Trabalho 2

Luana Pinho Bueno Penha - 2312082
Theo Jesus Canuto de Sousa - 2311293

Sistemas Operacionais - Turma 3WB

Prof. Markus Endler

24 de Junho de 2025, Rio de Janeiro

Funcionamento Geral do Programa

O sistema simula um Gerenciador de Memória Virtual (GMV) composto por um processo principal (pai) - que executa a função `GMV_Process()` - e quatro processos filhos (P1 a P4), responsáveis por gerar acessos a páginas virtuais. Na inicialização, o usuário passa entre dois e três parâmetros na linha de comando:

1. Algoritmo de substituição Not Recently Used, Second Chance, Least Recently Used ou Working Set ("`NRU`", "`2nCH`", "`LRU`" ou "`WS`", respectivamente)
2. Número de rounds a executa
3. (Opcional, só para "`WS`") tamanho da janela de tempo

A comunicação entre pai e filhos se dá por pipes dedicados (um par de descritores de leitura/escrita por processo), enquanto as tabelas de páginas de cada processo (32 entradas cada) e o vetor de flags de page-fault são mantidos em memória compartilhada (shmget + shmat). Cada entrada de tabela armazena o número do frame, os bits Presence, Modified, Referenced, além de campos auxiliares para Aging e timestamp de último acesso.

O escalonamento segue política Round Robin, com quantum fixo de 100 ms e atraso extra de 50 ms para processos que causam page-fault (implementação do ponto extra). O número de rodadas é definido pelo usuário via parâmetro na linha de comando, permitindo configurar a duração da simulação.

Gerenciador de Memória virtual (GMV)

A cada turno, o GMV:

1. Acorda o filho correspondente com SIGCONT
2. Lê do pipe o par <número da página> <tipo de acesso (R/W)> extraído sequencialmente do arquivo acessos_Px.txt (150 linhas)
3. Processa o acesso:
    a. Se a página já está presente, apenas atualiza os bits R/M e o timestamp.
    b. Caso contrário, registra um page-fault, tenta alocar um frame livre e, não havendo, invoca o algoritmo de substituição especificado na linha de comando ("`NRU`", "`2nCH`", "`LRU`" ou "`WS(k)`") para eleger uma página vítima.
    c. Se a vítima estava modificada, contabiliza uma escrita no swap.
    d. Atualiza as tabelas de páginas dos processos afetados.
4. Aplica SIGSTOP ao filho e passa para o próximo

Ao final dos N rounds o GMV imprime um resumo contendo:

- Algoritmo de substituição utilizado
- Total de rounds executados
- Número de page-faults por processo (P1 a P4)
- Total de páginas “sujas” gravadas no swap (contador de dirty writes)

- A tabela de processos de cada processo

Implementação dos algoritmos

- *Second Chance (substituição global)*

Implementado na função `select_2nCh()`. Esse algoritmo mantém um ponteiro circular, `clock_hand`, que aponta para o próximo frame a ser avaliado. A cada iteração, ele obtém o mapeamento atual daquele frame em `frame_map[clock_hand]` e verifica o bit de referência (`referenced`) na tabela de páginas compartilhada do respectivo processo. Se `referenced == false`, aquela página é escolhida como vítima: os índices de processo e página são preenchidos nos parâmetros de saída, e avançamos o ponteiro em uma única posição, retornando ao GMV. Caso contrário - ou seja, se a página tiver sido referenciada recentemente - nós zeramos o bit `referenced` (dando assim "segunda chance") e simplesmente avançamos o ponteiro, deixando essa página para uma próxima varredura. Esse ciclo continua até encontrarmos uma página com bit R em zero, garantindo tanto que não derrubemos páginas ainda em uso quanto que, eventualmente, todas as candidatas sejam consideradas.

Por operar de forma global, o Second Chance considera indiscriminadamente todas as páginas carregadas, independentemente de qual processo as originou. Apesar de, em tese, escanear até 16 quadros no pior caso, o custo amortizado se mantém baixo graças ao ponteiro circular, que "retoma" de onde parou a avaliação anterior. O resultado é uma política mais justa que o FIFO puro - pois evita retirar páginas ativas - sem a complexidade de contadores por página ou estruturas pesadas de algoritmos como LRU.

- *NRU (substituição global)*

Implementado em `select_NRU()`, o algoritmo classifica todos os frames ocupados em quatro classes, segundo os bits R (`referenced`) e M (`modified`) de cada página:

- R=0, M=0
- R=0, M=1
- R=1, M=0
- R=1, M=1

Ele escolhe aleatoriamente um frame da menor classe não vazia, garantindo preferência por páginas não referenciadas e não modificadas. Internamente, um contador estático `call_counter` é incrementado a cada chamada, e a cada 10 invocações todos os bits `R` são resetados a `0`, simulando o envelhecimento periódico das referências. Por operar globalmente, considera indiscriminadamente todas as páginas carregadas, e a seleção randômica dentro da classe evita vieses de FIFO simples.

- *LRU (substituição local)*

Implementado em `select_LRU()`, este método simula o comportamento "Least Recently Used" apenas sobre as páginas do processo que causou o page-fault (política local).

O mecanismo de aging é disparado em cada rodada pelo `update_aging_counters()`, que para todas as páginas presentes realiza `aging_counter >>= 1` e, se `referenced==true`, insere esse bit como MSB antes de resetar R. Em `select_LRU()`, percorremos as 32 entradas do processo afetado, localizamos aquelas com `present==true` e comparamos seus `aging_counter`; a página com menor contador (i.e., sem uso recente) é escolhida como vítima. Esse approach em software aproxima bem o LRU real, usando apenas um contador de 8 bits por página e um passo periódico de envelhecimento.

- *Working Set (substituição local)*

Implementado na função `select_WS()`. Neste algoritmo, cada processo carrega apenas as páginas que realmente está usando em um "conjunto de trabalho" de tamanho k, tornando a substituição estritamente local: só as páginas do próprio processo que gerou o page-fault são candidatas. Para isso, mantemos em cada `PageEntry` um campo `last_access` - um timestamp incrementado em `clock_counter` a cada acesso - e o parâmetro `k`, recebido na linha de comando quando o usuário escolhe WS.

Essa função primeiro percorre todas as 32 páginas do processo `victim_proc` procurando aquelas cuja última referência foi antes de `clock_counter - k`. Dentre essas, escolhe a que tiver o `last_access` mais antigo - ou seja, saiu do working set há mais tempo. Se não houver nenhuma fora da janela de trabalho, faz um "fallback": analisa todas as páginas presentes e seleciona novamente a mais antiga. Em último caso (situação teórica), cai em `page 0`. Como só examinamos páginas do processo ativo, garantimos isolamento de working set entre processos e evitamos que um processo "roube" quadros de outro, respeitando a política local.

Testes Realizados

Os testes realizados nesse T2, devido ao seu contexto, foram de input de usuário (tentativa de selecionar um algoritmo que não existe ou a falta de um valor `k` para `WS`). Além disso, a partir do momento que desenvolvemos o primeiro algoritmo, testamos para ver se o restante da estrutura do nosso código estava bem desenvolvida. Por fim, a cada algoritmo implementado, rodamos testes aumentando o número de rounds para analisar se estavam funcionando apropriadamente.

Resultados da Simulação

Analisamos 750 rounds (5x o número de linhas que nossos arquivos de acesso de cada processo possuem) para cada um dos 4 algoritmos implementados, com Working Set sendo rodado 2 vezes, para `k = 5` e `k = 3`, conforme sugerido no enunciado.

- NRU:

```
=== GMV SIMULATION SUMMARY: NRU ===

Total rounds executed: 750

Page-faults per process:
  P1: 685
  P2: 680
  P3: 648
  P4: 666

Total dirty writes to swap: 1346
```

- LRU:

```
=== GMV SIMULATION SUMMARY: LRU ===

Total rounds executed: 750

Page-faults per process:
  P1: 705
  P2: 660
  P3: 660
  P4: 620

Total dirty writes to swap: 1355
```

- 2nd Chance:

```
=== GMV SIMULATION SUMMARY: 2nd Chance ===

Total rounds executed: 750

Page-faults per process:
  P1: 694
  P2: 695
  P3: 640
  P4: 655

Total dirty writes to swap: 1369
```

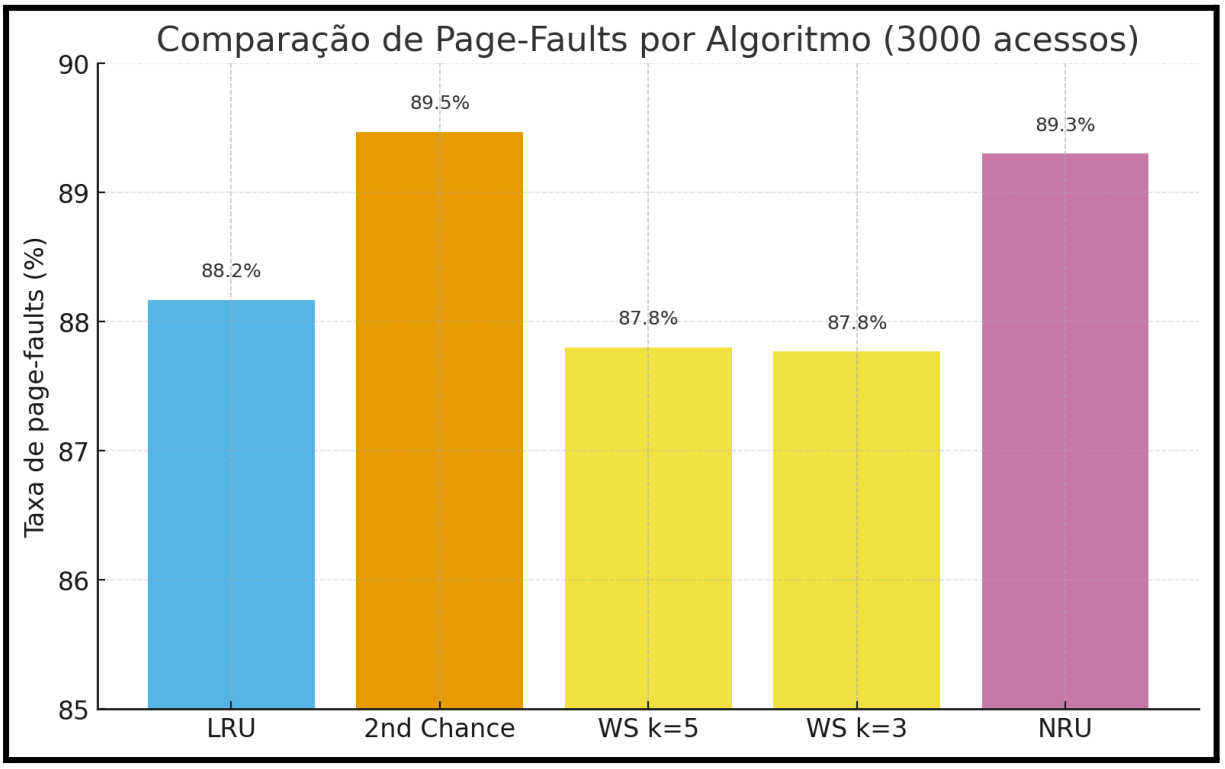
- Working Set para k = 3:

```
=== GMV SIMULATION SUMMARY: Working Set (k=3) ===

Total rounds executed: 750

Page-faults per process:
  P1: 688
  P2: 646
  P3: 659
  P4: 640

Total dirty writes to swap: 1279
```

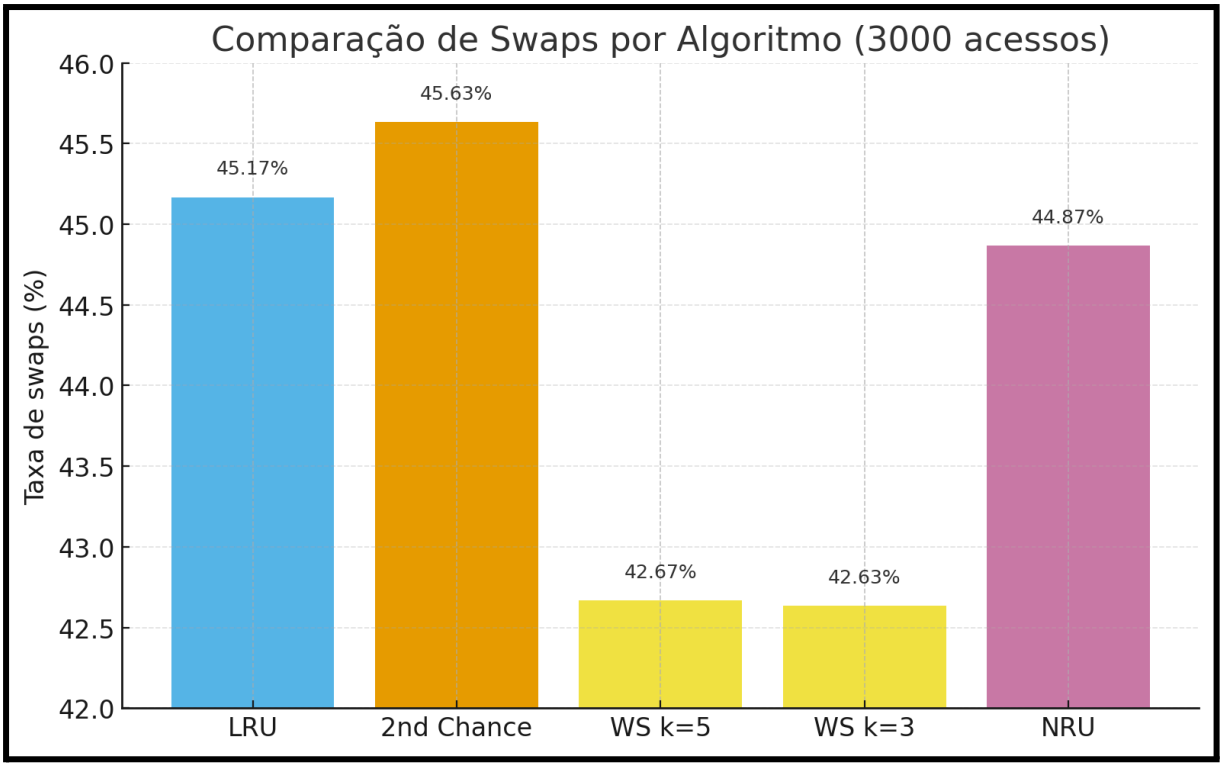
- Working Set para k = 5:

```
=== GMV SIMULATION SUMMARY: Working Set (k=5) ===

Total rounds executed: 750

Page-faults per process:
  P1: 689
  P2: 646
  P3: 659
  P4: 640

Total dirty writes to swap: 1280
```

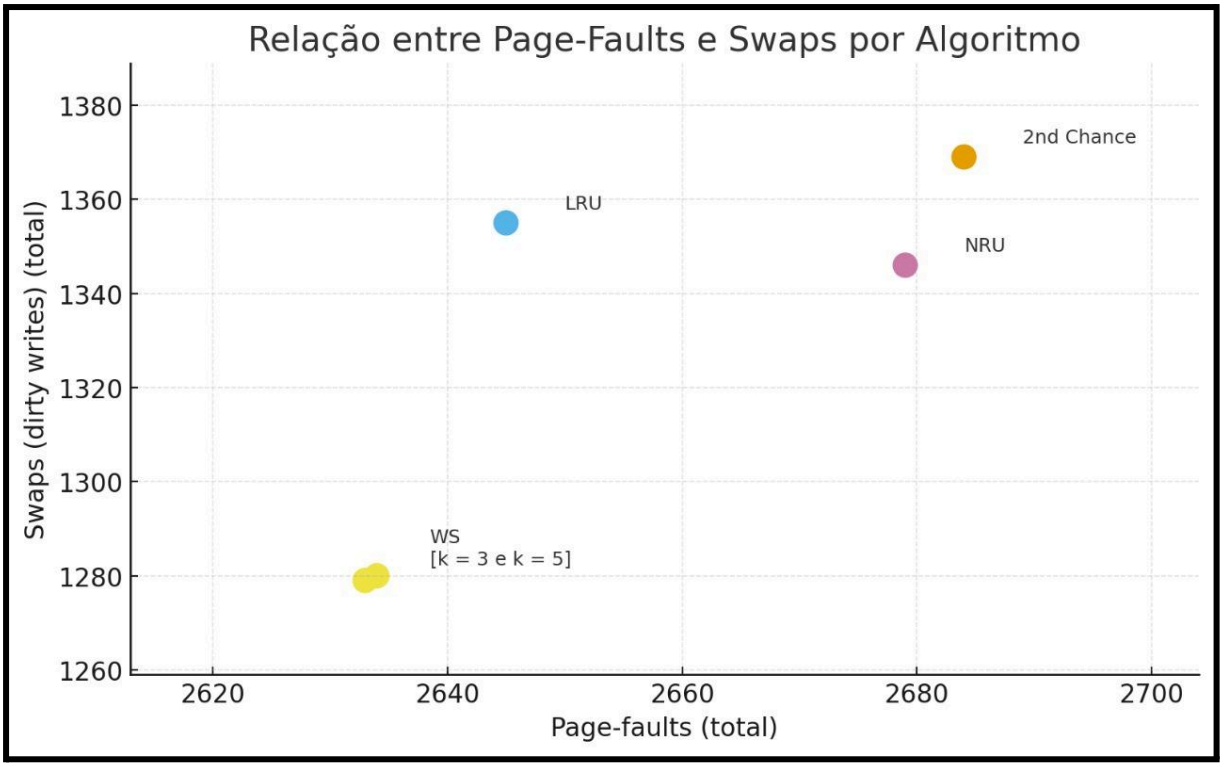
Análises Comparativas:



Percebe-se que o 2nd Chance e o NRU possuem o maior número de Page-Faults. A questão é que, na forma como operam, eles não conseguem ser tão eficientes em manter as páginas certas na memória quanto o LRU ou o Working Set. O NRU, por exemplo, trabalha com uma classificação das páginas que é um tanto simplificada, e ainda faz escolhas meio aleatórias dentro das categorias de menor prioridade, o que pode levar a ele despejar uma página que logo seria necessária. Já o 2nd Chance, embora seja uma melhoria do FIFO, ainda se baseia demais na ordem de chegada e usa apenas um bit de referência para decidir. Isso significa que ele tem menos "informação" sobre o uso real das páginas, fazendo com que ele, às vezes, segure páginas menos relevantes ou descarte páginas úteis muito cedo. No final das contas, a maneira como eles gerenciam as páginas se mostra menos otimizada para o nosso cenário, resultando em mais Page-Faults.



Na nossa simulação, nota-se que o LRU apresenta uma taxa de swaps alta - superada apenas pelo 2nd Chance - embora mantenha uma das menores porcentagens de Page-Faults. Isso pode ocorrer pois o LRU foca exclusivamente em substituir a página que não foi usada por mais tempo. Ele não diferencia se essa página é "limpa" (não modificada) ou "suja" (modificada). Se a página menos recentemente usada for uma página suja, o LRU a expulsará da memória para o disco, gerando um swap (write back), mesmo que haja páginas limpas menos úteis.



Ao comparar os resultados correlacionados, percebemos que Working Set - tanto com k = 3 quanto com k = 5 - entregou o melhor equilíbrio entre falta de página e escrita suja em swap. Nos 3.000 acessos simulados, ambas as janelas mantiveram a taxa de page-faults em torno de 87,8 %, quase um ponto percentual abaixo do LRU e cerca de dois pontos à frente do 2nd Chance.

Este diagrama de dispersão ilustra bem o resultado obtido: os dois pontos do WS concentram-se no canto inferior-esquerdo - a zona "ideal" de menos faults e menos swaps - enquanto LRU, NRU e 2nd Chance formam um aglomerado mais ao alto e à direita. Entre os dois valores de k, a diferença é mínima, mas o WS com k = 3 leva ligeira vantagem nos swaps, sem perder desempenho em faults.

Por fim, é interessante notar a semelhança entre as duas simulações de WS com valores diferentes para k. Isso se deve ao fato de que, no traço utilizado, o conjunto de trabalho real de cada processo raramente ultrapassa três páginas. Como o `clock_counter` avança apenas uma vez por rodada (isto é, por acesso de cada processo), passar de k = 3 para k = 5 representa só duas voltas extras do Round-Robin, insuficientes para que alguma página deixe de estar "recentemente usada". Quando todas as páginas presentes cabem na janela, o algoritmo cai no seu fallback (equivalente a um LRU local), e os resultados de k = 3 e k = 5 acabam praticamente idênticos. Para observar diferenças maiores seria necessário um padrão de acessos que explorasse mais de três páginas por processo, aumentar a granularidade do relógio ou reduzir ainda mais o número de quadros.