituição de páginas, é teoricamente possível, porém impraticável em sistemas reais. O algoritmo ótimo de substituição de página postula que, em uma falta de página, a página que levará mais tempo para ser referenciada novamente deve ser removida da memória. Este algoritmo levaria ao menor número possível de faltas de páginas, mas seu problema reside na impossibilidade de prever com precisão quando cada página será utilizada no futuro.Embora seja impossível prever o futuro uso das páginas em um sistema em tempo real, o autor sugere que é possível simular o algoritmo ótimo em um ambiente controlado.

2.2 Not Recently Used

Pergunta: 2) Explique o algoritmo de substituição de páginas Not Recently Used (não usada recentemente), apresentando um exemplo.

Resposta: O algoritmo de substituição de páginas Not Recently Used (NRU), é um algoritmo que visa remover páginas da memória com base em seu uso recente e estado de modificação. O NRU utiliza dois bits principais associados a cada página: o bit R (Referenciada) e o bit M (Modificada) [1, p. 145].

Bit R (Referenciada): Este bit é configurado pelo hardware sempre que uma página é acessada, seja para leitura ou escrita. O sistema operacional limpa periodicamente o bit R para identificar páginas que não foram usadas recentemente.

Bit M (Modificada): Este bit é configurado pelo hardware quando ocorre uma escrita em uma página. Ele indica se a página foi modificada desde que foi carregada do disco. Com base nos valores dos bits R e M, o algoritmo NRU classifica as páginas em quatro classes:

- Classe 0: Páginas não referenciadas e não modificadas.
- Classe 1: Páginas não referenciadas, mas modificadas.
- Classe 2: Páginas referenciadas, mas não modificadas.
- Classe 3: Páginas referenciadas e modificadas.

Quando ocorre uma falta de página, o algoritmo NRU seleciona aleatoriamente uma página para remoção da classe de menor ordem não vazia. A lógica por trás dessa escolha é que é preferível remover uma página modificada, mas não referenciada recentemente (Classe 1), do que uma página não modificada que está sendo utilizada ativamente (Classe 2).

Exemplo do Algoritmo NRU

Imagine um sistema com 4 páginas na memória e a seguinte configuração de bits R e M, como ilustrado na tabela 1:

Página	R	M	Classe
0	1	0	2
1	0	1	1
2	1	1	3
3	0	0	0

Tabela 1: Valores de exemplo para os bits R e M

Se ocorrer uma falta de página neste momento, o algoritmo NRU escolheria aleatoriamente uma página da Classe 0. No caso, a única opção seria a página 3. Após a remoção da página 3, a nova página seria carregada em seu lugar, e seus bits R e M seriam inicializados de acordo com o estado da página. O algoritmo NRU é fácil de entender e implementar, e oferece um desempenho razoável, embora não seja ótimo.

2.3 Least Recently Used

Pergunta: 3) Considere o algoritmo de substituição de página "Least Recently Used" (LRU) implementado com um hardware que mantém uma matriz de n x n bits. No exemplo, n será de 4 bits. Agora imagine a seguinte ordem de referência: 3 3 2 1 2 3 0. Mostre a sequência de matrizes de acesso do LRU para a ordem dada. Qual página que seria escolhida para substituição, se fosse necessário retirar uma?

Resposta: O algoritmo de substituição de página Least Recently Used (LRU) pode ser implementado com uma matriz de bits n×nn×n, onde nn é o número de páginas na memória. Essa matriz é atualizada a cada acesso, de forma que a linha correspondente à página acessada tenha seus bits ajustados.

- Cada elemento da matriz $M[i][j]$ representa se a página i foi acessada mais recentemente que a página j. Se $M[i][j] = 1$, significa que i foi acessada mais recentemente que j.
- Quando uma página é acessada, a linha correspondente a essa página é definida com 1 em todas as suas posições, e a coluna correspondente a essa página é definida com 0.

Considerando $n = 4$ (páginas 0, 1, 2 e 3), a matriz de acesso M é uma matriz 4×4 inicialmente zerada. Dada a ordem de referência $3, 3, 2, 1, 2, 3, 0$, as matrizes são atualizadas conforme descrito a seguir:

$$M_{inicial} = \begin{bmatrix} 0_{00} & 0_{01} & 0_{02} & 0_{03} \\ 0_{10} & 0_{11} & 0_{12} & 0_{13} \\ 0_{20} & 0_{21} & 0_{22} & 0_{23} \\ 0_{30} & 0_{31} & 0_{32} & 0_{33} \end{bmatrix}$$

$$PaginaReferenciada : 3 = \begin{bmatrix} 0_{00} & 0_{01} & 0_{02} & 0_{03} \\ 0_{10} & 0_{11} & 0_{12} & 0_{13} \\ 0_{20} & 0_{21} & 0_{22} & 0_{23} \\ 1_{30} & 1_{31} & 1_{32} & 0_{33} \end{bmatrix}$$

$$PaginaReferenciada : 2 = \begin{bmatrix} 0_{00} & 0_{01} & 0_{02} & 0_{03} \\ 0_{10} & 0_{11} & 0_{12} & 0_{13} \\ 1_{20} & 1_{21} & 0_{22} & 1_{23} \\ 1_{30} & 1_{31} & 0_{32} & 0_{33} \end{bmatrix}$$

$$PaginaReferenciada : 1 = \begin{bmatrix} 0_{00} & 0_{01} & 0_{02} & 0_{03} \\ 1_{10} & 0_{11} & 1_{12} & 1_{13} \\ 1_{20} & 0_{21} & 0_{22} & 1_{23} \\ 1_{30} & 0_{31} & 0_{32} & 0_{33} \end{bmatrix}$$

$$PaginaReferenciada : 2 = \begin{bmatrix} 0_{00} & 0_{01} & 0_{02} & 0_{03} \\ 1_{10} & 0_{11} & 0_{12} & 1_{13} \\ 1_{20} & 1_{21} & 0_{22} & 1_{23} \\ 1_{30} & 0_{31} & 0_{32} & 0_{33} \end{bmatrix}$$

$$PaginaReferenciada : 3 = \begin{bmatrix} 0_{00} & 0_{01} & 0_{02} & 0_{03} \\ 1_{10} & 0_{11} & 0_{12} & 0_{13} \\ 1_{20} & 1_{21} & 0_{22} & 0_{23} \\ 1_{30} & 1_{31} & 1_{32} & 0_{33} \end{bmatrix}$$

$$PaginaReferenciada : 0 = \begin{bmatrix} 0_{00} & 1_{01} & 1_{02} & 1_{03} \\ 0_{10} & 0_{11} & 0_{12} & 0_{13} \\ 0_{20} & 1_{21} & 0_{22} & 0_{23} \\ 0_{30} & 1_{31} & 1_{32} & 0_{33} \end{bmatrix}$$

Para identificar a página a ser substituída, verificamos a linha com o menor valor binário, pois ela não foi acessada a mais tempo em relação a todas as outras. Nesse caso, observando a matriz final, a página 1 seria escolhida para substituição, pois é a página com menor valor.

2.4 Page Fault Frequency

Pergunta: 4) O algoritmo PFF (Page Fault Frequency – Frequência de Falta de Página) é utilizado para controlar o tamanho do conjunto de páginas alocadas na memória RAM de um determinado processo, quando um algoritmo de alocação global é utilizado. Explique, com exemplos, como funciona este algoritmo.

Resposta: Imagine um processo com uma taxa de faltas de página de 10 faltas por segundo. Se o limite superior aceitável for de 5 faltas por segundo, o PFF alocará mais quadros de página para o processo, na tentativa de reduzir a taxa de faltas. Se, por outro lado, a taxa de faltas de página fosse de 1 falta por segundo, e o limite inferior aceitável fosse de 2 faltas por segundo, o PFF removeria alguns quadros de página do processo, liberando memória para outros processos. O PFF é baseado na observação de que, para a maioria dos algoritmos de substituição de página, como o LRU, a taxa de faltas de página diminui à medida que mais páginas são alocadas ao processo. No entanto, é importante observar que o PFF não especifica qual página deve ser substituída em caso de falta de página, ele apenas controla o tamanho do conjunto de alocação de cada processo.

2.5 Tradução de endereço MULTICS

Pergunta: 5) Explique como funciona a tradução de um endereço MULTICS para o endereço físico da máquina. O MULTICS trabalha com segmentação com paginação. Explique como funciona o descritor de segmento juntamente com as tabelas de páginas. Tanenbaum e Bos [1, p. 170]

Resposta: O MULTICS implementa um sistema de memória virtual que combina segmentação com paginação, o que significa que o espaço de endereço virtual é dividido em segmentos, e cada segmento é dividido em páginas. Esse sistema oferece as vantagens da segmentação, como a facilidade de programação, modularidade, proteção e compartilhamento, com as vantagens da paginação, como o tamanho uniforme da página e a capacidade de manter apenas as páginas necessárias na memória. A tradução de um endereço MULTICS para um endereço físico da máquina envolve as seguintes etapas:

- O endereço virtual MULTICS é composto por duas partes: o número do segmento e o endereço dentro do segmento.
- O número do segmento é usado como um índice na tabela de segmentos do programa. Cada entrada na tabela de segmentos corresponde a um descritor de segmento.
- O descritor de segmento contém informações sobre o segmento, incluindo um ponteiro para sua tabela de páginas, o tamanho do segmento, bits de proteção e outros itens. Se o segmento estiver na memória, o descritor conterá um ponteiro de 18 bits para a tabela de páginas do segmento.

- O endereço dentro do segmento é dividido em um número de página e um deslocamento dentro da página.
- O número da página é usado como um índice na tabela de páginas do segmento, que foi localizada através do descritor de segmento.
- A entrada da tabela de páginas contém o endereço físico do quadro de página na memória principal.
- O deslocamento é adicionado ao endereço do quadro de página para gerar o endereço físico da palavra na memória principal.

3 Desenvolver um programa

3.1

Enunciado: Nos dias atuais, sistemas com múltiplos cores dentro do processador estão em toda a parte. No Linux, podemos determinar a afinidade de uma determinada thread a um dado core com a função: sched-setaffinity. Faça um programa que gere 4 vertores de tamanho n com números aleatórios. Cada vetor deve ser ordenado por uma thread diferente. As threads devem ser divididas entre os cores existentes e cada thread deve sempre ser executada no mesmo core. Você deve verificar se a afinidade da thread foi escolhida corretamente com a função sched-getaffinity. Além disso, o comando top deve ser utilizado para a verificação do core que executa cada thread.

3.2 Código

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#include <unistd.h>
#include <time.h>

#define N 100000 // Tamanho dos vetores
#define NUM_THREADS 4

typedef struct
{
    int *vetor;
    size_t tamanho;
    int core_id;
} ThreadArgs;

void gera_vetor(int *vetor, size_t tamanho)
{
    for (size_t i = 0; i < tamanho; i++)
    {
        vetor[i] = rand() % 10000;
    }
}

void bubble_sort(int *vetor, size_t tamanho, int core_id)
{
    printf("Thread ordenando no core %d...\n", core_id);
```

```
30      for (size_t i = 0; i < tamanho - 1; i++)
31      {
32          for (size_t j = 0; j < tamanho - i - 1; j++)
33          {
34              if (vetor[j] > vetor[j + 1])
35              {
36                  int temp = vetor[j];
37                  vetor[j] = vetor[j + 1];
38                  vetor[j + 1] = temp;
39              }
40          }
41          if (i % (tamanho / 10) == 0)
42          { // Mostrar progresso a cada 10% do vetor ordenado
43              int current_core = sched_getcpu();
44              printf("Progresso: %.2f%%, executando no core atual %d (esperado:
                    %d)\n",
45                      (i * 100.0) / tamanho, current_core, core_id);
46          }
47      }
48      printf("Core %d: Ordering completed.\n", core_id);
49  }
50
51  void *ordena_vetor(void *args)
52  {
53      ThreadArgs *argumentos = (ThreadArgs *)args;
54      cpu_set_t cpuset;
55
56      // Configurar a afinidade da thread ao core especificado
57      CPU_ZERO(&cpuset);
58      CPU_SET(argumentos->core_id, &cpuset);
59      if (sched_setaffinity(0, sizeof(cpu_set_t), &cpuset) != 0)
60      {
61          perror("Erro ao configurar afinidade");
62          pthread_exit(NULL);
63      }
64
65      // Verificar a afinidade configurada
66      CPU_ZERO(&cpuset);
67      if (sched_getaffinity(0, sizeof(cpu_set_t), &cpuset) != 0)
68      {
69          perror("Erro ao obter afinidade");
70          pthread_exit(NULL);
71      }
72
73      if (CPU_ISSET(argumentos->core_id, &cpuset))
74      {
75          printf("Thread no core %d configurada corretamente.\n", argumentos->
                core_id);
76      }
77      else
78      {
79          printf("Erro: Thread no core %d not configurada corretamente.\n",
                argumentos->core_id);
80      }
```

```
81
82      // Ordenar o vetor usando Bubble Sort
83      bubble_sort(argumentos->vetor, argumentos->tamanho, argumentos->core_id);
84      pthread_exit(NULL);
85  }
86
87  int main()
88  {
89      srand(time(NULL));
90      pthread_t threads[NUM_THREADS];
91      ThreadArgs args[NUM_THREADS];
92      int *vetores[NUM_THREADS];
93
94      // Verificar o number de cores available
95      int num_cores = sysconf(_SC_NPROCESSORS_ONLN);
96      printf("Number of availables cores: %d\n", num_cores);
97
98      // Criar e inicializar os vetores
99      for (int i = 0; i < NUM_THREADS; i++)
100     {
101         vetores[i] = (int *)malloc(N * sizeof(int));
102         if (!vetores[i])
103         {
104             perror("Erro ao alocar memory");
105             return EXIT_FAILURE;
106         }
107         gera_vetor(vetores[i], N);
108
109         args[i].vetor = vetores[i];
110         args[i].tamanho = N;
111         args[i].core_id = i % num_cores; // Garantir distribution entre os
                cores
112     }
113
114     // Criar threads
115     for (int i = 0; i < NUM_THREADS; i++)
116     {
117         if (pthread_create(&threads[i], NULL, ordena_vetor, &args[i]) != 0)
118         {
119             perror("Erro ao criar thread");
120             return EXIT_FAILURE;
121         }
122     }
123
124     // Aguardar completion das threads
125     for (int i = 0; i < NUM_THREADS; i++)
126     {
127         pthread_join(threads[i], NULL);
128     }
129
130     // Liberar memory alocada
131     for (int i = 0; i < NUM_THREADS; i++)
132     {
133         free(vetores[i]);
```

```
134     }
135
136     return EXIT_SUCCESS;
137 }
```

3.3 Testes e Resultados

Como resultado da execução do código exibido na subceção 3.2, obtivemos a saída ilustrada na figura 1. O que mostrou que a afinidade com o core funcionou corretamente, já que salvamos a afinidade na estrutura, e posteriormente verificamos em qual core a ordenação está sendo realizada, e todos os outputs mostraram que o core corresponde com o esperado.

```
Progresso: 40.00%, executando no core atual 3 (esperado: 3)
Progresso: 40.00%, executando no core atual 0 (esperado: 0)
Progresso: 40.00%, executando no core atual 1 (esperado: 1)
Progresso: 50.00%, executando no core atual 3 (esperado: 3)
Progresso: 50.00%, executando no core atual 2 (esperado: 2)
Progresso: 50.00%, executando no core atual 0 (esperado: 0)
Progresso: 50.00%, executando no core atual 1 (esperado: 1)
Progresso: 60.00%, executando no core atual 3 (esperado: 3)
Progresso: 60.00%, executando no core atual 2 (esperado: 2)
Progresso: 60.00%, executando no core atual 0 (esperado: 0)
Progresso: 60.00%, executando no core atual 1 (esperado: 1)
Progresso: 70.00%, executando no core atual 3 (esperado: 3)
Progresso: 70.00%, executando no core atual 2 (esperado: 2)
Progresso: 70.00%, executando no core atual 0 (esperado: 0)
Progresso: 80.00%, executando no core atual 3 (esperado: 3)
Progresso: 80.00%, executando no core atual 2 (esperado: 2)
Progresso: 80.00%, executando no core atual 0 (esperado: 0)
Progresso: 70.00%, executando no core atual 1 (esperado: 1)
Progresso: 90.00%, executando no core atual 3 (esperado: 3)
Progresso: 90.00%, executando no core atual 2 (esperado: 2)
Progresso: 90.00%, executando no core atual 0 (esperado: 0)
Core 3: Ordenação concluída.
Core 2: Ordenação concluída.
Core 0: Ordenação concluída.
Progresso: 80.00%, executando no core atual 1 (esperado: 1)
Progresso: 90.00%, executando no core atual 1 (esperado: 1)
Core 1: Ordenação concluída.
```

Figura 1: Resultado da execução do programa

Referências

[1] A. S. Tanenbaum e H. Bos, *SISTEMAS OPERACIONAIS MODERNOS*, 4ª ed. Boston: Pearson, 2021.