Relatório - Simulador de Alocação de Páginas

Alunos: Gabriel Correa Terra (21105570), Lucas Ladwig (22100910), Pedro Ruschel Bressan (22100920)
Disciplina: INE5611 - Sistemas Operacionais
Turma: 04328A/B
Período: 2023.2

Implementação do Simulador

O simulador foi criado em *Python 3.11*, no modelo de orientação a objetos. O simulador foi implementado como uma única classe `Alocador de Páginas`, cujos atributos correspondem aos parâmetros necessários para as simulações, e cujos métodos correspondem aos diferentes algoritmos de substituição de páginas, além de getters e setters para os atributos e funcionalidades para gerar sequências de acesso e métricas de desempenho de cada algoritmo.

Nota: Os módulos `random`, `time` e `math` foram utilizadas para a implementação dessa classe, e estão presentes na biblioteca padrão *Python*, não necessitando instalar dependências adicionais.

Atributos

Os atributos a seguir descrevem os parâmetros necessários para as simulações:

- `num_quadros`: *Obrigatório*. Número de quadros disponíveis na memória física (16 a 1024).
- `max_pag_novas`: *Obrigatório*. Número máximo de páginas endereçáveis por processo (1x a 4x o número de quadros).
- `num_acessos`: *Opcional*. Número de acessos aleatórios para teste de algoritmos (1.000 ou mais).
- `tempo_memoria`: *Opcional*. Tempo de acesso a memória em nanossegundos (50 a 200).
- `tempo_tratamento`: *Opcional*. Tempo de tratamento de falhas de página em nanossegundos (8 a 20 milhões).

Algoritmos de Subsituição de Páginas

Foram implementados os seguintes algoritmos: FIFO, Segunda Chance, Relógio, NRU, LRU, NFU e Envelhecimento.

Fila simples (FIFO)

Chamada: `fifo()`

Algoritmo de fila simples, onde a página referenciada mais antiga é removida para entrar uma nova.

Segunda Chance

Chamada: `segunda_chance()`

Similar ao algoritmo de fila, mas implementado com mecanismo de segunda chance: um bit de referência (R) é utilizado para controlar se a página foi acessada no último ciclo. Caso a página tenha sido acessada no último ciclo (bit R = 1), ela volta ao final da fila com (bit R = 0) ao invés de ser excluída diretamente.

Relógio

Chamada: `relogio()`

Igual ao algoritmo de segunda chance, mas implementado com uma fila circular (ou 'relógio') para melhorar desempenho (final é "conectado" ao começo). Ajuda a evitar modificações repetidas na fila. Um ponteiro é mantido para apontar a próxima página candidata a sair da fila. També utiliza um bit de referência (R) para controlar se a página foi acessada no último ciclo.

NRU (Não Usado Recentemente)

Chamada: `nru()`

Algoritmo "não usado recentemente", que prioriza remover páginas não referenciadas, mesmo que modificadas recentemente. Define 4 classes com base em 2 bits: referenciado (R) e modificado (M), e remove aqueles de menor classe. Um bit presente/ausente (P/A) é usado para verificar se está na memória.

Nota: Nesta implementação, o período de "reset" dos bits R é simulado como um número aleatório de referências de página.

LRU (Menos Usado Recentemente)

Chamada: `lru_lista()`

Algoritmo "menos usado recentemente", que utiliza uma estrutura de lista ordenada das páginas carregadas em memória. A cada referência de página a lista é atualizada, colocando a página no início da lista. As páginas ao final da lista são as menos usadas recenemente e são as primeiras a serem removidas.

Nota: Nesta implementação a lista foi 'invertida', ou seja, as páginas usadas recentemente vão para o final e as páginas a serem removidas estão no início. Isto melhora um pouco o desempenho deste algoritmo nas listas em Python.

NFU (Não Usado Frequentemente)

Chamada: `nfu_contador()`

Algoritmo "não usado frequentemente", que prioriza remover as páginas menos referenciadas ao longo do tempo. Uma tabela com todas as páginas do processo é mantida, e cada página na tabela contém um contador de acessos, sendoincrementando com o valor do seu bit de referência (R) a cada interrupção de clock.

Nota: Nesta implementação, a interrupção de clock é simulada como um pequeno número aleatório de referências de página que vem da sequência de acessos. Em caso de falha de página, remove aleatoriamente uma página entre as de menor contador.

Envelhecimento

Chamada: `envelhecimento()`

Similar ao NFU, mas com um mecanismo de envelhecimento, que diminui os contadores ao longo do tempo para dar chanca de "renovar" as páginas a serem substituidas. O contador é uma sequência de bits, onde a

cada interrupção de clock os bits do contador são deslocados uma posição para a direita, e o valor do bit referência (R) é colocado na posição de bit mais significativo.

Nota: Nesta implementação, a interrupção de clock é simulada como um pequeno número aleatório de referências de página que vem da sequência de acessos. Em caso de falha de página, remove aleatoriamente uma página entre as de menor contador.

Outras Funcionalidades

Aqui estão descritas as outras funcionalidades do simulador: getters e setters, geradores de sequências de acesso e calculadores de desempenho.

Getters e Setters

Cada atributo listado anteriormente contém métodos *getters* e *setters* para que os atributos possam ser acessados e modificados de forma pública. Chamados pelos mesmos nomes dos atributos.

Geradores de Sequências de Acesso

Cada algoritmo responde de forma diferente a diferentes padrões de acesso à memória. Para melhor compreender o desempenho de cada algoritmo, foram então implementados **3 métodos geradores** de sequências de acesso às páginas: sequência aleatória, sequência localizada e sequência linear.

Nota: Para garantir que os algoritmos de substituição sejam comparados em condições iguais, cada gerador retorna uma mesma sequência de números, dados os mesmos parâmetros de números de quadros e páginas alocáveis. Além disso, os geradores foram implementados de forma que a sequência de acessos seja gerada número a número, evitando que uma sequência muito grande de acessos "estoure" a memória, caso fossem pré-carregados por inteiro.

Sequência Aleatória

Chamada: `sequencia_aleatoria()`

Gera uma sequência pseudoaleatória dentro da faixa possível de páginas referenciadas. Cada página tem a mesma chance de ser refernciada.

Sequência Localizada

Chamada: `sequencia_localizada()`

Gera uma sequência aleatória com maior localidade espacial. A sequência "avança" aleatoriamente a partir de uma página inicial. A localidade foi definida em 1/4 do total de páginas, sendo 1/8 para cada direção. Outros valores podem ser testados modificando o parâmetro interno ao método.

Sequência Linear

Chamada: `sequencia_linear()`

Gera uma sequência linear dentro da faixa possível de páginas referenciadas. Inicia a sequência a partir de uma pagina aleatória.

Calculadores de Desempenho de Algoritmos

Métodos que calculam métricas relevantes para analisar o desempenho dos algoritmos de substituição de páginas. Os métodos `__calcular_metricas()` e `medir_tempo()` são invocados por cada algoritmo e retornam as seguintes informações:

- **Total de Falhas:** número total de falhas ao longo da excecução do algoritmo.
- **Porcentagem de Falhas:** taxa proporcional de falhas ao longo da excecução do algoritmo.
- **Tempo Médio de Acesso a Memória:** tempo médio de acesso a memória (em milissegundos) considerando a taxa de falhas de página, bem como o tempo de acesso direto à memória e o tempo de tratamento das falhas.
  Este valor é dado pela fórmula $ t_a = (1 – p) \times t_m \times t_t $, onde $ t_a $ é o tempo médio de acesso a memória, $ p $ é a taxa de falhas de página, $ t_m $ é o tempo de acesso direto à memória e $ t_t $ é o tempo de tratamento de falhas de página.
- **Tempo Total de Execução:** tempo total de execução do algoritmo (em milissegundos). Calculado com auxílio do método decorador `medir_tempo()`, que "envolve" cada algoritmo, medindo o tempo inicial e final, retornando o tempo total de execução. **Nota:** o tempo de execução depende muito do sistema onde está sendo rodado.

Execução do Simulador

Para iniciar o simulador é preciso excutar o arquivo `main.py` e assegurar-se que o arquivo `alocador_de_paginas` esteja na mesma pasta. O simulador roda em linha de comando e apresenta alguns cenários pré-programados e ao final permite que o usuário insira seus próprios parâmetros para um cenário personalizado.

Para executar o simulador é **necessário** ter o módulo `tabulate` instalado. Ele facilita a apresentação dos dados em formato tabular. Como este módulo não faz parte da biblioteca padrão, talvez seja necessário instalá-lo antes de rodar o simulador. Para isso, basta rodar o seguinte código abaixo no terminal:

```
pip install tabulate
```

Nota: algumas instalações de *Python3* podem exigir que se use o seguinte comando (caso use outro gerenciador de pacotes, então siga suas instruções para a instalação):

```
pip3 install tabulate
```

Ao iniciar, o simulador deixa aberta a opção para executar os cenários predefinidos ou então simular cenários personalizados. Ambos os casos estão descritos a seguir.

Cenários Predefinidos

Ao escolher simular cenários predefinidos, os cenários abaixo são testados com os 3 tipos de sequência de acesso, totalizando 12 diferentes cenários. Os cenários estão descritos a seguir:

- **Poucos quadros (64)** disponíveis, com **carga média (1.5x)** de páginas endereçáveis
- **Poucos quadros (64)** disponíveis, com **carga alta (2x)** de páginas endereçáveis

- **Muitos quadros (1024)** disponíveis, com **carga média (1.5x)** de páginas endereçáveis
- **Muitos quadros (1024)** disponíveis, com **carga alta (2x)** de páginas endereçáveis

Todos os cenários são executados com os seguintes parâmetros:

- **Número de acessos aleatórios:** 100.000
- **Tempo de acesso a memória:** 100ns
- **Tempo de tratamento de falha de página:** 8ms ($ 8 \times 10^6 $ ns)

Nota: Após cada cenário ser simulado, é necessário apertar `enter` ou `return` para que o próximo cenário seja executado. Isto ajuda na legibilidade dos resultados.

```
Aperte ENTER para calcular próximo cenário:
```

Os resultados de cada cenário são apresentados no seguinte formato:

```
CENÁRIO 1
=============

- Quadros na memória: 64
- Páginas endereçáveis: 96
- Tipo de sequência de acesso: sequencia_aleatoria


ALGORITMO           Falhas (total)    Falhas (%)    Acesso (ms)
Execução (ms)
----------------  ----------------  ------------  -------------  ---------
------
FIFO                         33239        33.239          2.659
72.358
Segunda Chance               33300        33.3            2.664
108.063
Relógio                      33300        33.3            2.664
106.613
NRU                          33148        33.148          2.652
238.469
LRU com lista                33232        33.232          2.659
55.66
NFU com contador             33629        33.629          2.69
229.104
Envelhecimento               33372        33.372          2.67
256.009
```

Cenários Personalizados

Após a execução dos cenários predefinidos, o simulador permite ao usuário inserir, caso deseje, seus próprios parâmetros para a execução de um cenário personalizado. Neste caso, o usuário pode alterar *todos* os atributos do simulador, inclusive os opcionais, além de escolher qual tipo de sequência de acesso deseja utilizar.

Exemplo de prompt abaixo:

```
CENÁRIO PERSONALIZADO
=====================

Insira os parâmetros abaixo:
- Quadros na memória (16 a 1024): 32
- Páginas alocáveis (1x a 4x o número de quadros): 48
- Acessos aleatórios (1000 ou mais): 25000
- Tempo de acesso a memória em ns (50 a 200): 80
- Tempo de tratamento de falha de página em ns (5 a 20 milhões): 12000000
- Sequência de acesso (1 = ALEATÓRIA | 2 = LOCALIZADA | 3 = LINEAR): 2
```

Nota: Todos os parâmetros *devem* ser inseridos como números inteiros. Os valores de referência são dados em cada prompt.

Resultados e Discussões

Após analisar diversos cenários, percebeu-se que o fator determinante no desempenho dos algoritmos é o padrão de acessos à memória, derivado da sequência de acesso. Portanto, a discussão dos resultados está agrupada por este critério.

O número de quadros disponíveis e o número de páginas alocáveis, quando considerados em conjunto, tem uma menor influência, pelo menos nas condições apresentadas nestas simulações.

Cenários com Sequência Aleatória

Percebeu-se que nos casos em que os acessos à memória tem padrão aleatório, a tendência é que os algoritmos apresentem desempenhos muito parecidos em relação à taxa de falhas de página. Consequentemente, o tempo médio de acesso a memória é diretamente ligado à taxa de falhas de página.

Isso é esperado tendo em vista que qualquer página de um processo tem a mesma chance de ser referenciada, e, conforme aumenta o número de acessos, a tendência é que as falhas de página se distribuam igualmente indepente do algoritmo. Portanto, nesses casos o que mais conta é o melhor desempenho da execução do algoritmo em si.

Observações

- Todos algoritmos apresentaram taxas de falha de página muito similares.
- Algoritmos com melhores tempos de execução: LRU e FIFO.
- Algoritmos com piores tempos de execução: NRU, NFU e Envelhecimento.

Cenários com Sequência Localizada

Quando os acessos à memória tem um padrão com localidade espacial, a tendência também é que os algoritmos apresentem desempenhos parecidos, exceto nos casos que o número de quadros é baixo. Nestes casos, então, o algoritmo **NFU apresentou um desempenho pior** comparado aos outros algoritmos, em relação à taxa de falhas de página.

Observações

- Todos algoritmos, exceto NFU que é pior, apresentaram taxas de falha de página similares.
- Algoritmos com melhores tempos de execução: LRU e FIFO.
- Algoritmos com piores tempos de execução: NRU, NFU e Envelhecimento.

Cenários com Sequência Linar

Finalmente, nos casos em que o acesso à memória tem um padrão linear, foi observado que os algortimos FIFO, Segunda Chance, Relógio e LRU com lista apresentaram **100% de falhas** de página. Considerando que os próprios algoritmos gerenciam as páginas de forma linear, é esperado que estes algoritmos sejam praticamente inúteis nesses casos, já que uma nova página na sequência jamais vai ter sido referenciada na chamada anterior.

Considerando, então, as aplicações onde os acessos à memória tenham um aspecto mais linar, os algoritmos NFU (preferencialmente), NFU e Envelhecimento são os mais adequados entres os algoritmos analisados, devendo ser escolhidos com base no menor tempo de execução do algoritmo.

Observações

- Algortimos FIFO, Segunda Chance, Relógio e LRU com lista apresentaram **100% de falhas**.
- NFU apresentou menor taxa de falhas de página na maioria dos cenários.
- Melhores tempos de execução: NFU
- Piores tempos de execução: Envelhecimento

Conclusão

Após a extensa análise dos algoritmos, sequências de acessos e outros parâmetros, fica evidente que a eficiência dos algoritmos é **bastante** dependente dos padrões de acesso às páginas. É seguro dizer que, mantendo-se uma proporção entre o número de quadros na memória e o número de páginas endereçáveis, a diferença entre os algoritmos é pequena.

Portanto, para que um sistema seja operado da melhor forma possível, é necessário que se implemente um sistema de alocação de páginas adaptável a vários casos de uso, pois é muito difícil prever o padrão de acesso à memoria de um processo *a priori*.

No caso geral desta implementação, surpreendementemente, os algoritmos mais eficientes e com menor taxa de falhas são aqueles considerados os mais simples e "problemáticos": LRU, FIFO e Relógio (desde que o padrão de acesso não seja linear). Quando o padrãpo de acessos tem linearidade, então, o algoritmo NFU apresentou um melhor desempenho no geral.