DOUGLAS SANTOS

AVALIAÇÃO DO COMPORTAMENTO DE SISTEMAS OPERACIONAIS EM SITUAÇÃO DE THRASHING

Dissertação submetida ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para a obtenção do título de Mestre em Informática.

Curitiba PR
Maio de 2009

DOUGLAS SANTOS

AVALIAÇÃO DO COMPORTAMENTO DE SISTEMAS OPERACIONAIS EM SITUAÇÃO DE THRASHING

Dissertação submetida ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para a obtenção do título de Mestre em Informática.

Área de concentração: *Ciência da Computação*

Orientador: Carlos Maziero

Curitiba PR
Maio de 2009

Santos, Douglas Alan dos.

Avaliação do Comportamento de Sistemas Operacionais em Situação de Thrashing. Curitiba, 2009. 79p.

Dissertação - Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática.

1. Thrashing 2. Gerência de memória 3. Sistemas operacionais I. Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e de Tecnologia. Programa de Pós-Graduação em Informática II-t.

Esta folha deve ser substituída pela ata de defesa devidamente assinada, que será fornecida pela secretaria do programa após a defesa.

A minha querida esposa.

Resumo

Em um sistema operacional convencional, o mecanismo de memória virtual permite usar discos rígidos como uma extensão da memória física. Dessa forma, torna-se possível oferecer aos processos em execução mais memória que aquela fisicamente disponível no sistema. Todavia, como os dispositivos de armazenamento são bem mais lentos que a memória RAM, seu acesso pode constituir um gargalo de desempenho. Quando a demanda por memória no sistema se torna muito elevada, o número de operações de leitura/escrita em disco também cresce. Se o espaço disponível em memória RAM for insuficiente para atender todos os processos ativos no sistema, o número de acessos a disco aumenta muito, podendo levar o sistema a uma situação conhecida como *thrashing*. Nessa situação, os processos ativos praticamente não conseguem mais continuar suas execuções, pois precisam esperar pelos acessos a disco, tornando o sistema inoperante. Apesar de ter sido inicialmente estudado nos anos 60, o *thrashing* ainda constitui um sério problema nos sistemas operacionais atuais. Este trabalho apresenta uma avaliação do comportamento de alguns sistemas operacionais de mercado em condições de *thrashing*. Foi desenvolvida uma ferramenta de *benchmark* que induz o sistema uma situação de *thrashing* controlada e depois seu retorno à normalidade. Essa ferramenta possui vários parâmetros ajustáveis, cuja influência também é avaliada. Com isso, pode-se avaliar como cada sistema operacional gerencia a memória durante o *thrashing* e quão rápido consegue se recuperar dele, quando a demanda por memória retornar a níveis normais.

Palavras-chave: thrashing, gerência de memória, sistemas operacionais.

Abstract

The virtual memory mechanism of a conventional operating system allows to use disks as an extension of the physical memory. Using this, it offers to running process more memory space than that physically available in the system. However, as storage devices are much slower than RAM memory, its access may become a performance bottleneck. When the demand for memory space increases, the number of disk input/output operations also increases. If the space available in physical RAM is not sufficient to satisfy the active processes, the number of disk accesses increases significantly, bringing the system to a situation known as *thrashing*. During a thrashing, the active processes almost cannot execute, as they should wait for the disk accesses, making the system unresponsive. Although it was first studied during the 60's, memory thrashing constitutes a serious problems in current operating systems yet. This work presents an evaluation of the behavior of some commodity operating systems under memory thrashing situation. A benchmark tool was developed to induce the operating system to a thrashing situation and then its return to normality. This tool has some adjustable parameters, whose influence was also evaluated. Using it, we could evaluate how each operating system manages the system memory during a thrashing occurrence and how fast it recovers from it, when the demand for memory returns to normal levels.

Keywords: thrashing, memory management, operating system.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

BSD	Berkeley Software Distribution
DMA	Direct Memory Access
FIFO	First In First Out
GCC	GNU Compiler Collection
GPL	GNU Public License
IPC	InterProcess Communication
MIB	Management Information Base
MMU	Memory Management Unit
MPL	Multiprogramming level
LAU	Least Actively Used
LRU	Least Recently Used
RAM	Random Access Memory

Capítulo 1

Introdução

São cada vez mais variadas as necessidades de um usuário de computador. Ouvir música, editar um texto e navegar na Internet são algumas das principais tarefas que um usuário desktop geralmente precisa executar e que devem ser gerenciadas de maneira eficiente pelo sistema operacional. Estas tarefas, ou programas, também cada vez mais exigem recursos do computador. A memória geralmente é um dos recursos mais afetados, pois vários destes programas ficam abertos ao mesmo tempo. O sistema operacional, através do seu gerenciador de memória, deve prover memória suficiente para todos os programas em execução, mesmo que não exista memória física disponível. Neste caso, o gerenciador usa espaço do disco rígido como uma extensão da memória principal. A memória total disponível, ou seja, a soma da memória física com o espaço reservado em disco é chamada memória virtual.

O gerenciador de memória deve sempre manter os programas mais ativos na memória física, pois ela tem um tempo de acesso muito menor que o disco rígido, também chamado área de *swap*. Por isso, frequentemente, o gerenciador de memória precisa transferir dados entre a memória física e o disco rígido, para atender às necessidades dos programas em execução, em um procedimento denominado *paginação*.

Quando o espaço disponível de memória RAM é insuficiente para atender todos os processos ativos, o número de paginação aumenta muito, podendo levar o sistema a uma situação de *thrashing*. Nesse caso extremo os processos ativos não conseguem usar o processador porque os dados ou código de que necessitam não estão na memória e têm de ser trazidos de volta do disco. Para trazer esses dados do disco, outros dados dever ser enviados para o disco, para liberar espaço na memória, afetando outros processos e assim sucessivamente. Durante o *thrashing*, o sistema torna-se extremamente lento, pois sua evolução passa a ser determinada pelo tempo de acesso ao disco rígido, geralmente entre 10^4 e 10^6 vezes mais lento que a memória RAM.

Thrashing é um problema de longa data e muitos algoritmos se propõem a resolvê-lo, no entanto ele ainda ocorre com frequência nos sistemas operacionais atuais. Geralmente é causado por ações involuntárias, como erros de programação, mas também pode ocorrer através do consumo excessivo de memória pelos programas de usuário do sistema. O objetivo principal deste trabalho é avaliar quantitativamente o comportamento de alguns sistemas operacionais de mercado em condições de *thrashing*. Para tal, foram selecionados alguns sistemas operacionais de amplo uso e foi definido um *benchmark* próprio, que provoca uma situação de *thrashing* controlada e seu retorno posterior à normalidade, em cada um dos sistemas operacionais sob estudo. Estes sistemas podem operar em modo desktop ou como ambientes multi-usuários com

terminais gráficos remotos. Nesse contexto, o *thrashing* pode ser particularmente problemático, pois a ação de um usuário pode afetar diretamente a disponibilidade do sistema para os demais.

A ferramenta de *benchmark* possui um programa que é responsável por gerar uma carga de trabalho usada para provocar o *thrashing*. Essa carga, consiste em criar alguns processos que alocam certa quantidade de memória e acessam essa memória de modo aleatório. Essa ferramenta possui vários parâmetros ajustáveis, cuja influência sobre o fenômeno de *thrashing* pode ser avaliada nos experimentos. A ferramenta de *benchmark* possui também outro programa que responsável pela obtenção de informações, como consumo de processador e memória, do núcleo dos sistemas operacionais avaliados. Cada sistema operacional apresentou reações diferentes que serão avaliadas nesse trabalho.

Esta dissertação está organizada da seguinte maneira: O capítulo 2 apresenta os conceitos de gerência de memória, incluindo gerência de memória virtual e o mecanismo de gerência de memória de cada sistema operacional avaliado; o capítulo 3 descreve os conceitos de *thrashing* e também apresenta os mecanismos, existentes nos sistemas operacionais avaliados, para prevenir o *thrashing*; o capítulo 4 apresenta os conceitos de *benchmark*, bem como algumas das principais ferramentas de *benchmark* existentes; o capítulo 5 apresenta o estudo dos sistemas operacionais sob o efeito de *thrashing*. Também é descrito neste capítulo toda a metodologia e experimentação, além dos resultados obtidos. A dissertação é concluída no capítulo 6, com as considerações finais acerca do trabalho realizado.

Capítulo 2

Gerência de memória

Neste capítulo são apresentados os conceitos básicos de gerência de recursos de um sistema operacional, os conceitos de gerência de memória virtual e paginação. Também são apresentados os principais sistemas operacionais desktop de mercado e seus respectivos mecanismos de gerência de memória. Os sistemas operacionais estudados foram o FreeBSD, Linux, OpenSolaris e o Windows XP pois apresentam características distintas, como diferenças em seus mecanismos de gerência de memória, e também apresentam características em comum, como o funcionamento na mesma plataforma de hardware.

2.1 Aspectos básicos

O sistema operacional é uma camada de software que atua entre o hardware e os programas aplicativos voltados ao usuário final. O sistema operacional é uma estrutura de software ampla, muitas vezes complexa, que incorpora aspectos de baixo nível, como *drivers* de dispositivos e gerência de memória, e de alto nível, como programas utilitários e a própria interface gráfica.

Um sistema operacional desktop é voltado ao atendimento do usuário doméstico e corporativo para a realização de atividades corriqueiras, como edição de textos, navegação na Internet e reprodução de mídias simples. Suas principais características são a interface gráfica de usuário, o suporte a interatividade e a operação em rede.

O sistema operacional tem como objetivos básicos oferecer a abstração de recursos, ou seja, oferecer interfaces de acesso aos dispositivos, e oferecer a gerência de recursos para definir políticas de uso dos recursos de hardware pelos aplicativos. Para cumprir estes objetivos, o sistema operacional deve atuar em várias frentes. Cada recurso do sistema possui suas particularidades, o que impõe exigências específicas para gerenciar e abstrair os mesmos. As principais funcionalidades implementadas por um sistema operacional típico são [Maziero, 2008]:

- Gerência do processador : também chamada gerência de atividade, visa distribuir a capacidade de processamento de forma justa[1] entre as aplicações, evitando que uma aplicação monopolize esse recurso e respeite as prioridades dos usuários. O sistema operacional provê a ilusão de que existe um processador independente para cada tarefa, o que facilita

[1] Distribuir de forma justa, mas não necessariamente igual, pois as aplicações têm demandas de processamento distintas.

o trabalho dos programadores de aplicações e permite a construção de sistemas mais interativos. Também faz parte da gerência de atividades fornecer abstrações para sincronizar atividades interdependentes e prover formas de comunicação entre elas.

- Gerência de memória : tem como objetivo fornecer a cada aplicação uma área de memória própria, independente e isolada das demais aplicações inclusive do *kernel* ou núcleo do sistema. O isolamento destas áreas de memória melhora a estabilidade e segurança do sistema, pois impede aplicações com erros (ou aplicações maliciosas) de interferir no funcionamento das demais aplicações. Além disso, caso a Memória de Acesso Aleatório ou RAM (*Random Access Memory*) existente seja insuficiente para as aplicações, o sistema operacional pode aumentá-la de forma transparente às aplicações, usando o espaço disponível em um meio de armazenamento secundário como um disco rígido. Uma abstração importante construída pela gerência de memória é a noção de memória virtual, descrita na seção 2.1.2, que desvincula os endereços de memória vistos por cada aplicação dos endereços acessados pelo processador na memória RAM. Com isso, uma aplicação pode ser carregada em qualquer posição livre da memória, sem que seu programador tenha que se preocupar com os endereços de memória onde ela irá executar.

- Gerência de dispositivos : cada periférico do computador possui suas peculiaridades; assim, o procedimento de interação com uma placa de rede é completamente diferente da interação com um disco rígido. Todavia, existem muitos problemas e abordagens em comum para o acesso aos periféricos. Por exemplo, é possível criar uma abstração única para a maioria dos dispositivos de armazenamento como discos rígidos, disquetes, CDs, etc, na forma de um vetor de blocos de dados. A função da gerência de dispositivos, também conhecida como gerência de I/O ou Entrada/Saída é implementar a interação com cada dispositivo por meio de *drivers* e criar modelos abstratos que permitam agrupar vários dispositivos distintos sob a mesma interface de acesso.

- Gerência de arquivos : essa funcionalidade é construída sobre a gerência de dispositivos e visa criar arquivos e diretórios, definindo sua interface de acesso e as regras para seu uso. É importante observar que, os conceitos abstratos de arquivo e diretório são tão importantes e difundidos que muitos sistemas operacionais os usam para permitir o acesso a recursos que nada tem a ver com armazenamento. Exemplos disso são as conexões de rede, onde em alguns sistemas operacionais, cada *socket* TCP é visto como um descritor de arquivo no qual pode-se ler ou escrever dados, e informações do núcleo como o diretório `/proc` do Linux.

Além dessas funcionalidades básicas oferecidas pela maioria dos sistemas operacionais, várias outras vêm se agregar aos sistemas modernos, para cobrir aspectos complementares, como a interface gráfica, suporte de rede, fluxos multimídia, gerência de energia, etc.

2.1.1 Estruturas de memória

Existem diversos tipos de memória em um sistema de computação, cada uma com suas características, mas todas com o mesmo objetivo: armazenar dados. Em um sistema computacional típico, pode-se identificar vários locais onde os dados são armazenados como, os registradores, o *cache* interno e externo do processador, denominados *cache L1* e *cache L2*, e a memória principal. Além disso, discos rígidos e unidades de armazenamento externas como

pendrives, também podem ser considerados memória em um sentido mas amplo, pois também têm como função o armazenamento de dados.

Esses componentes de hardware são construídos usando diversas tecnologias e por isso têm características distintas, como a capacidade de armazenamento, a velocidade de operação, o consumo de energia e o custo por byte armazenado. Os registradores e os *caches* L1 e L2 são memória mais rápidas, mais caras e consomem mais energia que memórias mais lentas, como a memória RAM. As memória mais rápidas, incluindo a memória RAM, são voláteis, ou seja, perdem seu conteúdo ao ficarem sem energia. Memórias que preservam seu conteúdo mesmo quando não tiverem energia são denominadas não-voláteis.

Outra característica importante de uma memória é a rapidez de seu funcionamento, que pode ser detalhada em duas dimensões: *tempo de acesso*, ou latência, e *taxa de transferência*. O tempo de acesso caracteriza o tempo necessário para iniciar uma transferência de dados de/para um determinado meio de armazenamento. A taxa de transferência indica quantos bytes por segundo podem ser lidos/escritos naquele meio, uma vez iniciada a transferência de dados. A Tabela 2.1 apresenta os valores de tempo de acesso e taxa de transferência típicos de alguns meios de armazenamento usuais.

Tabela 2.1: Tempos de acesso e taxas de transferência típicas [Patterson and Henessy, 2005]

Meio	Tempo de acesso	Taxa de transferência
Cache L2	1 ns	1 GB/s
Memória RAM (100 MHz)	60 ns	800 MB/s
Memória flash (NAND)	2 ms	10 MB/s
Disco rígido IDE	10 ms (tempo necessário para o deslocamento da cabeça de leitura e rotação do disco até o setor desejado)	80 MB/s

2.1.2 Memória virtual

Um problema constante nos computadores é a disponibilidade de memória física. Os programas se tornam cada vez maiores e mais processos executam simultaneamente, ocupando toda a memória disponível. Observando o comportamento de um sistema computacional, constata-se que nem todos os processos estão constantemente ativos, e que nem todas as áreas de memória alocada pelos processos estão constantemente sendo usadas. Por isso, as áreas de memória alocadas mas pouco acessadas poderiam ser transferidas para um meio de armazenamento mais barato e abundante, como um disco rígido, liberando a memória RAM para outros usos. O uso de um armazenamento externo como extensão da memória RAM se chama memória virtual.

Nos primeiros sistemas a implementar estratégias de memória virtual, processos inteiros eram transferidos da memória para o disco rígido e vice-versa. Esse procedimento denominado *swapping* permite liberar grandes áreas de memória a cada transferência, e se justifica no caso de um armazenamento com tempo de acesso muito elevado, como os antigos discos rígidos. Os sistemas atuais raramente transferem processos inteiros para o disco; geralmente as transferências são feitas por páginas ou grupos de páginas, em um procedimento denominado paginação ou *paging*.

Uma característica do processador e de sua Unidade de Gerenciamento de Memória ou MMU (*Memory Management Unit*) é a tabela de páginas, a qual faz o mapeamento de todas as páginas virtuais, ou lógicas, em suas páginas físicas correspondentes. Quando o processador acessa um endereço virtual, a MMU usa a tabela de páginas para traduzir o acesso em um endereço físico, que é o endereço passado para o sistema de memória do computador.

Se a tradução de uma página virtual para um endereço físico falhar, ocorre uma falta de página ou *page fault*. O sistema de memória virtual chama um tratador de faltas de páginas para parar o programa atual e responder à falta, procurando uma página livre na memória física, transferindo os dados do disco para a memória e atualizando a tabela de páginas para que essa página apareça no endereço lógico correto. O termo *page out* é utilizado quando o sistema copia páginas da memória para o disco. Já o termo *page in* é utilizado quando o sistema copia as páginas do disco para a memória.

A idéia central do mecanismo de memória virtual em sistemas com memória paginada consiste em retirar da memória principal as páginas pouco usadas, salvando-as em uma área do disco rígido reservada para esse fim. As páginas retiradas são escolhidas de acordo com os algoritmos descritos a seguir. As entradas das tabelas de páginas relativas as páginas transferidas para o disco devem estão ser ajustadas de forma a referenciar os conteúdos correspondentes no disco rígido. Essa situação está ilustrada de forma simplificada na Figura 2.1. Por razões históricas, essa área de disco é geralmente denominada área de troca ou área de *swap*, embora armazene páginas.

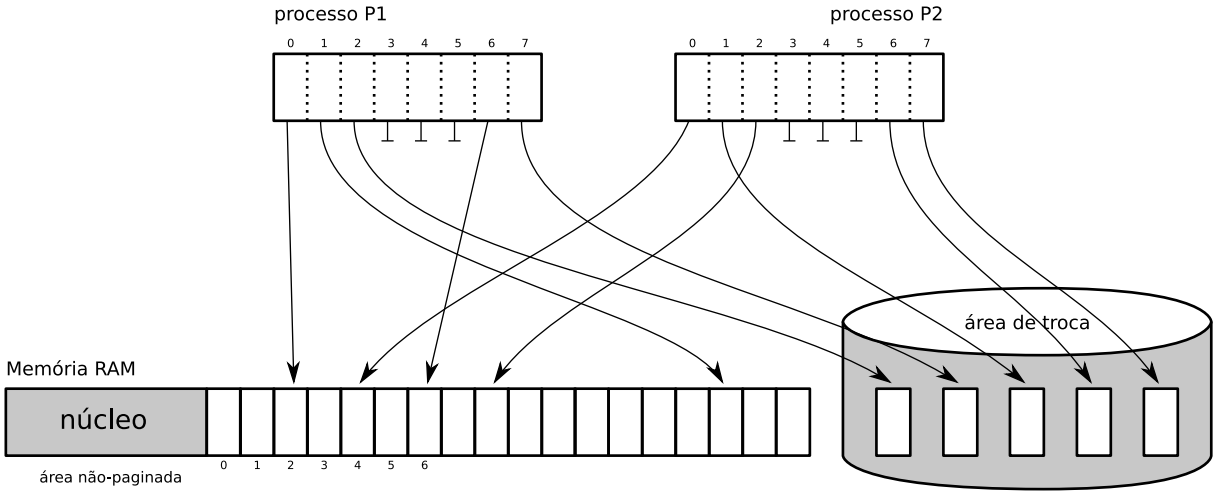

Figura 2.1: Memória virtual paginada [Maziero, 2008]

2.1.3 Princípio de localidade de referências

A maneira como os processos acessam a memória tem um impacto direto na eficiência dos mecanismos de gerência de memória, sobretudo o cache de páginas e o mecanismo de memória virtual. Processos que concentram seus acessos em poucas páginas de cada vez farão um uso eficiente desses mecanismos, enquanto processos que acessam muitas páginas distintas em um curto período irão gerar frequentes erros de cache e faltas de página, prejudicando seu desempenho no acesso a memória.

A propriedade de um processo ou sistema concentrar seus acessos em poucas áreas da memória a cada instante é chamada *localidade de referências* [Denning, 2005]. A localidade de referência de uma implementação depende de um conjunto de fatores, que incluem:

- As estruturas de dados usadas pelo programa: estruturas como vetores e matrizes têm seus elementos alocados de forma contígua na memória, o que leva a uma localidade de referências maior que estruturas mais dispersas, como listas encadeadas e árvores.
- Os algoritmos usados pelo programa: o comportamento do programa no acesso à memória é definido pelos algoritmos que ele implementa.
- A qualidade do compilador: cabe ao compilador analisar quais variáveis e trechos de código são usadas com frequência juntos e colocá-los nas mesmas páginas de memória, para aumentar a localidade de referência do código gerado.

2.1.4 Conjunto de trabalho

A localidade de referências, descrita anteriormente, mostra que os processos normalmente acessam apenas algumas de suas páginas a cada instante. O conjunto de páginas acessadas na história recente de um processo é chamado conjunto de trabalho *(working set)*. A composição do conjunto de trabalho é dinâmica, variando à medida em que o processo executa e evolui seu comportamento, acessando novas páginas e deixando de acessar outras.

O tamanho e a composição do conjunto de trabalho dependem do número de páginas. Se um processo tiver todas as páginas de seu conjunto de trabalho carregadas na memória, ele sofrerá poucas faltas de página, pois somente acessos a novas páginas poderão gerar faltas.

2.1.5 Algoritmos de substituição de páginas

A escolha correta das páginas a remover da memória física é um fator essencial para a eficiência do mecanismo de memória virtual. Escolhas ruins poderão remover da memória páginas muitos usadas, aumentando a taxa de faltas de página e diminuindo o desempenho geral do sistema operacional. Vários critérios podem ser usados para escolher páginas vítimas, ou seja, páginas a transferir da memória para a área de *swap*:

- Idade da página, há quanto tempo a página está na memória, páginas muito antigas talvez sejam pouco usadas.
- Frequência de acesso à página, páginas muito usadas possivelmente ainda serão usadas no futuro.
- Data do último acesso, páginas há mais tempo sem uso possivelmente serão pouco acessadas no futuro.
- Prioridade do processo proprietário, processos de alta prioridade, ou tempo-real, podem precisar de suas páginas de memória rapidamente.
- Conteúdo da página, páginas cujo conteúdo seja código executável exigem menos esforço do mecanismo de memória virtual, porque seu conteúdo já está mapeado no disco.

- Páginas especiais, páginas contendo *buffers* de operações de entrada/saída devem sempre estar na memória.

Existem vários algoritmos para a escolha de páginas na memória visando reduzir a frequência de falta de páginas, que levam em conta alguns dos fatores descritos acima:

Algoritmo Ótimo

Idealmente, a melhor página a remover da memória em um dado instante é aquela que ficará mais tempo sem ser acessada pelos processo. Essa idéia simples define o algoritmo ótimo, ou simplesmente OPT. Entretanto, como o comportamento futuro dos processos geralmente não pode ser previsto com precisão, esse algoritmo não é implementável.

Algoritmo FIFO

Um critério a considerar para a escolha das páginas a substituir poderia ser sua idade, ou seja, o tempo em que estão na memória. Assim, páginas mais antigas podem ser removidas para dar lugar a novas páginas. Esse algoritmo é muito simples de implementar: basta organizar as páginas em uma fila de números de páginas com política FIFO (*First In, First Out*).

Os números das páginas recém carregadas na memória são registrados no final da fila, enquanto os números das próximas páginas a substituir na memória são obtidos no início da fila. Apesar de ter uma implementação simples, na prática esse algoritmo não oferece bons resultados. Seu principal defeito é considerar somente a idade da página, sem levar em conta sua importância.

Algoritmo LRU

Uma aproximação implementável do algoritmo ótimo é proporcionada pelo algoritmo LRU (*Least Recently Used*) ou "menos recentemente usado". Nesse algoritmo, a escolha recai sobre as páginas que estão na memória há mais tempo sem serem acessadas. Assim, páginas antigas e menos usadas são escolhas preferenciais. Páginas antigas mas de uso frequente não são penalizadas por esse algoritmo, ao contrário que ocorre no algoritmo FIFO.

Pode-se observar facilmente que esse algoritmo é simétrico do algoritmo OPT em relação ao tempo: enquanto o OPT busca as páginas que serão acessadas "mais longe" no futuro do processo, o algoritmo LRU busca as páginas que foram acessadas "mais longe" no seu passado. O algoritmo LRU parte do pressuposto que páginas recentemente acessadas no passado provavelmente serão acessadas em um futuro próximo, e então evita removê-las da memória. Essa hipótese se verifica na prática, sobretudo se os processos respeitam o princípio da localidade descrito na seção 2.1.3.

Embora possa ser implementado, o algoritmo LRU básico é pouco usado na prática, porque sua implementação exigiria registrar as datas de acesso às páginas a cada leitura ou escrita na memória. Assim, na prática a maioria dos sistemas operacionais implementam aproximações do LRU, como as apresentadas na sequência.

Algoritmo da segunda chance

O algoritmo FIFO move para a área de *swap* as páginas há mais tempo na memória, sem levar em conta seu histórico de acessos. Uma melhoria simples desse algoritmo consiste

em analisar o bit de acesso, também chamado de bit de referência, de cada página candidata para saber se ela foi acessada recentemente. Caso tenha sido, essa página recebe uma "segunda chance", voltando para o fim da fila com seu bit de referência ajustado para zero. Desta forma, evita-se substituir páginas muito acessadas. Todavia, caso todas as páginas sejam muito acessadas, o algoritmo vai varrer todas as paginas e acabará por escolher a primeira página da fila, como faria o algoritmo FIFO.

Uma forma eficiente de implementar esse algoritmo é através da uma fila circular de números de página, ordenados de acordo com seu ingresso na memória. Um ponteiro percorre a fila sequencialmente, analisando os bits de referência das páginas. Quando uma página vítima é encontrada, ela é movida para o disco e a página desejada é carregada na memória no lugar da vítima. Essa implementação é conhecida como algoritmo do relógio.

Algoritmo NRU

O algoritmo da segunda chance leva em conta somente o bit de referência de cada página ao escolher as vítimas para substitui-lo. O algoritmo NRU (*Not Recently Used*), ou não usada recentemente, melhora essa escolha ao considerar também um bit de modificação. Este bit indica se o conteúdo da página foi modificado após ela ter sido carregada na memória.

Algoritmo do envelhecimento

Outra possibilidade de melhoria do algoritmo da segunda chance consiste em usar os bits de referência das páginas para construir contadores de acesso às mesmas. A cada página é associado um contador e periodicamente o algoritmo varre as tabelas de página lendo os bits de referência e carrega seus valores aos contadores de acessos das respectivas páginas. Uma vez lidos os bits de referencia são ajustados para zero, para registrar as referências que ocorrerão durante o próximo período.

O valor lido de cada bit de referência não é simplesmente somado ao contador, cada contador é deslocado para a direita 1 bit, descartando o bit menos significativo. Em seguida, o valor do bit de referência é colocado na primeira posição à esquerda do contador, ou seja, em seu bit mais significativo. Dessa forma, acessos mais recentes têm um peso maior que acessos mais antigos, e o contador nunca ultrapassa seu valor máximo.

O contador construído por esse algoritmo constitui uma aproximação razoável do algoritmo LRU: páginas menos acessadas "envelhecem", ficando com contadores menores, enquanto que páginas mais acessadas permanecerão "jovens", com contadores maiores. Por essa razão, essa estratégia é conhecida como algoritmo do envelhecimento [Tanenbaum, 2001].

2.1.6 Gerência de memória no FreeBSD

O FreeBSD [Lehey, 2001] é um sistema operacional Unix, descendente do 4.4BSD *Berkeley Software Distribution*, e começou a ser desenvolvido em 1993. Atualmente o FreeBSD é desenvolvido por um grupo de voluntários espalhados pelo mundo. Ele roda em diversas plataformas de hardware como AMD64, ARM, DEC Alpha, Intel x86 IA-32 e IA-64, PowerPC, Sun UltraSPARC, entre outros. Este sistema é considerado bastante confiável e robusto. É um dos sistemas operacionais, de código fonte aberto, com o maior tempo de *uptime*, ou seja, maior tempo sem reinícios ou travamentos do sistema [Netcraft, 2008].

O FreeBSD aloca a estrutura de *swap* para um objeto apenas quando é realmente necessário. Em vez de usar uma lista encadeada para rastrear as reservas de espaço de *swap*, utiliza um *bitmap* de blocos em uma árvore, o que torna a alocação e liberação de *swap* um algoritmo com complexidade O(1). Toda a árvore é alocada para evitar ter que alocar mais memória em situações criticas de operações de *swap* [McKusick and Neville-Neil, 2004].

O algoritmo de troca de páginas é uma aproximação do LRU, porém é baseado no histórico de uso de páginas de memória, também chamado de LAU (*Least Actively Used*). Para conseguir esse histórico, o sistema usa um *bit* de referência. O histórico de uso é desenvolvido verificando esse *bit* regularmente. Posteriormente, quando o sistema de memória virtual precisa liberar algumas páginas, verificar esse histórico é determinante para escolher a melhor página a ser reutilizada. O objetivo deste algoritmo é preservar páginas que tenham histórico de uso recente, ou seja, estas páginas serão favorecidas durante períodos de muita paginação.

O FreeBSD usa várias filas para refinar a seleção de páginas para reutilizar, bem como para determinar quando as páginas devem ser movidas para o disco. Quando uma página candidata é encontrada, ela é movida para a fila inativa se possuir dados, e posteriormente esses dados são escritos no disco antes que ela possa ser reutilizada. Se a página não tem dados ela é movida para a fila cache. Um algoritmo determina quando uma página na fila inativa pode ser movida para o disco. As páginas na fila cache podem ser reutilizadas a qualquer momento através de uma falta de página. No entanto, estas páginas serão as primeiras a serem reutilizadas de maneira LRU quando o sistema precisar alocar mais memória.

Quanto um programa é mapeado na memória mas não está mapeado na tabela de páginas, todos os acessos a páginas irão gerar faltas de página. Então o FreeBSD tenta preencher a tabela de páginas do processo com as páginas que estão no cache, reduzindo assim o número de faltas. No entanto, quando uma falta ocorrer o sistema não deve apenas alocar novas páginas, ele deve também preencher as páginas com zeros. Para otimizar essa função, o sistema possui um mecanismo que entra em funcionamento sempre que a CPU está livre e o número de páginas zeradas está limitado. A alocação de memória do FreeBSD implementa o algoritmo de coloração de páginas, o que significa que o sistema tenta alocar páginas livres que são contíguas do ponto de vista do cache.

2.1.7 Gerência de memória no Linux

O nome Linux vem do núcleo Linux [Kernel, 2008] originalmente desenvolvido por Linus Torvalds em 1991. Uma distribuição Linux, é um termo que se refere ao núcleo Linux mais um conjunto de pacotes de aplicativos. O Linux pode ser instalado em uma variedade de plataformas de hardware, que inclui dispositivos móveis até computadores de grande porte. O Linux é um software livre baseado no Unix e pode ser distribuído sobre os termos da GPL *GNU Public License*. O núcleo atualmente é desenvolvido por um grupo de voluntários espalhados ao redor do mundo.

O núcleo Linux 2.6 adota 4 KB como padrão de unidade de alocação de memória [Bovet and Cesati, 2005], basicamente por duas razões:

- Exceções de falta de páginas são facilmente interpretadas.
- 4 Kb é múltiplo de todos os blocos de disco, que na maioria dos casos é mais eficiente quando blocos pequenos são utilizados na transferência de dados entre a memória e o disco.

O Linux particiona a memória física em três zonas que são descritas a seguir: *a)* ZONE_DMA que contém páginas de memória abaixo de 16 MB; *b)* ZONE_NORMAL que contém páginas de memória acima de 16 MB e abaixo de 896 MB; *c)* ZONE_HIGHMEM contém páginas de memória acima de 896 MB. Quando o núcleo chama uma função de alocação de memória, deve especificar a zona que contém as páginas requisitadas.

O subsistema do núcleo que gerencia as requisições de alocação de memória para grupos contínuos de páginas é chamado de *Zone Allocator*. Este componente recebe as requisições para alocação e desalocação de memória. Para as requisições de alocação esse componente procura uma zona de memória que inclui grupos contíguos de páginas que satisfaçam a requisição. As páginas, dentro de cada zona, são gerenciadas pelo *algoritmo companheiro* ou algoritmo *buddy* [Knuth, 1997]. Para conseguir um melhor desempenho do sistema, uma pequena quantidade de páginas são mantidas em cache para satisfazer rapidamente as requisições de página través do esquema de alocação *slab* [Bonwick and Microsystems, 1994].

O algoritmo de recuperação de páginas é uma versão simplificada do algoritmo de páginas LRU para classificar as páginas em uso e as páginas não usadas. Se uma página não foi acessada por um longo período de tempo, a probabilidade dela ser acessada no futuro é baixa e pode ser considerada não usada. Por outro lado, se a página foi acessada recentemente, a probabilidade é que continue a ser acessada e é considerada em uso. O algoritmo tenta recuperar apenas páginas não usadas. A idéia do algoritmo LRU é associar um contador para armazenar a idade de uma página na RAM, ou seja, o intervalo de tempo desde o último acesso a página. Esse contador permite ao algoritmo recuperar as páginas mais antigas de qualquer processo.

Apesar dos esforços do algoritmo de recuperação em manter uma reserva de páginas livres, é possível que toda a memória disponível seja esgotada. Essa situação pode causar atrasos nas atividades dos processos do sistema. O núcleo tenta liberar memória para satisfazer uma requisição urgente, mas não tem sucesso porque a área de *swap* e todos caches de disco estão cheios. A consequência é que nenhum processo consegue executar, e nenhum processo consegue liberar mais páginas. Para resolver esta situação crítica, um novo algoritmo foi implementado no núcleo 2.6. Esse algoritmo usa um método chamado *Out of Memory killer* [OOM, 2009], o qual seleciona um processo no sistema e mata (via `SIGKILL`) esse processo para liberar suas páginas. Uma função seleciona um processo vítima, entre os processos existentes no sistema, para o sacrifício.

O critério de seleção de um processo vítima é baseado nos seguintes requisitos:

- A vítima deve possuir um grande número de páginas, assim o total de memória a ser liberada é significante.
- A vítima deve ser um processo recente.
- A vítima deve ter prioridade baixa, não pode ser um processo com privilégio de superusuário ou *root* e não pode ser nenhum dos processos de núcleo.
- A vítima não pode acessar diretamente os dispositivos de hardware, como o servidor X Window.

2.1.8 Gerência de memória no OpenSolaris

O OpenSolaris é um sistema operacional, de código fonte aberto, baseado no sistema Solaris da Sun Microsystems, sendo derivado do Unix System V Release 4. O Solaris foi lan-

çado originalmente pela Sun em 1991, e o projeto de desenvolvimento do OpenSolaris começou em 2004.

O sistema operacional Solaris usa dois tipos comuns de paginação em seu sistema de memória virtual. São eles: *a) swapping*, que faz a troca de toda a memória associada a um processo; *b)* paginação por demanda, que faz a troca páginas não usadas recentemente. O método usado em uma dada situação é determinado pela comparação da memória física disponível com diversos parâmetros como: contagem total de páginas, fila de E/S de *swap* e limites de memória definidos pelo administrador do sistema [Sun, 2006].

A partir do Solaris 8 é utilizado um novo algoritmo para remover páginas de memória. Esta nova arquitetura é conhecida como *cyclical page cache* e foi projetada para resolver a maioria dos problemas de cache nos sistemas de arquivos. Esse algoritmo usa uma lista no sistema de arquivos para fazer cache apenas de dados do sistema de arquivos. Outros objetos de memória, como binários de aplicações e bibliotecas, são gerenciados em uma lista separada.

Se a memória disponível do sistema está abaixo de um nível mínimo de memória livre, por um determinado período de tempo, o escalonador de memória começa a fazer a troca de processos. Inicialmente o escalonador procura por processos que estão inativos por mais tempo. Esse modo é chamado de *soft swapping*. Se há mais de um processo na fila de processos ativos e a atividade de paginação ultrapassa um valor pré definido o sistema começa a fazer o *hard swapping*. Nesse estado, o núcleo descarrega todos os módulos e cache do sistema que não estão ativos e começa a fazer a troca de processos sequencialmente até que exista memória livre disponível. Quando grandes sequências de E/S são necessárias é usado um mecanismo de E/S direto, para evitar problemas de desempenho devido ao uso excessivo de cache de páginas de memória.

2.1.9 Gerência de memória no Windows XP

O Windows XP é um sistema operacional desenvolvido pela Microsoft para uso em computadores pessoais, que inclui usuários domésticos e corporativos. Ele é descendente da família Windows 2000, e é o primeiro sistema Desktop produzido pela Microsoft sobre a arquitetura e o núcleo do Windows NT. A primeira versão do Windows XP foi lançada em 2001. De acordo com a pesquisa [MarketShare, 2008], o Windows XP é o sistema para desktop operacional mais utilizado no mundo, com mais de 60% da fatia de mercado nesse segmento.

A gerência de memória no Windows XP permite o sistema alocar e liberar memória virtual, compartilhar memória entre os processos, mapear arquivos em memória, copiar páginas de memória virtual para o disco, recuperar informação de um conjunto de páginas virtuais, mudar a permissão das páginas, e bloquear páginas virtuais na memória [Microsoft, 2003].

O Windows XP organiza a memória RAM da seguinte forma: *a)* Área não paginada, contém partes importante do sistema que nunca podem sofrer *page out*; *b)* Pool de páginas, que contém códigos e dados de programas, e também um espaço para *cache*. Para melhorar o desempenho, o Windows XP também permite que um processo, que possua os privilégios necessários, bloqueie páginas na memória, fazendo que essas páginas nunca sofram paginação.

A gerência de memória do Windows XP é baseada em paginação com tamanho de páginas variando entre 4 KB e 64 KB, dependendo do processador. O algoritmo do paginação do Windows é baseado em demanda por grupos ou *cluster*. Nesse modelo, quando ocorre uma falta de página, o gerenciador carrega na memória a página que faltava e mais algumas páginas ao seu redor com o objetivo de minimizar o número de acessos ao disco, explorando o princí-

pio de localidade, descrito na seção 2.1.3. A política de substituição de páginas do Windows, depende do processador, por exemplo, na arquitetura monoprocessada Intel o algoritmo de seleção de páginas é basicamente o LRU, já na arquitetura multiprocessada o algoritmo é FIFO [de Oliveira et al., 2003].

O Windows sempre tenta usar toda a memória RAM disponível, ele mantém o código dos programas na RAM, mesmo que os programas já tenham terminado, no caso dos programas precisarem desse código novamente. Assim, uma quantidade insignificante de memória vai ser reportada pelo sistema como memória livre ou disponível. Se há pouco espaço na RAM, então partes de código ou dados de um programa que não estão sendo usados podem sofrer paginação, para liberar mais espaço na RAM.

2.2 Conclusão

Neste capítulo foram apresentados os conceitos de gerência de recursos, principalmente gerência de memória, de um sistema operacional. Também foram apresentados os mecanismos de gerência de memória dos sistemas operacionais que serão avaliados neste trabalho.

Todos os sistemas operacionais mencionados utilizam alguma variação do algoritmo de páginas não utilizadas recentemente ou LRU. Todos eles também utilizam o conceito de listas ou filas para fazer referência às páginas atualmente em uso ou disponíveis. O FreeBSD, Linux e o Solaris possuem um processo ou *daemon* para fazer a troca de páginas. O mecanismo para tratar o espaço de endereços nestes sistemas operacionais são similares, porém o nome e as estruturas de dados utilizadas são completamente diferentes. A Tabela 2.2 apresenta um resumo dos algoritmos de substituição de páginas utilizados nesses sistemas operacionais.

Tabela 2.2: Algoritmos de substituição de páginas

FreeBSD	aproximação LRU baseada no histórico de uso
Linux	aproximação LRU baseada na idade da página
OpenSolaris	aproximação LRU baseada no algoritmo do relógio
Windows XP	aproximação LRU e FIFO para construir um *working set*

O capítulo seguinte apresenta os conceitos de *thrashing* e como ele pode ser resolvido ou minimizado nos sistemas atuais. Além disso, esse capítulo também apresenta os principais mecanismos propostos na literatura visando resolver o problema de *thrashing* ou atenuar seus efeitos.

Capítulo 3

Thrashing

Este capítulo descreve em detalhes o fenômeno conhecido por *thrashing*. Aqui são apresentados os conceitos e os modelos de tratamento já existentes nos sistemas operacionais avaliados. Além destes modelos, também são apresentados as principais propostas de melhorias ou mecanismos avançados de gerência de memória que visam minimizar o *thrashing*.

3.1 Conceitos

O termo *thrashing* foi inicialmente utilizado para descrever o som que as fitas magnéticas faziam quando estavam escrevendo e lendo dados rapidamente [Denning, 1968]. Hoje esse termo designa a situação de um computador durante um episódio de consumo excessivo de memória RAM. Nesse caso, o sistema operacional precisa escrever ou ler muitos dados do disco rígido por causa do mecanismo de paginação, fazendo com que os processos fiquem esperando muito tempo pelo uso da CPU. Nos sistemas de computação atuais, o *thrashing* geralmente provoca sérios problemas de desempenho, tornando o sistema inutilizável.

O fenômeno de *thrashing* tem uma dinâmica relativamente simples: o sistema operacional monitora o uso da CPU, se a utilização de CPU estiver baixa, o nível de multiprogramação é aumentado, ou seja, novos processos podem ser introduzidos no sistema. Se um novo processo entrar em execução e precisar de mais páginas de memória, este novo processo então começa a gerar muitas faltas de páginas, "roubando" páginas do outros processos. No entanto, estes processos também precisam destas páginas e também começam a geram muitas faltas de página. Estes processos então começam a sofrer operações de *page in* e *page out*. Enquanto eles estão na fila, esperando por páginas de memória, a fila de processos aptos a executar *(ready queue)* fica vazia. Com isso a utilização de CPU diminui, pois todos os processos estão esperando por páginas de memória. Por outro lado o escalonador de processos percebe a diminuição da utilização de CPU e aumenta o nível de multiprogramação. Então, novos processos começam a executar, tomando mais páginas de memória e gerando mais faltas de páginas.

Nesse ponto o *thrashing* já está acontecendo, e o desempenho do sistema diminui significativamente. A taxa de faltas de página aumenta rapidamente, com isso o tempo de acesso a memória também aumenta, e nenhum trabalho ou processamento é feito [Silberschatz et al., 2002]. A Figura 3.1 ilustra esse comportamento. O nível de multiprogramação, ou MPL (*Multiprogramming level*), é definido como o número de processos ativos em um sistema [Denning, 1995].

O fenômeno de *thrashing* é influenciado pelas seguintes condições:

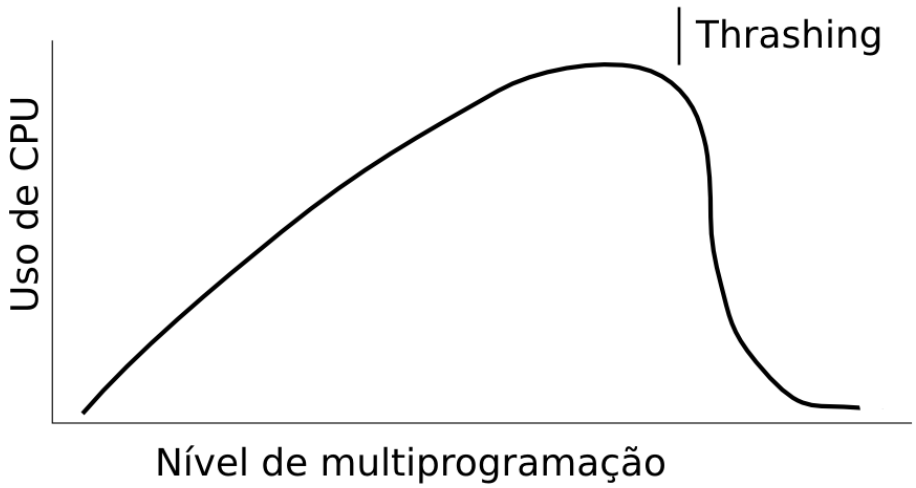

Figura 3.1: Thrashing [Silberschatz et al., 2002]

- O tamanho do espaço de memória do sistema;
- O número de processos ativos no sistema;
- A demanda dinâmica de memória de cada processo;
- O algoritmo de troca de páginas.

Algumas soluções para evitar o *thrashing* poderiam ser, por exemplo, adicionar mais memória ao sistema, porém essa solução nem sempre é viável e a estrutura dos computadores atuais limita a quantidade máxima de memória que pode ser instalada. Outra solução poderia ser um algoritmo de troca de páginas local, que quando um processo começar a fazer *thrashing* não possa "roubar" páginas de outros processos. No entanto o problema não é completamente resolvido, pois o processo irá precisar de páginas e, como não pode emprestar de outros processos, ele continua na fila de processos esperando por páginas, aumentando seu tempo de execução.

O aspecto mais destrutivo de *thrashing* é acionado quando ocorre um pico de carga de trabalho, como por exemplo, todos os usuários do sistema pressionarem a tecla *Enter* no mesmo segundo. O *thrashing* também pode ser gerado por programas que apresentam baixa localidade de referência, porque o programa não pode ser mantido de maneira eficiente na memória, gerando constantes movimentos na área de *swap*.

3.2 Tratamento de *thrashing*

Esta seção descreve os métodos de tratamento de *thrashing* já existentes nos sistemas operacionais sob estudo nesta dissertação. A detecção do *thrashing* é tão importante quanto o seu tratamento propriamente dito, pois quanto mais cedo um sistema consegue detectar a ocorrência de *thrashing*, mais eficiente é a redução deste fenômeno.

3.2.1 FreeBSD

O FreeBSD tenta detectar situações de *thrashing* observando a quantidade de memória livre [McKusick and Neville-Neil, 2004]. Quando o sistema possuir poucas páginas de memória disponíveis e uma taxa elevada de novas requisições de memória, ele se considera em *thrashing*.

O sistema reduz o *thrashing* fazendo com que o último processo que conseguiu executar não consiga mais executar, ou seja, não permite que esse processo volte ao processador durante algum tempo. Isto permite que o serviço, ou *daemon*, de paginação coloque todas as páginas associadas a esse processo no disco.

O efeito desta ação é fazer com que os processos sofram *swap out*. A memória liberada por estes processos bloqueados pode ser distribuída entre os processos remanescentes. Se o *thrashing* ainda continuar, outros processos são selecionados para serem bloqueados, até que exista memória suficiente para os demais processos.

Mesmo que não exista memória disponível, é permitido aos processos bloqueados que voltem a executar depois de aproximadamente 20 segundos. Geralmente a condição de *thrashing* volta a aparecer, o que requer que outros processos sejam selecionados para serem bloqueados.

3.2.2 Linux

O método clássico de redução de *thrashing* usado em sistemas BSD limita o número de processos que executam simultaneamente, ou seja, suspende temporariamente alguns processos. Isto garante uma redução de carga no gerenciador de memória do sistema, porém não garante que os processos consigam fazer um progresso significativo.

No Linux, a partir da versão de núcleo 2.6.10, foi incorporado uma técnica chamada *token swap* [Jiang and Zhang, 2005] para tentar resolver o problema de *thrashing*. Nessa técnica, um *token* é atribuído a um único processo no sistema permitindo que esse processo seja excluído do algoritmo de recuperação de páginas, ou seja, não sofrerá *swap* no período que estiver com o *token*. Assim, esse processo pode conseguir um progresso substancial em sua execução.

O *token* não é considerado, por exemplo, quando o algoritmo de recuperação alcançou o último nível de prioridade. O Linux implementa alguns níveis de prioridade no seu algoritmo de recuperação de páginas. Quando o algoritmo alcança o último nível, mais nenhuma página de memória pode ser recuperada e o sistema é forçado a selecionar um processo vítima (descrito na seção 2.1.7) para ser morto.

Uma função determina se o *token* deve ser atribuído ao processo atual. Ela faz algumas verificações antes de atribuir o *token*:

- Pelo menos 2 segundos se passaram desde a última chamada desta função.
- O processo atualmente com *token* não fez uma falta de página desde a última execução desta função ou tem o *token* desde o ultimo período de tempo em que esteve com o *token*.
- O *token* não foi atribuído recentemente ao processo atual.

O tempo de permanência do *token* deve ser grande o suficiente, com o objetivo de permitir ao processo terminar sua execução. O administrador do sistema pode configurar esse valor através do arquivo */proc/sys/vm/swap_token_default_timeout*.

3.2.3 OpenSolaris

Manter dinamicamente um nível de multiprogramação ótimo para conseguir uma alta utilização de CPU é uma questão fundamental no projeto de um sistema operacional. Os projetistas de sistemas operacionais tentam desenvolver a solução mais eficiente para o problema de uso dos recursos de CPU e memória dentro de um ambiente de multiprogramação, tentando evitar o *thrashing* que a multiprogramação pode causar.

Geralmente, quando o nível de multiprogramação aumenta, o número de processos em execução também aumenta, e como consequência o consumo de CPU cresce. No entanto, quanto o nível de multiprogramação cresce a um certo ponto, a competição por páginas de memória se torna crítica, o que eventualmente causa o *thrashing* do sistema e reduz significativamente o uso da CPU.

Considerando grandes variações de demanda de memória, praticamente não é possível definir um nível de multiprogramação ótimo para evitar completamente o *thrashing* e ainda permitir um grande número de processos no sistema. Deste modo, alguns sistemas operacionais incluindo o OpenSolaris, oferecem uma facilidade de controle de carga para fazer *swap in* e *swap out* de processos para a proteção contra *thrashing*.

Adicionalmente ao mecanismo de paginação, onde apenas algumas páginas de cada processo são movimentadas entre o disco e a memória, o gerenciador de memória pode mover um processo inteiro para o disco, com o objetivo de conservar memória. Esta facilidade permite que sistema possa reduzir o nível de multiprogramação de forma dinâmica. Porém, o *swaping* de um processo inteiro, dependendo do uso de memória deste processo, pode ter um custo muito alto para o sistema. Esta ação só tomada em extremas circunstâncias [Mauro and McDougall, 2001].

3.2.4 Windows XP

O Windows XP, de maneira similar a alguns outros sistemas operacionais como o Apple MacOS X [Singh, 2001], implementa um arquivo chamado *pagefile.sys* como área dinâmica de *swap*. Ou seja, diferentemente dos sistemas Unix tradicionais que implementam uma partição fixa para a área de *swap*, o Windows XP implementa um mecanismo que gerencia, sob demanda, o tamanho deste arquivo. Isto proporciona um espaço extra para as situações mais críticas.

O Windows XP também tenta fazer um uso eficiente do seu algoritmo de paginação de memória, que é baseado em *Working sets* com agrupamento de páginas [Russinovich and Solomon, 2004]. Uma *thread* de núcleo periodicamente revisa os tamanhos dos conjuntos de trabalho, removendo páginas caso haja pouca memória disponível. O gerenciador de memória tenta recuperar as páginas através de um *cluster* de páginas. Assim, o Windows XP consegue explorar de maneira mais eficiente o princípio de localidade de referência, pois as páginas são sempre recuperadas em grupo. Outra *thread* de núcleo migra gradativamente processos há muito tempo suspensos para a área de swap. Na documentação analisada, não foi encontrada menção a uma política explícita para thrashing no Windows XP.

3.3 Propostas

Nesta seção são apresentados as principais propostas ou mecanismos de gerência de memória que pretendem resolver, ou pelo menos minimizar, o problema de *thrashing*. Seja através do uso de memória remota, ou melhorando o desempenho do algoritmo de trocas de páginas, ou ainda modificando o escalonador de processos para agrupar os processos de acordo com a quantidade de memória que cada processo necessita.

3.3.1 Uso de memória remota

O artigo [Markatos, 1996] apresenta uma proposta que busca aproveitar a baixa latência das redes de alta velocidade para tornar possível o uso de memória remota, ou compartilhamento de memória via rede, como uma alternativa para armazenar os dados de aplicações. A arquitetura utilizada foi um sistema distribuído, composto de máquinas com processador e memória local. A memória de todas as outras máquinas compõem a memória remota.

Uma simulação de aplicações foi utilizada para avaliar a eficiência deste mecanismo de memória remota. O artigo explora a possibilidade de usar a memória remota para ter: *a)* área de armazenamento mais rápida que o disco rígido, *b)* extensão da memória principal, *c)* uma combinação de ambos os casos.

Foi mostrado que em alguns casos o uso de memória remota para armazenar os dados de aplicações é mais rápido que armazenar em disco, especialmente quando o sistema está sofrendo *thrashing*. Alguma melhoria de desempenho pode ser notada através desta técnica, porém ela pode ser ineficiente em uma rede com muitas máquinas, ou seja, uma rede que apresente um grande tráfego de dados. Além disso, o artigo não apresenta soluções para tolerância a faltas, caso uma máquina na rede, que possua uma certa página, venha a falhar.

3.3.2 TPF & Adaptive Page Replacement

Os desenvolvedores do Linux tentaram resolver o problema de *thrashing* melhorando o desempenho da troca de páginas. A idéia utilizada foi fazer com que um ou mais processos com uso intensivo de memória liberem mais páginas de memória. Isso permite que outros processos construam seu conjunto de trabalho ou *working set* mais rapidamente e tenham maior utilizacão de ciclos de CPU [Jiang and Zhang, 2001].

Considerando o conflito de interesse entre a utilizacão de CPU e memória, esse artigo demonstra que os esforços nessa direção são limitados e foi proposta uma modificação [Jiang and Zhang, 2002] para o núcleo Linux 2.2, visando melhorar a capacidade dos algoritmos de troca para tentar diminuir o *thrashing*, monitorando dinamicamente as condições do sistema. Para que essa proposta funcione, o algoritmo de trocas de páginas deve ser modificado. Esta implementação consiste de duas rotinas:

- Monitoração: usada para monitorar dinamicamente a taxa de faltas de página em cada processo e a utilizacão da CPU do sistema.
- Proteção: utilizada para ajustar o mecanismo de troca de página quando a utilizacão da CPU for menor que um certo limite e quando a taxa de faltas de página de mais de um processo ultrapassar um limite. Isto garante um privilégio para um determinado processo, ou seja, o processo em questão não fará *swap* enquanto estiver no modo de proteção.

A rotina de monitoração também verifica se um dado processo diminuiu sua taxa de faltas até um certo nível, caso afirmativo o privilegio é então removido. A Figura 3.2 descreve a transição entre os três estados possíveis. No estado normal, nenhuma atividade de monitoramento é executada e o algoritmo de trocas de páginas tradicional do Linux é mantido. As grandezas CPU_L e CPU_H apresentadas na figura representam respectivamente um limiar mínimo e máximo de uso da CPU. A grandeza PF_L representa um limiar mínimo de faltas de página.

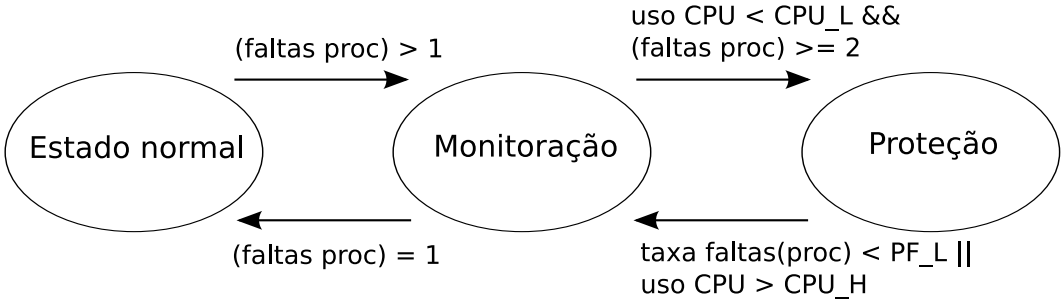

Figura 3.2: Transição entre os estados [Jiang and Zhang, 2001]

Nessa implementação, sempre é selecionado o processo que tem a maior possibilidade de faltas de página, o que permite que esse processo seja privilegiado mais que os outros. Este método é eficiente apenas quando a alocação de memória não excede a memória física. Caso contrário, acaba atuando como uma fila seqüencial ou FIFO (*First in First Out*). Além disso, essa implementação requer modificações no núcleo do Linux.

3.3.3 Medium-term scheduler

Este artigo [Reuven and Wiseman, 2006] descreve uma técnica que modifica o escalonador tradicional do Linux 2.6. Essa modificação ajuda o sistema operacional a fazer menos trocas de páginas, assim minimizando as paradas do sistema causadas por *thrashing*. O conjunto de todos os processos na memória é dividido em grupos, onde o tamanho total de memória usada por cada grupo é o mais próximo do tamanho da memória física disponível. Esse problema da melhor divisão entre os processo e grupos é conhecido como o problema do empacotamento (*bin packing*) [Scholl et al., 1997].

Esse novo escalonador, adicionado ao Linux, opera como um escalonador intermediário. Ele é responsável por colocar os grupos na fila de processos prontos do Linux usando uma política *round-robin*. O escalonador tradicional do Linux faz o escalonamento dentro de cada grupo da mesma maneira como é originalmente feito em qualquer sistema Linux. A fatia de tempo de cada grupo nesse escalonador intermediário será maior que o tempo médio alocado aos processos pelo escalonador tradicional do Linux. Os processos na memória física não poderão causar *thrashing* durante a execução do seu grupo, porque seu tamanho total não é maior que o tamanho da memória física disponível.

Para o cálculo do tamanho de um grupo são obtidas informações do sistema de arquivos */proc*, como a quantidade de memória residente utilizada por cada processo. O escalonador intermediário soma total de memória usado pelos processos ativos no sistema e divide essta soma pela quantidade de memória física disponível, para definir o número de grupos ou *bins*.

Nessa implementação, a fatia de tempo do grupo pode ser de meio segundo ou um segundo, enquanto o escalonador tradicional do Linux trabalha com milissegundos para cada processo dentro de um grupo. Os processos que não fazem parte do grupo em execução são mantidos em uma fila distinta, de modo que o escalonador tradicional do Linux não possa vê-los. Quando o último grupo termina sua execução, o escalonador intermediário reconstrói os grupos levando em conta os estados dos processos, como processos parados ou novos processos. Os processos que possuem memória compartilhada entre si são colocados no mesmo grupo. Os processos interativos são tratados de maneira diferente, sendo mantidos em todos os grupos, porém com uma fatia de tempo reduzida. Todavia, demora ao menos uma rodada para que um processo seja reconhecido e tratado como interativo. Os processo de tempo-real (processos que fazem parte da fila *SCHED_RR*) também são colocados em todos os grupos, para que possam ser escalonados rapidamente. A prioridade média para cada grupo é calculada de modo que um grupo com alta prioridade receba uma maior fatia de tempo.

No começo da execução de cada grupo ocorre um *swap* intensivo, porque as páginas dos processos do novo grupo precisam ser colocadas na memória. A fatia de tempo de cada grupo não é dinâmica, o que prejudica o desempenho se o grupo possui poucos processos. Além disso, o escalonador do Linux foi modificado para se atingir esses resultados.

3.4 Conclusão

Este capítulo apresentou o conceito de *thrashing* e os modelos de tratamento existentes nos sistemas operacionais estudados. Também foram descritas as principais propostas ou mecanismos avançados que visam minimizar o *thrashing*.

Pode-se perceber que o tratamento de *thrashing* é limitado nos sistemas operacionais avaliados e também existem poucos artigos disponíveis na literatura a respeito. A Tabela 3.1 apresenta um resumo dos mecanismos de tratamento de *thrashing* existentes nos sistemas operacionais avaliados. Os mecanismos avançados de gerência de memória descritos apresentam um resultado satisfatório, no entanto nenhum deles oferece uma solução definitiva para o problema. Alguns mecanismos apresentados modificam o núcleo do sistema operacional, o que dificulta a portabilidade desses mecanismos para outros sistemas.

Tabela 3.1: Mecanismos de tratamento de *thrashing*

FreeBSD	O último processo que usou a CPU fica bloqueado por 20 segundos. Nesse intervalo, suas páginas são liberadas para outros processos.
Linux	Um processo é privilegiado com um *token*, enquanto o processo estiver com esse *token* não sofrerá *swap*.
OpenSolaris	Trabalha com três estados. No estado mais crítico (*hard swap*), o sistema descarrega módulos de núcleo sem uso e move processos inteiros para o disco.
Windows XP	Na documentação analisada não foi encontrada menção a uma política explícita para *thrashing*.

O capítulo seguinte apresenta o conceito de desempenho da gerência de memória e um resumo das principais ferramentas de *benchmark* de memória existentes no mercado.

Capítulo 4

Benchmarking de memória

Este capítulo apresenta uma breve descrição do *benchmarking* de memória. Também apresenta o que é possível medir nesse tipo de *benchmark*, bem como as principais ferramentas de *benchmark* de memória existentes no mercado.

4.1 Benchmarking

Um *benchmarking* computacional não é nada mais que executar um conjunto de operações com o objetivo de avaliar o desempenho de um objeto específico, normalmente executando uma série de testes ou ensaios sobre esse objeto. Geralmente um *benchmark* é associado com a avaliação de desempenho de hardware de um computador, como o desempenho de operações de ponto flutuante de uma CPU. No entanto, também pode ser aplicado a software como, por exemplo, em compiladores ou sistemas de banco de dados. Ferramentas de *benchmark* em geral, oferecem um método de comparação de desempenho entre diferentes arquiteturas e sistemas [Musumeci and Loukides, 2002].

Os *benchmarks* geralmente são desenvolvidos para "imitar" o comportamento de um determinado componente ou sistema e podem ser classificados em:

- *Benchmarks* sintéticos: são programas especialmente desenvolvidos para impor um determinado comportamento no objeto em avaliação. Esse tipo de *benchmark* é mais usado no teste de componentes individuais, como CPU, disco rígido ou memória.
- *Benchmarks* de aplicação: avaliam programas já existentes como, por exemplo, editores de texto ou banco de dados.

4.1.1 *Benchmark* de memória

Ferramentas para análise de desempenho de memória podem ser classificadas basicamente de acordo dois parâmetros de operação:

- Largura de banda: ou *bandwidth*, significa a taxa de leitura e escrita de dados na memória. A unidade geralmente utilizada é bytes por segundo, no entanto isto pode variar, depenando do tamanho dos dados utilizados. No caso específico de sistema de memória é Megabye por segundo. A movimentação de dados é fundamental para o desempenho de

um sistema. O *benchmark* de largura de banda tenta medir a capacidade, ou *throughput*, que um sistema possui na movimentação de dados na memória.

- Latência: é definida pelo tempo de resposta entre a requisição inicial de um byte na memória e a obtenção deste byte. Se esse dado não estiver no cache do processador, leva mais tempo para obter estes dados porque o processador precisa se comunicar com a memória. Latência é uma medida fundamental para medir a velocidade da memória. Quanto menor a latência, mais rápida é a reação do sistema operacional. Ela não pode ser confundida com a largura de banda, que mede a "vazão" da memória. A unidade da latência geralmente é medida em microssegundos.

4.2 Principais ferramentas de *benchmark* de memória

Esta seção descreve as principais ferramentas de *benchmark* de memória existentes no mercado. Vale lembrar que a escolha destas ferramentas de *benchmark* foi baseada na sua portabilidade, ou seja, ferramentas que rodam em mais de uma plataforma de hardware ou em mais de um sistema operacional e que possuem código fonte aberto, para que possa ser estudado quais as funções ou chamadas de sistemas estão sendo utilizadas.

Esses parâmetros descritos acima são fundamentais para que possa ser desenvolvida uma nova ferramenta de *benchmark* que consiga avaliar o comportamento de um sistema operacional sob condições de *thrashing*, pois, as ferramentas estudadas não são destinadas a avaliar situações de *thrashing*, apenas uso normal da memória.

4.2.1 Bandwidth

Bandwidth [bandwidth, 2008] é um programa de *benchmark* escrito em linguagem C, que visa medir a largura de banda de memória. Ele pode medir: *a)* leitura e escrita sequencial do cache L2; *b)* velocidade de leitura e escrita da memória RAM; *c)* velocidade de leitura e escrita da memória de video; *d)* velocidade de algumas funções como *memset, memcpy e bzero*.

Este programa permite confirmar algumas expectativas, baseadas no senso comum, como: *a)* a leitura na memória é mais rápida que a escrita em memória; *b)* o tempo de leitura em cache L2 é próximo ao tempo de leitura do cache L1; *c)* o tempo de escrita em cache L2 é mais lento que tempo do cache L1; *d)* a leitura e escrita em cache L2 á mais rápida que a RAM. No entanto, esse programa avalia apenas a largura de banda e assim como as demais ferramentas apresentadas, ele não avalia situações de *thrashing* no sistema.

4.2.2 CacheBench

O programa CacheBench [Mucci et al., 1998] está disponível livremente na Internet e faz parte de um pacote de *benchmark* chamado LLCBench que funciona em sistemas Unix. Ele é utilizado para avaliar o desempenho da hierarquia de memória de um computador. Seu foco é parametrizar o desempenho de vários níveis de *cache* dentro e fora do processador. O desempenho é medido pela largura de banda, em Megabytes por segundo.

Atualmente são incorporadas nesse sistema uma série de medições, cada uma é executada de maneira repetida e acessam dados, geralmente através de um vetor de tamanho variável.

O tempo é medido para cada tamanho de vetor e o produto desses tamanhos e do número de iterações dá o total de dados acessados em bytes. Basicamente as medições que podem ser feitas pelo CacheBench são: *a)* leitura em *cache*; *b)* escrita em *cache*; *c)* leitura/modificação/escrita em *cache*; *d)* uma repetição dos três primeiros testes, porém ajustados para um melhor desempenho; *e)* função *memset()*, para inicializar regiões de memória; *f)* função *memcpy()*, para copiar regiões de memória.

A ferramenta realiza seis testes. Os quatro primeiros testes são propostos para avaliar a eficiência de uso da memória pelo código binário, que é consequência direta da qualidade do compilador que o gerou. Os últimos dois testes são propostos apenas para comparação, porque são rotinas geralmente utilizadas em aplicações. Todas essas medições executam por um período de tempo fixo.

O resultado do *benchmark* efetuado pelo CacheBench está diretamente relacionado com o *cache* de memória. Nessa ferramenta, assim como as demais aqui apresentadas, também é inexistente a preocupação com o *thrashing* ou mesmo com o *swapping* do sistema.

4.2.3 LMbench

O LMbench [McVoy and Staelin, 1996] oferece um conjunto de ferramentas para medir os problemas de desempenho mais comuns, encontrados pelas aplicações de um sistema. É capaz de rodar na maioria dos sistemas da família Unix e está disponível livremente na Internet, sob a licença GPL (*GNU Public License*). O LMbench consegue medir a latência e a largura de banda de dados entre o processador e a memória, a rede, o sistema de arquivos e o disco. O foco nessas áreas é porque os problemas de desempenho geralmente são causados por problemas de latência ou problemas largura de banda, ou a combinação de ambos.

Nessa ferramenta, a largura de banda pode ser medida através de diferentes métodos, como:

- Avaliação das funções de cópia, leitura e escrita com diferentes tamanhos de dados.
- Avaliação do IPC (*InterProcess Communication*) através da criação, de *pipes* e *sockets*, e transferência de dados também com tamanhos variados.
- Avaliação do *cache* de páginas de disco, que pode medido através de funções *read* e *mmap*. A interface *read* copia dados do *cache* do sistema de arquivos do núcleo para um *buffer* de um processo. A interface *mmap* oferece um meio de acessar o cache de arquivos do núcleo sem copiar os dados, sendo implementada pelo mapeamento de um arquivo em memória.

Já a latência, pode ser medida através dos seguintes métodos:

- Leitura de memória, onde um ponteiro pode ser incrementado até atingir um determinado tamanho em um vetor.
- Tempo de chamadas de sistema, onde pode ser medido a escrita repetida de uma palavra no */dev/null*, um dispositivo que simplesmente descarta os dados recebidos. Assim é possível medir a eficiência da chamada de sistema *write*.
- Tempo de tratamento de sinais, que pode ser medido pela instalação de um tratador de sinal, onde repetidamente um sinal é enviado para o mesmo processo.

- Custo de criação de um processo, por exemplo, através de uma função *fork*.
- Troca de contexto, ou seja o tempo necessário para salvar o estado de um processo e restaurar o estado de um outro processo.
- A latência de IPC pode ser medida pela passagem de uma pequena mensagem, de ida e volta entre dois processos. Podem ser usados *pipes* e *sockets* TCP e UDP.
- A latência do sistema de arquivos é definida pelo tempo necessário para criar ou apagar um arquivo vazio. Um laço no LMbench cria 1.000 arquivos e depois os apaga.
- A latência de disco, pode ser avaliada através de um grande número operações fazendo leitura e transferências sequenciais de alguns bytes, além disso, também pode ser medida a carga do processador associada com cada operação de disco.

O LMbench mede apenas a capacidade de transferência de dados entre o processador, cache, memória, rede e disco. Condições de *thrashing* e *swapping* não são levadas em consideração. O objetivo final do LMbench é comparar máquinas ou plataformas de hardware diferentes, ao contrário do trabalho aqui apresentado cujo, objetivo é comparar diversos sistemas operacionais sob as mesmas condições de *thrashing* e *swap* na mesma plataforma de hardware.

4.2.4 nBench

O nBench [nbench, 2008] é um programa que foi portado do BYTEmark benchmark da revista BYTE Magazine para o Unix. Ele explora algumas capacidades do sistema como CPU e largura de banda de memória. O código usado nos testes deste programa simula algumas operações reais utilizadas por aplicações populares de escritório.

Todo o programa de *benchmark* roda em menos de 10 minutos, e compara o resultado da máquina em questão com dois outros *benchmark* pré definidos. O código fonte deste programa foi compilado com sucesso em várias plataformas e sistemas operacionais diferentes, incluindo SunOS, MS-DOS e Linux. Esta ferramenta é composta por um conjunto de 10 testes que incluem: *a)* avaliação de vetores com valores inteiros e caracteres; *b)* funções de manipulação de bits; *c)* cálculos baseados funções matemáticas e algoritmos de compressão e cifragem de dados;

O nBech não avalia o desempenho do sistema sob condições de *thrashing*.

4.2.5 STREAM

STREAM [Stream, 2008] é uma ferramenta simples e mede o tempo necessário para copiar regiões de memória. Ou seja, ela mede a largura de banda no “mundo real”, e não o que é chamado de largura de banda de “pico” ou largura máxima.

Existem três meios de contar quantos dados são transferidos em uma única operação:

- Método via hardware, conta quantos bytes são transferidos fisicamente, porque o hardware pode mover um número diferente de bytes que o usuário especificou. Isto pode ocorrer devido ao cache de hardware.

- Método *bcopy*, conta quantos bytes são movidos de um local da memória para outro. Se a máquina levar 1 segundo para ler um certo número de bytes de um lugar e mais um segundo para gravar o mesmo número de bytes em outro lugar, a largura de banda resultante é esse número em bytes por segundo.

- Método STREAM, conta quantos bytes o usuário especificou para serem lidos e e quantos para serem escritos. Isto é precisamente duas vezes o número obtido pelo método *bcopy*.

STREAM é software livre e seu código fonte compila e roda em diversas plataformas de hardware e sistemas operacionais diferentes. Essa ferramenta também não oferece nenhum tratamento específico para avaliação de sistemas sob *thrashing*.

4.2.6 Unixbench

O UnixBench [UnixBench, 2008] vem de uma data bastante antiga, 1983, e foi melhorado pela revista Byte Magazine. É uma ferramenta de *benchmark* de propósitos gerais que oferece uma avaliação básica do desempenho do sistema.

O UnixBench apresenta alguns micro *benchmark* como:

- Cópia de arquivos, que mede a taxa de dados transferidos de um arquivo para outro, com diversos tamanhos de *buffers*.

- O número de vezes que um processo pode ler e escrever alguns bytes em um *pipe*.

- O tempo que um processo leva para criar outro processo via função *fork*.

- O número de vezes que um processo pode iniciar diversas cópias de um script shell.

O UnixBench também pode fazer alguns medições de desempenho gráfico em 2D e 3D. As avaliações de memória foram removidas já nas primeiras verões do UnixBench. o UnixBench também não trata da questão de *thrashing* do sistema.

SPEC

Uma organização, sem fins lucrativos, chamada SPEC (*Standard Performance Evaluation Corporation*) mantém um conjunto padrão de *benchmarks* que podem ser aplicados aos computadores modernos. Esta organização desenvolve ferramentas de *benchmark* e divulga os resultados obtidos pelos membros associados [Henning, 2006].

A ferramenta fornecida pela SPEC, chamada SPEC CPU, é multiplataforma e funciona em diversos sistemas operacionais da família Unix e Windows. Com essa ferramenta, pode-se medir o desempenho do processador, desempenho do compilador e largura de banda de memória.

Esta ferramenta também não avalia o desempenho do sistema sob condições de *thrashing*.

4.2.7 Outras ferramentas

Esta seção apresenta um resumo de outras ferramentas de *benchmark*. As ferramentas descritas aqui não são compatíveis com as restrições impostas no início deste capítulo, geralmente porque são ferramentas comerciais e porque rodam em apenas uma família de sistemas operacionais.

Bapco SYSmark

SYSmark [Bapco, 2008] é uma ferramenta comercial de *benchmark* para a plataforma Windows que tem como foco principal o teste de aplicações desktop como, editores de texto, leitores de e-mail, navegadores de Internet.

Esse sistema foi criado para simular a interação humana com o computador. Nenhum teste específico de memória é aplicada nessa simulação. Esta ferramenta também não avalia situações de *thrashing* no sistema.

Futuremark PCMark

PCMark [Futuremark, 2008] é uma ferramenta de *benchmarking* comercial, que consegue medir componentes específicos da família de sistemas operacionais Windows, além de medir componentes de hardware que rodam sob a plataforma Windows.

Diferentes componentes podem ser testados como, CPU, memória e desempenho de gráficos 2D e 3D. A memória pode ser medida pela largura de banda e latência.

Assim como as demais ferramentas aqui apresentadas, esta ferramenta também não avalia situações de *thrashing* no sistema.

SiSoftware Sandra

Sandra [Sissoftware, 2008] é uma ferramenta de *benchmark* comercial, da empresa SiSoftware, que funciona apenas na família Windows. Ela é baseada na ferramenta STREAM, descrita anteriormente.

Enquanto que a ferramenta STREAM mede apenas largura de banda, com a ferramenta Sandra é possível obter diversas informações do sistema como, CPU, placa de vídeo, placa de som, placa de rede, largura de banda de memória e componentes internos do Windows.

Esta ferramenta também não avalia situações de *thrashing* no sistema.

4.3 Conclusão

Este capítulo apresentou brevemente o *benchmark* de memória e as principais ferramentas multiplataforma de *benchmark* existentes. Estas ferramentas geralmente buscam avaliar a largura de banda e a latência no acesso a memória. Nenhuma das ferramentas analisadas oferece mecanismos para a medição do desempenho de sistemas em condições de *thrashing*. Ferramentas de *benchmark* são geralmente mais conservadoras e avaliam apenas um componente específico do sistema. Uma avaliação de *thrashing*, requer uma análise mais detalhada de outros componentes do sistema, além disso, a situação de *thrashing* pode causar o congelamento

de alguns processos que nem sempre pode ser útil para os usuários de em uma ferramenta genérica de *benchmark*. Por isso, se faz necessário o uso de uma ferramenta específica para avaliar o comportamento dos sistemas em condições de *thrashing*.

O capítulo a seguir apresenta o estudo dos sistemas operacionais sob condições de *thrashing*. São apresentados o objetivo e a metodologia, bem como os resultados obtidos com os experimentos realizados.

Capítulo 5

Estudo de sistemas operacionais sob *thrashing*

5.1 Objetivo

O objetivo principal deste trabalho é avaliar quantitativamente o comportamento de alguns sistemas operacionais de mercado em condições de *thrashing*. Para tal, foram selecionados alguns sistemas operacionais de amplo uso e foi definido um *benchmark* próprio, que conduz o sistema, de um estado normal de operação, ao *thrashing* e depois de volta a uma situação normal. Esse *benchmark* possui vários parâmetros ajustáveis, cuja influência também é avaliada. Com isso, pode-se avaliar como cada sistema operacional gerencia a memória durante o *thrashing* e quão rápido consegue se recuperar dele, quando a demanda por memória retornar aos níveis normais. Além desta avaliação quantitativa, este trabalho também considera a impressão subjetiva, ou seja a interação, do usuário de um sistema sob *thrashing*.

Nesse capítulo serão apresentados os sistemas operacionais avaliados, a metodologia de experimentação, o programa de *benchmark* desenvolvido e seus parâmetros, as eventuais adaptações do mesmo para cada sistema operacional, a forma de medição dos parâmetros operacionais em cada sistema e os resultados obtidos.

5.2 Os sistemas operacionais escolhidos

Foram selecionados quatro sistemas operacionais desktop de amplo uso para que se possa avaliar o comportamento de cada um destes sistemas sob condições de *thrashing*. Os sistemas escolhidos foram o FreeBSD, o Linux, o OpenSolaris e o Windows XP.

Além de serem os principais sistemas operacionais desktop disponíveis no mercado, estes sistemas possuem características muito distintas em suas estrutura de núcleo, principalmente seus mecanismos de gerência de memória. No entanto, este sistemas apresentam características em comum, como o funcionamento na mesma plataforma de hardware. Esta característica foi fundamental para a escolha destes sistemas. Outro sistema de amplo uso, o Apple Mac OS X, um sistema baseado no núcleo BSD, por exemplo, não pode ser incluído na lista pois requer uma plataforma de hardware específica pra rodar.

A arquitetura Intel x86 foi escolhida como plataforma de hardware, justamente por ser uma plataforma de fácil acesso, e por possuir um bom suporte de hardware, como gerenciadores de dispositivos, nos sistemas operacionais escolhidos.

5.3 Metodologia

A metodologia usada para a avaliação consistiu em provocar uma situação de *thrashing* controlada em cada um dos sistemas operacionais sob estudo e avaliar o comportamento dos mesmos, através dos seguintes indicadores: percentual de uso de processador em modo usuário, percentual de uso de processador em modo sistema, número de páginas escritas em disco (*page-outs*) e número de páginas lidas do disco (*page-ins*). Os sistemas foram observados antes, durante e depois do *thrashing*, para avaliar sua rapidez de reação e também de recuperação do mesmo.

A carga de trabalho usada para provocar o *thrashing* consiste em um conjunto de *N* processos, denominados *processos consumidores*. Cada processo consumidor aloca uma área de memória própria de tamanho razoável (100 MB são alocados com apenas uma chamada à função *malloc()*) e executa ciclos de escrita intensiva nessa área, alternados com períodos de espera. Deste modo, é forçado uma localidade de referência baixa, pois as escritas são feitas em posições aleatórias.

O comportamento de cada processo consumidor e seus parâmetros ajustáveis são descritos com mais detalhes na seção 5.4. Além disso, para cada sistema operacional foi construído um *processo de medição*, que efetua a medição periódica dos indicadores de desempenho sob estudo. Esses processos de medição são específicos para cada sistema operacional, pois as interfaces de acesso às informações de núcleo em cada sistema são distintas.

O programa usado como carga de trabalho é definido com detalhes na seção 5.4. A forma de medição de informações de núcleo empregada em cada sistema é detalhada na seção 5.5. A seção 5.6 descreve a plataforma de hardware e as ferramentas de software usadas nos experimentos.

5.4 O programa de *benchmark*

Foi desenvolvido um programa de *benchmark* específico com o objetivo de gerar um *thrashing* controlado, que conduz o sistema operacional, de um estado normal de operação, ao *thrashing* e depois de volta ao estado normal. Para isso, foi feita uma simulação onde diversos processos começam a consumir memória aleatoriamente forçando o mecanismo de gerência de memória a fazer muita paginação, e consequentemente causando *thrashing* do sistema. As funções aleatórias utilizados no programa também forçam uma localidade de referência muito baixa, de forma a não favorecer nenhum sistema específico. Através deste *benchmark* é possível avaliar quantitativamente o comportamento de alguns sistemas operacionais em condições de *thrashing*. A ferramenta de *benchmark* desenvolvida possui três componentes:

- Processo pai, é responsável por criar *N* processos no sistema.
- Processo consumidor, que é responsável por alocar uma certa quantidade de memória e por escrever alguns valores aleatórios em posições aleatórias de memória.

- Processo de medição, que é responsável por obter informações como consumo de CPU e memória do sistema operacional.

O programa de *benchmark* possui vários parâmetros ajustáveis, cuja influência sobre o fenômeno de *thrashing* pode ser avaliada. Os principais parâmetros cuja influência foi observada durante os experimentos são descritos a seguir. Esses parâmetros foram escolhidos pois influenciam diretamente na maneira como a memória é acessada.

- Avaliação do número de escritas na memória: o número de escritas (W) se refere ao número de vezes no qual o processo consumidor escreve na memória. O processo consumidor é descrito com mais detalhes na seção 5.4.

- Avaliação da duração da espera entre ciclos de escrita na memória: a duração da espera entre ciclos de escritas (t_w) é o período no qual o processo consumidor fica dormindo até começar um novo ciclo de escritas.

- Avaliação da duração da espera entre dois processos consumidores: a duração da espera entre dois processos (t_c) é o período de espera para o início da atividade de um novo processo.

A duração total (T) de cada experimento é calculada pelo número de processos (N) multiplicado pela duração da espera entre dois processos (t_c). Como são duas etapas, de criação dos processos e de finalização dos processos, este valor ainda é multiplicado por dois. Este cálculo é descrito pela seguinte fórmula $T = (N \times t_c) \times 2$.

Também foi acrescentado na fórmula um intervalo inicial que foi utilizado em todas as avaliações. Ele é definido como o período de espera entre a criação dos processos, instante T=0, e o início da atividade de cada processo individual. Ou seja, todos os processos são criados no instante T=0, porém o primeiro processo espera 10 segundos antes de começar suas operações. Esse intervalo foi utilizado para que o sistema se estabilize, após a criação dos processos consumidores. Por exemplo, considerando 25 processos, com intervalo de 30 segundos entre a criação de cada processo o tempo total é de 1510 segundos, $(10 + ((25 \times 30) \times 2))$.

Um resumo de todos os valores utilizados nas medições é descrito na Tabela 5.1 a seguir. Esses valores foram escolhidos após um longo processo de experimentação e influenciam diretamente o comportamento dos sistemas.

Tabela 5.1: Valores utilizados nas medições

Número de processos (N)	25
Memória alocada (M)	100 MB
Número de escritas (W)	1.000, 10.000, 50.000
Duração da espera entre ciclos de escrita (t_w)	100ms, 500ms, 1.000ms
Duração da espera entre processos consecutivos (t_c)	1s, 10s, 30s

Basicamente o processo consumidor pode ser descrito pelo Algoritmo 1 que é apresentado a seguir. A Figura 5.1 ilustra o funcionamento desse algoritmo.

Os componentes, processo consumidor e processo de medição, da ferramenta de *benchmark* foram escritos em linguagem C, pois facilita a portabilidade de código. Com exceção do processo de medição na plataforma Windows, no qual foi utilizado um programa nativo da

Algoritmo 1 Processo consumidor de memória pc_i

1: N : numero total de processos consumidores
2: i : índice do processo corrente ($1 \leq i \leq N$)
3: W : número de escritas na memória em cada ciclo de escritas
4: t_p : duração da atividade de cada processo consumidor
5: t_c : duração da espera entre dois processos consumidores
6: t_w : duração da espera entre ciclos de escrita na memória
7: t : relógio do sistema (indica o instante atual)
8:
9: sleep (10) // aguarda que o sistema se estabilize
10: sleep ($i \times t_c$) // cada processo inicia em um momento próprio
11: *mem* = malloc (100×1024^2) // aloca 100 MB de memória RAM
12: $t_p = (N \times t_c) + t_c$
13: $t_f = t + t_p$ // data do fim da execução deste processo
14: **while** $t \leq t_f$ **do**
15: **for** $k = 1$ to W **do**
16: *val* = random ($0 \ldots 255$)
17: *pos* = random ($0 \ldots 100 \times 1024^2$)
18: *mem* [*pos*] = *val* // escreve valor aleatório em posição aleatória
19: **end for**
20: sleep (t_w) // espera entre ciclos de escritas
21: **end while**
22: free (*mem*) // libera a memória alocada

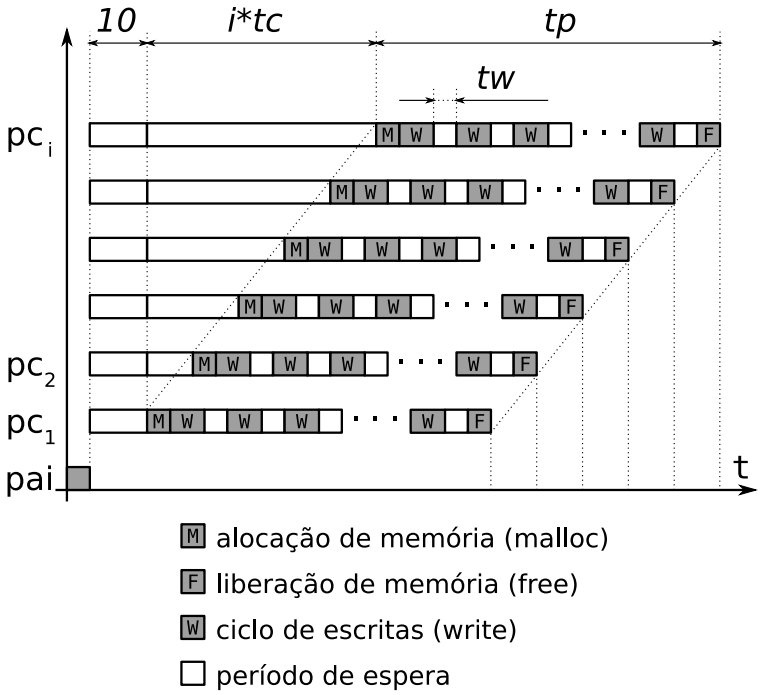

Figura 5.1: Funcionamento do programa de *benchmark*

plataforma. Já o processo pai foi escrito em linguagem Java, além de facilitar a portabilidade, suas funções de criação de processos permitem um maior controle sobre os processos criados.

O processo consumidor foi escrito de forma portável, ou seja, é exatamente o mesmo programa nos quatro sistemas avaliados. Para conseguir isso, foram utilizadas funções simples da biblioteca C, como *sleep()*, *random()* e *malloc()*. O processo pai não gera influência no resultado, pois ele simplesmente cria os processos consumidores e encerra sua execução.

5.5 Medição das informações do núcleo

O processo de medição precisou de ajustes em cada um dos sistemas avaliados, pois cada sistema oferece uma interface diferente para acesso aos dados do núcleo. Basicamente o processo de medição lê as informações de processador e memória de uma determinada interface e salva essas informações em um arquivo, a cada segundo. Os detalhes da interface de informações de núcleo provida por cada sistema são descritos a seguir.

5.5.1 Informações de núcleo no sistema FreeBSD

O sistema operacional FreeBSD dispõe de uma ferramenta de configuração, denominada *sysctl*, permite fazer modificações ou acessar informações de estado no núcleo do sistema operacional. Este estado é descrito de maneira similar a uma MIB (*Management Information Base*). Mais de 500 variáveis, que vão desde configurações TCP/IP a gerência de memória, podem ser consultadas ou configuradas através desta ferramenta.

A ferramenta *sysctl* foi utilizada nos experimentos para obter as informações em tempo real sobre a utilização de CPU e de memória do sistema. As variáveis consultadas referentes à utilização de CPU, representadas em *ticks*, são descritas a seguir:

- `kern.cp_time[CP_USER]` : representa o tempo de CPU gasto por processos de usuário.
- `kern.cp_time[CP_SYS]` : representa o tempo de CPU gasto por atividades de sistema.

As variáveis referentes à utilização de memória são descritas a seguir:

- `vm.stats.vm.v_swappgsin` : representa o número de *page in* ocorridos no sistema durante o último segundo.
- `vm.stats.vm.v_swappgsout` : representa o número de *page out* ocorridos no sistema durante o último segundo.

5.5.2 Informações de núcleo no sistema Linux

O sistema de arquivos */proc* [Cowardin, 1997] é um sistema de arquivos virtual que permite a comunicação entre o núcleo Linux e os processos de usuário, através da leitura ou escrita de arquivos virtuais. Estes arquivos apresentam uma série de informações sobre o estado atual do sistema. A maioria das ferramentas de medição de desempenho, como o *top* e *vmstat*, no Linux usam o sistema */proc* como fonte de informação.

Nos experimentos aqui descritos, as informações como o consumo de CPU por processos de usuário e por atividades de sistema foram obtidas do arquivo */proc/stat*. Este arquivo apresenta estas informações em *ticks*. Já as informações referentes à memória foram obtidas do arquivo */proc/vmstat*. Este arquivo possui diversos campos que apresentam estatísticas de memória do sistema. Os campos utilizados pela ferramenta de *benchmark* são descritos a seguir:

- `pgpgin` : representa o número de *page in* ocorridos no sistema durante o último segundo.
- `pgpgout` : representa o número de *page out* ocorridos no sistema durante o último segundo.

5.5.3 Informações de núcleo no sistema OpenSolaris

O OpenSolaris oferece um conjunto de funções e estruturas de dados para que informações de vários módulos do núcleo estejam disponíveis para o usuário. Desta maneira pode-se obter informações e dados estatísticos do OpenSolaris sem precisar acessar diretamente a área de memória do núcleo. Esta estrutura é chamada *kstat*, e é representada pelo dispositivo */dev/kstat*. Apenas programas com acesso privilegiado ou *root* podem acessar estas informações.

A função *kstat_data_lookup* foi utilizada para procurar por um nome de variável em específico. As seguintes variáveis, disponíveis no *kstat*, foram utilizadas para se obter os valores, em *ticks*, de uso da CPU:

- `cpu_ticks_user` : representa o tempo de CPU gasto por processos de usuário.
- `cpu_ticks_kernel` : representa o tempo de CPU gasto por atividades de sistema.

As variáveis utilizadas, pela ferramenta de *benchmark*, referentes ao uso de memória são descritas a seguir. Estes dados também foram obtidos através da função *kstat_data_lookup*.

- `pgswapin` : representam o número de *page in* ocorridos no sistema durante o último segundo.
- `pgswapout` : representam o número de *page out* ocorridos no sistema durante o último segundo.

5.5.4 Informações de núcleo no sistema Windows XP

O Windows, ao contrário dos demais sistemas operacionais sob estudo, não apresentou uma interface de fácil acesso para coletar os dados do núcleo. Por este motivo, se fez necessário o uso de uma ferramenta nativa do Windows que será descrita a seguir.

O Monitor do Sistema no Windows, chamado de *perfmon*, permite que sejam coletados dados de desempenho em tempo real de um ou mais computadores. O *perfmon* classifica cada recurso do computador como um objeto, por exemplo, objeto CPU ou objeto memória. Cada um desses objetos possui uma lista de contadores. Para o objeto CPU, por exemplo, existem os contadores de consumo de CPU por processos de usuário, por atividades de sistema, e assim por diante. Os contadores utilizados, na ferramenta de *benchmark*, para o objeto CPU, representado em (%), e para o objeto memória foram os descritos a seguir:

- `% User Time` : representa o tempo de CPU gasto processos de usuário.
- `% Privileged Time` : representam o tempo de CPU gasto por atividades de sistema.
- `Pages Input/sec` : representam o número de *page in* ocorridos no sistema durante o último segundo.
- `Pages Output/sec` : representam o número de *page out* ocorridos no sistema durante o último segundo.

5.6 Ambiente de experimentação

O equipamento utilizado nos experimentos foi um PC IBM ThinkCentre S50, com um processador Intel Pentium 4 2.6 GHz, placa mãe Intel 865G com componentes *on-board* e barramento frontal de 533 MHz. A máquina possui 1 GB de memória RAM Kingston DDR SDRAM PC2700, com *clock* de 333 MHz e tempo de acesso de 6ns. O disco rígido utilizado foi um Seagate Barracuda ST340014A de 40 GB IDE 7200RPM, com conector ATA-100. Este disco conta com 2 MB de tamanho de buffer interno. A taxa de transferência deste disco é de 100 MB/s e o tempo de busca é de 8.5ms[1].

O disco foi particionado em 4 partições iguais. Um espaço de 2 GB foi criado em arquivo para a área *swap* em cada partição, com a finalidade de deixar a estrutura dos sistemas mais próxima possível uma vez que a plataforma Windows não utiliza partição *swap* como nos sistemas derivados do Unix. Cada sistema operacional então conta com 3 GB de espaço de endereçamento de memória. A Tabela 5.2 descreve a máquina utilizada. A placa de rede, bem como outros dispositivos, a não ser teclado e mouse, estavam desconectados.

Tabela 5.2: Descrição da máquina

Máquina	IBM ThinkCentre S50
Placa mãe	Intel 865G e componentes On-board
Barramento	533 MHz
Processador	Intel Pentium 4
Clock	2.66 GHz
Cache L1	32 KB
Cache L2	512 KB
Disco rígido	40 GB IDE ATA-100 7200RPM
Memória RAM	1 GB DDR SDRAM
Clock de memória	333MHz

A Tabela 5.3 descreve o modelo de particionamento utilizado e a formatação do sistema de arquivos para cada sistema operacional avaliado. Esta formatação segue o padrão recomendado peles respectivos fabricantes/fornecedores dos sistemas. A partição do Windows foi configurada como inicializável, ou como partição de *boot*, e contém o gerenciador de *boot*.

O compilador escolhido foi o GCC (*GNU Compiler Collection*) [GCC, 2008], pois ele é amplamente utilizado e possui versões para todos os sistemas operacionais avaliados. Nenhuma configuração específica de otimização no compilador foi utilizada.

[1]Dados fornecidos pelo fabricante.

Tabela 5.3: Modelo de particionamento

Sistema Operacional	Partição	Sistema de arquivos
FreeBSD	3	UFS2
Linux	4	EXT3
OpenSolaris	2	ZFS 5.11-0.101
Windows XP	1	NTFS 5.1

As medições foram efetuadas a cada segundo, por um processo de medição, e salvas em arquivo. O processo de medição foi executado com prioridade normal, porém com permissões de super usuário (*root*) ou administrador do sistema. A máquina foi reiniciada após cada medição. É importante ressaltar que nenhum ajuste ou configuração de desempenho foi aplicado nos sistemas em estudo.

Foi feita uma instalação desktop default para cada sistema operacional e nenhuma outra atividade, a não ser as atividades default do sistema operacional, estava sendo executada durante a experimentação. A seguir são descritas as configurações usadas em cada sistema operacional sob estudo:

FreeBSD : os experimentos foram medidos no PC-BSD versão 7, com núcleo FreeBSD 7.0 e compilador GCC versão 4.2.1. O ambiente gráfico é o KDE 4. Após a carga do sistema operacional, 66 processos estavam na fila do processador e aproximadamente 630 MB de memória RAM estavam disponíveis (ou seja, memória livre) para processos de usuário, além disso, aproximadamente 70 MB estavam sendo sendo utilizadas pelo *cache* do sistema. Este sistema não apresentou consumo de memória *swap* antes de iniciar os experimentos. As informações descritas acima foram obtidas através dos comandos *top* e *vmstat*.

Linux : os experimentos foram efetuados na distribuição OpenSUSE 10.3 com núcleo 2.6.22 padrão e compilador GCC versão 4.2.1. O ambiente gráfico é o Gnome 2.20. Após a carga do sistema operacional, 62 processos estavam na fila do processador e aproximadamente 730 MB de memória RAM estavam disponíveis para processos de usuário, além disso, aproximadamente 140 MB estavam sendo sendo utilizadas pelo *cache* do sistema. O Linux também não apresentou consumo de memória *swap* antes de iniciar os experimentos. Estas informações foram obtidas através dos comandos *top* e *free*.

OpenSolaris : o OpenSolaris 2008.11 foi utilizado nos experimentos com compilador GCC versão 3.4.3. Na data que foram feitos os experimentos, esta era a única versão do GCC disponível para esse sistema. Como foram utilizadas funções muitos simples de alocação de memória, esta versão não tem influência nos resultados. O ambiente gráfico default do sistema é o Gnome 2.22. Após a carga do sistema operacional, 79 processos estavam na fila do processador e aproximadamente 370 MB de memória RAM estavam disponíveis para processos de usuário, além disso, aproximadamente 100 MB estavam sendo sendo utilizadas pelo *cache* do sistema. Este sistema não apresentou consumo de memória *swap* antes de iniciar os experimentos. Estas informações foram obtidas através dos comandos *top* e *vmstat*.

Windows : os experimentos foram medidos no Windows XP Profissional, com ServicePack 3 e compilador GCC (djgpp) versão 4.2.3. Após a carga do sistema operacional, 16 processos

estavam na fila do processador e aproximadamente 830 MB de memória RAM estavam disponíveis para processos de usuário, além disso, aproximadamente 70 MB estavam sendo sendo utilizadas pelo *cache* do sistema. Este sistema apresentou consumo de 80 MB de memória *swap* antes de iniciar os experimentos. Estas informações foram obtidas através do comando *taskmgr*.

A configuração de software usada em cada sistema operacional sob estudo está sumarizada na Tabela 5.4. As condições iniciais de operação de cada sistema estão sumarizadas na Tabela 5.5.

Tabela 5.4: Versões dos sistemas operacionais

Sistema Operacional	Núcleo	Ambiente gráfico	GCC
FreeBSD	7.0	KDE 4	4.2.1
Linux	2.6.22	Gnome 2.20	4.2.1
OpenSolaris	SunOS 5.11	Gnome 2.22	3.4.3
Windows XP	SP3	Windows XP	4.2.3

Tabela 5.5: Número de processos, RAM e *swap* utilizados antes dos experimentos

Sistema Operacional	Núm. de processos	RAM disponível / *cache*	*swap* utilizada
FreeBSD	66	630 MB / 70 MB	0
Linux	62	730 MB / 140 MB	0
OpenSolaris	79	370 MB / 100 MB	0
Windows XP	16	830 MB / 70 MB	80 MB

5.7 Resultados obtidos

Esta seção descreve os resultados obtidos através dos experimentos. Foi avaliada a influência dos parâmetros da seção 5.3, ou seja, o número de escritas na memória, a duração da espera entre ciclos de escrita na memória e a duração da espera entre a atividade dois processos. Para cada parâmetro, foram registrados os valores de consumo de CPU por processos de usuário e atividade de sistema e o número *page in* e *page out* durante os experimentos. Foram registrados os valores de consumo de CPU, pois quando ocorre muita paginação no sistema os processos ficam esperando muito tempo pelo uso da CPU.

As figuras foram plotadas com a ferramenta *gnuplot*, que inclui algumas rotinas para fazer interpolação e aproximação dos dados chamada *smooth*. Esta rotina foi aplicada a uma função que aproxima os dados com uma curva *Bezier*. Esse procedimento foi abordado porque existe uma forte variação entre medidas consecutivas, tornando os gráficos difíceis de ler e interpretar. Com a interpolação, os gráficos não representam os dados exatos medidos, mas sua tendência ao longo do tempo. A Figura 5.2 apresenta dois gráficos do mesmo experimento, um sem a suavização de *Bezier* e outro com essa suavização, para ilustrar esse problema. Todos os demais gráficos apresentados nesta dissertação são construídos usando essa suavização. A estabilidade dos resultados obtidos durante os experimentos é discutida na seção 5.8.

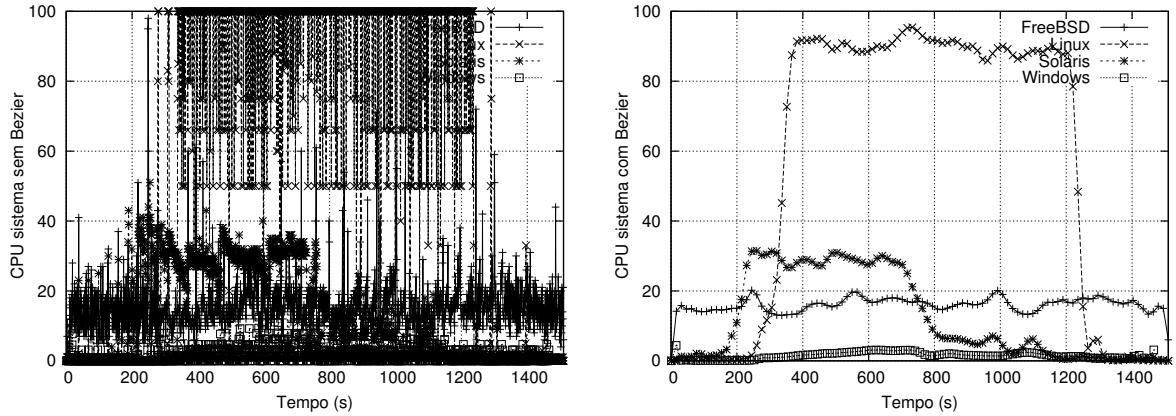

Figura 5.2: Plotagem dos dados brutos (esq.) e com suavização por interpolação de Bezier (dir.)

5.7.1 Influência do número de escritas

O comportamento dos sistemas em função do número de escritas (W) realizado em cada ciclo de escritas foi avaliado para 1.000, 10.000, e 50.000 escritas/ciclo. Foram criados 25 processos, cada um alocando 100 MB de memória RAM. O tempo de espera entre o ciclos de escritas (t_w) foi de 100ms e o tempo de espera entre duas atividades de processos (t_c) foi de 30 segundos. Os dados usados para a plotagem dos gráficos foram coletados com período de um segundo.

A figuras a seguir apresentam o consumo de CPU por processos de usuários e por atividades de sistema em função da variação do número de escritas em cada ciclo de escritas. A Figura 5.3 indica os resultados para 1.000 escritas/ciclo, a Figura 5.4 indica os resultados para 10.000 escritas/ciclo e a Figura 5.5 indica os resultados para 50.000 escritas/ciclo.

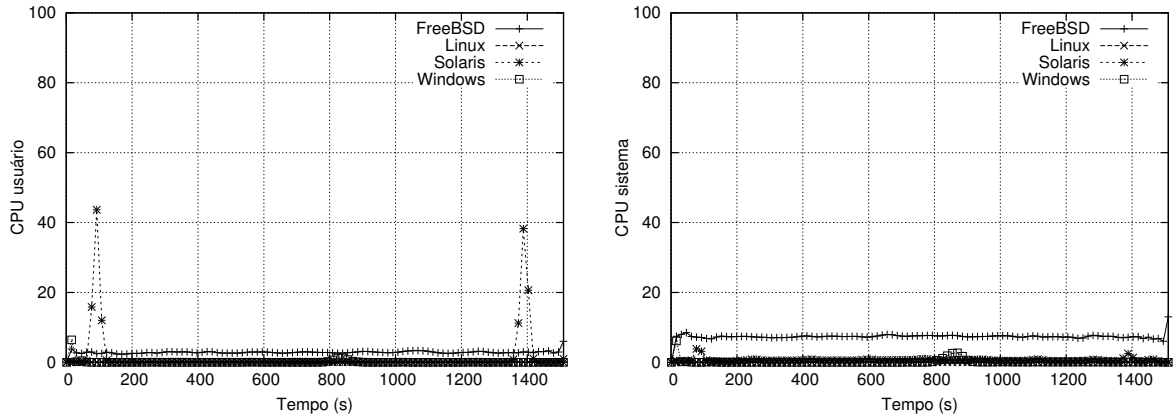

Figura 5.3: Consumo de CPU em modo usuário e sistema, para 1.000 escritas/ciclo

O consumo de CPU em modo usuário foi bastante modesto quando avaliado em 1.000 escritas/ciclo. Um destaque foi o sistema operacional OpenSolaris, consumindo um pouco mais que 40% de CPU quando os processos estão sendo ativados e quando eles estão terminando. O consumo de CPU em modo sistema avaliado em 1.000 escritas/ciclo também foi bastante modesto. O FreeBSD apresentou um consumo bastante estável, na faixa de 7% de consumo de CPU em modo sistema.

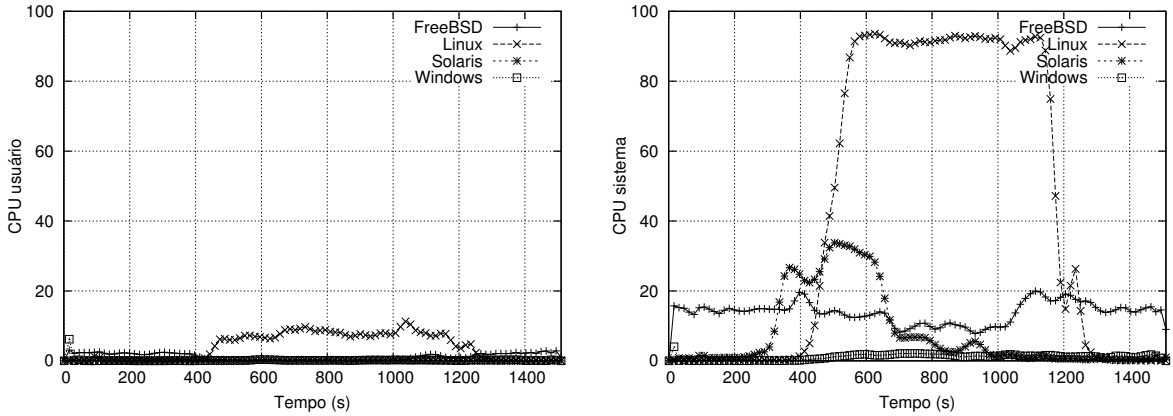

Figura 5.4: Consumo de CPU em modo usuário e sistema, para 10.000 escritas/ciclo

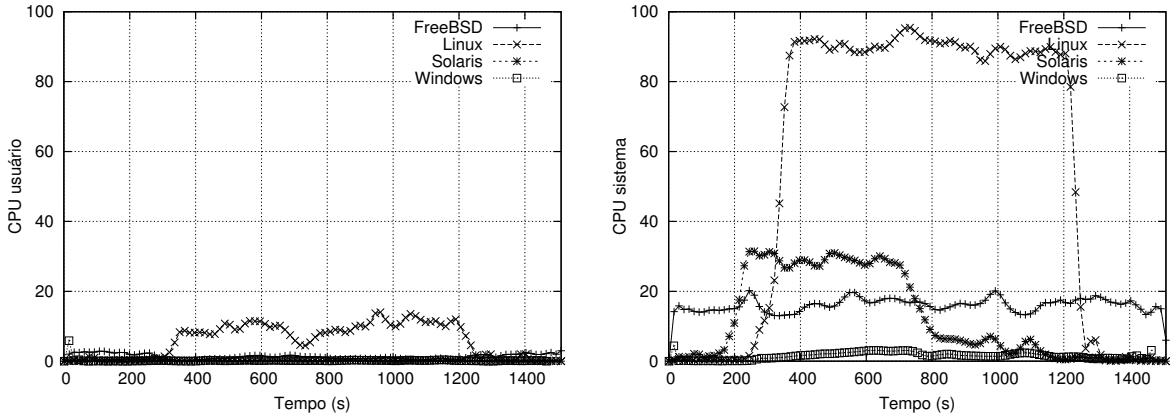

Figura 5.5: Consumo de CPU em modo usuário e sistema, para 50.000 escritas/ciclo

Quando avaliado em 10.000 e 50.000 escritas/ciclo, o comportamento individual também é bastante similar aos demais. É possível observar que, a não ser o Linux, os demais sistemas praticamente não consomem CPU em modo usuário. O consumo de CPU em modo usuário do Linux também foi bastante modesto, não passando de 14% de uso da CPU. No entanto, o Linux consome praticamente 90% de CPU com atividades de sistema neste mesmo período, conforme mostrado nas Figuras 5.4 e 5.5. Podemos considerar que o Linux consome 100% de CPU, somando o uso de CPU em modo sistema e em modo usuário, como apresentado nas figuras anteriores.

O Windows apresenta pouco consumo de CPU por processos em modo usuário, mesmo quando avaliado em 50.000 escritas/ciclo. Pode-se observar um consumo aproximado de 6% apenas quando os processos então sendo ativados. O FreeBSD novamente apresenta um consumo estável, com um máximo de 20% de consumo de CPU. Pode-se observar facilmente o surgimento de *thrashing* principalmente no sistema OpenSolaris, como mostram as Figuras 5.4 e 5.5. Esse sistema apresenta um consumo inicial de CPU, em modo sistema, de aproximadamente 30% que é praticamente interrompido logo após o intervalo (t) = 600. Após esse intervalo, todos os processos consumidores já estão ativos e apresentam uma influência maior sob o sistema, pois estão gerando mais faltas de páginas.

Pode-se observar que o Windows é o sistema que apresenta pouco consumo de CPU quanto está fazendo operações na memória. No entanto, um ponto a levar em consideração é que o Windows não conseguiu terminar de executar todos os processos dentro do tempo total (T) previsto, descrito na seção 5.3. Isto gerou atrasos na atividade de alguns processos, que permaneceram executando após esse tempo total.

As Figuras 5.6, 5.7 e 5.8 apresentam o número de *page in* e *page out*, ocorridos no sistema, sob as mesmas condições de variação do número de escritas/ciclo.

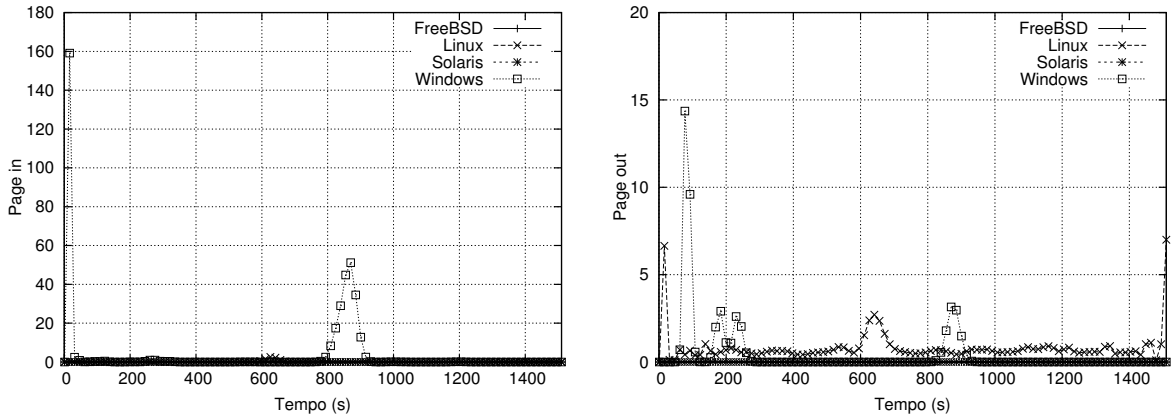

Figura 5.6: Número de page in/out, para 1.000 escritas

O número de *page in/out* é bastante modesto quando avaliado em 1.000 escritas/ciclo. O Windows apresentou um maior número nesta avaliação. Já quando avaliado em 10.000 e 50.000 escritas/ciclo, o número de *page in/out* é significativamente maior. O Linux é o sistema que apresenta maior número de movimentação de páginas entre o disco e a memória, atingindo picos de quase 5.000 *page out* por segundo. Os demais sistemas não ultrapassaram 1.500 *page out* por segundo.

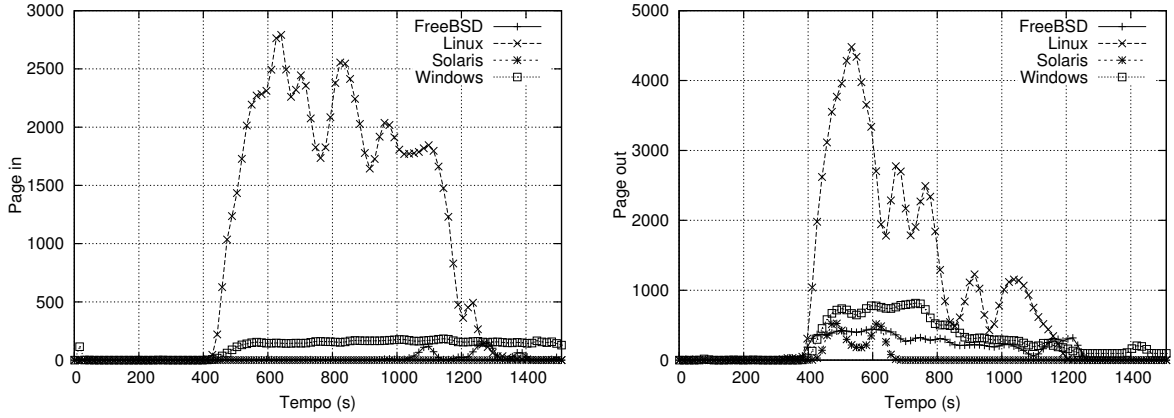

Figura 5.7: Número de page in/out, para 10.000 escritas

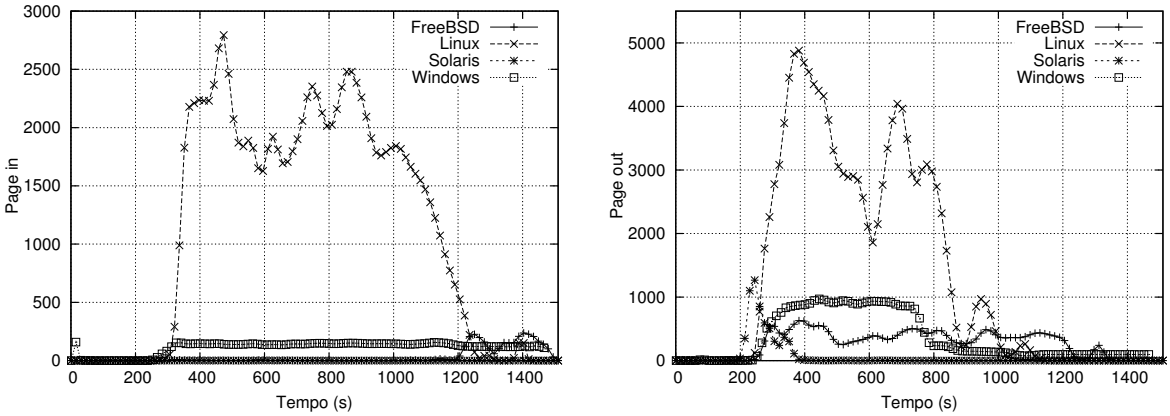

Figura 5.8: Número de page in/out, para 50.000 escritas

5.7.2 Influência do tempo de espera entre ciclos de escritas

O tempo de espera entre ciclos de escritas (t_w) foi variado entre 100, 500 e 1.000 milissegundos. Foram criados 25 processos, cada um alocando 100 MB de memória RAM. O número de escritas (W) foi avaliado em 10.000 e o tempo de espera para ativação de cada processo (t_c) foi de 30 segundos. As medidas foram efetuadas a cada segundo.

As Figuras 5.9, 5.10 e 5.11 apresentam um gráfico do consumo de CPU por processos de usuário em função da variação do tempo de espera entre as escritas (100, 500 e 1.000ms respectivamente).

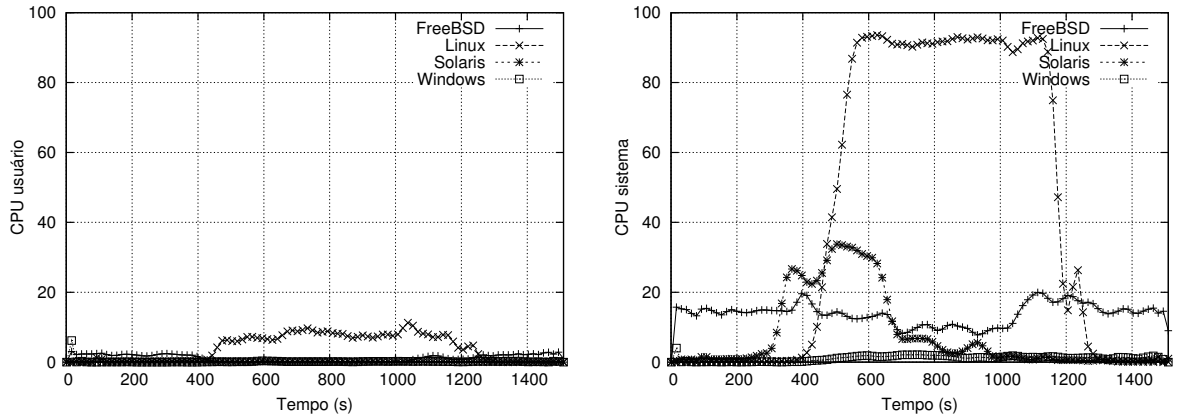

Figura 5.9: Consumo de CPU em modo usuário e sistema, com tempo de espera de 100ms

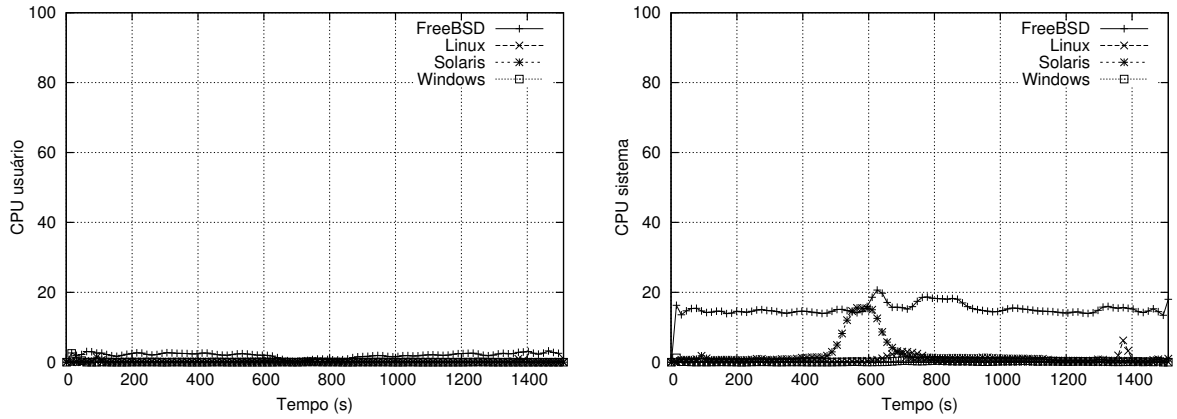

Figura 5.10: Consumo de CPU em modo usuário e sistema, com tempo de espera de 500ms

Pode-se observar que, em geral, quanto menor o tempo de espera entre ciclos de escritas maior é o consumo de CPU. O Linux é o sistema que apresenta maior consumo de CPU em modo usuário e sistema quando avaliado com tempo de espera de 100ms. O FreeBSD apresenta um consumo médio de 15% de CPU em modo sistema, independentemente do tempo de espera entre ciclos de escrita. O Windows apresentou, em média, baixo consumo de CPU em modo usuário e sistema, no entanto, quando avaliado em 1.000ms apresentou picos de mais de 10% de uso de CPU em modo sistema.

As Figuras 5.12, 5.13 e 5.13 apresentam o número *page in/out*, ocorridos no sistema, sob as mesmas condições de variação do tempo de espera entre as escritas.

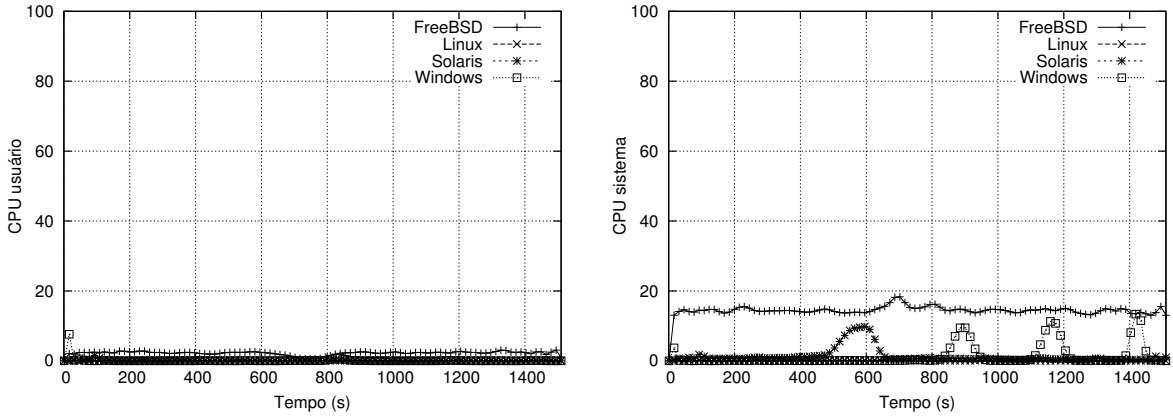

Figura 5.11: Consumo de CPU em modo usuário e sistema, com tempo de espera de 1.000ms

Pode-se observar que, em geral, quanto maior o tempo de espera entre ciclos de escritas menor é o número de *page in/out*. O Linux apresenta o maior pico de *page in/out* por segundo, nos três primeiros experimentos. No entanto, o Windows apresentou maior taxa de *page in* quando avaliado em 1.000ms.

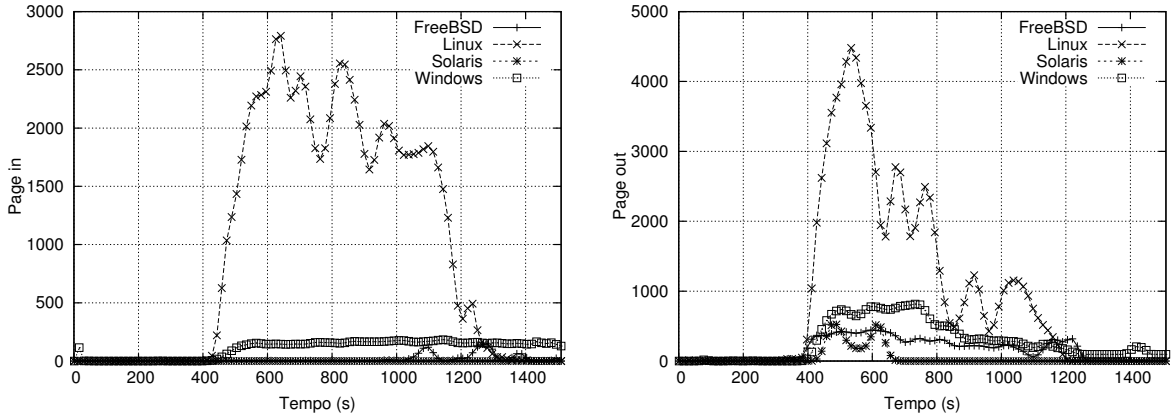

Figura 5.12: Número de *page in/out*, com tempo de espera de 100ms

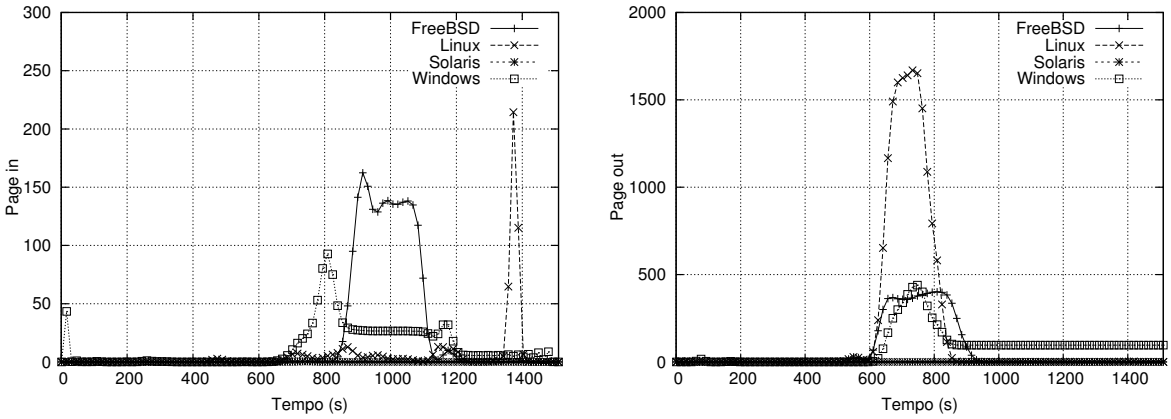

Figura 5.13: Número de *page in/out*, com tempo de espera de 500ms

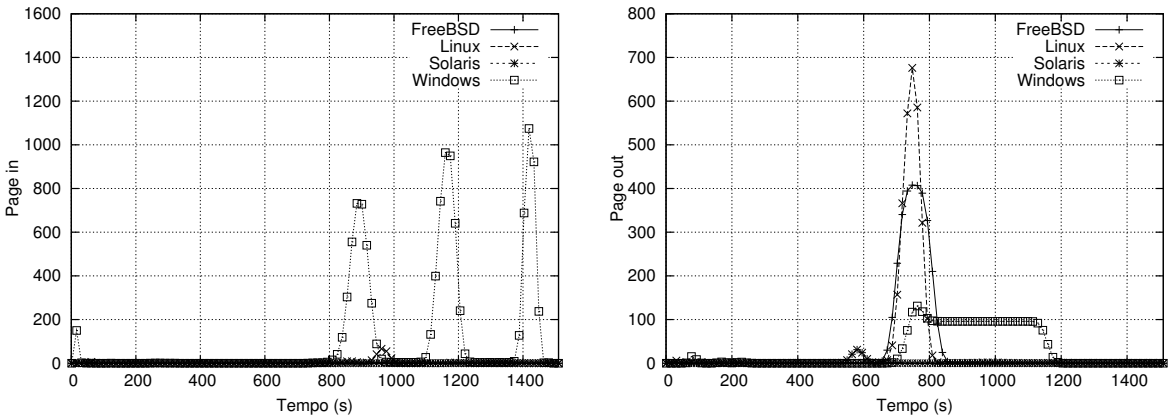

Figura 5.14: Número de *page in/out*, com tempo de espera de 1.000ms

5.7.3 Influência do tempo entre duas ativações de processos

O tempo de espera entre duas ativações consecutivas de processos consumidores (t_c) foi avaliado em 1, 10 e 30 segundos. Foram criados 25 processos, cada um alocando 100 MB de memória RAM. O número de escritas (W) foi mantido em 10.000 escritas/ciclo e o tempo de espera entre ciclos de escritas (t_w) foi mantido em 100 milissegundos.

As Figuras 5.15, 5.16 e 5.17 apresentam um gráfico do consumo de CPU em modo usuário e sistema em função da variação do tempo entre duas ativações de processos (1, 10 e 30 segundos respectivamente).

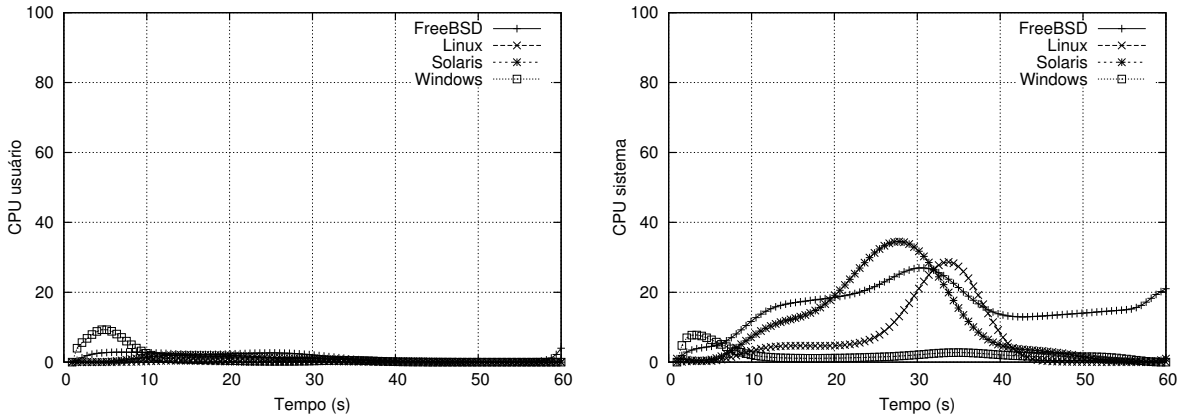

Figura 5.15: Consumo de CPU em modo usuário e sistema, com 1s de espera entre duas ativações de processos

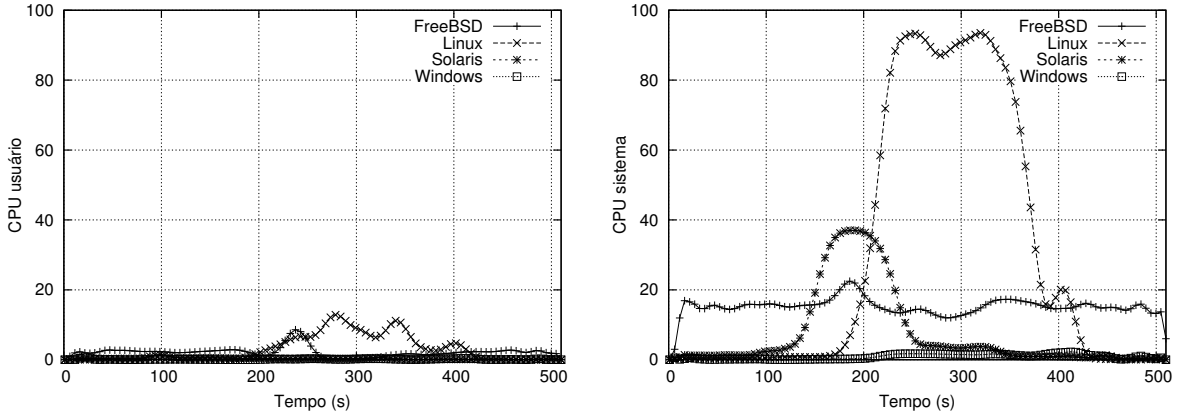

Figura 5.16: Consumo de CPU em modo usuário e sistema, com 10s de espera entre duas ativações de processos

O OpenSolaris apresentou maior consumo de CPU em modo sistema quando avaliado com 1s de espera. Este pico se manteve próximo a 40% nos três experimentos. Já o Linux, apresentou maior consumo de CPU em modo usuário e sistema quando avaliado em 10 e 30 segundos. O Linux novamente atingiu picos de 100% de uso de CPU, somando o consumo de CPU em modo usuário e sistema.

As Figuras 5.18, 5.19 e 5.20 apresentam o número *page in/out*, ocorridos no sistema, sob as mesmas condições de espera entre duas ativações de processos.

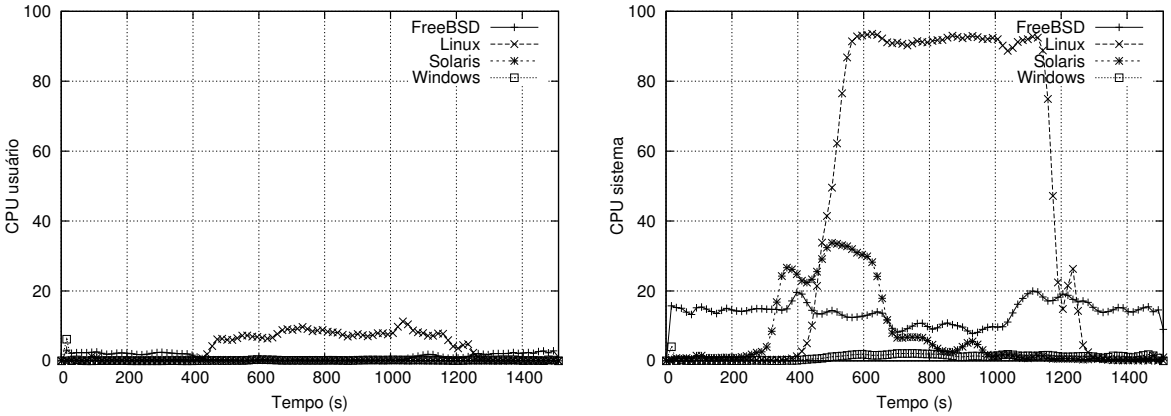

Figura 5.17: Consumo de CPU em modo usuário e sistema, com 30s de espera entre duas ativações de processos

O Linux é o sistema que apresenta o maior número de *page in/out* nos três experimentos. Com 10 segundos de tempo de espera, o Linux atingiu mais de 7.000 *page out* por segundo. O Windows atingiu picos de aproximadamente 1.500 *page out* por segundo. O OpenSolaris, de modo geral, foi o sistema que apresentou o menor número de *page in/out* nos três experimentos.

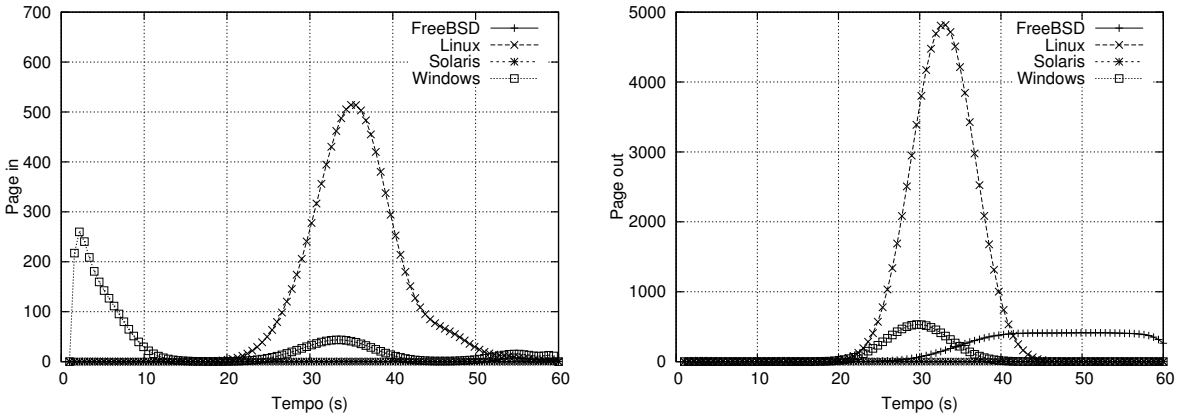

Figura 5.18: Número de *page in/out*, com 1s de espera entre duas ativações de processos

Pode-se observar, principalmente na Figura 5.19, que logo após o intervalo (t) = 200 todos os sistemas apresentam um número considerável de paginação, caracterizando o surgimento do *thrashing*. Pois é justamente logo após esse intervalo que todos os processos consumidores estão ativos no sistema, gerando mais faltas de página e consequentemente reduzindo a utilização de CPU.

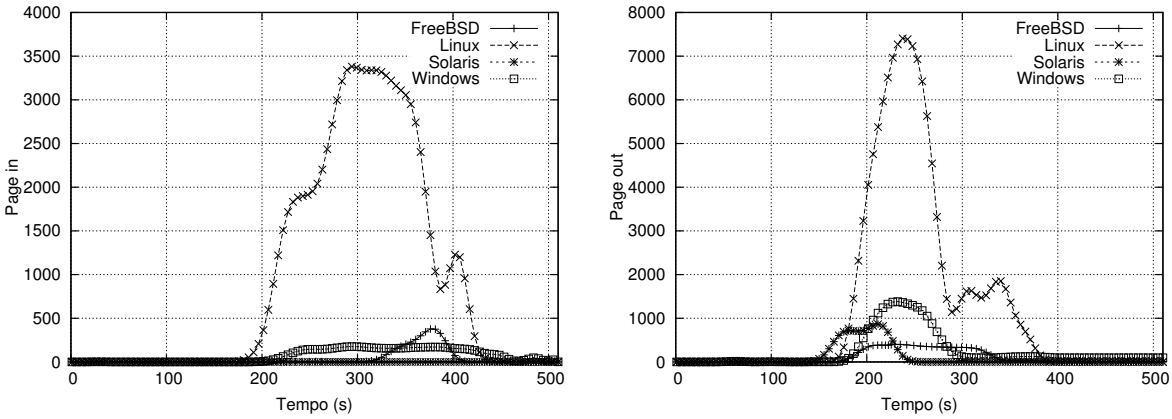

Figura 5.19: Número de *page in/out*, com 10s de espera entre duas ativações de processos

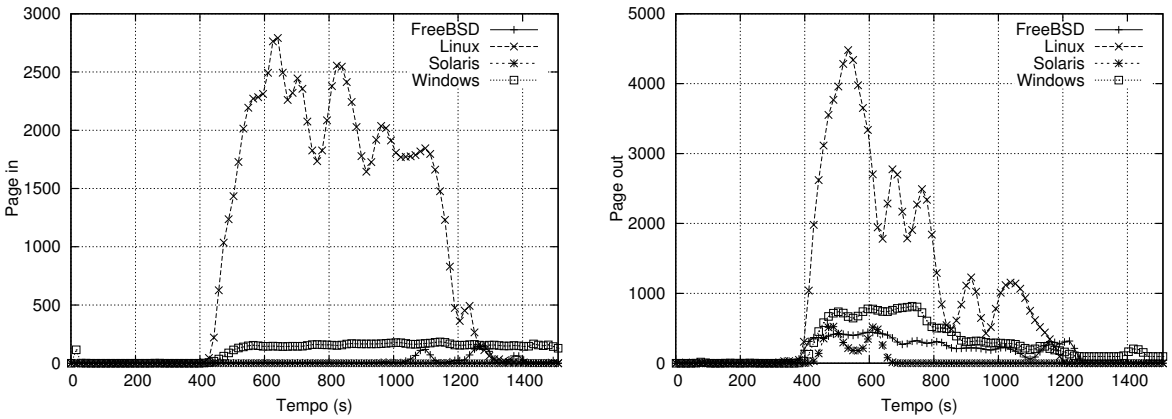

Figura 5.20: Número de *page in/out*, com 30s de espera entre duas ativações de processos

5.7.4 Influência sem o tempo de espera entre ativações de processos

Também foi avaliado o comportamento dos sistemas operacionais sem o tempo de espera entre ativações de processos. Ou seja, todos os processos consumidores começam sua atividade logo após o intervalo de 10 segundos de estabilização do sistema. Foram criados 25 processos, cada um alocando 100 MB de memória RAM. O número de escritas (W) foi mantido em 10.000 escritas/ciclo e o tempo de espera entre ciclos de escritas (t_w) foi mantido em 100 milissegundos.

A Figura 5.21 apresenta o gráfico do consumo de CPU em modo usuário e sistema, sem o tempo de espera entre ativações de processos. Pode-se observar um consumo de CPU em modo sistema logo no início da atividade dos processos, principalmente pelo sistema OpenSolaris. Novamente o Linux foi o sistema que apresentou maior consumo de CPU, tanto em modo usuário como em modo sistema. O FreeBSD manteve um consumo bastante constante e o Windows XP apresentou baixo consumo de CPU em modo sistema.

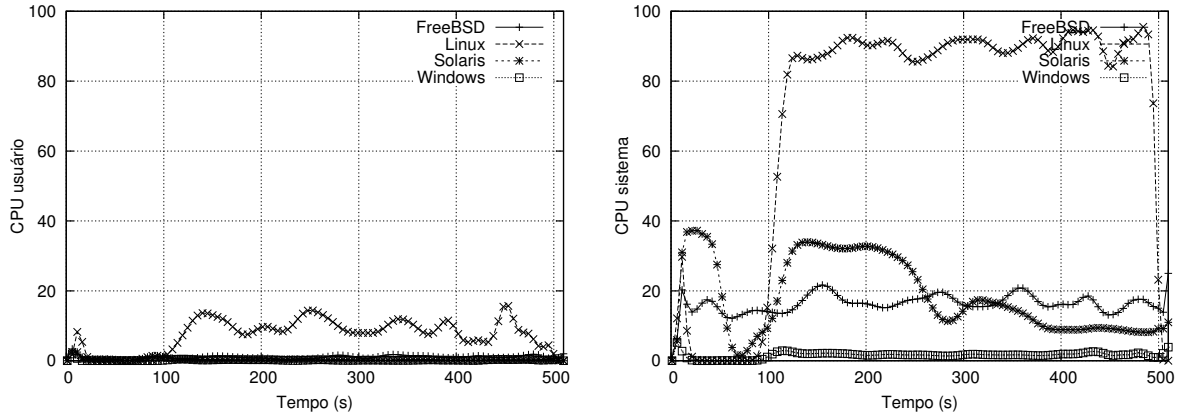

Figura 5.21: Consumo de CPU em modo usuário e sistema, sem espera entre duas ativações de processos

A Figura 5.22 apresenta o número *page in/out*, ocorridos no sistema, sob as mesmas condições descritas anteriormente. Pode-se também observar um pico logo que os processos consumidores são ativados. O Linux é novamente o sistema que apresenta o maior número de *page in/out*, atingindo picos de quase 9.000 *page out*. O Windows atingiu picos de aproximadamente 2.000 *page out* por segundo.

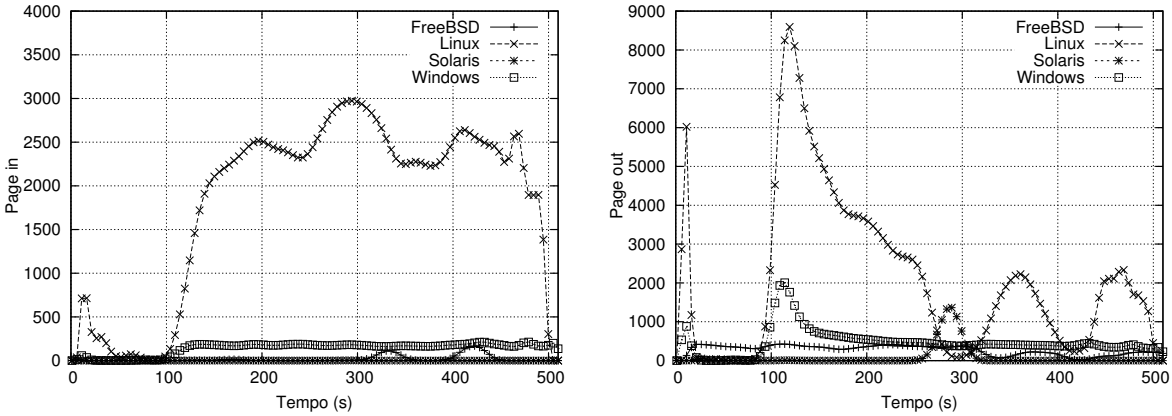

Figura 5.22: Número de *page in/out*, sem espera entre duas ativações de processos

5.8 Confiabilidade dos resultados obtidos

Em qualquer procedimento experimental, a reprodutibilidade dos experimentos é importante para que os resultados obtidos possam ser considerados confiáveis. Tendo em vista o caráter bastante dinâmico dos mecanismos de gerência de memória implementados pelos sistemas operacionais estudados, era de se esperar uma grande variabilidade nos resultados obtidos durante os experimentos. Para nossa surpresa, essa variabilidade se mostrou relativamente modesta, o que permite confirmar a validade dos resultados apresentados.

Os diagramas apresentados nas Figuras 5.23 a 5.30 representam os resultados obtidos em três experimentos idênticos realizados nos sistemas operacionais sob estudo. Os parâmetros de avaliação foram ajustados da seguinte forma: 10.000 escritas por ciclo, 100 ms de tempo de espera entre ciclos de escrita e 10 segundos de tempo de espera entre duas ativações de processos. Foram criados ao todo 25 processos, cada um alocando 100 MB de memória RAM. O computador foi reiniciado antes de cada experimento.

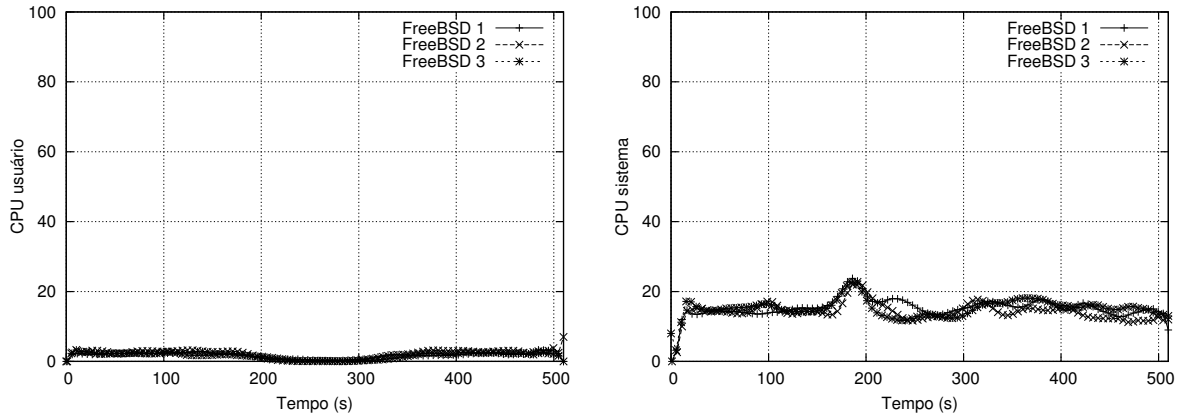

Figura 5.23: Variação de CPU em modo usuário e sistema no FreeBSD

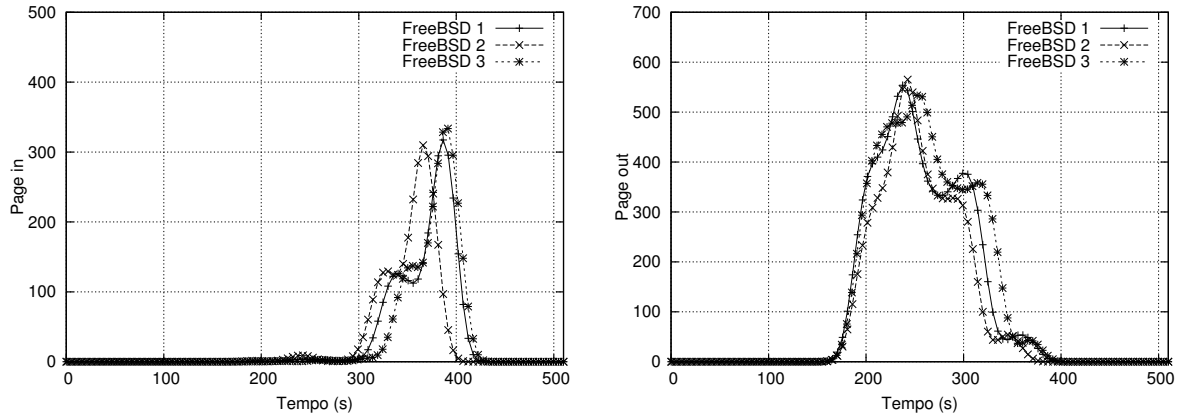

Figura 5.24: Variação de número de *page in/out* no FreeBSD

Como pode ser observado nos diagramas, cada sistema operacional se comportou aproximadamente da mesma forma ao longo dos três experimentos mantendo a tendência geral. Essa estabilidade dos resultados nos permite afirmar que os dados obtidos nos experimentos anteriores são estáveis e podem ser usados com base de comparação entre os sistemas.

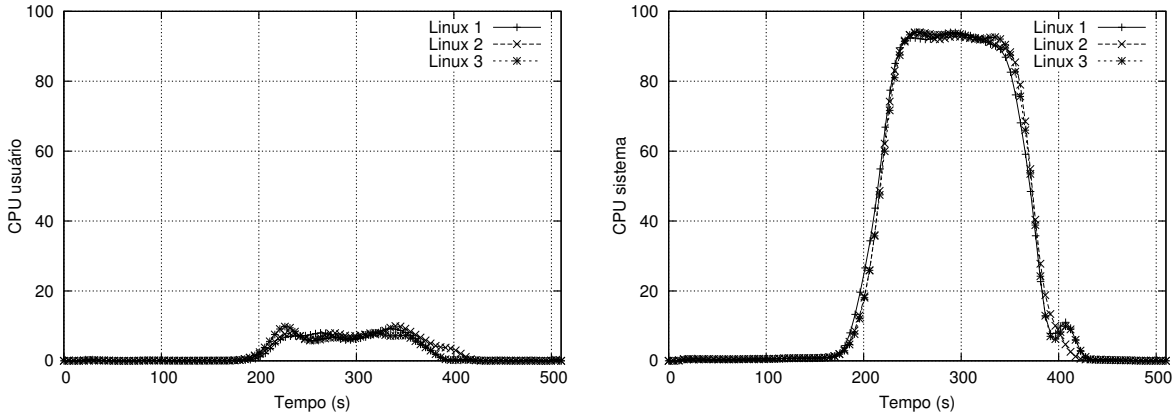

Figura 5.25: Variação de CPU em modo usuário e sistema no Linux

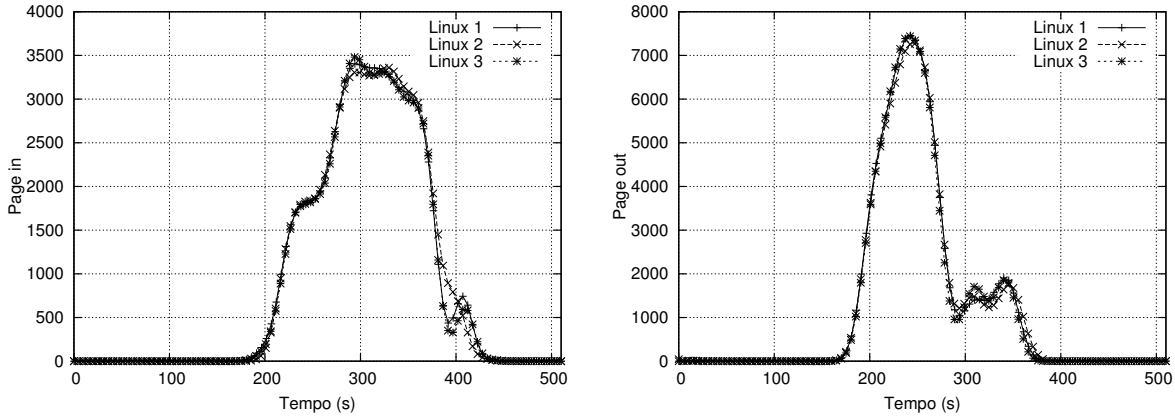

Figura 5.26: Variação de número de *page in/out* no Linux

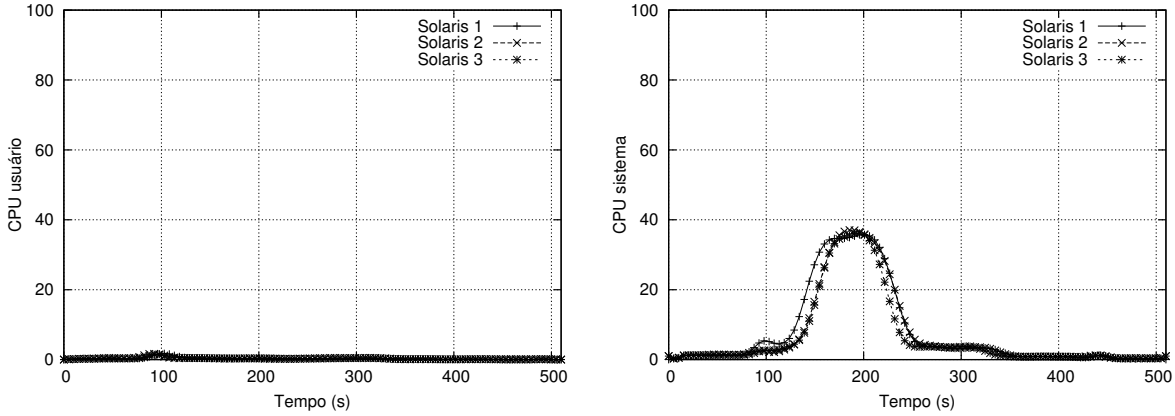

Figura 5.27: Variação de CPU em modo usuário e sistema no OpenSolaris

Como o programa de carga de trabalho (processo consumidor) emprega números pseudo-aleatórios de forma intensiva, a qualidade dos geradores de números aleatórios providos pelos sistemas pode influenciar nos resultados: se um gerador não for suficientemente aleatório, o padrão de acesso à memória do programa de carga de trabalho pode apresentar uma localidade de referência significativa e beneficiar o algoritmo de paginação.

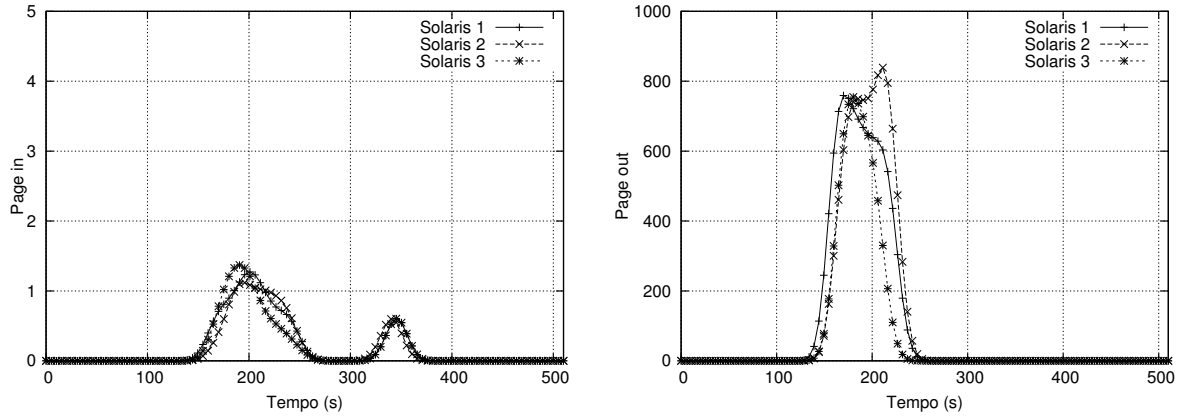

Figura 5.28: Variação de número de *page in/out* no OpenSolaris

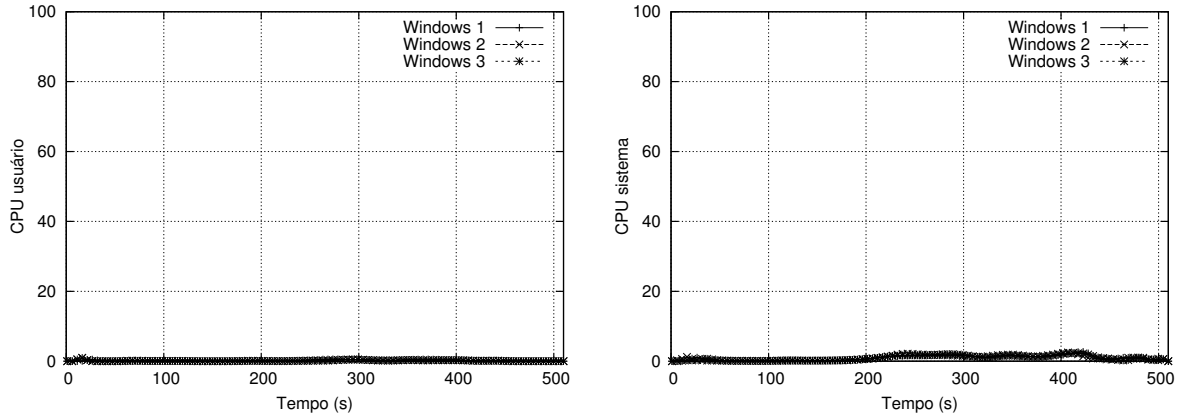

Figura 5.29: Variação de CPU em modo usuário e sistema no Windows XP

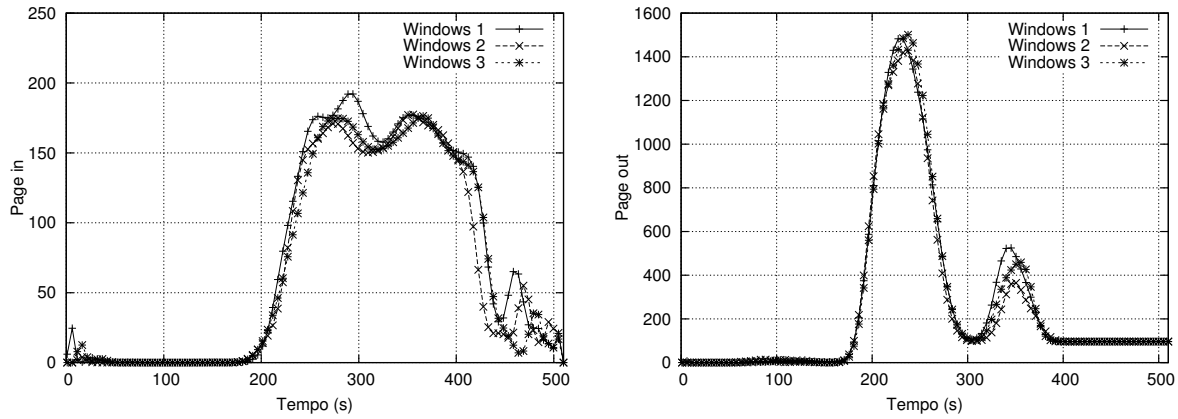

Figura 5.30: Variação de número de *page in/out* no Windows XP

Para avaliar a qualidade dos geradores de números aleatórios dos sistemas sob estudo, foi usada a técnica de plotagem do espaço de fase [Hegger et al., 1999], muito usada para a análise de sistemas dinâmicos. Para converter uma sequência unidimensional de números aleatórios $s_{1...n}$ em coordenadas (x, y) para plotagem do espaço de fase, foi usada a técnica de coordenadas atrasadas (*delayed coordinates* [Fraser and Swinney, 1986]), que permite aumentar o espaço de

coordenadas de um conjunto de dados $n-$dimensional. As coordenadas adicionais são construídas a partir das coordenadas já existentes. Em nosso caso, cada ponto (x_i, y_i) de plotagem foi construído a partir da sequência s da seguinte forma:

$$\begin{aligned} x_i &= s_{i-t} - s_{i-2t} \\ y_i &= s_i - s_{i-t} \end{aligned}$$

Onde t é um inteiro positivo ajustável ($t = 1, 2, 3, \ldots$). O conjunto de pontos assim obtido é plotado e observado para diversos valores de t. No caso de números aleatórios de boa qualidade, a plotagem deve mostrar uma "nuvem" de dados relativamente homogênea e sem pontos de concentração (ou "atratores"). Caso sejam percebidos atratores, existe uma correlação entre os valores sucessivos e o gerador de aleatórios deve ser colocado sob suspeita.

Analisamos os geradores de números aleatórios dos sistemas sob estudo usando essa técnica para vários valores de t, e em todos os casos os resultados obtidos foram satisfatórios. A Figura 5.31 mostra o resultado dessa análise para $t = 1$ nos sistemas Windows XP, OpenSolaris, Linux e FreeBSD. Nessa figura podemos perceber que todos os geradores de números aleatórios se comportaram de forma satisfatória, sem a ocorrência de atratores que pudessem denunciar comportamentos tendenciosos.

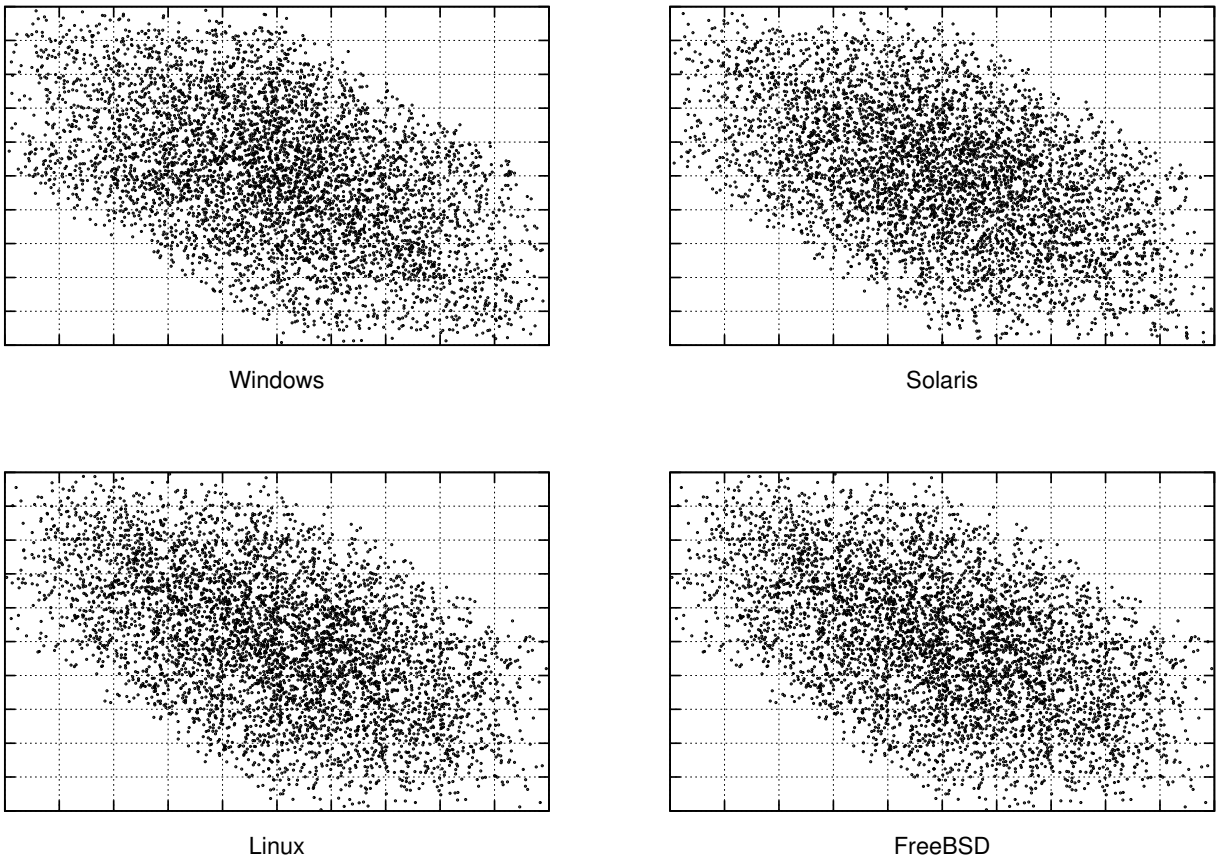

Figura 5.31: Análise de números aleatórios

5.9 Avaliação dos resultados

Esta seção apresenta a avaliação dos resultados obtidos através dos experimentos descritos nas seções anteriores. Cada sistema operacional avaliado foi submetido a alguns experimentos, que conduziu o sistema ao *thrashing* e de volta ao estado normal de operação. Cada sistema apresentou uma reação diferente, e a avaliação destas reações serão descritas nos parágrafos a seguir.

Além do comportamento demonstrado nos gráficos apresentados, há que se considerar também a impressão subjetiva do usuário de um sistema sob *thrashing*. De forma geral, o FreeBSD é o sistema que apresentou a maior estabilidade de consumo de CPU, seja em modo usuário ou em modo sistema, durante os experimentos. Esse sistema, assim como o OpenSolaris, apresentaram demora no uso do sistema em geral, de acordo com o que foi percebido pela interação do usuário com o sistema.

O Linux, diferente dos demais sistemas, em alguns momentos apresentou 100% de consumo de CPU, dois quais 90% são referentes a consumo de CPU por atividades de sistema. Além disso, o Linux apresentou os maiores picos de *page in/out* na memória. Pode-se concluir que esse consumo elevado de CPU em modo sistema foi utilizado para o processamento de leitura/escrita de páginas entre a memória RAM e o disco rígido. Na prática, Linux foi o sistema que apresentou maior responsividade após os experimentos. Ou seja, se recupera mais rapidamente do *thrashing*. Essa responsividade não pode ser medida de uma maneira fácil, mas pode ser percebida pela interação do usuário com o sistema.

O OpenSolaris apresenta um crescimento inicial de consumo CPU, principalmente em modo de sistema, após a ativação dos processos. Esse consumo começa a decrescer de forma significativa a medida que mais processos vão sendo ativados. O OpenSolaris foi o sistema que apresentou maior consumo de memória RAM antes dos experimentos, e também foi o sistema que apresentou maior demora no uso do sistema em geral, de acordo com o que foi percebido pela interação do usuário.

O Windows apresentou, em geral, baixo consumo de CPU durante os experimentos. Diferente dos demais sistemas operacionais, o Windows atrasou a atividade de alguns processos que não terminaram no tempo (T) esperado. Além disso foi o sistema que apresentou menor responsividade, percebido pela interação do usuário, após os experimentos. Ou seja, leva mais tempo para se recuperar do *thrashing*.

Como pode ser observado, o mecanismo de tratamento de *thrashing* existentes nos sistemas operacionais avaliados não é suficiente para conter de forma eficiente este fenômeno. Ajustes de desempenho mais finos como, por exemplo descritos em [Ciliendo and Kinimasa, 2008], podem diminuir significativamente a quantidade de paginação do sistema melhorando efetivamente o desempenho.

Capítulo 6

Conclusão

A gerência de memória é um mecanismo fundamental para qualquer ambiente computacional. Ela é responsável por entregar páginas de memória de acordo com as requisições dos processos, e também é responsável pelo mapeamento da memória física em memória virtual. Quando ocorrem muitas requisições de memória, o mecanismo de gerência de memória é forçado a fazer páginação, movendo páginas de memória de alguns processos entre o disco e a memória RAM. Em períodos de muita paginação, os processos que sofreram paginação não conseguem fazer uso da CPU pois estão aguardado suas páginas de memória. Este fenômeno é chamado de *thrashing*.

O estudo apresentado neste trabalho comprova que não existem soluções ideais para o *thrashing* e que os sistemas operacionais modernos ainda estão sujeitos a este fenômeno. Além disso, cada sistema operacional aborda esse problema de uma forma própria.

Esta dissertação buscou construir uma análise comparativa do comportamento de alguns sistemas operacionais desktop de mercado quando confrontados a situação de *thrashing*. Para tal, foi desenvolvido um modelo de *benchmarking* de uso da memória que provoca uma situação de thrashing controlada e seu retorno posterior à normalidade. Além disso, foram definidas técnicas de extração de informações sobre uso de recursos para cada um dos sistemas operacionais avaliados.

Foi observado nos experimentos que o Linux é o sistema que se recupera mais rapidamente do *thrashing*, pois esse sistema consegue fazer bastante processamento nesse período. Pode-se observar que o mecanismo de *token* implementado no Linux consegue um desempenho melhor que os demais, justamente por ceder privilégios para um determinado processo em um dado instante. Assim, ao menos um processo consegue um avanço substancial em sua execução. Sob a ótica de usuário, o Linux foi o sistema que apresentou a menor degradação de interatividade e a mais rápida recuperação do thrashing. Essa percepção não é facilmente mensurável, mas pode ser facilmente percebida pela interação do usuário.

Já o Windows é o sistema mais lento durante o *thrashing* e o sistema que demora mais tempo para se recuperar, pois as atividades no sistema são comprometidas devido ao baixo processamento efetuado nos períodos de *thrashing*. O FreeBSD apresentou um consumo estável de CPU durante os experimentos. Assim como o OpenSolaris, o FreeBSD apresentou demora no uso em geral do sistema, ou seja, os maiores atrasos na interatividade do sistema, de acordo com o que foi percebido pela interação do usuário.

Pode-se pensar, como trabalho futuro, a definição de um padrão para extração de informações do núcleo do sistema operacional com o objetivo de facilitar futuras medições ou

mesmo para definir um modelo de *benchmark* mais completo, que possa avaliar outros parâmetros no sistema. Além disso, com os resultados apresentados nesse trabalho, pode-se pensar no aperfeiçoamento de técnicas ou métodos para minimizar o *thrashing*. O presente estudo também pode ser estendido para abranger outros sistemas operacionais populares, como o MacOSX.

Referências Bibliográficas

[bandwidth, 2008] bandwidth (2008). *Bandwidth*. http://home.comcast.net/∼fbui/bandwidth.html.

[Bapco, 2008] Bapco (2008). *SYSmark*. http://www.bapco.com/products.

[Bonwick and Microsystems, 1994] Bonwick, J. and Microsystems, S. (1994). The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98.

[Bovet and Cesati, 2005] Bovet, D. P. and Cesati, M. (2005). *Understanding the Linux Kernel, 3rd Edition*. O'Reilly & Associates, Inc., USA.

[Ciliendo and Kinimasa, 2008] Ciliendo, E. and Kinimasa, T. (2008). *Linux Performance and Tuning Guidelines*. IBM Redbooks.

[Cowardin, 1997] Cowardin, J. (1997). A proc buffer for kernel instrumentation. Master's thesis, The College of William & Mary.

[de Oliveira et al., 2003] de Oliveira, R. S., da Silva Carissimi, A., and Toscani, S. S. (2003). *Sistemas Operacionais*. Bookman.

[Denning, 1968] Denning, P. J. (1968). Thrashing: its causes and prevention. In *AFIPS '68 (Fall, part I): Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 915–922, New York, NY, USA. ACM.

[Denning, 1995] Denning, P. J. (1995). A short theory of multiprogramming. In *MASCOTS '95: Proceedings of the 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 2–7, Washington, DC, USA. IEEE Computer Society.

[Denning, 2005] Denning, P. J. (2005). The locality principle. *Communications of the ACM*, 48(7).

[Fraser and Swinney, 1986] Fraser, A. M. and Swinney, H. L. (1986). Independent coordinates for strange attractors from mutual information. *Physical Review A*, 33(2):1134–1140.

[Futuremark, 2008] Futuremark (2008). *PCMarck*. http://www.futuremark.com/products.

[GCC, 2008] GCC (2008). *GNU Compiler Collection*. http://gcc.gnu.org.

[Hegger et al., 1999] Hegger, R., Kantz, H., and Schreiber, T. (1999). Practical implementation of nonlinear time series methods: The [small-caps tisean] package. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 9(2):413–435.

[Henning, 2006] Henning, J. L. (2006). *SPEC CPU2006 Benchmark Descriptions*. http://www.spec.org.

[Jiang and Zhang, 2001] Jiang, S. and Zhang, X. (2001). Adaptive page replacement to protect thrashing in Linux. In *ALS '01: Proceedings of the 5th annual conference on Linux Showcase & Conference*, pages 16–16, Berkeley, CA, USA. USENIX Association.

[Jiang and Zhang, 2002] Jiang, S. and Zhang, X. (2002). Tpf: a dynamic system thrashing protection facility. *Software Practice and Experience*, 32(3):295–318.

[Jiang and Zhang, 2005] Jiang, S. and Zhang, X. (2005). Token-ordered LRU: an effective page replacement policy and its implementation in Linux systems. In *Performance Evaluation Vol .60, Issue 1-4*, pages 5–29.

[Kernel, 2008] Kernel (2008). *Linux kernel*. http://www.kernel.org.

[Knuth, 1997] Knuth, D. (1997). *The Art of Computer Programming Volume 1: Fundamental Algorithms*. Addison-Wesley.

[Lehey, 2001] Lehey, G. (2001). *Explaining BSD*. http://www.freebsd.org.

[Markatos, 1996] Markatos, E. P. (1996). Using remote memory to avoid disk thrashing: A simulation study. In *In Proceedings of the ACM International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '96*, pages 69–73.

[MarketShare, 2008] MarketShare (2008). *Windows XP Market Share*. http://marketshare.hitslink.com.

[Mauro and McDougall, 2001] Mauro, J. and McDougall, R. (2001). *Solaris Internals: Core Kernel Components*. Prentice Hall PTR.

[Maziero, 2008] Maziero, C. (2008). *Sistemas Operacionais*. Open Book http://www.ppgia.pucpr.br/∼maziero/.

[McKusick and Neville-Neil, 2004] McKusick, M. K. and Neville-Neil, G. V. (2004). *The Design and Implementation of the FreeBSD Operating System*. Pearson Education.

[McVoy and Staelin, 1996] McVoy, L. W. and Staelin, C. (1996). LMbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294.

[Microsoft, 2003] Microsoft (2003). *Kernel Enhancements for Windows XP*. http://www.microsoft.com/whdc/archive/XP_kernel.mspx.

[Mucci et al., 1998] Mucci, P. J., London, K., and Thurman, J. (1998). *The CacheBench Report*. http://www.cs.utk.edu/∼mucci/DOD/cachebench.ps.

[Musumeci and Loukides, 2002] Musumeci, G.-P. D. and Loukides, M. (2002). *System Performance Tuning*. O'Reilly.

[nbench, 2008] nbench (2008). *nbench*. http://www.tux.org/∼mayer/linux/bmark.html.

[Netcraft, 2008] Netcraft (2008). *Netcraft Internet Research*. http://www.netcraft.com.

[OOM, 2009] OOM (2009). *Out of Memory Killer*. http://linux-mm.org/OOM_Killer.

[Patterson and Henessy, 2005] Patterson, D. and Henessy, J. (2005). *Organização e Projeto de Cpmpitadores*. Campus.

[Reuven and Wiseman, 2006] Reuven, M. and Wiseman, Y. (2006). Medium-term scheduler as a solution for the thrashing effect. *Computer Journal*, 49(3):297–309.

[Russinovich and Solomon, 2004] Russinovich, M. and Solomon, D. (2004). *Microsoft Windows Internals*. Microsoft Press.

[Scholl et al., 1997] Scholl, A., Klein, R., and Jürgens, C. (1997). Bison: a fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers and Operations Research*, 24(7).

[Silberschatz et al., 2002] Silberschatz, A., Galvin, P. B., and Gagne, G. (2002). *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA.

[Singh, 2001] Singh, A. (2001). *Mac OS X Internals: A Systems Approach*. Addison-Wesley.

[Sissoftware, 2008] Sissoftware (2008). *Sandra*. http://www.sisoftware.net/.

[Stream, 2008] Stream (2008). *STREAM bench*. http://www.streambench.org.

[Sun, 2006] Sun (2006). *System Administration Guide: Solaris Containers–Resource Management and Solaris Zones*. Sun Microsystems Inc, Santa Clara, USA.

[Tanenbaum, 2001] Tanenbaum, A. S. (2001). *Modern Operating Systems*. Prentice Hall.

[UnixBench, 2008] UnixBench (2008). *UnixBench*. http://www.hermit.org/Linux/Benchmarking.