gerenciamento de memória

Nathan Benedetto Proença
8941276
Victor Sena Molero
8941317

classes

O EP confia fortemente em Orientação a Objetos para organizar o código.

São centrais para a simulação as seguintes classes:

1. Runner
2. Memory
3. Page

classes auxiliares

Além das classes principais, várias classes auxiliares foram utilizadas neste projeto:

- `BinaryIO` serve de interface para as operações feitas nos arquivos `/tmp/ep3.mem` e `/tmp/ep3.vir`
- `Helper` contém uma função auxiliar para tratamento de erros
- `Trace` contém uma função que realiza a leitura do trace

classes auxiliares

Além das classes principais, várias classes auxiliares foram utilizadas neste projeto:

- `Process` abstrai os processos lidos no trace para facilitar sua ordenação e manipulação nas classes principais
- `Task` abstrai os acessos à memória realizados pelos processos e ajuda na abstracão realizada por `Process`

Runner

A classe `Runner` executa toda a simulação e cuida de todos os detalhes necessários para que isso ocorra, isto é, limpa a memória, ordena os processos, chama as funções de outras classes, etc.

Isso faz com que a função `main` tenha o papel de gerenciar a interface iterativa do EP, executando as funções chamadas por ele, inclusive a chamada para `Runner::execute`, a função principal desta classe.

Memory

Esta classe gerencia a memória usada na simulação, guardando e alterando funções importantes para a simulação. Estão, guardadas lá, por exemplo

- um vetor de bits que indica se cada posição está ocupada ou não na memória virtual chamado `Memory::used`
- os parâmetros total, virtual, s e p dados no começo do trace

Ela também serve de interface para acessos à memória.

Page

Esta classe representa o gerenciador de páginas usado na simulação guardando a tabela de páginas em `Memory::table` e os bits r de cada página em `Memory::r`.

Ela também serve de interface para acessos ao gerenciador de páginas.

Algorithm

Tanto `Page` quanto `Memory` possuem uma subclasses Algorithm. E um membro `manager` que é um ponteiro para uma instância dessa subclasse.

Esta subclasse representa o algoritmo utilizado para realizar tanto a substituição de páginas quanto o gerenciamento de memória.

Quando é escolhido um algoritmo para alguma dessas duas funções, é instanciada uma classe Algorithm e o seu ponteiro é guardado em `Page::manager` ou `Memory::manager`, de acordo com o algoritmo em questão.

Algorithm

A subclasse `Page::Algorithm` tem quatro classes herdadas

1. `PageOptimal`
2. `PageSecond`
3. `PageClock`
4. `PageLRU`

A subclasse `Memory::Algorithm` tem quatro classes herdadas

1. `MemoryFirst`
2. `MemoryNext`
3. `MemoryBest`
4. `MemoryWorst`

Memory::Algorithm

As 4 classes herdadas de `Memory::Algorithm` possuem implementações bastante simples e parecidas entre si.

Para todas elas basta iterar pela memória mantendo um ponteiro para o início do intervalo de memória livre atual. Quando se alcança uma memória ocupada, decide-se o que fazer com o intervalo atual.

O que muda de algoritmo para algoritmo é o bloco em que se começa a buscar estes espaços além do que se faz assim que se encontra um espaço.

PageOptimal

Para poder implementar este algoritmo, foi necessário primeiro chamar uma simulação “fantasma” de todo o processo, ou seja, chamar a função `Runner::execute` e salvar os passos feitos por ele ao invés de executá-los.

Tendo salvos os acessos à memória virtual em filas separadas pelas páginas, foi possível, a cada momento, descobrir qual das páginas mapeadas iria ser acessada mais tarde em tempo linear na quantidade de páginas da memória virtual.

PageSecond e PageClock

Em ambos os algoritmos, a implementação seguiu as instruções explícitas da descrição do algoritmo.

`PageSecond` utiliza uma fila (`queue` da biblioteca padrão do C++) enquanto `PageClock` utiliza uma lista ligada (`list` da biblioteca padrão do C++) para acessar as páginas guardadas na memória a fim de descobrir qual será substituída por uma nova página.

LRU (4a versão)

Neste algoritmo, a cada segundo simulado, foi realizada a atualização dos contadores p.

Em todos os algoritmos de substituição de página, o bit r é atualizado a cada 4 segundos de simulação.

Foi usado um inteiro de 64 bits para os valores dos contadores p.

teste gerado

O teste utilizado era gerado de forma aleatória. Contudo, foi feito de forma a garantir algumas propriedades:

- Memória física muito menor que a virtual
- Localidade de acessos à memória
- Duração dos processos proporcionais ao espaço na memória por ele utilizado
- Tamanho do espaço ocupado pelo processo gerado proporcional a memória disponível

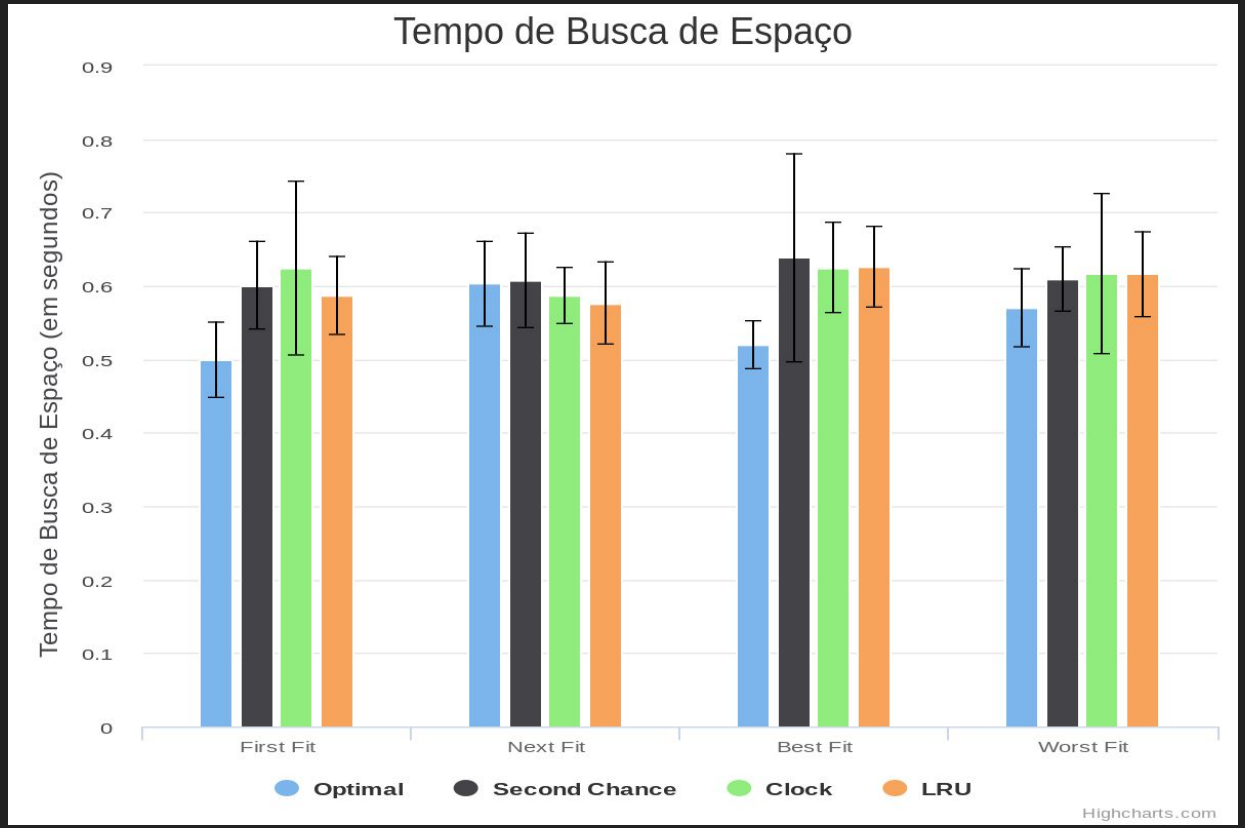
dados

Algoritmo de memória	Algoritmo de paginação	Tempo médio de busca	Média de page faults
First Fit	Optimal	0.498921233416	4427.0
First Fit	Second chance	0.600801320871	6013.0
First Fit	Clock	0.624453097582	6013.0
First Fit	LRU	0.586885497967	6050.0
Next Fit	Optimal	0.603107345104	4953.0
Next Fit	Second chance	0.60745151639	6105.0
Next Fit	Clock	0.586393698057	6105.0
Next Fit	LRU	0.575860087077	6199.0

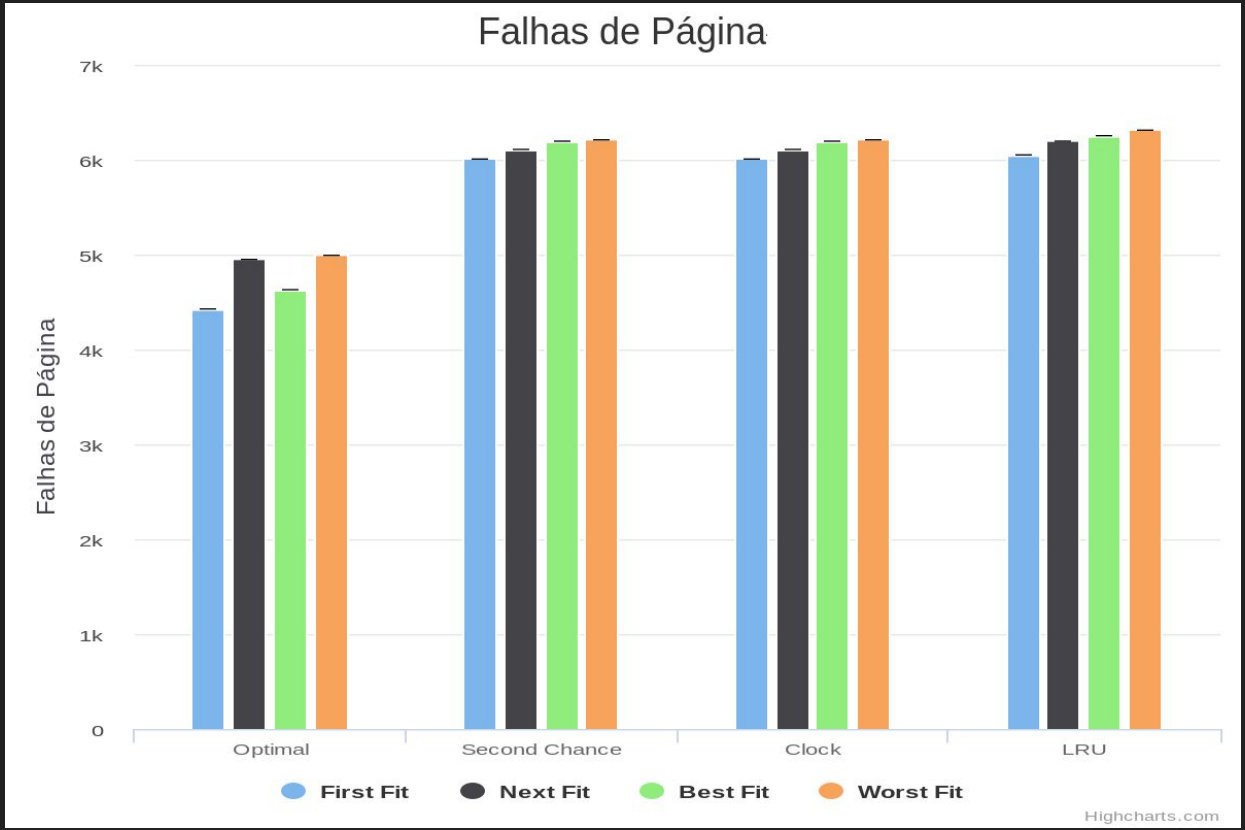
dados

Algoritmo de memória	Algoritmo de paginação	Tempo médio de busca	Média de page faults
Best Fit	Optimal	0.520543255409	4627.0
Best Fit	Second Chance	0.638332796097	6195.0
Best Fit	Clock	0.625025401513	6195.0
Best Fit	LRU	0.625275321802	6250.0
Worst Fit	Optimal	0.570575400194	4994.0
Worst Fit	Second Chance	0.609489645561	6217.0
Worst Fit	Clock	0.617309451103	6217.0
Worst Fit	LRU	0.615873142083	6318.0

tempo para alocações



falhas de páginas



comentários sobre teste

O tamanho da memória física com relação a virtual realmente ocasionou diversos pagefaults.

A localidade não destacou o LRU, como era de se esperar.

A duração proporcional ao espaço utilizado foi apenas uma escolha coerente com a realidade.

O tamanho do espaço ocupado pelo processo ser proporcional à memória destacou o first fit, com sesu baixo overhead.