Pontifícia Universidade Católica do Rio Grande do Sul

Faculdade de Informática

Pós-Graduação em Ciência da Computação

Um estudo sobre Migração de Páginas no Linux

Guilherme Antônio Anzilago Tesser

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação.

Orientador: Prof. Dr. Avelino F. Zorzo

Porto Alegre, julho de 2006.

Av. Ipiranga, 6681 - Prédio 16
CEP 90619-900 - Porto Alegre - RS - Brasil
http://www.pucrs.br/biblioteca

Fone: 0 (xx) 51 3320-3544
FAX: 0 (xx) 51 3320-3548
e-mail:bceadm@pucrs.br

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada **"Um Estudo Sobre Migração de Páginas no Linux"**, apresentada por Guilherme Antônio Anzilago Tesser, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 30/08/2006 pela Comissão Examinadora:

Prof. Dr. Avelino Francisco Zorzo – Orientador — PPGCC/PUCRS

Prof. Dr. Celso Maciel da Costa – FACIN/PUCRS

Prof. Dr. Fernando Luís Dotti – PPGCC/PUCRS

Homologada em 07/05/2007, conforme Ata No. 010 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P. 16 – sala 106 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@inf.pucrs.br
www.pucrs.br/facin/pos

A todos que acreditaram que isso era possível!

Agradecimentos

Começo agradecendo ao professor Avelino pela paciência que teve com o trabalho e principalmente pela minha ausência, por algumas vezes, um tanto longas. Na parte técnica deste curso, não posso mensurar a ajuda da colega Mônica, que sempre esteve à disposição para esclarecer dúvidas teóricas, discutir código fonte e resolver problemas técnicos que muitas vezes nem cabiam a ela. Esse trabalho não sairia do lugar se essa guria não atendesse ao telefone no final de semana ou me bloqueasse no *messenger*! Devo agradecer ao grande amigo Chanin, por ter me ajudado a não desistir no meio do caminho, me convencendo de que "no final tudo iria acabar bem" e que quando a gente fosse lembrar disso tudo a gente só iria rir. Pode vir te rindo já, então "hehe"! Não tenho como me esquecer do também grande amigo Joan, que me ensinou todos os atalhos do mestrado, sempre com ótimas dicas de como gerenciar o tempo entre trabalho e lazer. Mestrado gera traumas, mas esse cara sabe tudo sobre como não se estressar com isso. Acho que um pouco eu aprendi! Agradeço também aos colegas de trabalho com quem muito aprendi e aprendo diariamente através de troca de experiências, conversas técnicas e de assuntos gerais. Apesar de não estarem diretamente envolvidos neste trabalho, são também como mestres responsáveis pela minha formação profissional.

Não posso deixar de agradecer à minha família pelo apoio que me deu desde que decidi fazer o curso. Com certeza, saber que vocês acreditam em mim, é um fator de motivação muito grande para o meu trabalho.

Especialmente, agradeço à minha namorada! Por toda ajuda que me deu durante esse tempo, a paciência e compreensão pelos finais de semana longe escrevendo a dissertação, e também pelos ótimos momentos de estudo juntos, cada um no seu assunto! Era muito bom estar no meio do trabalho e olhar para o lado e te ver! Te agradeço também pela correção ortográfica dos trabalhos, principalmente aqueles que tu teve que ler inteiro na véspera do dia da entrega!

Resumo

Este trabalho discute o desempenho de algoritmos de migração de memória em computadores do tipo NUMA. É apresentada uma breve descrição do algoritmo de gerenciamento de memória do Linux e, também, dos algoritmos de migração de memória propostos (migração total e migração sob demanda). Em seguida, é descrito o modelo de avaliação de desempenho. Neste trabalho, foi usado modelo de simulação com o desenvolvimento de um simulador que modela os algoritmos de gerência de memória do Linux e os algoritmos de migração de memória propostos. No final, são apresentados os resultados obtidos com o uso de migração de memória, que mostraram que há melhor desempenho quando comparado com o atual algoritmo de gerenciamento de memória do Linux.

Abstract

This work discusses the performance of memory migration algorithms on NUMA machines. It presents a brief description of the Linux memory management algorithm and also the memory migration algorithms proposed (full migration and on demand migration). In order to compare the memory migration strategies a simulation model was used. At the end of this dissertation a set of results acquired from the simulation model is presented. This results were obtained for two actual computers: SGI Altix and HP Superdome.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Símbolos e Abreviaturas

UMA	*Uniform Memory Access*	17
SMP	*Symetric Multiprocessor*	17
NUMA	*Non-Uniform Memory Access*	17
cc-NUMA	*Cache-Coherent Non-Uniform Memory Access*	17
ACPI	*Advanced Configuration and Power Interface*	21
RSDP	*Root System Description Pointer*	21
RSDT	*Root System Description Table*	21
SLIT	*System Locality Information Table*	21
LRU	*Least Recently Used*	34
I/O	*Input/Output*	42

Capítulo 1

Introdução

A complexidade de cálculos para se obter resultados satisfatórios para várias questões, que nos dias de hoje são fundamentais para diferentes tipos de negócio, exigem cada vez mais que tenhamos máquinas com grande poder computacional. Para questões como a previsão do tempo, simulação de trânsito, entre outras que se encaixam nesse perfil, o mercado oferece supercomputadores, *clusters* e máquinas multiprocessadas como algumas das possíveis soluções para esses usuários que necessitam de alta capacidade de processamento. Dentre essas soluções, as máquinas multiprocessadas são um tema de bastante estudo e pesquisa [Bircsak et al., 2000, Chapin et al., 1995, Mu et al., 2003]. Máquinas multiprocessadas são computadores formados por um conjunto de processadores. Essas máquinas podem ser classificadas em dois tipos segundo a sua topologia [Paterson and Hennessy, 1998]: UMA (*Uniform Memory Access*), também conhecida como SMP (*Symetric Multiprocessor*), onde temos somente um banco de memória e todos os processadores ligados a esse banco, e NUMA (*Non-uniform Memory Access*), onde temos diversos bancos de memória arranjados entre os processadores, porém, com todos os processadores compartilhando esses bancos [Paterson and Hennessy, 1998]. A Figura 1.1 mostra uma máquina NUMA formada por dois nodos (neste caso, duas máquinas SMP) onde qualquer um dos quatro processadores tem acesso ao banco de memória de qualquer um dos nodos.

A arquitetura das máquinas multiprocessadas fornece uma série de características que proporcionam melhor desempenho. Em máquinas multiprocessadas, por exemplo, podemos ter diversos processos executando em paralelo, cada um em um processador. Estes processos são colocados na memória, e cabe ao sistema operacional alocá-los de forma a garantir o melhor desempenho possível ao sistema. A parte do sistema operacional responsável pelo tratamento da memória é o gerente de memória. Em máquinas SMP, as funções desse gerente são facilitadas devido à máquina ter apenas um banco de memória, assim, uma série de questões que, em outras máquinas são problemáticas, em máquinas SMPs, acabam não aparecendo.

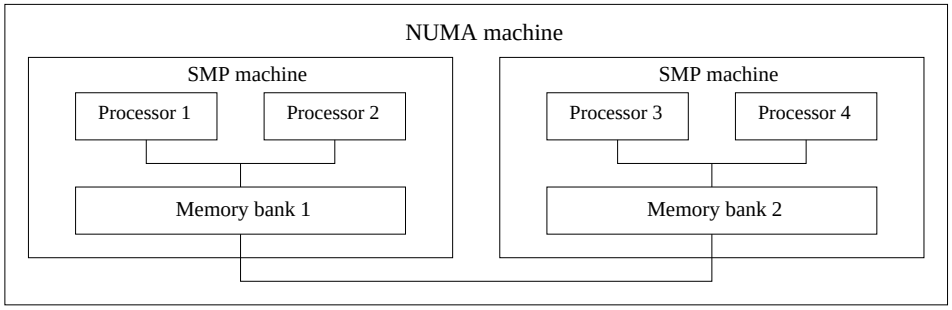


Figura 1.1: Máquina NUMA

Para máquinas NUMA[1], esta gerência não mais se dá de forma tão simples, já que o sistema operacional tem de lidar com os diversos bancos de memória, em que o tempo de acesso a eles já não é o mesmo para todos os processadores (ou conjunto de processadores). Dessa forma, diversas questões como balanceamento de carga, por exemplo, tem de ter um tratamento diferenciado para que a máquina seja melhor explorada e tenha melhor desempenho.

Atualmente, existem diversos paradigmas em uso para o balanceamento de carga em máquinas multiprocessadas [Chandra et al., 1994]. A maioria deles baseia-se em ambientes de programação paralela que decidem onde cada processo irá rodar no momento em que o processo é criado. Nestes ambientes, uma vez que um processo inicia em um nodo, ele deve se manter no mesmo nodo até o seu término. Mesmo se a carga do sistema, desta forma, não for a melhor, não se poderá voltar atrás sobre decisões de escalonamento já tomadas. Geralmente, é impossível prever como a carga do sistema irá variar no tempo, e assim, impossível de otimizar o escalonamento de processos em processadores [Chandra et al., 1994]. Em função disso, uma série de esforços foi feita a fim de se desenvolver uma classe diferente de ambientes de programação paralela que permitisse mover um processo de um nodo para outro dinamicamente durante qualquer momento da vida do processo. Essa mudança de local de execução de um processo durante o seu tempo de vida é chamada "migração de processo" [Milojicic et al., 2000]. Através de mudança dinâmica de processos durante o seu tempo de vida, o sistema pode se adaptar às mudanças de carga que não podiam ser previstas no início da execução das tarefas. Essa migração pode ser interessante do ponto de vista de balanceamento de carga, mas, novamente, considerando arquiteturas onde o tempo de acesso à memória não é igual para todos os processadores (NUMA, por exemplo) a migração pode não beneficiar o desempenho do sistema, já que o custo do acesso de um processo migrado ao seu espaço de endereçamento (que pode estar em um banco de memória

[1]Nesta dissertação, o termo NUMA refere-se a cc-NUMA [Culler and Singh, 1999].

mais "distante", devido à migração do processo) pode não pagar o custo (compensar) de um balancemento de carga. Para o melhor aproveitamento da migração de processos, um sistema de migração de páginas é importante pois fará com que as áreas de memória do processo fiquem mais próximas ao processador em que este está executando, para fazer com que estes dois custos sejam os menores possíveis e resultem em um melhor desempenho do sistema global.

Dada a importância de um sistema de migração de páginas, este trabalho propõe duas estratégias para migração de páginas e mostra um estudo, através de um processo de simulação, sobre o desempenho destas estratégias se implementadas no sistema operacional Linux[2].

A seguir, serão apresentadas algumas questões relevantes sobre gerenciamento de memória em máquinas NUMA (Capítulo 2), e, em seguida, o gerenciamento de memória feito pelo Sistema Operacional Linux (Seção 2.2). Os modelos de migração de páginas propostos no trabalho, então, são apresentados, primeiramente com o modelo de migração completa (migração total), mostrado na Seção 2.3.1 e, em seguida, o modelo de migração sob demanda, mostrado na Seção 2.3.2. Para a avaliação do desempenho do sistema foi utilizado um modelo simulado, que será apresentado ao final, no Capítulo 3.

[2]Este trabalho faz parte do projeto PeSO [PeSO, 2006] que tem como um dos seus requisitos, fazer pesquisa sobre o sistema operacional Linux, por isso, este foi o Sistema Operacional utilizado.

Capítulo 2

Migração de Páginas

A fim de garantir a eficiência em um sistema, a memória deve ser muito bem gerenciada [Haeberlen and Elphinston, 2003]. Em um sistema operacional, o gerente de memória é responsável por gerenciar o uso de toda a memória disponível no sistema. Gerência de memória em máquinas monoprocessadas tem sido assunto comum de pesquisa nos últimos anos, contudo, gerência de memória ainda apresenta alguns desafios quando aplicada a máquinas multiprocessadas, embora já tenha sido estudada por um tempo considerável.

Conforme mencionado, máquinas multiprocessadas com memória compartilhada podem ser classificadas como *Symmetric Multiprocessor* (SMP) ou *Non-Uniform Memory Access* (NUMA). Máquinas SMP são sistemas multiprocessados onde cada processador acessa qualquer área da memória em um tempo constante. Sistemas do tipo NUMA são sistemas multiprocessados organizados em nodos. Cada nodo tem um conjunto de processadores e parte da memória principal. A distância entre os nodos não é a mesma, portanto, existem diferentes tempos de acesso de cada processador para diferentes áreas da memória. A Figura 1.1 mostra uma máquina NUMA com quatro processadores organizados em dois nodos.

Com o objetivo de proporcionar o correto gerenciamento da memória, o sistema operacional precisa conhecer a estrutura de memória do computador. Atualmente, grande parte dos computadores suportam o padrão ACPI (*Advanced Configuration and Power Interface*) [Hewlett-Packard et al., 2004], o qual provê diversas informações para o sistema operacional.

2.1 ACPI

ACPI [Hewlett-Packard et al., 2004] é uma especificação de inteface que provê informações sobre a configuração do *hardware* permitindo ao sistema operacional realizar o gerenciamento de energia dos dispositivos de um computador. Todos os

dados da ACPI estão hierarquicamente organizados em tabelas de descrição que são montadas pelo *firmware* do computador. A estrutura base do padrão ACPI é o *Root System Description Pointer* (RSDP), o qual é carregado na memória em um endereço padrão e aponta para a *Root System Description Table* (RSDT). A RSDT contém ponteiros para todas as outras tabelas de descrição que provém informações a respeito da configuração da máquina, como por exemplo, informações de dispositivos *plug & play*, temporizadores, informação sobre a memória, etc.

Uma destas tabelas, chamada *System Locality Information Table* (SLIT), descreve a distância relativa (latência de memória) entre localidades ou domínios de proximidade. Especificamente no caso de computadores tipo NUMA, cada nodo é uma localidade. A Seção 2.1.1 apresenta em detalhes a SLIT.

2.1.1 ACPI SLIT

Em uma SLIT o valor da posição $P_{i,j}$ representa a distância do nodo i para o nodo j. A distância entre um nodo e ele mesmo é chamada distância SMP e possui o valor padrão de 10. Todas outras distâncias são relativas à distância SMP. Como processadores e blocos de memória estão dentro de um nodo, a SLIT provê a distância entre processadores e áreas de memória, isto é, diferentes distâncias na SLIT representam diferentes níveis de acesso à memória.

A Tabela 2.1.1 mostra uma possível SLIT para a máquina NUMA da Figura 1.1. De acordo com esta tabela, a distância entre o nodo 1 e o nodo 2 é duas vezes a distância SMP. Isto significa que um processador no nodo 1 acessa a área de memória do nodo 2 duas vezes mais lento que uma área de memória no nodo 1 [Hewlett-Packard et al., 2004].

Tabela 2.1: SLIT para uma máquina NUMA

	Nodo 1	Nodo 2
Nodo 1	10	20
Nodo 2	20	10

2.2 O Gerente de Memória do Linux

O gerente de memória implementado pelo Linux trabalha de forma similar para máquinas NUMA e SMP. Em máquinas NUMA, contudo, a memória é organizada em bancos chamados nodos, e o gerente de memória do Linux aloca memória para um processo no banco mais perto do nodo no qual o processo está executando. Esta estratégia se mostra interessante se um processo não muda de nodo durante

seu tempo de execução. O escalonador do Linux, entretanto, tenta manter a carga de todos os processadores a mais balanceada possível, reatribuindo processos a processadores em três diferentes situações: (i) quando um processador entra no estado ocioso; (ii) quando um processo executa a chamada de sistema *exec* ou *clone*; e (iii) periodicamente, em intervalos de tempo específicos, definidos para cada domínio de escalonamento [Corrêa et al., 2006]. Esta migração de processos pode causar o "afastamento" da área de memória do processo em relação ao nodo no qual o processo está executando. Atualmente, o gerente de memória do Linux não implementa qualquer tipo de migração de páginas, isto é, depois da migração de um processo, nenhum esforço é feito para trazer o seu espaço de endereçamento para um banco de memória mais próximo do processador no qual ele foi reescalonado para rodar. O único esforço feito pelo Linux para trazer as páginas mais perto do nodo onde o processo está executando ocorre se as páginas são trazidas à memória em função de uma falta de página (*page fault*).

Apesar de não haver um tratamento diferenciado com relação à memória do processo migrado, o que pode degradar o desempenho do sistema, o algoritmo de balanceamento de carga do Linux está estruturado de forma a tentar evitar a migração de processos entre nodos em máquinas NUMA. Isto é feito baseado no conceito de domínios de escalonamento e grupos de CPUs que hierarquicamente estruturados representam a topologia da máquina. Um domínio de escalonamento é formado por um ou mais grupos de CPUs, e estes são formados pelos processadores de um mesmo nodo. Na máquina da Figura 1.1, teríamos dois grupos de CPU (um formado pelos processadores 1 e 2 e outro formado pelos processadores 3 e 4) e um domínio de escalonamento (composto pelos dois grupos de CPU) como podemos ver na Figura 2.1. Assim, o balanceador de carga trabalha dentro do escopo definido por estes grupos e domínios, ou seja, ele tenta primeiramente resolver o desbalanceamento dentro do nodo e posteriormente entre os nodos caso necessário, evitando assim a migração de processos entre nodos. Informações mais detalhadas sobre o balanceador de carga do Linux em [Corrêa, 2005].

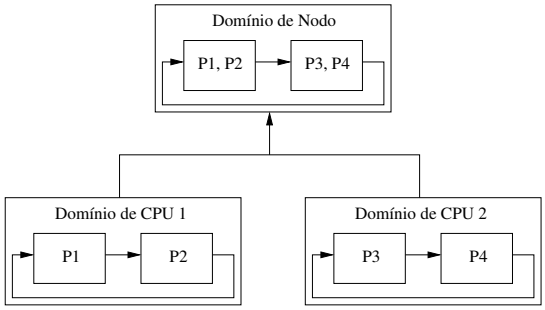


Figura 2.1: Domínios de CPU e escalonamento

2.3 Estratégias para Migração de Páginas

Em sistemas SMP, a escolha de migrar processos de um processador sobrecarregado para um processador livre (*idle*) não causa nenhum efeito maior. Como a distância entre todos os processadores e a memória é a mesma, migrar um processo de qualquer processador para outro não diminui o desempenho global do processo. Isto não acontece em máquinas NUMA pois migrar um processo de um processador para outro processador no mesmo nodo é melhor do que migrá-lo de um processador para outro nodo. Isto se deve às diferentes distâncias entre processadores que estão em diferentes nodos.

A Figura 2.2 mostra um exemplo de migração de processo. Suponhamos que dois processos $S1$ e $S2$ compartilham o mesmo processador $P1$. O Linux aloca memória para ambos $S1$ e $S2$ no banco de memória do nodo 1, que é o banco de memória mais próximo do processador $P1$. Linux, então, decide migrar o processo $S1$ para o processador $P3$ porque $P3$ está livre (*idle*). Isto irá balancear a carga dos processadores, provavelmente, melhorando o desempenho do sistema já que os processos $S1$ e $S2$ irão agora estar rodando em paralelo. Contudo, toda vez que o processo $S1$ acessar sua memória, este acesso será mais lento. Esta situação pode causar o declínio do desempenho do processo, e, provavelmente, de todo o sistema. O atual escalonador do Linux e o gerente de memória não levam em consideração a localização da memória do processo.

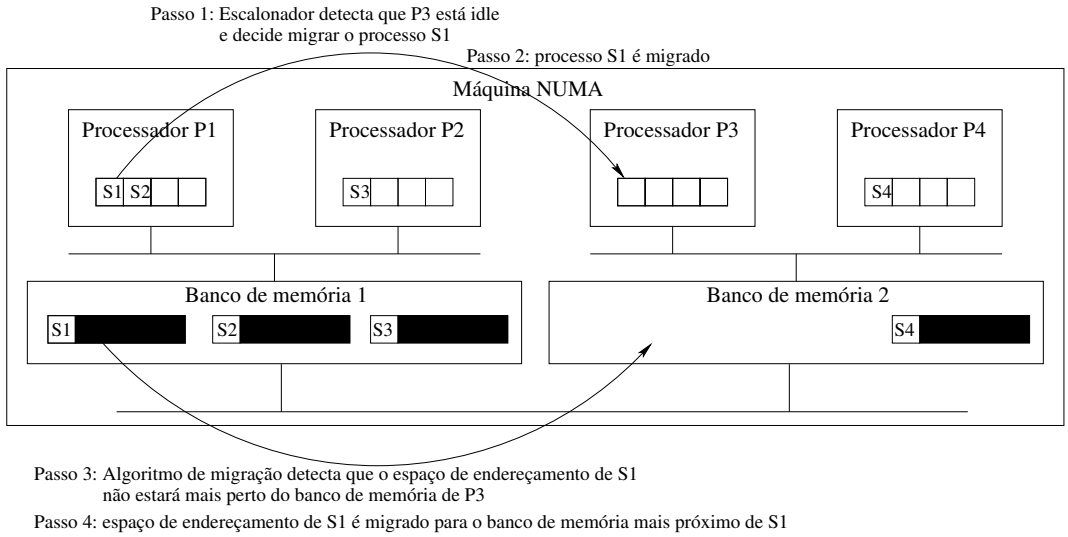


Figura 2.2: Migração do espaço de endereçamento de um processo

Este trabalho analisa duas abordagens que consideram a migração do espaço de endereçamento de um processo migrado para o banco de memória mais próximo ao

processador ao qual o escalonador migrou o processo: (i) migrar todo o conteúdo da memória do processo no mesmo momento em que o processo migra (Migração Total de Memória); e (ii) migração do conteúdo da memória do processo conforme o processo continua a executar (Migração de Memória sob Demanda).

2.3.1 Migração Total de Memória

Esta estratégia considera a possibilidade de migrar toda a memória do processo no momento em que o processo está sendo migrado. Normalmente, o Linux migra um processo quando um processador está livre, portanto, esta estratégia não irá diminuir o poder de processamento global do sistema. O maior *overhead* trazido ao sistema está relacionado ao acesso ao barramento da memória, isto é, mover páginas de um processo migrado de um nodo para outro pode fazer com que processos executando tenham de esperar até esta migração ser concluída.

Com o objetivo de implementar esta abordagem foram investigadas as estruturas de dados e funções providas pelo sistema operacional Linux. As estruturas e funções básicas usadas para implementar esta estratégia são: `task_struct` (Código 2), `mm_struct` (Código 3), `alloc_pages_node`, `__migrate_task` (Código 1) e `memcpy()`. Algumas dessas funções foram usadas para extrair informações necessárias no modelo simulado apresentado no Capítulo 3.

Dentre as funções envolvidas no balanceamento de carga do Linux, no Código 1 pode-se examinar a `__migrate_task()` que é a que realiza a migração de um determinado processo (`p`) de um processador (`src_cpu`) para outro (`dest_cpu`).

Ao final desta função, o processo alvo da migração já estará atribuído ao processador `dest_cpu` e com as estruturas disponíveis nela, possui-se todas as informações necessárias para a migração das páginas do processo. Assim, no caso de um sistema de migração total de páginas ser implementado seria necessário adicionar, ao final desta função, o código que tratará da migração das páginas deste processo para o nodo do processador para o qual o processo foi movido (`dest_cpu`). Como se sabe quem é o processador destino da migração, pode-se, através dele, se chegar ao nodo ao qual este processador pertence, e, além disso, como se sabe quem é o processo que foi migrado (`p`), pode-se chegar até as páginas que devem ser migradas.

Para obter o nodo ao qual `dest_cpu` pertence, basta utilizar o vetor `cpu_2_node`. Este vetor é preenchido durante a inicialização do sistema operacional e mapeia os processadores aos nodos do sistema. Assim, o valor do vetor na posição `dest_cpu` (`cpu_2_node[dest_cpu]`) retornará o identificador do nodo ao qual `dest_cpu` pertence.

```
1  static void __migrate_task(struct task_struct *p, int src_cpu, int dest_cpu)
2  {
3          runqueue_t *rq_dest, *rq_src;
4
5          if (unlikely(cpu_is_offline(dest_cpu)))
6                  return;
7
8          rq_src = cpu_rq(src_cpu);
9          rq_dest = cpu_rq(dest_cpu);
10
11         double_rq_lock(rq_src, rq_dest);
12         /* Already moved. */
13         if (task_cpu(p) != src_cpu)
14                 goto out;
15         /* Affinity changed (again). */
16         if (!cpu_isset(dest_cpu, p->cpus_allowed))
17                 goto out;
18
19         set_task_cpu(p, dest_cpu);
20         if (p->array) {
21                 p->timestamp = p->timestamp - rq_src->timestamp_last_tick
22                                 + rq_dest->timestamp_last_tick;
23                 deactivate_task(p, rq_src);
24                 activate_task(p, rq_dest, 0);
25                 if (TASK_PREEMPTS_CURR(p, rq_dest))
26                         resched_task(rq_dest->curr);
27         }
28
29         //exec_full_migration(p->mm, cpu_2_node[dest_cpu]);
30
31 out:
32         double_rq_unlock(rq_src, rq_dest);
33 }
```

Código 1: Função *migrate_ tasks*

Em posse do nodo para onde serão migradas as páginas resta se obter as informações das páginas do processo migrado para se iniciar a execução da migração das páginas.

Como se está em posse, também, da estrutura do processo que foi migrado (`task_struct *p`, mostrada resumidamente no Código 2), pode-se chegar a estrutura `mm_struct *mm` (linha 6) que representa a estrutura de memória deste processo.

```
1   struct task_struct {
2           volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
3           struct thread_info *thread_info;
4           atomic_t usage;
5           ...
6           struct mm_struct *mm, *active_mm;
7           ...
8           pid_t pid;
9           ...
10  };
```

Código 2: Estrutura *task_struct*

Na estrutura `mm_struct` estão todas as referências às informações da memória do processo: código, dados, pilha, argumentos, endereço da área de *swap*, contexto, etc. como pode ser visto no Código 3.

```
1   struct mm_struct {
2       ...
3       pgd_t * pgd;
4       unsigned long start_code, end_code, start_data, end_data;
5       unsigned long start_brk, brk, start_stack;
6       unsigned long arg_start, arg_end, env_start, env_end;
7       unsigned long swap_address;
8       mm_context_t context;
9       ...
10  };
```

Código 3: Estrutura *mm_struct*

No caso da migração de páginas, o que necessita ser alterado nessa estrutura são exatamente as referências ao código, dados, argumentos e pilha (linhas 4, 5 e 6). Estas referências serão modificadas a fim de refletirem a migração desses dados para outro nodo. Essa migração é feita através da cópia das informações referenciadas por estas estruturas para o nodo destino da migração, que já foi descoberto através do vetor `cpu_2_node`, anteriormente explicado. Desta forma, necessita-se calcular o tamanho do espaço ocupado por estas estruturas, e, em posse deste valor, alocar espaço na memória do nodo destino para receber a cópia das mesmas. Esta alocação deve ser feita através da função `alloc_pages_node` que permite a alocação de memória em um determinado nodo. Sendo assim, após feita a alocação, basta copiar o conteúdo referenciado pelas variáveis com as informações de código, dados, argumentos e pilha do processo (linhas 4, 5 e 6 do Código 3)

para a área recém alocada no novo nodo, atualizando as referências das mesmas para o nodo destino e liberando a área de memória do nodo fonte.

O Código 4 mostra uma possível implementação do algoritmo de migração total de páginas descrito no parágrafo acima, que é chamada mediante a remoção do comentário da linha 29 do Código 1.

```
1  void exec_full_migration (struct mm_struct *mm, int dest_node)
2  {
3          byte *buffer;
4          long size;
5
6          size = mm->end_code - mm->start_code;
7          buffer = alloc_pages_node(dest_node, GFP_KERNEL, sqrt(size))
8          memcpy(buffer, mm->start_code, size);
9          mm->start_code = buffer;
10         mm->end_code = mm->start_code + size;
11
12         size = mm->end_data - mm->start_data;
13         buffer = alloc_pages_node(dest_node, GFP_KERNEL, sqrt(size))
14         memcpy(buffer, mm->start_data, size);
15         mm->start_data = buffer;
16         mm->end_data = mm->start_data + size;
17
18         size = mm->arg_end - mm->arg_start;
19         buffer = alloc_pages_node(dest_node, GFP_KERNEL, sqrt(size))
20         memcpy(buffer, mm->arg_start, size);
21         mm->arg_start = buffer;
22         mm->arg_end = mm->arg_start + size;
23 }
```

Código 4: Código que realiza a migração de páginas do processo

2.3.2 Migração de Memória sob Demanda

Embora migrar todo o espaço de endereçamento do processo ao mesmo tempo que o processo está migrando pode aumentar o desempenho do sistema, migrar todo o espaço de endereçamento do processo toda vez que um processo é migrado pode causar a diminuição do desempenho se o processo migra diversas vezes durante seu tempo de execução. A segunda abordagem analisada neste trabalho considera a mesma estratégia usada pelo sistema de memória virtual do Linux quando este faz *swap* de páginas do disco. Nossa estratégia marca todas as páginas do processo como não presentes na memória (*swapped out*) e, quando uma falta de página (*page fault*) ocorre, é verificado se a mesma foi gerada por um processo migrado ou por um processo que tinha, de fato, páginas armazenadas no disco. Se a falta

de página for relacionada a um processo migrado, nosso sistema move as páginas do nodo remoto para o nodo atual do processo. A implementação deste algoritmo no *kernel* do Linux está em andamento e está sendo realizada em parceria com a Hewlett-Packard.

A idéia básica da implementação atual é a de quando um tratador de falta de página (`do_swap_page`, `filemap_nopage`, ...) é acionado e a página que gerou esta falta se encontra na *cache* sem mapeamento[1] e sem *writebacks* pendentes (ou seja, o conteúdo da página na *cache* e na memória estão sincronizados), é um momento interessante para se verificar se está página se encontra na memória do nodo mais próximo do processador no qual está executando. Assim, as funções que fazem o tratamento de uma falta de página foram alteradas, passando a fazer uma chamada para a função `check_migrate_misplaced_page`, mostrada no Código 5, onde se faz a verificação da necessidade de migrar a página que gerou a falta.

```
1   static inline struct page *check_migrate_misplaced_page(struct page *page,
2                   struct vm_area_struct *vma, unsigned long address)
3   {
4           int polnid, misplaced;
5
6           if (!(current->flags & PF_MIGONFAULT) || page_mapcount(page) ||
7                           PageWriteback(page) ||
8                           page_count(page) != 2 + !!PagePrivate(page) ||
9                           !page_mapping(page)->a_ops->migratepage)
10                  return page;
11
12          misplaced = mpol_misplaced(page, vma, address, &polnid);
13          if (!misplaced)
14                  return page;
15
16          return migrate_misplaced_page(page, polnid,
17                          misplaced_is_interleaved(misplaced));
18
19  }
```

Código 5: Função `check_migrate_misplaced_page`

Se o algoritmo de migração sob demanda (`PF_MIGONFAULT`) estiver habilitado para a *task* que gerou a falha (`current`), e a página procurada não tiver mapeamento, nem tiver algum *writeback* pendente, e esta página tendo suporte a migração, então é verificado se a mesma se encontra no nodo mais próximo do processador no qual a tarefa atual (*current*) está executando. Esta verificação

[1] Quando o processador procura uma página na *cache* e não a encontra, o tratador da falta de página aloca uma página e a lê da memória, zerando sua estrutura `mapping`.

é feita através da chamada da função `mpol_misplaced` (linha 12). Verificando-se que a página não está no nodo mais próximo do processador, a migração da mesma é feita através da chamada da função `migrate_misplaced_page` (linhas 16 e 17), senão, se por alguma razão a página não pode ou não deve migrar, a mesma é retornada pela função `check_migrate_misplaced_page` causando efeito nenhum sobre o algoritmo de gerência de memória do Linux.

A função `mpol_misplaced`, mostrada nos Códigos 6 e 7, conforme já dito anteriormente, é a responsável por verificar se a página que gerou a falta precisa ser migrada ou não. A função faz esta análise com base na política de alocação de páginas configurada para o sistema de gerência de memória. O suporte a NUMA oferecido pelo Linux permite ao usuário dar dicas sobre em que nodo(s) a memória deve ser alocada. Estas dicas são dadas através das políticas de alocação de memória. As políticas implementadas pelo Linux são:

- *Interleave* - Aloca memória de forma entrelaçada entre um conjunto de nodos;
- *Bind* - Aloca memória somente de um conjunto específico de nodos;
- *Preferred* - Tenta a alocação em um nodo específico, se não conseguir utiliza a política *Default*. O valor -1 significa alocação no nodo local.
- *Default* - Aloca sempre no nodo local.

O algoritmo implementado pela função `mpol_misplaced` é bastante simples. Ele somente pesquisa o identificador do nodo onde a página que gerou a falta de página se encontra e o compara com o identificador do nodo onde a *task* em questão se encontra. Se forem iguais, não há necessidade de migração, sendo retornado o valor 0, caso contrário, retorna-se um valor diferente assinalando a necessidade de migração da página. Como já dito anteriormente, esta pesquisa é feita com base na política de alocação páginas configuradas. Na linha 15, faz-se o tratamento para o caso da política estar configurada como *Interleaved*. Nesta política, o nodo alvo da alocação é definido pelo *offset* do endereço da página, portanto, para descobrir o identificador deste nodo são feitas operações sob o endereço da página (linhas 21 e 22), obtendo-se em `polnid` o identificador procurado. Caso este identificador (que referencia o nodo no qual o processo que gerou a falta de página se encontra) for igual ao do nodo onde a página em questão se encontra (linha 25), não há necessidade de migração pois ambos já estão no mesmo nodo, caso contrário, a função retorna sinalizando que a migração é necessária e o nodo alvo da migração é passado por referência através da variável `newnid` (linha 28).

```
1  int mpol_misplaced(struct page *page, struct vm_area_struct *vma,
2                          unsigned long addr, int *newnid)
3  {
4          struct mempolicy *pol;
5          struct zonelist *zl;
6          int curnid = page_to_nid(page);
7          int polnid = -1;
8          int i;
9
10         BUG_ON(!vma);
11
12         cpuset_update_task_memory_state();
13         pol = get_vma_policy(current, vma, addr);
14
15         if (unlikely(pol->policy == MPOL_INTERLEAVE)) {
16                 unsigned long pgoff;
17
18                 BUG_ON(addr >= vma->vm_end);
19                 BUG_ON(addr < vma->vm_start);
20
21                 pgoff = vma->vm_pgoff;
22                 pgoff += (addr - vma->vm_start) >> PAGE_SHIFT;
23                 polnid = offset_il_node(pol, pgoff);
24
25                 if (curnid == polnid)
26                         return 0;
27
28                 *newnid = polnid;
29                 return MPOL_MIGRATE_INTERLEAVED;
30         }
```

Código 6: Função `mpol_misplaced` - parte 1.

Os Códigos 7 e 8 mostra a seqüência da função `mpol_misplaced`, onde são tratadas as demais políticas de alocação de páginas. Para descobrir o nodo no qual o processo se encontra quando a política utilizada é a *Preferred*, basta-se verificar qual o nodo foi configurado para a alocação (linha 33), verificando se o caso especial (nodo configurado para a alocação igual a -1), onde então deve-se alocar a página no nodo local (linha 35).

No caso de se estar usando a política *Bind*, verifica-se se o nodo onde a página que gerou a falta ocorreu se encontra na lista de nodos configurados para terem páginas alocadas (linhas 39 a 43). Caso se encontre, a migração não é necessária (a política de alocação tem prioridade maior do que a migração), caso contrário, a migração é feita para o primeiro nodo da lista (linhas 45, 46 e 47).

```
31          switch (pol->policy) {
32          case MPOL_PREFERRED:
33                  polnid = pol->v.preferred_node;
34                  if (polnid < 0)
35                          polnid = numa_node_id();
36                  break;
37          case MPOL_BIND:
38                  zl = pol->v.zonelist;
39                  for (i = 0; zl->zones[i]; i++) {
40                          int nid = zl->zones[i]->zone_pgdat->node_id;
41
42                          if (nid == curnid)
43                                  return 0;
44
45                          if (polnid < 0 &&
46                                  node_isset(nid, current->mems_allowed))
47                                  polnid = nid;
48                  }
49                  if (polnid >= 0)
50                          break;
```

Código 7: Função `mpol_misplaced` - parte 2.

```
51          case MPOL_INTERLEAVE: /* should not happen */
52          case MPOL_DEFAULT:
53                  polnid = numa_node_id();
54                  break;
55          default:
56                  polnid = 0;
57                  BUG();
58          }
59
60          if (curnid == polnid)
61                  return 0;
62
63          *newnid = polnid;
64          return MPOL_MIGRATE_NONINTERLEAVED;
65  }
```

Código 8: Função `mpol_misplaced` - parte 3.

No último caso, se configurada a política de alocação *Default*, a alocação é sempre feita no nodo local, este é, portanto, o nodo onde se encontra o processo e para onde a página deve ser migrada caso necessário. A necessidade da migração é

verificada no código das linhas 60 e 61. Como no caso da política *Interleave*, caso a migração seja necessária, o nodo para onde a página deve ser migrada retorna através da variável `newnid`.

Após a função `mpol_misplaced` executar, ela retorna para a `check_migrate_misplaced_page` sinalizando a necessidade de se realizar a migração. No caso de ser necessária a migração, a função `migrate_misplaced_page` para que esta gerencie a cópia do conteúdo de uma página para a outra, bem como a configuração dos atributos da mesma e outras estruturas de controle do sistema de gerência de memória.

A função `migrate_misplaced_page` mostrada nos Códigos 9 e 10, inicia fazendo algumas verificações para certificar-se de que os requisitos para a migração estão satisfeitos. A primeira verificação analisa se não houve mudança em relação ao mapeamento da página (linha 9), já verificado anteriormente. Em seguida, na linha 11, remove a página que sofrerá migração da lista das páginas mais recentemente acessadas. Após, da linha 15 a 18, realiza-se a alocação de uma página no nodo alvo da migração (conforme a política de alocação usada), e, então, inicia-se o procedimento de cópia dos atributos e conteúdo da página.

```
1   struct page *migrate_misplaced_page(struct page *page,
2                                       int dest, int interleaved)
3   {
4           struct page *newpage;
5           struct address_space *mapping = page_mapping(page);
6           unsigned int gfp;
7           int rc;
8
9           if (page_mapcount(page))
10                  goto out_nolru; /* race in fault path? */
11          if (isolate_lru_page(page, NULL)) /* incrs page count on success */
12                  goto out_nolru; /* we lost */
13          gfp = (unsigned int)mapping_gfp_mask(mapping);
14
15          if (interleaved)
16                  newpage = alloc_page_interleave(gfp, 0, dest);
17          else
18                  newpage = alloc_pages_node(dest, gfp, 0);
19
20          if (!newpage)
21                  goto out;       /* give up */
```

Código 9: Função `migrate_misplaced_page` - parte 1.

As linhas 24 e 25 do Código 10 fazem a cópia de estruturas de controle da

página, e, na linha 26, é chamada a função que realizará a cópia do conteúdo da página de um nodo para o outro. Esta função é dependente da arquitetura da máquina e implementada diretamente em linguagem de baixo nível. Caso a cópia do conteúdo tenha falhado (linha 28), realiza-se a liberação da memória da página recém alocada e cancela-se a migração. Caso contrário (linha 31), as funções `get_page` e `set_page` são chamadas para setar o número de acessos à página e, por fim, na linha 39, a página no nodo alvo da migração é inserida na lista LRU.

```
22          lock_page(newpage);
23          newpage->index = page->index;
24          newpage->mapping = page->mapping;
25          rc = mapping->a_ops->migratepage(mapping, newpage,
26                                                   page, 1);
27          if (rc) {
28                  unlock_page(newpage);
29                  __free_page(newpage);
30          } else {
31                  get_page(newpage);
32                  put_page(page);
33                  unlock_page(page);
34                  put_page(page);
35                  page = newpage;
36          }
37  out:
38          move_to_lru(page);              /* ultimately, drops a page ref */
39  out_nolru:
40          return page;                    /* locked, to complete fault */
41  }
```

Código 10: Função `migrate_misplaced_page` - parte 2.

Ao final de todo o processo de migração de página, tendo essa ocorrido ou não, o tratamento da falta de página por parte do gerente de memória do sistema operacional continua normalmente, sendo a migração da página somente uma nova configuração do sistema, transparente ao atual gerente de memória do Linux.

2.4 Considerações Finais

Neste capítulo foram mostrados os algoritmos de migração de memória propostos e também possíveis implementações destes dois algoritmos no código fonte do sistema operacional Linux. No Capítulo 3 será apresentada a ferramenta de simulação utilizada no desenvolvimento do simulador do sistema de gerência de

memória, bem como o modelo simulado utilizado no trabalho para avaliar o desempenho dos algoritmos de migração de memória propostos.

Capítulo 3

Modelo Simulado

A fim de comparar o desempenho do algoritmo do gerente de memória do Linux quando ocorre migração de processos em máquinas NUMA com o mesmo algoritmo usando os modelos propostos, foi implementado um modelo simulado utilizando a ferramenta de simulação *JavaSim* [Javasim-Group, 2005].

3.1 Javasim

JavaSim é a implementação Java da biblioteca C++SIM [C++SIM-Group, 2005, Little and McCue, 1994] que suporta um modelo de simulação de eventos contínuos em um tempo discreto. O escalonamento de eventos é orientado a processos, isto é, o gerenciamento dos eventos é implícito no gerenciamento dos processos [Banks et al., 2001]. Modelos de sistemas específicos podem ser construídos através da herança das classes do *JavaSim*.

O *framework Javasim* faz a gerência da simulação de uma forma que se assemelha ao gerenciamento de processos de um sistema operacional. Este *framework* possui um escalonador que é o responsável pelo ordenamento da execução dos processos *Javasim* submetidos à simulação. Este gerenciamento se dá através do tempo de ativação dos processos, sendo assim, pode-se verificar que a passagem do tempo na simulação se dá através do tempo de ativação e o tempo de execução dos processos. As figuras a seguir exemplificam o gerenciamento da execução dos processos *Javasim* feito pelo escalonador do *framework*.

O cenário exemplificado na Figura 3.1 mostra uma simulação em que 3 processos ($P1$, $P2$ e $P3$) devem ser executados. Cada um dos processos possui um tempo de ativação (TA) que representa o momento no espaço de tempo em que o processo inicia sua execução. Da mesma forma, cada processo tem ainda um tempo de execução (TE) que determina o tempo que ele precisa para realizar toda a sua execução.

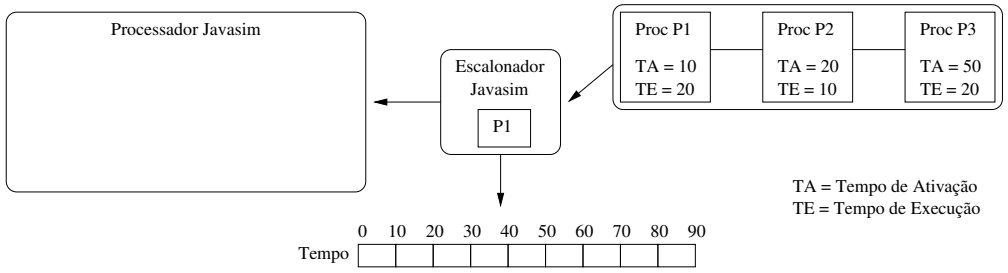


Figura 3.1: Funcionamento do Javasim - Parte 1 (Estrutura das tarefas)

O escalonador do *Javasim* funciona baseado em eventos que são marcados pelo tempo de ativação e o tempo de execução dos processos da simulação. Assim, em posse dessas informações, o escalonador do *Javasim* gerencia o tempo de inicialização dos processos, bem como o andamento do "relógio" do sistema. Pode-se pensar na estruturação das tarefas a serem realizadas pelo escalonador do *Javasim* como a Tabela 3.1 (que estrutura os eventos da simulação da Figura 3.1).

Tabela 3.1: Tempos e eventos da simulação.

Tempo	**Ação**
10	Inicializar $P1$
20	Inicializar $P2$
30	Finalizar $P1$ e $P2$
50	Inicializar $P3$
70	Finalizar $P3$

Como mostra a tabela, pode-se verificar que do tempo 0 até o 10 não há evento, portanto o tempo do sistema, "salta" diretamente para o tempo do primeiro evento, que é a inicialização do processo $P1$. A Figura 3.2 mostra o cenário neste momento.

Em seguida, o simulador verifica que o próximo evento está marcado para o tempo 20. Configura o tempo do sistema para tal e inicializa este, como mostrado na Figura 3.3. Neste cenário, pode-se notar como o *Javasim* pode ser configurado para a execução de processos em paralelo. Neste momento, tem-se $P1$ e $P2$ em execução.

Os próximos eventos ocorrerão no tempo 30. Portanto, o tempo do sistema "salta" para este valor e os dois processos em execução são finalizados, conforme os eventos estão configurados para serem simulados. A Figura 3.4 apresenta o cenário neste tempo.

No tempo 50, o processo $P3$ está marcado para ser inicializado, e, então, no-

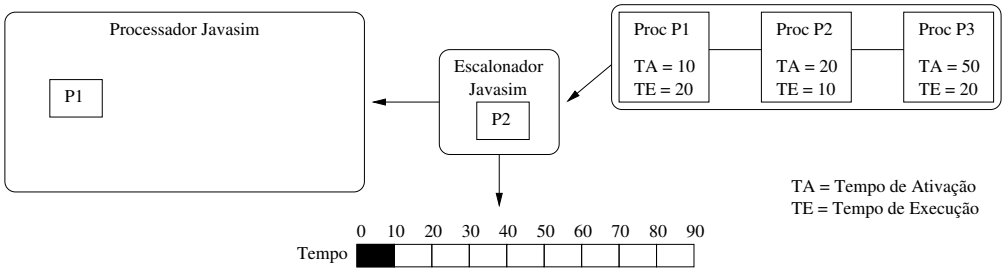


Figura 3.2: Funcionamento do Javasim - Parte 2 (Inicializar $P1$)

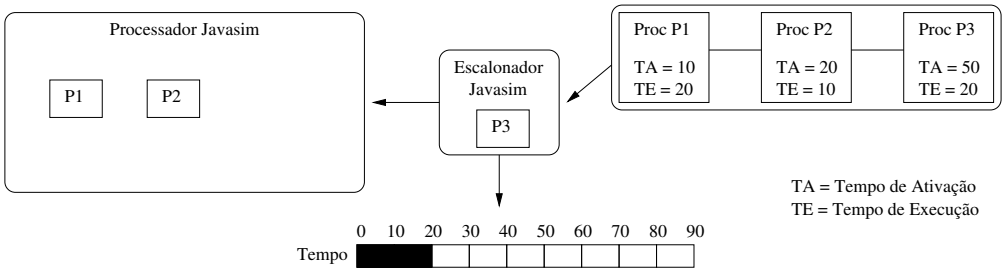


Figura 3.3: Funcionamento do Javasim - Parte 3 (Inicializar $P2$)

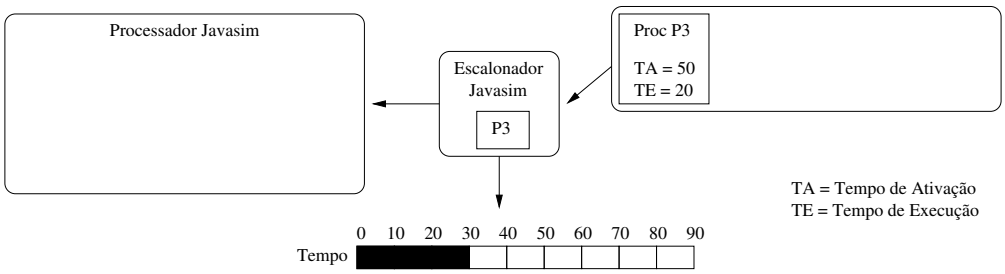


Figura 3.4: Funcionamento do Javasim - Parte 4 (Finalizar $P1$ e $P2$)

vamente o "relógio" do sistema é alterado, agora ao tempo 50, e tal processo é inicializado. A Figura 3.5 mostra esta situação. Da mesma forma como ocorrido com os demais processos já finalizados, o "relógio" do sistema irá direto ao tempo 70, onde existe o evento de finalização do último processo a ser executado que finalizará a simulação.

Este exemplo mostrou o funcionamento básico do *Javasim*, porém, diversas outras situações podem ser criadas gerando-se eventos conforme a necessidade do que se deseja simular. Um processo pode ser configurado para ser executado em

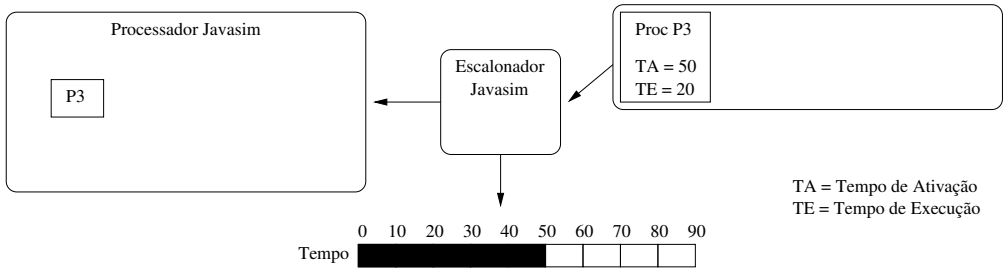


Figura 3.5: Funcionamento do Javasim - Parte 5 (Inicializar $P3$)

duas partes, por exemplo. Nesse caso, a partir do seu tempo de inicialização, o processo executará até ser removido do processador e novamente passado a fila dos processos a serem executados, agora, entretanto com seu tempo de ativação alterado para algum tempo no futuro e o seu tempo de execução decrescido do tempo que este já executou na primeira vez em que esteve no processador.

3.2 Implementação do Simulador

O sistema modelado simula os algoritmos do escalonador de processos e do balanceador de carga do Linux para diferentes arquiteturas e cargas de trabalho [Corrêa et al., 2005]. Internamente à modelagem do escalonador, foi adicionado código para a configuração do sistema de gerência de memória da máquina simulada. A Figura 3.6 mostra o diagrama de classes do modelo simulado.

Existem três classes que implementam processos de simulação, ou seja, classes que estendem a classe *SimulationProcess* do *Javasim*: *NumaMachine*, *Arrivals* e *Processor*. O modelo contém outras classes que são usadas pelos processos da simulação, como por exemplo, a classe *Task*.

Em uma simulação, existe somente uma instância de *NumaMachine*, a qual contém informações sobre a topologia da máquina simulada, bem como é a responsável por configurar a simulação de acordo com os parâmetros informados pelo usuário. O objeto da classe *NumaMachine* controla a execução da simulação e a criação e ativação inicial de todos os outros processos da simulação.

Existe também somente uma instância da classe *Arrivals*. Este objeto é responsável pela criação das *tasks* e pela atribuição destas para os processadores de acordo com as políticas do Linux. A taxa de criação das *tasks* é definida por uma distribuição exponencial sendo a média informada na inicialização da simulação pelo usuário. A distribuição exponencial é usada neste caso (e em alguns outros que serão mencionados) pois possui a característica *memoryless* [Chanin et al., 2006b],

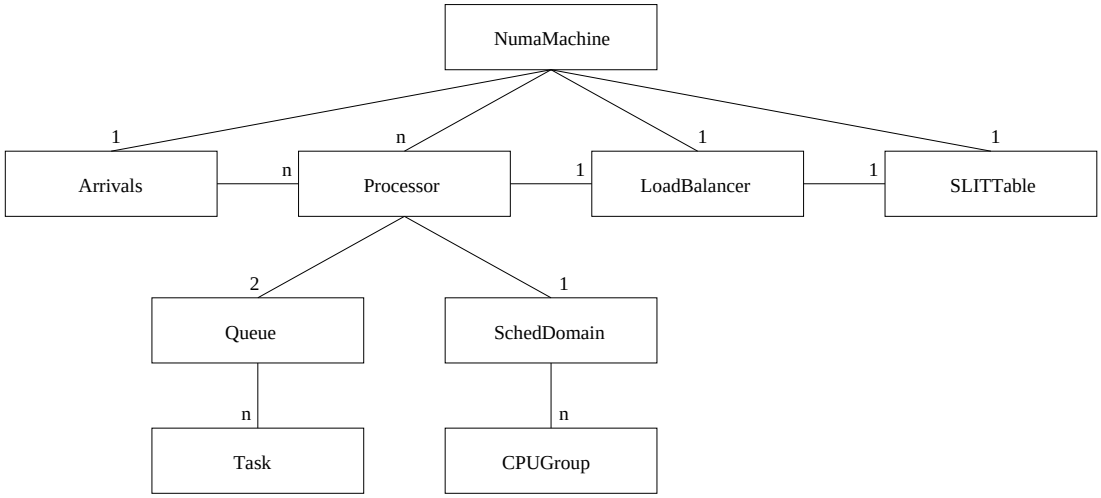


Figura 3.6: Diagrama de classes do modelo simulado

que significa que saber o que aconteceu no passado não ajuda a prever o futuro[1].

Objetos da classe *Processor* são os principais processos da simulação. Como o escalonador do Linux executa de forma independente em cada processador do sistema, os algoritmos de escalonamento e balanceamento de carga são executados por esses objetos. Esta classe é onde se encontra, também, a implementação do algoritmo de simulação de migração de páginas que será detalhado posteriormente. Todos os processadores do sistema possuem duas filas associadas (implementadas por objetos da classe *Queue*): a fila *ready*, que contém os processos que estão prontos para serem executados, e a fila *expired*, que possui os processos expirados.

Objetos da classe *Task* representam processos do Linux, estes, que possuem dois atributos que são usados pelo escalonador para definir a ordem de execução dos processos no processador. Tais atributos funcionam configurando a prioridade do processo no sistema, são eles: prioridade estática, conhecida internamente no Linux como *nice* e a prioridade dinâmica. No simulador, o valor do *nice* (prioridade estática) de cada processo é determinado por uma distribuição uniforme de -20 a 20 (estes valores são extraídos do *kernel* do Linux). O valor da prioridade dinâmica, bem como o valor da fatia de tempo de cada processo no processador são calculados baseado no valor do *nice*. O tempo de execução de uma *task* também é definido por uma distribuição exponencial com a média usada sendo informada pelo usuário. Este valor define o tempo que esta *task* executaria se não fosse interrompida.

[1]Por exemplo, considerando um evento contínuo como a chegada de *tasks* em um sistema com distribuição exponencial e média 5s, o fato de até o tempo 4s não ter chegado nenhuma *task* não aumenta a probabilidade de uma chegar no segundo seguinte.

Similar ao Linux, *tasks* não são classificadas como totalmente *CPU-bound* ou *I/O-bound*, elas podem ser mais ou menos *CPU* ou *I/O-bound*, dependendo do tempo que elas gastam executando (*processing time*) e o tempo que elas esperam pelo término de operações de I/O (*waiting time*). Portanto, o tempo que uma *task* irá executar sem perder o processador é definido por uma distribuição exponencial e é sempre menor ou igual a fatia de tempo deste processo. Se a *task* executar por menos do que a sua fatia de tempo, significa que ela perdeu o processador à espera de alguma operação de I/O. O tempo que a tarefa ficará no estado de espera, a taxa de acesso à memória do processo, assim como a taxa de acessos a posições de memória já acessadas por ele e o tamanho do processo (*task*) também são definidos por uma distribuição exponencial.

Além das informações sobre as características dos processos do Linux, os seguintes valores são necessários para iniciar uma simulação: (i) distância dos nodos; (ii) número de *tasks* que serão criadas e executadas pelos processadores; (iii) número de processadores e nodos da máquina simulada; (iv) tipo de migração de páginas que define se será utilizado o algoritmo de migração total, migração sob demanda, ou se será utilizado o algoritmo do Linux que não implementa migração; (v) taxa média de criação de *tasks*; (vi) tempo médio de execução das *tasks*; (vii) tempo médio de processamento das *tasks*; (viii) tempo médio de espera das *tasks* (*waiting time*); e (ix) tamanho médio de uma *task*.

3.2.1 Implementação de Migração de Páginas no Simulador

O simulador proposto implementa os dois algoritmos de migração estudados (migração total e migração sob demanda) e também o algoritmo de gerência de memória do Linux, o qual não implementa nenhum tipo de migração de páginas.

No modelo simulado, a implementação da parte do gerente de memória responsável por lidar com a migração de processos e páginas encontra-se no escalonador (na classe *LoadBalancer*).

Todos os processos (instâncias da classe *Task*) encontram-se em uma fila (*ready* ou *expired*) de algum processador. Assim, a migração de um processo é realizada pelo simulador simplesmente movendo-se o processo selecionado para sofrer migração da fila onde se encontra para a mesma fila, só que de outro processador (para maiores detalhes sobre migração de processos no Linux, ver [Corrêa, 2005]).

O impacto causado no desempenho do sistema quando um processo é migrado é verificado pelo tempo que o processo migrado levará para executar. Quando um processo é criado, ele tem um tempo de execução. Como já dito na Seção 3.1, este tempo representa o tempo que o processo leva para executar sem ser interrompido no processador ao qual foi atribuído ao ser criado (lembre-se de que ao ser criado o processo é carregado no banco de memória mais "próximo" do processador ao qual foi atribuído). Considerando-se a implementação do Linux do gerente de

memória, o tempo de execução deste processo poderá ser alterado, já que o acesso à memória deste processo poderá ter um "custo" maior, dependendo da "distância" entre o processador que recebeu o processo migrado e o banco de memória onde o processo foi carregado inicialmente.

Desta forma, o que ocorre no simulador quando um processo sofre migração, é o recálculo do tempo de execução do processo, ou seja, recalcula-se quanto tempo a mais o processo irá levar para executar depois de mudar de processador. O Código 11 mostra o código do simulador que faz o recálculo do tempo final do processo migrado quando o algoritmo do gerente de memória usado é o implementado pelo sistema operacional Linux.

```
1  double correction = 0;
2
3  if (task.getLastTimeToFinish() != 0) {
4          double addedTime=task.getExecutionTime()-task.getLastExecutionTime();
5          double ranTime = (task.getExecutionTime() - task.getTimeToFinish()) -
6                  (task.getLastExecutionTime() - task.getLastTimeToFinish());
7          correction = (addedTime - ranTime);
8  }
9
10 if (NUMAMachine.MMType == 0) { //algoritmo do Linux
11         nrMemoryAccesses = (task.getTimeToFinish()*task.getMemBound()/100) /
12         (NUMAMachine.LOCAL_MEMORY_ACCESS_TIME * slit.nodeDistance(sourceNode,
13         destNode)/10);
14         if (nrMemoryAccesses > task.getSize()) {
15                 nrMemoryAccesses = task.getSize();
16         }
17         newTime = currentTime + nrMemoryAccesses *
18         NUMAMachine.LOCAL_MEMORY_ACCESS_TIME * slit.nodeDistance(sourceNode,
19         destNode)/10 - correction;
20 }
```

Código 11: Algoritmo do Linux para recálculo do tempo de execução do processo migrado

A linha 10 do Código 11 verifica se a simulação está configurada para simular o algoritmo do Linux. As linhas 11, 12 e 13 são as responsáveis por calcular quantos acessos à memória este processo ainda fará. Este cálculo é feito da seguinte forma: pega-se o tempo que resta para este processo executar (`task.getTimeToFinish()`) e multiplica-se pela porcentagem do tempo que este processo acessa a memória (`task.getMemBound()`). Este valor é o tempo que resta para este processo terminar. Dividindo-se este valor pelo tempo de acesso à memória do processador ao qual ele será migrado até o banco de memória onde o processo foi carregado,

resultará no total de acessos que o processo fará até o seu término (caso ele execute até o final no processador ao qual está sendo migrado). Em posse do número de acessos restantes, o tempo a mais que o processo levará para executar devido à migração é igual ao número de acessos calculado multiplicado pelo tempo de cada acesso (agora "remoto"). As linhas 17, 18 e 19 fazem este cálculo. A variável `currentTime` contém o tempo do término do processo antes da migração, portanto, ela é somada ao incremento gerado pela migração (explicação acima) e é subtraído um fator de correção (`correction`). Este fator de correção é necessário, pois quando uma migração é feita, o recálculo do tempo de execução do processo é feito supondo-se que o processo execute até o seu final no processador ao qual migrou. Porém, no caso deste processo migrar novamente, o tempo que resta para o final de sua execução deve desconsiderar o tempo que se supôs que ele executaria no processador do qual foi migrado. Este cálculo é feito pelo código das linhas 3 à 8 do Código 11. Na linha 4, calcula-se o tempo adicionado na migração anterior (`addedTime`). Isto é feito simplesmente subtraindo-se o tempo de execução total do processo (`task.getExecutionTime()`) pelo tempo de execução total do processo antes da última migração ocorrida (`task.getLastExecutionTime()`) do mesmo. Nas linhas 5 e 6, calcula-se quanto tempo o processo executou depois da última migração. Este cálculo é feito subtraindo-se o tempo que o processo executou até antes da última migração ocorrida, do tempo que este processo executou até o atual momento (`ranTime`). Subtraindo-se `ranTime` de `addedTime` (linha 7) tem-se o tempo a ser descontado na migração corrente (`correction`).

3.2.1.1 Migração Total

O algoritmo para a simulação da migração total não precisa lidar com o tempo de acesso à memória, pois como toda vez que um processo migrar para outro processador, seu espaço de endereçamento, automaticamente, migrará para o banco de memória mais próximo deste processador. Sendo assim, todos acessos à memória terão o mesmo custo. O algoritmo que faz o recálculo do tempo de execução de um processo migrado é mostrado no Código 12.

```
21  else if (NUMAMachine.MMType == 1) { //full migration
22          newTime = currentTime + (task.getSize() *
23          NUMAMachine.LOCAL_MEMORY_ACCESS_TIME * slit.nodeDistance(sourceNode,
24          destNode)/10);
25  }
```

Código 12: Algoritmo de migração total

A linha 21 do Código 12 verifica se o algoritmo de migração de páginas a ser

usado é o de migração total. As linhas 22, 23 e 24 do mesmo código fazem o recálculo do tempo de execução do processo migrado. Este cálculo é realizado acrescentando-se ao tempo final do processo inicialmente calculado, o tempo da migração de todo o espaço de endereçamento do processo migrado para o banco de memória mais próximo do processador ao qual o processo migrou. O tempo desta migração do espaço de endereçamento do processo é o resultado da multiplicação do tamanho do processo (`task.getSize()`) pelo tempo de acesso à memória do processador ao qual ele será migrado até o banco de memória onde o processo foi carregado (linhas 23 e 24).

3.2.1.2 Migração sob Demanda

Diferentemente do algoritmo de migração total de páginas e do algoritmo implementado pelo Linux que foram de simples implementação no simulador, a implementação do simulador para tratar do algoritmo de migração sob demanda possui uma complexidade um pouco maior por não só ter de considerar as migrações já ocorridas durante a simulação, como também de que nodo e para que nodo foram estas migrações.

O código do simulador que implementa o algoritmo de migração de páginas sob demanda está dividido em duas partes: uma que trata da primeira migração de páginas do processo migrado (Código 13) e outra parte que cuida das demais migrações (Códigos 14 e 15). Assim, o código do simulador que trata do algoritmo de migração sob demanda inicia verificando se a simulação está configurada para utilizar este algoritmo de migração de páginas (linha 26). Neste caso, o simulador busca uma estrutura (do tipo `Hashtable` chamada, no código, de `memory`) que armazena as informações de como está distribuído o espaço de endereçamento do processo (`task.getMemoryDescriptor()`) entre os nodos. Em seguida, é verificado se o processo ainda não sofreu migração na simulação corrente (linha 30). Neste caso, calcula-se o número de acessos à memória que este processo fará até o seu término. Este cálculo é feito aplicando-se a taxa de acesso à memória do processo (`task.getMemBound()`) ao tempo que resta para o processo terminar (`task.getTimeToFinish()` - linha 31), posteriormente, subtraindo-se o número de acessos a posições de memória já acessadas (pois já estarão no banco de memória mais próximo do processador, não impactando, assim, no tempo final de término do processo) e finalmente dividindo este valor pelo tempo de acesso do nodo destino da migração até o nodo origem da mesma (linhas 33, 34 e 35). O algoritmo em seguida configura a estrutura `memory` para refletir a migração dos bytes de um banco de memória para outro (linhas 41, 42 e 43) e finalmente, nas linhas 48, 49 e 50, adiciona ao tempo final do processo o tempo de todos os acessos "remotos" necessários para a migração.

Quando ocorre a migração de um processo que já havia migrado durante a simu-

lação corrente, o tratamento é diferenciado. Conforme dito anteriormente, quando um processo é migrado, recalcula-se o tempo de execução do mesmo (adicionando-se o tempo da migração das páginas), configurando-o, então, para o tempo de execução no caso deste processo não mais migrar. Juntamente com isso, no algoritmo de migração sob demanda, cada vez que o processo migra, parte de sua memória migra junto com o mesmo resultando em uma distribuição da memória do processo por diferentes nodos. Essa migração da memória reflete-se na simulação na estrutura `memory`. Da mesma forma como ocorre no recálculo do tempo de execução do processo quando este migra, o número de *bytes* de memória migrados (e configurados na estrutura `memory`) é calculado considerando-se que este processo executará até o seu final no mesmo processador.

```
26   else if (NUMAMachine.MMType == 2) { //on demand
27
28     Hashtable memory = task.getMemoryDescriptor();
29
30     if (task.getLastDestNode() == -1) {
31           nrMemoryAccesses = (task.getTimeToFinish()*task.getMemBound()/100);
32           nrMemoryAccesses -= nrMemoryAccesses*task.getRepeatedAccess()/100;
33           nrMemoryAccesses = nrMemoryAccesses /
34                   (NUMAMachine.LOCAL_MEMORY_ACCESS_TIME *
35                   slit.nodeDistance(sourceNode, destNode)/10);
36
37           if (nrMemoryAccesses > task.getSize()) {
38                   nrMemoryAccesses = task.getSize();
39           }
40
41           memory.put(new Integer(destNode), new Double(nrMemoryAccesses));
42           memory.put(new Integer(sourceNode),
43                   new Double(task.getSize()-nrMemoryAccesses));
44
45           task.setMemoryDescriptor(memory);
46           task.setLastDestNode(destNode);
47
48           newTime = currentTime + nrMemoryAccesses *
49                   NUMAMachine.LOCAL_MEMORY_ACCESS_TIME *
50                   slit.nodeDistance(sourceNode, destNode)/10;
51     }
```

Código 13: Algoritmo de migração sob demanda parte 1

Sendo assim, quando isso não ocorre, significa que a simulação utilizou no cálculo mais bytes do que efetivamente seria migrado pelo algoritmo sob demanda.

Desta forma, é necessário "devolver"[2] esses bytes migrados a mais para o nodo ao qual pertenciam antes da migração anterior.

A parte do código responsável por fazer esta "devolução" funciona da seguinte forma: calcula-se quantos bytes do processo atual se encontram em nodos diferentes daquele que atualmente contém o processo (Código 14 - linhas 56, 57 e 58).

```
52      else {
53            newTime = currentTime - correction;
54            // Distributes the bytes that should not have been migrated
55            double bytesToCorrect = 0;
56            double value = ((Double) memory.get(new Integer(
57                                    task.getLastDestNode()))).doubleValue();
58            double bytesInQuestion = task.getSize() - value;
59            Enumeration enum = memory.keys();
60            while(enum.hasMoreElements()) {
61                    int key = ((Integer) enum.nextElement()).intValue();
62                    if (key != task.getLastDestNode()) {
63                            value = ((Double) memory.get(
64                                            new Integer(key))).doubleValue();
65
66                            double nodeRate = value * 100 / bytesInQuestion;
67                            double temp = correction * nodeRate / 100;
68                            temp = temp/(NUMAMachine.LOCAL_MEMORY_ACCESS_TIME *
69                                   slit.nodeDistance(key, task.getLastDestNode())
70                                   /10) * PAGE_SIZE;
71
72                            bytesToCorrect += temp * PAGE_SIZE;
73                            value += temp * PAGE_SIZE;
74                            memory.put(new Integer(key), new Double(value));
75                    }
76            }
77            value = ((Double) memory.get(new Integer(
78                    task.getLastDestNode()))).doubleValue();
79            memory.put(new Integer(task.getLastDestNode()),
80                    new Double(value-bytesToCorrect));
```

Código 14: Algoritmo de migração sob demanda parte 2

A partir deste valor, para cada nodo da máquina calcula-se qual a porcentagem da memória do processo que este nodo possui (linha 63 à 66). Assim, aplicando-se esta porcentagem ao valor da variável `correction` (linha 67), obtem-se o tempo a mais que se supôs que o processo acessaria a sua memória no nodo em questão

[2] raciocínio similar ao da funcionalidade da variável `correction` para o tempo de execução do processo

caso este processo não mais migrasse. Dividindo-se esse tempo pelo tempo de um acesso à memória do nodo em questão a partir do nodo destino da última migração ocorrida, se transforma este tempo no número de acessos à memória a mais que se supôs seriam feitos no caso da migração anterior ter sido a última do processo. Com este valor se pode obter a quantidade de *bytes* que não deveria ter sido migrada (linha 72) e portanto é "devolvida" ao seu nodo de origem simplesmente corrigindo-se este valor na estrutura `memory` (linha 74). Note que variável `bytesToCorrect` armazena a quantidade total de *bytes* que estão sendo "devolvidos" ao seus respectivos nodos.

```
81          // Calculates number of bytes to migrate
82          double timeToAdd = task.getTimeToFinish() * task.getMemBound() /100;
83          timeToAdd -= timeToAdd * task.getRepeatedAccess() / 100;
84
85          double bytesMigrated = 0;
86          enum = memory.keys();
87          while(enum.hasMoreElements()) {
88                  int key = ((Integer) enum.nextElement()).intValue();
89                  if (key != destNode) {
90                          value = ((Double) memory.get(
91                                  new Integer(key))).doubleValue();
92                          double migrationRate = value * 100 / task.getSize();
93                          double localTimeToAdd = timeToAdd*migrationRate/100;
94                          double bytesMigratedFromThisNode = localTimeToAdd /
95                                  NUMAMachine.LOCAL_MEMORY_ACCESS_TIME *
96                                  slit.nodeDistance(key, destNode) / 10;
97                          value -= bytesMigratedFromThisNode;
98                          bytesMigrated += bytesMigratedFromThisNode;
99                          memory.put(new Integer(key), new Double(value));
100                         newTime += localTimeToAdd;
101                 }
102         }
103
104         if(memory.get(new Integer(destNode)) == null) {
105             value = 0;
106         } else {
107             value=((Double)memory.get(new Integer(destNode))).doubleValue();
108         }
109         memory.put(new Integer(destNode), new Double(value+bytesMigrated));
110
111         task.setMemoryDescriptor(memory);
112     }
113 }
```

Código 15: Algoritmo de migração sob demanda parte 3

No final da redistribuição dos *bytes* que não deveriam ter sido migrados, corrige-se o valor da quantidade de memória no nodo destino da última migração (que foi de onde foram removidos os *bytes* corrigidos) nas linhas 79 e 80.

Após o processo de correção do real estado da memória do processo nos nodos do sistema, pode-se, então, iniciar o processo de migração do qual se está tratando.

Tomando-se o tempo que resta para o processo em questão terminar sua execução (`task.getTimeToFinish()`) e multiplicando-o pela taxa de acesso à memória deste processo, obtém-se o tempo (`timeToAdd`) que este processo ainda fará acesso à memória (linha 82 - Código 15). No algoritmo de migração sob demanda, acessos a páginas de memória que já foram acessadas pelo processo não geram migração porque já foram migradas no primeiro acesso às mesmas. Assim, a taxa de acesso a posições de memória já acessadas pelo processo (`task.getRepeatedAccess()`) é aplicada sobre `timeToAdd` para que estes acessos repetidos sejam descontados do tempo que será adicionado ao tempo de término do processo. Isto é feito na linha 83 do Código 15. Após, para cada nodo do sistema que possui parte da memória do processo, o algoritmo migra a porcentagem correspondente à quantidade da memória desse processo para o nodo destino da migração. Pode-se ver que no Código 15 as linhas 90 e 91 buscam a quantidade da memória do processo que se encontra no nodo em questão (nodo da iteração corrente do laço), e a linha 92 calcula a porcentagem que este nodo possui da memória total do processo e portanto, a porcentagem do tempo que será adicionada ao tempo de término do processo devido à migração de páginas deste nodo. Dividindo-se este valor pelo tempo de acesso à memória deste nodo a partir do nodo destino da migração, se tem o número de *bytes* que será migrado deste nodo (linha 94 à 96). Em seguida, diminui-se esse número de *bytes* do nodo do qual ele migrou, atualizando-se a estrutura *memory* (linha 99). Depois de todos os nodos que possuem parte da memória do processo migrado terem passado por este processo, o nodo destino da migração possuirá todas as páginas migradas destes nodos. Desta forma, atualiza-se a estrutura `memory` novamente, para refletir esta mudança (linha 109).

3.2.2 Considerações Finais

Nesta seção foram discutidos aspectos técnicos da implementação do simulador utilizado para verificar o desempenho dos algoritmos de migração de páginas propostos na Seção 2.3.

Como visto, o modelo simulado é suficientemente flexível para prover diversos resultados. Porém, o modelo simulado não contempla o suporte à memória compartilhada entre processos, assim como não se considera, durante a simulação, o tamanho da memória *cache* dos processadores e também a capacidade da memória principal dos nodos do sistema. Neste trabalho serão apresentados os resultados das simulações, utilizando o tempo médio de execução de todos os processos criados

como base para análise. Serão simulados diversos cenários com diferentes cargas no sistema, com processos de diferentes características, em máquinas diferentes. A Seção 4 apresenta os resultados (desempenho) encontrados pelo modelo simulado, usando os algoritmos propostos no trabalho. Juntamente com a apresentação dos resultados, o desempenho dos algoritmos propostos são avaliados juntamente de forma comparativa.

Capítulo 4

Resultados Numéricos

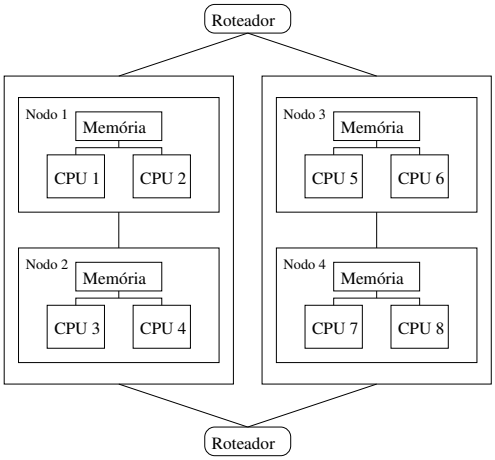
Neste capítulo são apresentados os resultados obtidos com a execução do simulador descrito. As simulações foram feitas em diversos cenários e testadas em duas diferentes máquinas (SGI Altix 3000 - Seção 4.1 e HP Integrity Superdome - Seção 4.2) verificando-se o desempenho dos algoritmos de migração de memória propostos na Seção 2.3. Além disso, também foi simulado e obtido o desempenho destes algoritmos quando o escalonador de processos é alterado (Seção 4.3), variando-se ainda mais os casos de simulação testados. Todos os resultados apresentados nessa seção possuem erro máximo de 0,83% e 5,22%, com um grau de confiança de 99%. A seguir, estes resultados são descritos.

4.1 Resultados da Simulação: SGI Altix 3000

Para a primeira execução das simulações foi usada uma máquina baseada nas arquiteturas Altix da SGI [Woodacre et al., 2005]. A máquina possui quatro nodos, oito processadores e quatro níveis de acesso à memória. A Figura 4.1 ilustra a máquina descrita, mostrando também a SLIT, em que se pode verificar os 4 níveis de acesso à memória através das distâncias entre os nodos.

Considerando os resultados das simulações, estes foram baseados no tempo total de execução de todos os processos submetidos ao sistema. A configuração do ambiente foi variada da seguinte forma: tempo médio de execução dos processos, número de processos no sistema, tamanho do processo, taxa de acesso à memória do processo e taxa de repetição de acesso a uma mesma área de memória já acessada pelo processo. Para cada caso de teste foram feitas 200 simulações considerando-se como resultado a média das mesmas. As variáveis que configuram o ambiente foram definidas a fim de cobrir a maioria dos casos possíveis.

Dentre as variáveis anteriormente citadas, o tempo médio de execução dos processos foi utilizado com o objetivo de se analisar o comportamento do sistema



SLIT

Nodo	1	2	3	4
1	10	20	30	40
2	20	10	40	30
3	30	40	10	20
4	40	30	20	10

Figura 4.1: Máquina Altix SGI usada nas simulações

quando submetido a uma carga de processos que ficam um curto espaço de tempo executando em comparação a uma carga de processos que possuem maior tempo de processamento. Isto, devido ao fato de que em um ambiente com grande quantidade de processos sendo submetidos, quanto maior o tempo que os processos tem para executar, maior a chance destes migrarem. Desta forma, configura-se um cenário interessante para o estudo, já que se pode ter, assim, a área de memória do processo migrado em um banco de memória "remoto", ou seja, em um banco de memória que não é o mais próximos do processador para cujo este processo migrou.

A Figura 4.2 mostra o resultado das simulações com a variação do tempo médio de execução de cada processo nos seguintes valores: 0,5 até 100s com variação de 20s para cada simulação. Estes valores foram simulados em um ambiente com 50 processos, todos de 50KB, que ficam 60% do seu tempo fazendo acessos à memória sendo 40% desses acessos a posições de memória já alguma vez acessadas.

Como podemos observar, o algoritmo de migração total teve um desempenho menor comparado aos demais. Os algoritmos do Linux e o de migração de páginas sob demanda apresentam um desempenho muito similar, sendo o ganho do segundo algoritmo ficando entre 9 e 13%. Este ganho inicia em 10,8% e cresce atingindo o pico de 12,7% com processos de tempo médio de execução de 3s. A partir de então, este ganho decresce até 7,5% com processos de 15s. Pode-se explicar este comportamento porque quanto mais tempo os processos ficam executando, mais páginas da memória estes podem acessar, causando uma maior quantidade de migrações de páginas no caso do algoritmo sob demanda. Assim, o ganho deste algoritmo acaba se "suavizando", pois na medida em que esse número de migrações é maior, e sendo as páginas migradas acessadas poucas vezes, o custo do acesso

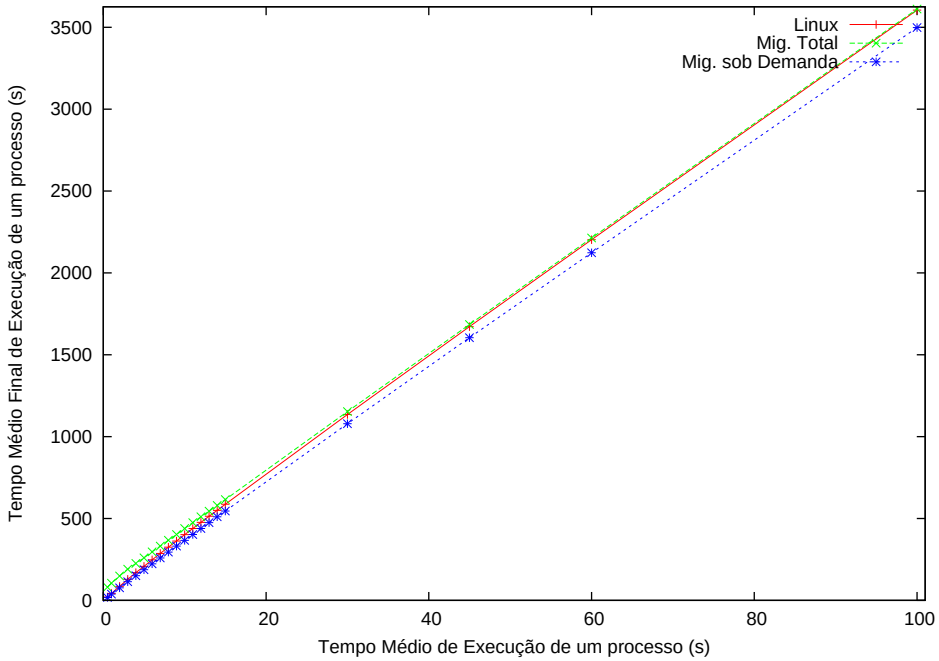


Figura 4.2: 50 processos de 50KB, taxa de acesso à memória de 60% e taxa de acesso a posições de memória já acessadas de 40%

a uma página fica cada vez mais similar ao do algoritmo do Linux. O custo do primeiro acesso a uma página depois da migração do processo é o mesmo tanto para o algoritmo do Linux quanto para o de migração sob demanda. O ganho do algoritmo de migração sob demanda se dá através dos acessos repetidos a páginas já migradas, assim, quanto menos acessos repetidos temos a uma página, menor esse ganho, e quanto mais páginas são acessadas, mais parecido fica o desempenho dos dois algoritmos, pois o custo do primeiro acesso após a migração, como já dito, é o mesmo. Com o mesmo raciocínio, podemos verificar o desempenho do algoritmo de migração total. Quanto mais páginas são acessadas pelo processo, melhor o desempenho deste algoritmo quando comparado aos outros dois. E como já dito, quanto mais tempo o processo fica executando, maior a chance disto acontecer, refletindo na diferença do ganho do algoritmo do Linux quando comparado a este. O ganho começa com quase 200%, e decresce (146%, 74%, 48%, 33%, 24%, ...) até se aproximar de 0,2% com processos de tempo médio de execução de 100s.

A fim de verificar outro fator variável no sistema, um caso interessante de se analisar é a influência do tamanho dos processos no desempenho dos algoritmos de migração. Isto, porque com processos de tamanhos maiores, temos mais espaço da memória sendo alocado para os processos, e, portanto, em caso de migração, temos

maior probabilidade de se ter o espaço de endereçamento do processo distribuído entre os bancos de memória da máquina, diferenciando o custo do acesso à esta memória dependendo de onde o processo precisa buscar dados.

A Figura 4.3 apresenta o resultado das simulações com a variação do tamanho do processo de 50, 100, 200, 300, 400 e 500KB. Estes valores foram simulados em um ambiente com 50 processos, todos com tempo médio de execução de 10s, que ficam 60% do seu tempo fazendo acessos à memória, sendo 40% desses acessos a posições de memória já alguma vez acessadas.

O algoritmo de migração total apresentou um resultado já esperado. Como neste algoritmo toda vez que ocorre migração do processo, toda sua área de memória é migrada, quanto maior o tamanho do processo, mais tempo este leva para migrar de um banco de memória para outro, sendo assim, maior proporcionalmente será o seu tempo de execução. A curva de desempenho dos algoritmos do Linux e o de migração sob demanda ficaram muito parecidas, com um ganho novamente do algoritmo de migração sob demanda. O ganho inicial de 9,3% com processos de 50KB aumentou gradativamente até 13,6% com processos de 300KB, quando se estabilizou. A estabilização deste ganho pode ser entendida neste cenário pensando-se que o fato de se aumentar o tamanho do processo não causa diferença no desempenho se o processo não tem tempo de acessar todos os seus dados. Assim, pode-se pensar que aumentando-se o tamanho do processo e da mesma forma, aumentando-se o seu tempo de execução, pode-se ter um ganho diferente do que o visto neste cenário.

Visto que a taxa de acessos repetidos a posições de memória já acessadas é um fator que diferencia o algoritmo de migração sob demanda. Algumas simulações foram executadas a fim de verificar a influência desta taxa no desempenho final do algoritmo. A taxa de acesso a posições de memória já acessadas foi variada nos seguintes valores: 20, 40, 60 e 80%. Estes valores foram simulados em diferentes ambientes, em que se variou a taxa de acesso à memória dos processos simulados.

As figuras a seguir apresentam o resultado das simulações com a taxa de acesso à memória variando nos seguintes valores: 20% (Tabela 4.1 - a), 40% (Tabela 4.1 - b), 60% (Tabela 4.1 - c) e 80% (Tabela 4.1 - d).

Pode-se verificar que o ganho do algoritmo de migração sob demanda aumenta conforme aumenta-se a taxa de acessos a posições de memória já acessadas, quando comparando-o com o algoritmo do Linux. A Tabela 4.1 apresenta o ganho do algoritmo proposto (migração sob demanda) quando comparado ao implementado pelo Linux.

Além disso, podemos verificar que conforme aumenta-se a taxa de acesso à memória, mesmo mantendo-se constante a taxa de acessos a posições de memória já acessadas, o ganho do algoritmo de migração sob demanda sob o do Linux também aumenta. A Tabela 4.1 mostra graficamente esse ganho.

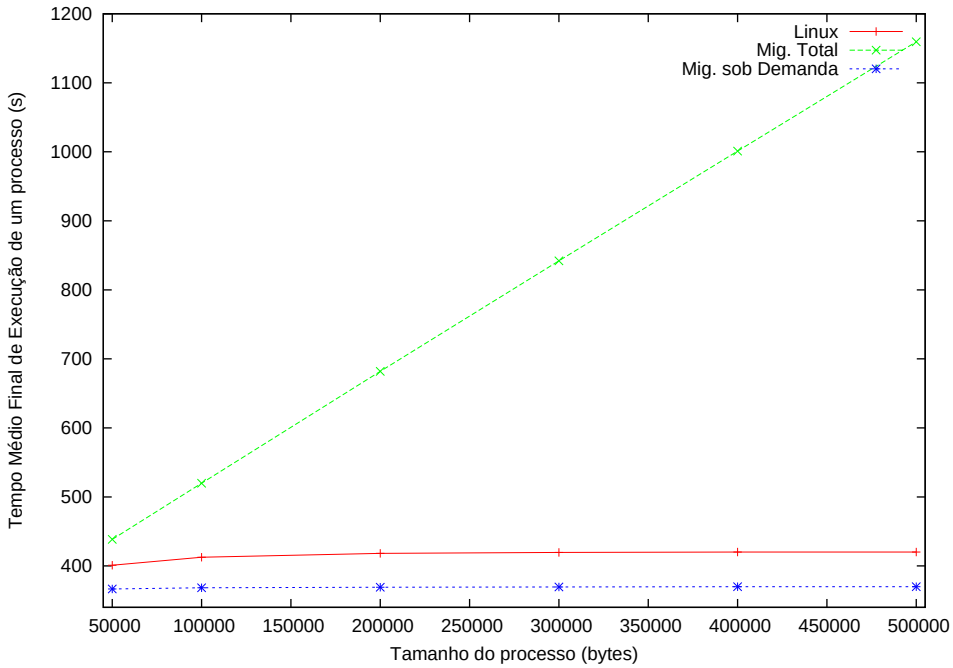


Figura 4.3: 50 processos de tempo médio de execução de 10s, taxa de acesso à memória de 60% e taxa de acesso a posições de memória já acessadas de 40%

Interessante notar que ao aumentarmos em 20% tanto a taxa de acessos a posições de memória já acessadas, quanto a taxa de acesso à memória, o aumento no ganho final do algoritmo de migração sob demanda é de no mínimo 27%, chegando a atingir quase 100% em alguns casos.

Aumentando-se a carga do sistema, submetendo-se 400 processos para serem executados, obtém-se os resultados mostrados na Tabela 4.1. Novamente, verifica-se que aumentando-se em 20% tanto a taxa de acesso à memória quanto a taxa de acessos a posições de memória já acessadas, o ganho do algoritmo de migração sob demanda é de no mínimo 30%.

Pode-se verificar, com estes resultados, que quanto maior a taxa de acesso à memória e quanto maior a taxa de acesso a posições de memória já acessadas, maior o ganho do algoritmo de migração sob demanda comparado ao algoritmo do Linux. O ganho quando aumenta-se taxa de acesso a posições de memória já acessadas pode ser explicado porque quanto mais o processo acessa páginas que já acessou, maior a chance dessas páginas já estarem no banco de memória mais próximo do processador onde o processo está executando no momento. Assim sendo, o custo do acesso a essas páginas é menor para o algoritmo de migração sob demanda em relação ao implementado pelo Linux. A explicação do ganho ao

Tabela 4.1: Ambiente com 200 processos de 50KB, tempo médio de execução de 0,5s e taxa de acesso à memória de 20%, 40%, 60% e 80%.

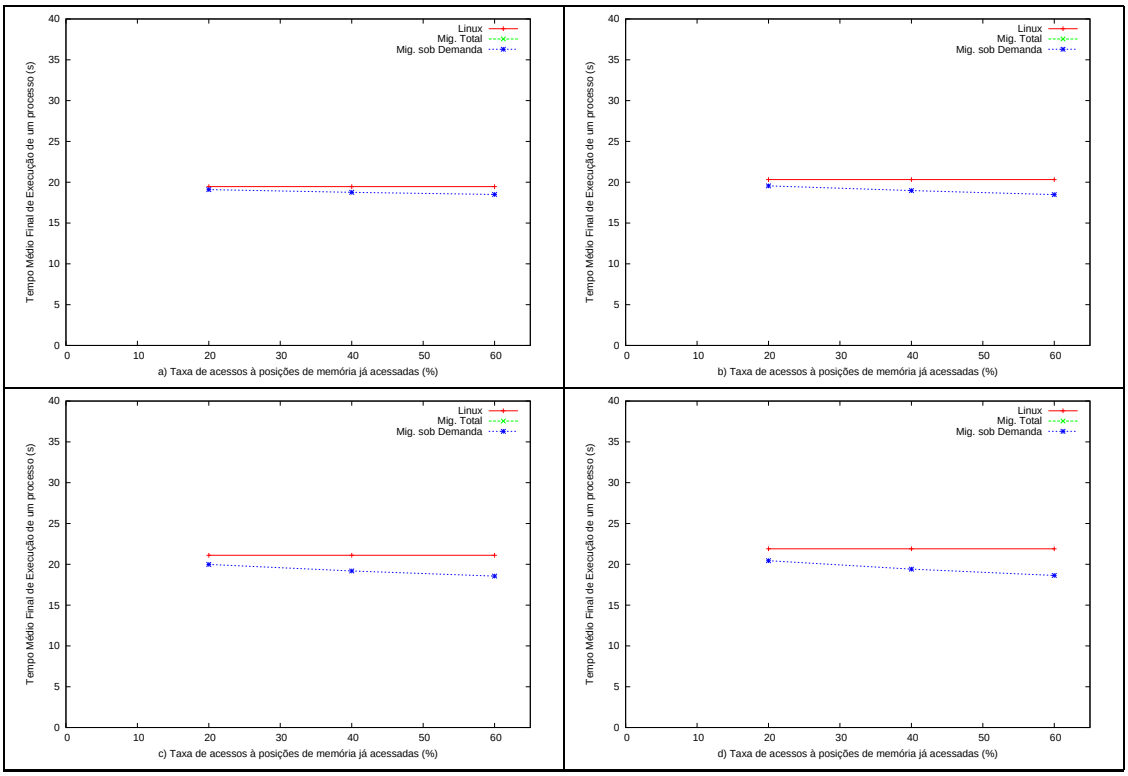


Tabela 4.2: Ganho do algoritmo de migração sob demanda comparado ao do Linux em um cenário com 200 processos de 50KB e de tempo médio de execução de 0,5s.

Tx Acesso Memória/Tx Acesso Rep. Memória	**20%**	**40%**	**60%**
20%	1,94%	3,69%	5,18%
40%	3,94%	7,1%	9,94%
60%	5,59%	10,04%	13,72%
80%	7,12%	12,83%	17,5%

se aumentar a taxa de acesso à memória vem do fato de que quando um processo migra para um processador que o deixa "longe" do banco de memória onde foi carregado, quanto mais acesso à memória este processo faz, mais se aplica o ganho causado pelo acesso a posições de memória já acessadas, descrito anteriormente.

Pode-se notar também que este ganho diminui conforme a carga do sistema, como se pode ver comparando as Tabelas 4.1 e 4.1. Além disso, pode-se verificar

Tabela 4.3: Ambiente com 200 processos de 50KB, tempo médio de execução de 0,5s e taxa de acesso a posições de memória já acessadas de 20% (a), 40% (b) e 60% (c).

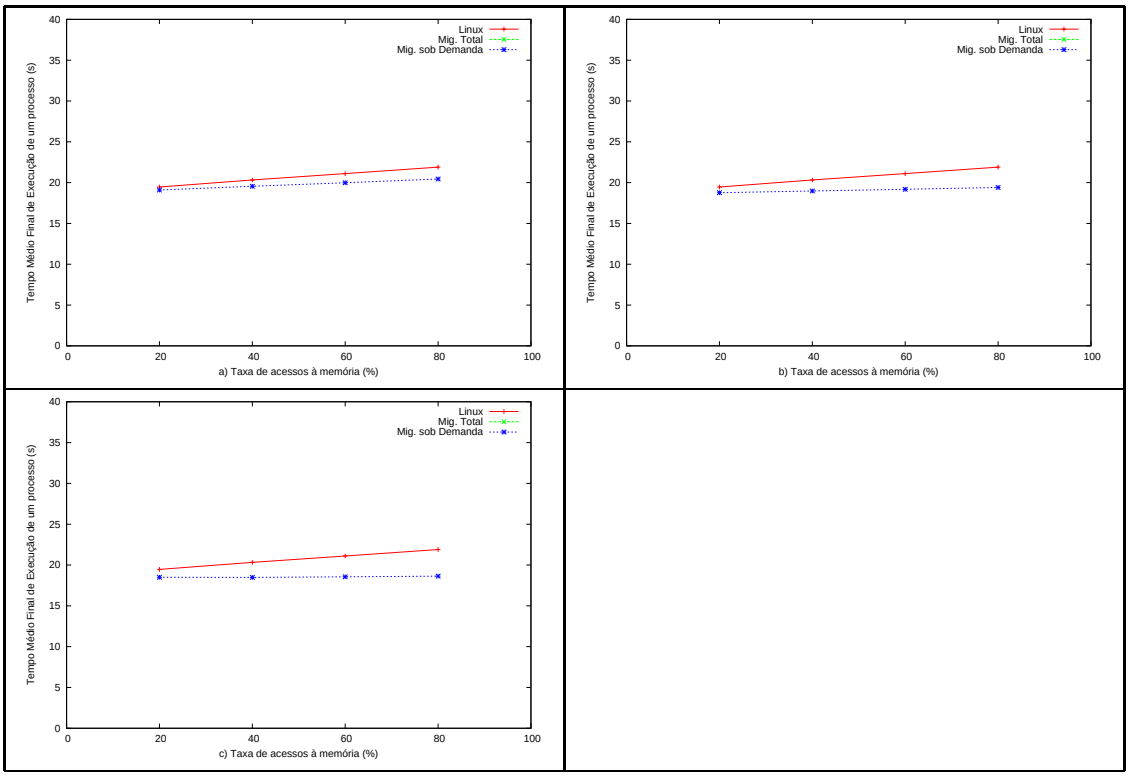


Tabela 4.4: Ganho do algoritmo de migração sob demanda comparado ao do Linux em um cenário com 400 processos de 50KB e de tempo médio de execução de 0,5s.

Tx Acesso Memória/Tx Acesso Rep. Memória	**20%**	**40%**	**60%**
20%	1,73%	3,44%	4,72%
40%	3,59%	6,6%	8,79%
60%	5,28%	9,47%	12,59%
80%	7%	12,43%	16,3%

que ao se aumentar a taxa de acessos a posições de memória já acessadas, o ganho do algoritmo de migração sob demanda é proporcionalmente maior do que o valor desse aumento. Isso mostra o que já havia sido dito anteriormente neste capítulo, de que o diferencial deste algoritmo em relação ao do Linux se dá na taxa de acessos a posições de memória já acessadas.

Como visto nas simulações anteriores, o número de processos no sistema é outro fator que influencia o desempenho dos algoritmos estudados. Assim, para se verificar esta influência, foram feitas simulações submetendo o sistema a diferentes cargas. Para isso, variou-se o número de processos de de 10 à 750. A Figura 4.4 mostra o desempenho dos algoritmos no cenário descrito.

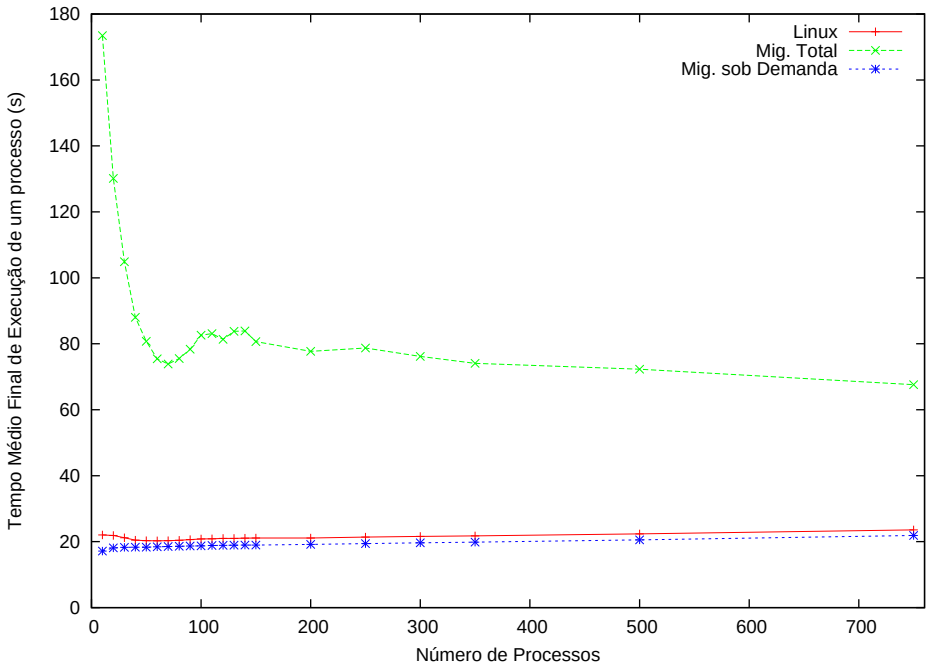


Figura 4.4: Processos de 50KB, tempo médio de execução de 0,5s, taxa de acesso à memória de 60% e taxa de acesso a posições de memória já acessadas de 40%

Pode-se notar uma discrepância bastante grande do desempenho do algoritmo de migração total comparado aos outros dois, quando temos um cenário com pequeno número de processos. No cenário mostrado na Figura 4.4, em um sistema com 10 processos, tem-se um desempenho quase que 800% pior que os outros dois algoritmos avaliados. Isto pode ser explicado pelo alto impacto que uma migração total causa em um sistema com poucos processos, pois o custo de migrar toda área de memória de um processo de um banco para outro é bastante superior ao custo de execução do processo. Como exemplo, pode-se imaginar o cenário simulado, em que temos 10 processos de tempo médio de execução de 0,5s. Supondo-se que 9 desses processos executem no cenário em exatos 0,5s e que um desses processos seja migrado (migração total), tem-se que o tempo de execução do processo migrado aumentará consideravelmente. Como tem-se poucos processos no sistema, é grande a chance de que após se passarem 0,5s, tenha-se somente o processo mi-

grado executando. Com o aumento do número de processos, o tempo da migração acaba se diluindo no paralelismo do processamento, pois se tem mais processos a executar. Naturalmente que a situação mencionada pode continuar a acontecer, mas o fato de existir mais processos no sistema, faz com que o impacto causado no desempenho final se "suavize".

Como pode-se verificar na Tabela 4.1, o ganho do algoritmo de migração sob demanda sobre o do Linux é maior em um sistema com poucos processos (10) que com um número maior (60). Desta vez pode-se explicar este comportamento devido ao grande impacto que um acesso "remoto" à memória quando temos poucos processos no sistema. A idéia é a mesma da situação descrita anteriormente. Em um sistema com poucos processos, quando um processo é migrado, o seu custo de acesso à memória aumenta, aumentando, assim, o seu tempo de execução. Desta maneira, tem-se o cenário descrito no parágrafo anterior novamente, em que depois de um tempo, ter-se-á apenas o processo migrado executando. Com um maior número de processos, esse custo é "suavizado" pelo paralelismo da execução dos demais processos.

Tabela 4.5: Ganho do algoritmo de migração sob demanda comparado ao do Linux em um cenário com processos de 50KB e de tempo médio de execução de 0,5s, com taxa de acesso à memória de 60% e taxa de acesso a posições de memória já acessadas de 40%.

Nro Processos	**Ganho**	**Nro Processos**	**Ganho**
10	28,48%	**120**	10,97%
20	20,75%	**130**	10,83%
30	15,78%	**140**	11,11%
40	11,73%	**150**	11,3%
50	10,81%	**200**	10,4%
60	9,94%	**250**	10,11%
70	9,62%	**300**	9,89%
80	9,96%	**350**	9,58%
90	10,53%	**500**	8,99%
100	11,35%	**750**	7,65%
110	10,78%		

4.2 Resultados da Simulação: HP Integrity Superdome

Além da máquina da SGI (Figura 4.1), uma máquina da arquitetura *Superdome* da Hewlett-Packard também passou pela mesma série de simulações. A máquina simulada, também conhecida como Orca, possui 64 processadores, distribuídos em 16 nodos, conforme mostra a Figura 4.5. Cada nodo desta máquina é composto por um SMP de 4 processadores.

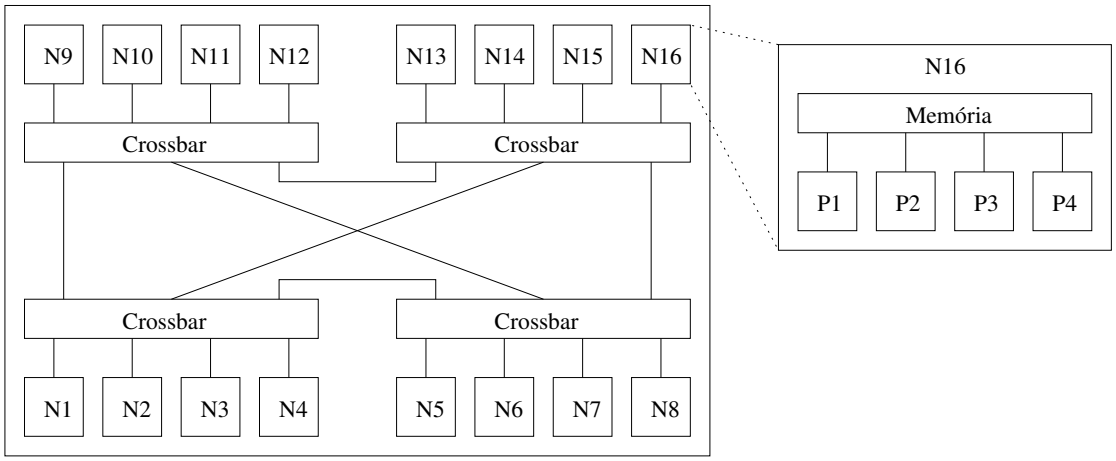


Figura 4.5: Máquina Orca usada nas simulações

Os resultados encontrados foram muito similares aos obtidos na simulação da máquina da SGI que foram apresentados anteriormente. Pode-se notar que o tempo médio de execução final dos processos submetidos à simulação na máquina Orca foram sempre sutilmente maiores que os mesmos na máquina Altix. Isso se deve à diferença da arquitetura das duas máquinas. Apesar da máquina Orca possuir maior número de processadores, isso não influencia no tempo de execução final do processo. Isto se explica, pois no modelo simulado, quando se submete um processo $P1$ de tempo médio de execução T à simulação em uma máquina $M1$, ao se submeter um processo com o mesmo tempo de execução T à simulação em uma máquina $M2$, não significa que este processo se trata do mesmo $P1$. Simplesmente se trata de um processo que na máquina $M2$ tem um tempo de execução igual ao que $P1$ tinha em $M1$.

Com resultados similares aos da máquina anteriormente simulada, o algoritmo de migração completa teve um desempenho sempre inferior ao demais (Linux e migração sob demanda). Novamente o algoritmo de migração de memória sob demanda teve um desempenho melhor do que o algoritmo utilizado pelo sistema

Tabela 4.6: SLIT da máquina Orca.

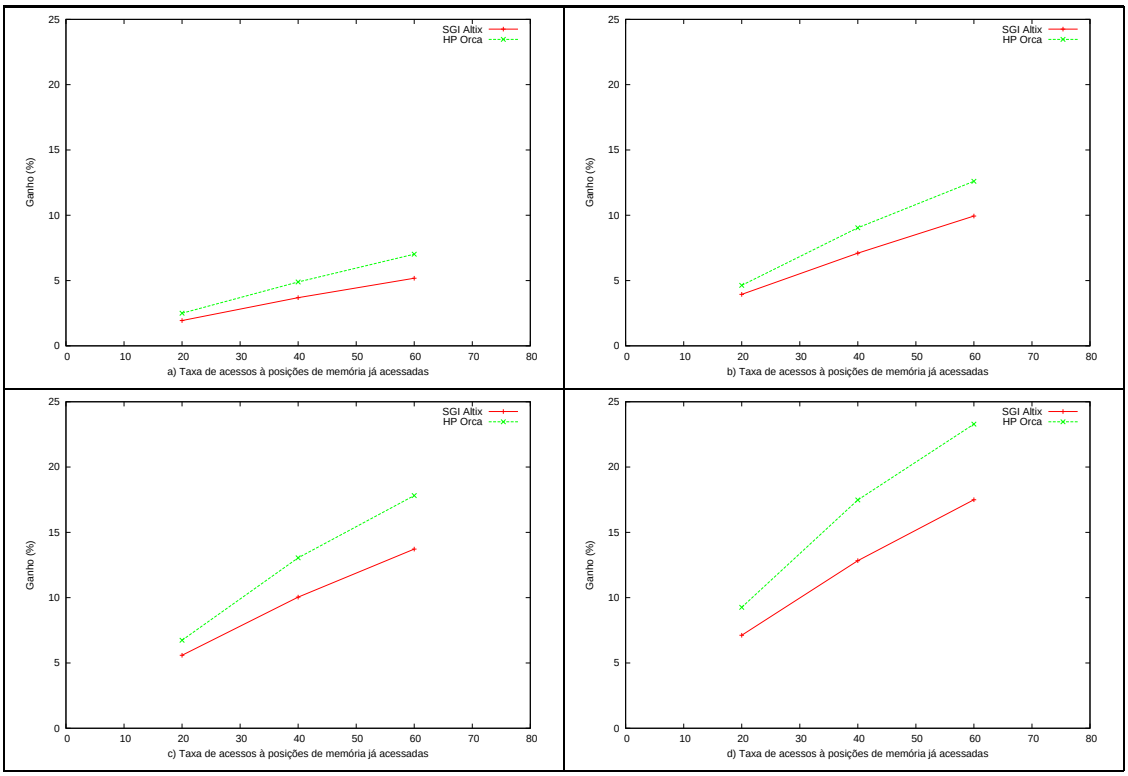
Nodo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
1	10	17	17	17	29	29	29	29	29	29	29	29	29	29	29	29
2	17	10	17	17	29	29	29	29	29	29	29	29	29	29	29	29
3	17	17	10	17	29	29	29	29	29	29	29	29	29	29	29	29
4	17	17	17	10	29	29	29	29	29	29	29	29	29	29	29	29
5	29	29	29	29	10	17	17	17	29	29	29	29	29	29	29	29
6	29	29	29	29	17	10	17	17	29	29	29	29	29	29	29	29
7	29	29	29	29	17	17	10	17	29	29	29	29	29	29	29	29
8	29	29	29	29	17	17	17	10	29	29	29	29	29	29	29	29
9	29	29	29	29	29	29	29	29	10	17	17	17	29	29	29	29
10	29	29	29	29	29	29	29	29	17	10	17	17	29	29	29	29
11	29	29	29	29	29	29	29	29	17	17	10	17	29	29	29	29
12	29	29	29	29	29	29	29	29	17	17	17	10	29	29	29	29
13	29	29	29	29	29	29	29	29	29	29	29	29	10	17	17	17
14	29	29	29	29	29	29	29	29	29	29	29	29	17	10	17	17
15	29	29	29	29	29	29	29	29	29	29	29	29	17	17	10	17
16	29	29	29	29	29	29	29	29	29	29	29	29	17	17	17	10

operacional Linux. Este ganho mostrou-se maior na máquina da Hewlett-Packard do na da SGI.

A Tabela 4.2 mostra a diferença entre o ganho obtido entre as duas máquinas. Nesse caso, se está avaliando o ganho do algoritmo de migração sob demanda em um cenário variando-se a taxa de acessos a posições de memória já acessadas pelo processo.

O ganho obtido por este algoritmo na máquina Orca é sempre superior ao obtido na Altix. Em todos os casos, o desempenho do algoritmo na Orca é pelo menos 17% melhor do que na máquina da SGI, chegando a mais de 36% em alguns casos. Um fato que pode explicar esta diferença encontra-se novamente na diferença da arquitetura das máquinas. Pode-se observar através da comparação entre as SLIT das duas máquinas que considerando-se a "distância" entre nodos separados por um único nível, esta é menor na máquina Orca. Isto torna menor o custo da migração de memória entre estes nodos, fazendo com que o tempo de execução do processo migrado seja menor nesta máquina, acabando, assim, por apresentar um desempenho superior ao da Altix. Este fato pode também explicar o melhor desempenho do algoritmo de migração sob demanda na máquina da HP quando avalia-se cenários onde há processos com diferentes taxas de acesso à memória, mostrado na Tabela 4.2.

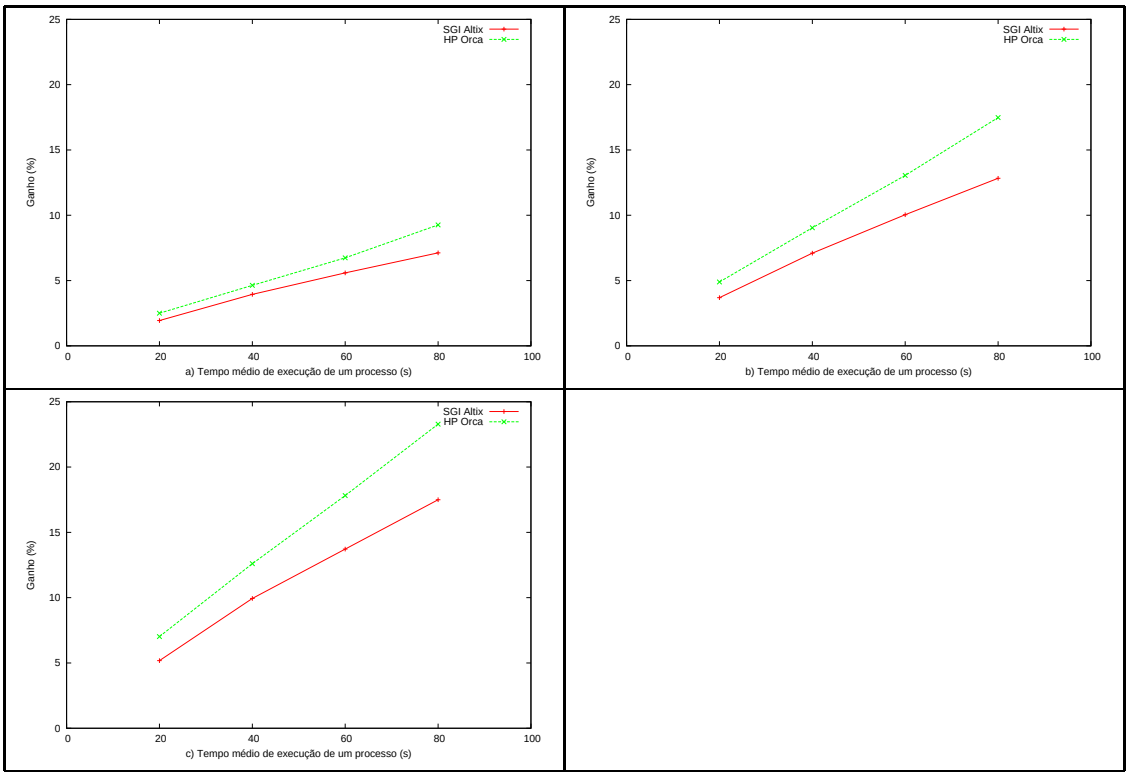
Tabela 4.7: Comparativo de ganho do algoritmo de migração sob demanda em um ambiente com 200 processos de 50KB, tempo médio de execução de 0,5s e taxa de acesso à memória de 20% (a), 40% (b), 60% (c) e 80% (d).



4.3 Resultados da Simulação: Escalonador alterado

Além de executar o simulador em diferentes arquiteturas, também foram feitos testes alterando o mesmo a fim de verificar o comportamento do desempenho dos algoritmos de migração de memória quando o escalonador é modificado para forçar a migração de processos. O escalonador foi alterado para que este gerasse migrações de um processo conforme uma determinada taxa (20, 33, 50 e 100% das vezes em que ele é executado). Esse teste foi simulado na máquina Altix (Figura 4.1), em um cenário com 50 processos, todos de 50KB, com 60% de taxa de acesso à memória e 40% de taxa de acesso a posições de memória já acessadas. O tempo médio de execução destes processos foi variado em 50, 80 e 100s e os resultados foram baseados na comparação dos algoritmos de migração de memória sob demanda e o algoritmo implementado pelo sistema operacional Linux.

Tabela 4.8: Comparativo de ganho do algoritmo de migração sob demanda em um ambiente com 200 processos de 50KB, tempo médio de execução de 0,5s e taxa de acesso a posições de memória já acessadas de 20% (a), 40% (b) e 60% (c).



Independente do tempo de execução dos processos do sistema, quando a taxa de migração forçada é menor que 50%, o algoritmo original do escalonador apresenta um desempenho melhor. Já quando a taxa de migração forçada é de exatos 50%, o desempenho fica praticamente igual, tendo uma pequena vantagem de desempenho o sistema simulando o algoritmo do escalonador modificado. Porém, quando a taxa de migração forçada é configurada para 100%, em todos os casos ocorreu um ganho de aproximadamente 3,7%. Pode-se, assim, notar a influência do sistema de migração de memória quando o ambiente é submetido a uma maior quantidade de migrações de processos.

Capítulo 5

Conclusão

Dada a escalabilidade de máquinas multiprocessadas do tipo NUMA, tem-se máquinas com arquiteturas cada vez mais diferentes, o que torna o trabalho do sistema operacional bastante complexo quando se objetiva o melhor desempenho da máquina. A gerência de memória é um dos fatores que influencia bastante no desempenho do sistema neste tipo de máquinas. Neste trabalho foram apresentadas duas estratégias para a migração de páginas da memória. A forma de estudo destas foi através de simulação, visto que é uma abordagem flexível e permite que sejam analisados diferentes cenários, evitando, assim, o estudo de apenas um conjunto de casos que aparentemente cobrem grande parte dos casos de uso desse tipo de máquina, mas que é uma forma questionável de se fazê-lo [Tsafrir and Dror, 2006]. Para uma melhor construção do modelo simulado, este trabalho analisou com cuidado a estrutura interna (código fonte) do Linux, propondo formas de implementar os algoritmos estudados. Nesse trabalho foram mostrados alguns resultados obtidos com as simulações. Diversos outros cenários foram simulados e apresentaram resultados semelhantes. Em [Corrêa et al., 2006], foi estudada uma proposta para a migração de processos no Linux que mostrou, através de simulação e posterior implementação, que o desempenho do sistema aumenta quando processos são migrados entre processadores. Em [Chanin et al., 2006a], utilizando modelos estocásticos, verificou-se da mesma forma este ganho. Este trabalho mostrou que, levando-se em conta a migração de páginas juntamente com a migração dos processos, pode-se obter um desempenho ainda melhor. O algoritmo de migração de memória sob demanda mostrou ter um desempenho superior ao algoritmo implementado pelo Linux em todos os cenários simulados. Alguns resultados obtidos nas simulações foram publicados em [Tesser and Zorzo, 2006]. Por ser um trabalho feito em parceria com a Hewlett-Packard, o desenvolvimento do algoritmo de migração sob demanda apresentado está sendo feito por um HP *Lab*.

Diversos outros aspectos relacionados com as características de processos não foram mostrados neste trabalho e são parte de um trabalho futuro, por exemplo,

compartilhamento de páginas entre processos. Outras questões como as já mencionadas na Seção 3.2.2, tais como o uso da memória *cache* e a configuração dos bancos de memória não são consideradas, mas podem ser facilmente integradas ao atual modelo simulado, fazendo parte também de trabalhos futuros. Uma questão interessante de ser estudada, também, seria a avaliação de um sistema de desfragmentação de memória a ser usada em conjunto com a migração de memória dos processos entre os nodos. Uma vez que migrar a memória do processo (algoritmo de migração sob demanda) mostrou um melhor desempenho, avaliar o impacto nesse desempenho de se ter um sistema que tente juntar a memória de um processo migrado a fim de diminuir o custo do acesso as partes da memória destes processos que se encontram distribuídas entre os nodos, seria interessante na tentativa de aprimorar o sistema de migração de memória. O modelo simulado é flexível para ser facilmente alterado e ter o escalonador implementando essa característica.

Referências Bibliográficas

[Banks et al., 2001] Banks, J., II, J. C., Nelson, B., and Nicol, D. (2001). *Discrete-Event System Simulation*. Prentice-Hall.

[Bircsak et al., 2000] Bircsak, J., Craig, R. C., Cvetanovic, Z., Harris, J., Nelson, C. A., and Offner, C. D. (2000). Extending OpenMP For NUMA Machines. *Scientific Programming*, 8(3):163–181.

[Chandra et al., 1994] Chandra, R., Devine, S., Verghese, B., Gupta, A., and Rosenblum, M. (1994). Scheduling and Page Migration for Multiprocessor Compute Server. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, pages 12–24, San Jose, Estados Unidos.

[Chanin et al., 2006a] Chanin, R., Corrêa, M., Fernandes, P. H. L., Sales, A., Scheer, R., and Zorzo, A. (2006a). Analytical Modeling for Operating System Schedulers on NUMA Systems. *Electronic Notes In Theoretical Computer Science*, 151(3):131–149.

[Chanin et al., 2006b] Chanin, R., Dotti, F. L., Sales, A., and Fernandes, P. (2006b). Avaliação Quantitativa de Sistemas. http://www.inf.pucrs.br/~fldotti/aqs/recursos/AQS-tudo.pdf, último acesso julho/2006.

[Chapin et al., 1995] Chapin, J., Herrod, S. A., Rosenblum, M., and Gupta, A. (1995). Memory system performance of UNIX on CC-NUMA multiprocessors. In *Proceedings of the Joint International Conference on Measurement & Modeling of Computer Systems*, pages 1–13, Ottawa, Canadá.

[Corrêa, 2005] Corrêa, M. (2005). Algoritmo de Construção de Hierarquia de Domínios de Escalonamento Multinível para Máquinas NUMA. Master's thesis, PPGCC, PUCRS, Brasil.

[Corrêa et al., 2005] Corrêa, M., Chanin, R., Sales, A., Zorzo, A., and Scheer, R. (2005). Performance Evaluation of a Multilevel Load Balancing Algorithm. Technical Report TR 048, PUCRS, Porto Alegre.

[Corrêa et al., 2006] Corrêa, M., Zorzo, A., and Scheer, R. (2006). Operating System Multilevel Load Balancing. In *Proceedings of ACM Symposium on Applied Computing*, pages 1467–1471, Dijon, França.

[C++SIM-Group, 2005] C++SIM-Group (2005). C++sim web site. http://cxxsim.ncl.ac.uk, último acesso maio/2005.

[Culler and Singh, 1999] Culler, D. E. and Singh, J. P. (1999). *Parallel Computer Architecture*. Morgan Kaufmann.

[Haeberlen and Elphinston, 2003] Haeberlen, A. and Elphinston, K. (2003). User-Level Management of Kernel Memory. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, volume 2823, pages 277–289, Aizu-Wakamatsu, Japão.

[Hewlett-Packard et al., 2004] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba (2004). Advanced Configuration and Power Interface Specification. http://www.acpi.info/DOWNLOADS/ACPIspec30.pdf, último acesso setembro/2004.

[Javasim-Group, 2005] Javasim-Group (2005). Javasim web site. http://javasim.ncl.ac.uk, último acesso maio/2005.

[Little and McCue, 1994] Little, M. C. and McCue, D. L. (1994). Construction and Use of Simulation Package in C++. *C User's Journal*, 3(12):1–18.

[Milojicic et al., 2000] Milojicic, D., Douglis, F., Paindaveine, Y., Wheeler, R., and Zhou, S. (2000). Process Migration. *ACM Computing Surveys*, 32(3):241–299.

[Mu et al., 2003] Mu, T., Tao, J., Schulz, M., and McKee, S. A. (2003). Interactive Locality Optimization on NUMA Architectures. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 133–213, San Diego, Estados Unidos.

[Paterson and Hennessy, 1998] Paterson, D. and Hennessy, J. (1998). *Computer Organization and Design: the hardware/software interface*. Morgan Kaufmann.

[PeSO, 2006] PeSO (2006). Grupo de Pesquisa em Sistemas Operacionais Escaláveis. http://www.inf.pucrs.br/∼peso, último acesso julho/2006.

[Tesser and Zorzo, 2006] Tesser, G. and Zorzo, A. (2006). Uma Proposta para Migração de Páginas no Linux. In *III Workshop de Sistemas Operacionais*, volume 1, pages 95–106, Campo Grande, Brasil.

[Tsafrir and Dror, 2006] Tsafrir, D. and Dror, G. F. (2006). Instability in Parallel Job Scheduling Simulation: The Role of Workload Flurries. In *IEEE International Parallel & Distributed Processing Symposium*, Rhodes Island, Grécia.

[Woodacre et al., 2005] Woodacre, M., Robb, D., and Feind, K. (2005). The SGI AltixTM 3000 Global Shared-Memory Architecture. http://www.sgi.com/products/servers/altix/whitepapers, último acesso maio/2005.