UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

ANÁLISE DE LOCALIDADE DE REFERÊNCIA NA JPC

WILLIAN GOEDERT

BLUMENAU
2009

2009/2-29

WILLIAN GOEDERT

ANÁLISE DE LOCALIDADE DE REFERÊNCIA NA JPC

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Mauro Marcelo Mattos – Orientador

BLUMENAU
2009

2009/2-29

ANÁLISE DE LOCALIDADE DE REFERÊNCIA NA JPC

Por

WILLIAN GOEDERT

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Mauro Marcelo Mattos, Doutor – Orientador, FURB
Membro:	Prof. Antônio Carlos Tavares, Mestre – FURB
Membro:	Prof. Miguel Alexandre Wisintainer, Mestre – FURB

Blumenau, 08 de dezembro de 2009

Dedico este trabalho a todos os amigos e família, que em todos os momentos do curso me apoiaram e incentivaram.

AGRADECIMENTOS

A Deus, pelo seu imenso amor, graça e força que me da para alcançar meus objetivos.

À minha família e meus amigos pelo incentivo e que sempre acreditaram em mim.

Ao meu orientador, Mauro Marcelo Mattos, por ter acreditado na minha capacidade e pelo seu incentivo.

A mente que se abre a uma nova idéia jamais voltará ao seu tamanho original.

Albert Einstein

RESUMO

O presente trabalho descreve o desenvolvimento de um módulo de geração de gráficos de padrões de acesso a memória produzidos durante a execução de programas na máquina JPC. A JPC é uma implementação de uma máquina virtual 8086/386 escrito em Java que permite a execução de programas e sistemas operacionais escritos para executar no hardware real.

Palavras-chave: Gerenciamento de memória. *Working-set*. Localidade de referência.

ABSTRACT

This academic work describes the development of a module of generating graphics with the patterns of memory access made during the execution of programs on the machine JPC. The JPC is an implementation of a virtual machine 8086/386 written in Java that allows the execution of programs and operating systems written to run in almost real hardware.

Key-words: Memory management. Working-set. Reference locality.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

ABI – *Application Binary Interface*

ATS – *Address Translation Simulator*

DCS – *Data Coordinate System*

DDCS – *Device-Dependent Coordinate System*

DICS – *Device-Independent Coordinate System*

FIFO – *First In First Out*

IBM – *International Business Machine*

ISA – *Instruction Set Architecture*

JCCKit – *Java Chart Construction Kit*

JPC – *Java Personal Computer*

JVM – *Java Virtual Machine*

LFU – *Least Frequently Used*

LRU-WAR – *Least Recently Used – Working Area Restriction*

LRU-WAR Lock – *Least Recently Used – Working Area Restriction Lock*

MMU – *Memory Management Unit*

MRU – *Most Recently Used*

PC – *Personal Computer*

PDA – *Personal Digital Assistant*

PGO – *Profile Guided Optimization*

RAM – *Random Access Memory*

VMM – *Virtual Machine Monitor*

VXT – *Virtual XT*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

A memória é um recurso importante para guardar as informações de execução dos programas fazendo com que executam de forma eficiente onde deve ser gerenciada com muito cuidado. Apesar de atualmente os computadores pessoais possuírem milhares de vezes mais memória do que o IBM 7094 (o maior computador do mundo no início dos anos 60), os programas crescem muito mais rapidamente do que as memórias (TANENBAUM, 2003, p. 139).

A medida que o hardware evolui em termos de capacidade de processamento e memória, mais recursos são disponibilizados aos programadores para que os mesmos desenvolvam aplicações mais robustas e amigáveis. Isso os leva a desenvolver aplicações que consomem boa parte dos recursos disponíveis, pressionando os desenvolvedores de hardware a produzir equipamentos com cada vez mais memória e capacidade de processamento. Este é o princípio que norteia a evolução da informática de uma forma geral (MATTOS, 2005).

Sob o ponto de vista de gerência de memória, a maioria dos computadores utiliza um mecanismo de hierarquia de memórias que combina: uma pequena quantidade de memória *cache* volátil, muito rápida e de custo alto; uma grande memória principal, volátil, com dezenas de megabytes, de velocidade e custo médios; e uma memória secundária, constituída de armazenamento não volátil em disco, com dezenas de centenas de gigabytes, velocidade e custo baixos. Cabe ao sistema operacional coordenar a utilização destas memórias (TANENBAUM, 2003, p. 139).

Conforme Piantola e Midorikawa (2008, p. 1), mesmo com os avanços na área de arquitetura de computadores, ainda não foi resolvida a questão sobre o desempenho das memórias em relação aos processadores. Ainda, segundo Tanenbaum (2003, p. 165), os processos apresentam uma propriedade denominada localidade de referência, a qual diz que, durante qualquer uma das fases de sua execução, o processo só vai referenciar uma fração relativamente pequena de suas páginas. O conjunto de páginas referenciadas em determinado momento denomina-se *working set*.

O estudo de mecanismos de gerência de memória envolve a obtenção dos padrões de acesso à memória e posterior análise de comportamento dos mesmos em relação as políticas de gerenciamento de memória implementadas em cada sistema operacional. Para isso se fazem necessários recursos que viabilizem a obtenção destes dados. Uma das formas é a utilização de simuladores (CASSETTARI; MIDORIKAWA, 2004). Outra forma é através de

técnicas de virtualização de hardware (OXFORD UNIVERSITY, 2008).

Neste contexto, o propósito deste trabalho é desenvolver um sistema que utilizando a JPC como base, permita a geração de gráficos de acesso a memória para posterior análise.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um módulo de análise de localidade de referência de programas que executam na máquina JPC.

Os objetivos específicos do trabalho são:

a) desenvolver um algoritmo de contabilização e registro de acessos a memória da JCP;
b) desenvolver um módulo de visualização das informações coletadas.

1.2 ESTRUTURA DO TRABALHO

O trabalho esta dividido em cinco capítulos.

O primeiro capítulo apresenta uma introdução do trabalho, os objetivos a serem apresentados e a estrutura do trabalho.

O segundo contempla a fundamentação teórica do trabalho, onde é feito um levantamento sobre os assuntos abordados no trabalho.

O terceiro capítulo descreve o mecanismo de gerência de memória e também mostra os trabalhos correlatos.

No quarto capítulo é descrito todo o processo para desenvolvimento do módulo de análise de localidade de referência, sua especificação, os requisitos que o sistema deverá ter e a sua implementação.

O quinto capítulo contém as conclusões do trabalho, assim como sugestões de extensões para o mesmo.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os conceitos de virtualização de hardware e a arquitetura da JPC exemplificando seu desenvolvimento baseado na arquitetura X86.

2.1 VIRTUALIZAÇÃO DE HARDWARE

Virtualização é o modo de apresentação ou agrupamento de um subconjunto lógico de recursos computacionais de modo que possam ser alcançados resultados e benefícios como se o sistema estivesse executando sobre a configuração nativa. Um outro tipo popular de virtualização e atualmente muito utilizado é a virtualização de hardware para rodar mais de um sistema operacional ao mesmo tempo, através de *microkernels* ou de camadas de abstração de hardware, como por exemplo o XEN (SANTOS, 2006, p. 342).

Segundo Santos (2006, p. 342), há várias soluções de virtualização. O nível mais baixo denomina-se emulação e possibilita que um sistema operacional execute sem modificações no hardware emulado. Um exemplo é a máquina JPC.

O segundo nível denomina-se virtualização nativa e envolve a utilização do conjunto de instruções de hardware do processador real, mas a simulação dos dispositivos periféricos ocorre através de uma camada de software denominada *hypervisor* ou *Virtual Machine Monitor* (VMM). Neste contexto, o sistema operacional executa sem alterações no hardware virtualizado, embora em certas situações sejam necessários *device drivers* adicionais para acesso aos dispositivos virtuais. Por exemplo, no VMWare (SANTOS, 2006, p. 344), há a necessidade da instalação de *device drivers* específicos para acesso ao hardware de rede (placa de rede virtual) e ao hardware de vídeo (placa de vídeo/monitor virtual).

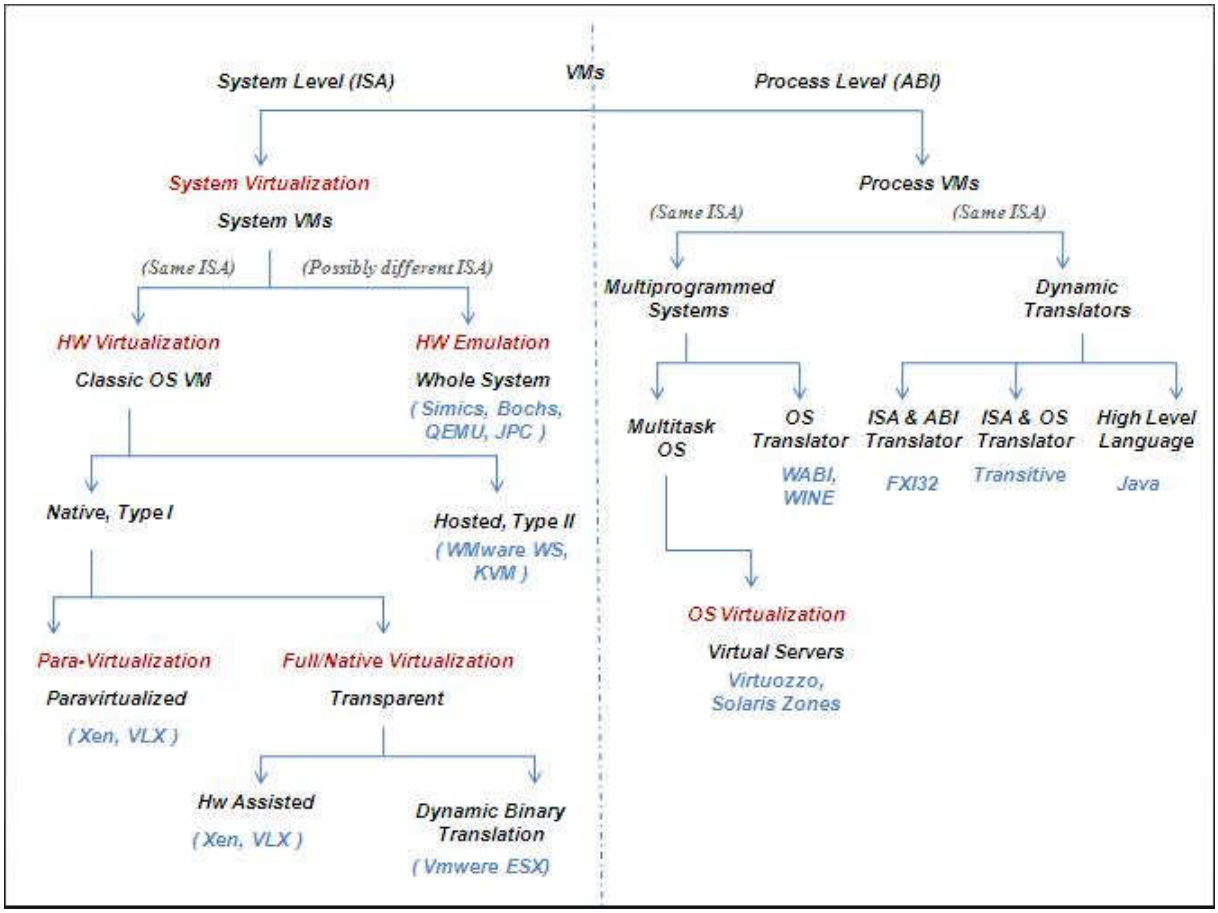
Outro nível de virtualização denomina-se paravirtualização e demanda alterações no sistema operacional hóspede de modo a adequar-se ao hardware virtualizado. O XEN é um exemplo de sistema que adota este conceito (SANTOS, 2006, p. 348).

Adams e Agesen (2006) identificaram três características essenciais para que um software seja considerado um monitor de máquina virtual, quais sejam:

a) fidelidade: o software no monitor de máquina virtual executa identicamente a sua execução no hardware;

b) desempenho: a grande maioria das instruções do hardware virtual são executadas pelo hardware real sem a intervenção do monitor de máquina virtual;
c) segurança: o monitor de máquina virtual gerencia todos os recursos do hardware virtual.

Segundo Scope (2008, p. 10), a figura 1 apresenta uma classificação das tecnologias de virtualização. Esta classificação é baseada em software que fornece um *Instruction Set Architecture* (ISA) implementadas com objetivo de funcinar como máquinas virtuais de sistemas ou um *Application Binary Interface* (ABI) implementadas com objetivo de funcioanr com máquinas virtuais de processo. Como pode ser visto, uma ISA marca a divisão entre hardware e software, onde a área de usuário inclui aspectos referentes às aplicações e a área de sistema em gerenciar os recursos de hardware. Já uma ABI permite a um programa acessar os recusrsos de *hardware* e os serviços disponíveis através da interface de chamadas do sistema. Uma ABI não inclui instruções de sistema, portanto todas as aplicações interagem com os recursos de hardware indiretamente pela invocação de serviços do sistema operacionalvia interfaces de chamadas de sistema. Como exemplo, o lado esquerdo da figura caracteriza tecnologias que suportam máquinas virtuais e o lado direito caracteriza tecnologias que suportam processos que implementam máquinas virtuais. A principal diferença é que uma máquina virtual é auto contida, ou seja, necessita de todo o software necessário (sistema operacional) para executar uma aplicação enquanto um processo de máquina virtual suporta somente aplicações.

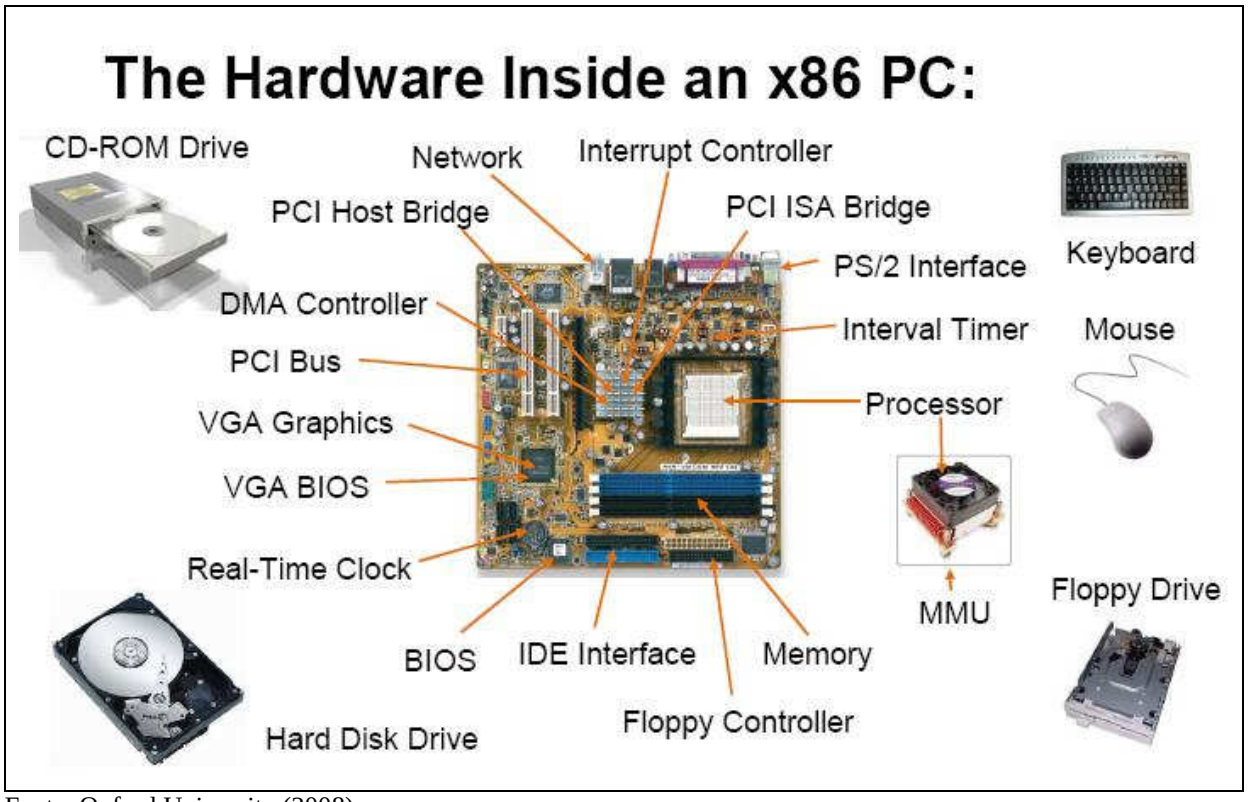


Fonte: Adams e Agesen (2006).

Figura 1 - Classificação das tecnologias de virtualização

2.2 A ARQUITETURA DA JPC

A JPC é um projeto que vem sendo desenvolvido desde 2002 por pesquisadores do Departamento de Física da Universidade de Oxford (OXFORD UNIVERSITY, 2008). Sua finalidade é a construção de um emulador de hardware de um X86 usando puramente a tecnologia Java, tendo completamente todos os seus periféricos virtuais. Para uma melhor compreensão a figura 2 destaca o interior de um X86 e todos os seus periféricos que são implementados no software. A arquitetura da JPC possui cinco pacotes de código fonte e 219 classes escritas em Java. O X86 tem sua arquitetura extremamente complexa e com uma longa história de melhorias implementadas. Sendo assim, este emulador levou muito tempo para ser escrito e para ser disponibilizado para seu uso.



Fonte: Oxford University (2008).

Figura 2 – Vista interna de um X86 com todos os seus periféricos

Este emulador tem dois modos comuns de operação que são:

a) real: utilizado em processadores 8086, a memória atinge somente 1 Mbyte de tamanho e é dividida em dois blocos: a memória convencional com 640 Kbytes e a memória estendida que corresponde a 384 Kbytes (OXFORD UNIVERSITY, 2008);

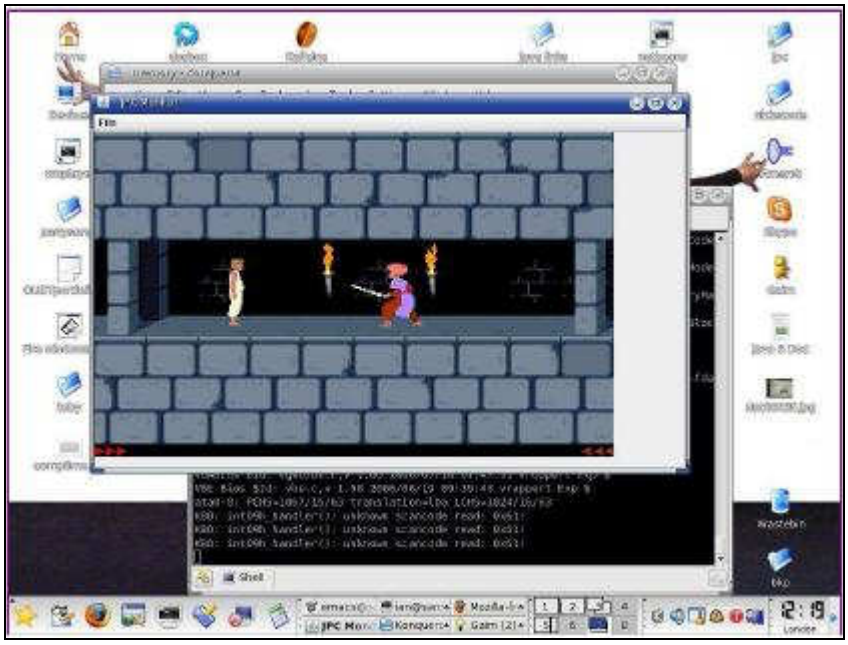
b) protegido: pode emular processadores 186, 286 e 386. Neste caso é possível utilizar os recursos de memória virtual contemplando também memória secundária (disco). Neste modo é possível emular o modo virtual 8086 que tem por objetivo simular vários ambientes em modo real com limite de memória em 1 Mbyte e com total acesso ao hardware virtual.

Tendo sido construída em Java, a aplicação é compatível com as maiores plataformas existentes hoje em dia, incluindo *Windows* (Figura 3), Linux (figura 4) e *MacOS* (figura 5).

Além dessas, a JPC roda em plataforma ARM (figura 6) e foi portada para dispositivos móveis como exemplificado na figura 7.

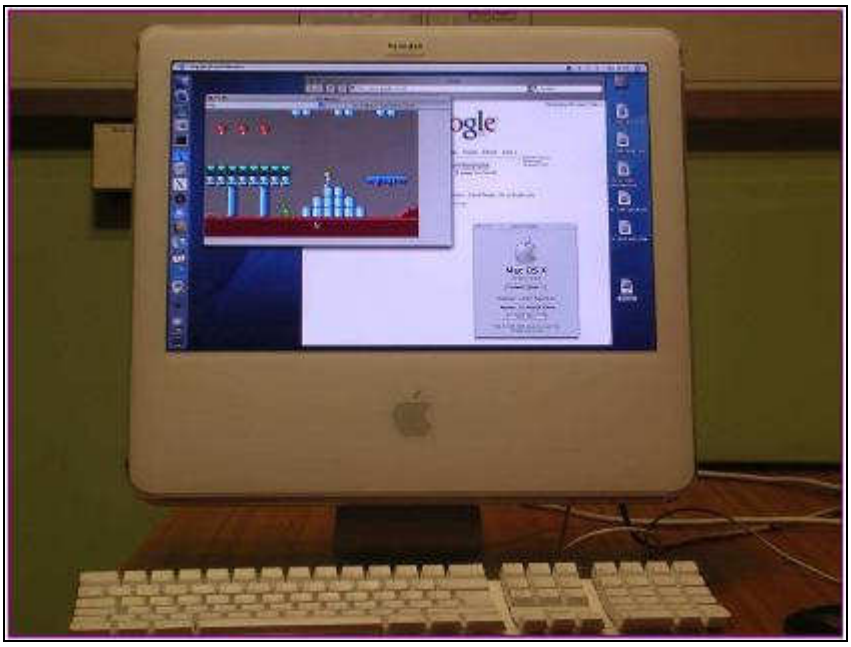
Fonte: Oxford University (2008).

Figura 3 - JPC em Windows XP rodando a versão DOS do Mario Bros



Fonte: Oxford University (2008).

Figura 4 - JPC em Linux rodando versão do DOS do jogo *Prince of Persia*



Fonte: Oxford University (2008).

Figura 5 - JPC rodando *Commander Keen 1* em DOS num Mac

Fonte: Oxford University (2008).

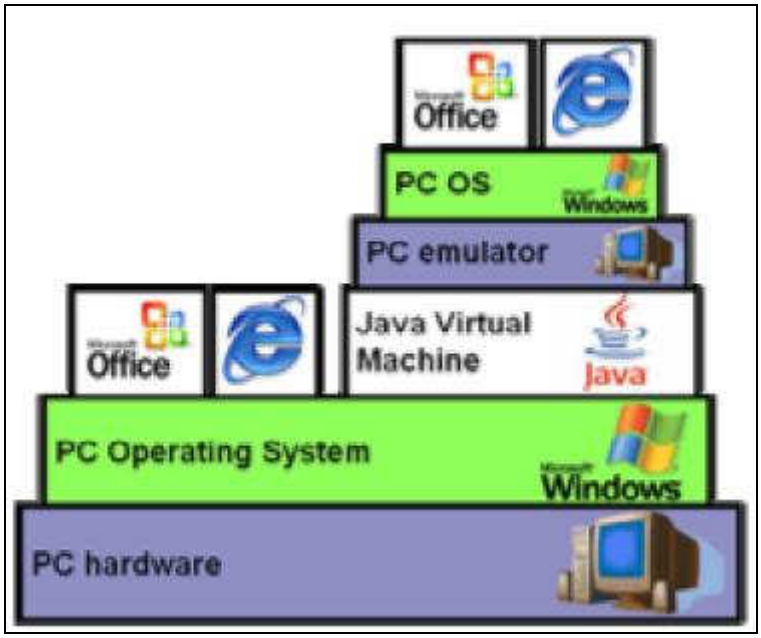
Figura 6 - JPC rodando em uma placa ARM (Celular Nokia N95)

Fonte: Oxford University (2008).

Figura 7 - Nokia N95 executando *booting* DOS e Mário Bros na JPC

Outra característica importante da JPC refere-se a sua segurança, visto que a mesma não tem acesso direto ao hardware.

A figura 8 ilustra a hierarquia dos programas que estão sendo executados no hardware do computador real. Como pode ser visto, a primeira camada é o hardware real; a segunda camada vem com o sistema operacional real (usado Windows como exemplo); na terceira camada estão os softwares aplicativos (*Offic*e e JVM); na quarta camada esta a JPC e sobre ela aplicam-se as mesmas camadas citadas anteriormente. Sendo assim tem-se uma quinta camada com um sistema operacional e sobre ele executam as aplicações.



Fonte: Oxford University (2008).

Figura 8 – Virtualização com a JPC

A estrutura do software JPC representado na forma de diagrama de classe apresenta-se nas figuras abaixo. Em função do número de classes optou-se pela divisão do diagrama de classes em quatro partes. Cabe destacar que, embora o projeto JPC tenha sido disponibilizado o código fonte, pouca documentação sobre o projeto foram localizadas.

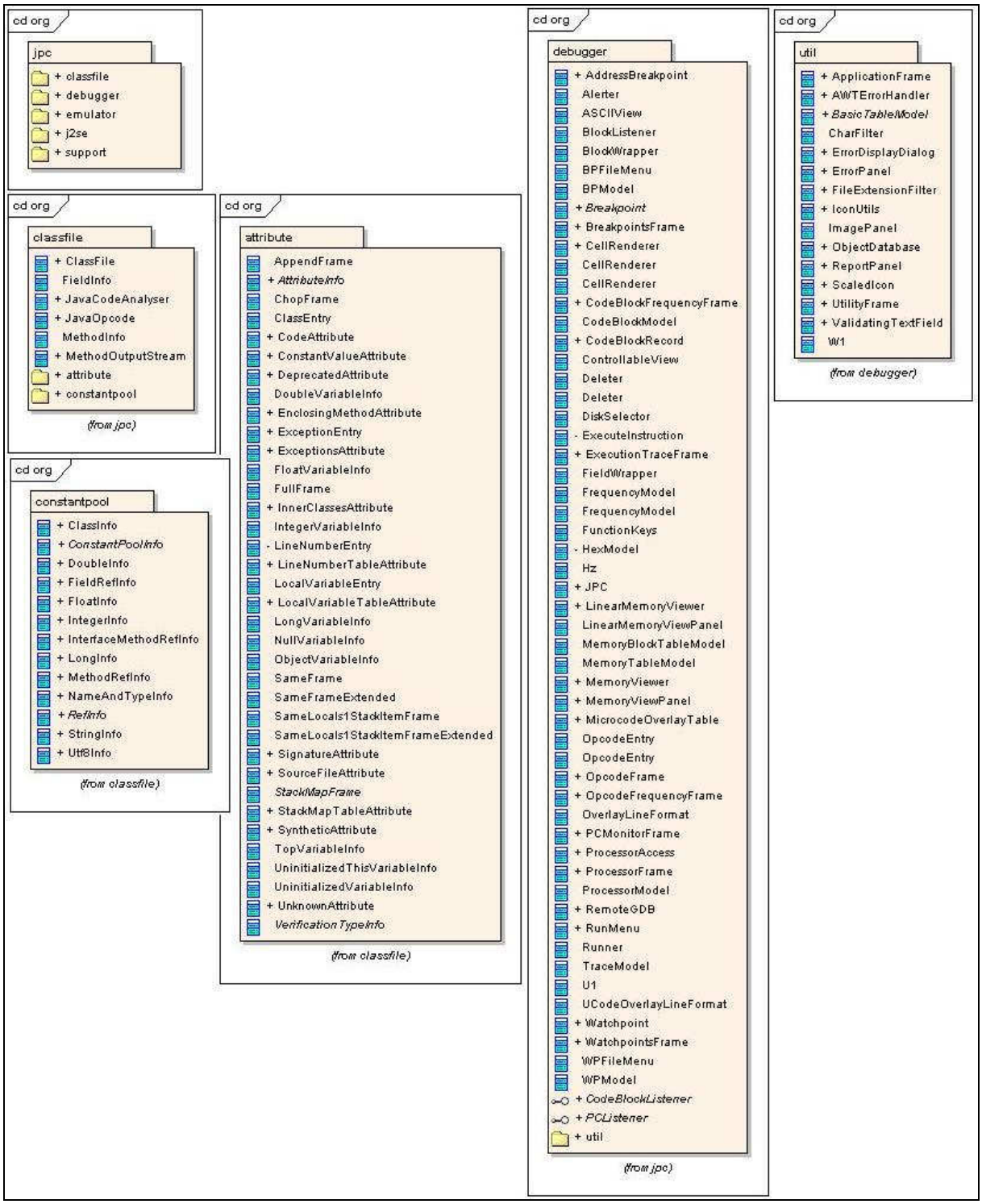


Figura 9 – Estrutura de classes da JPC – parte 1

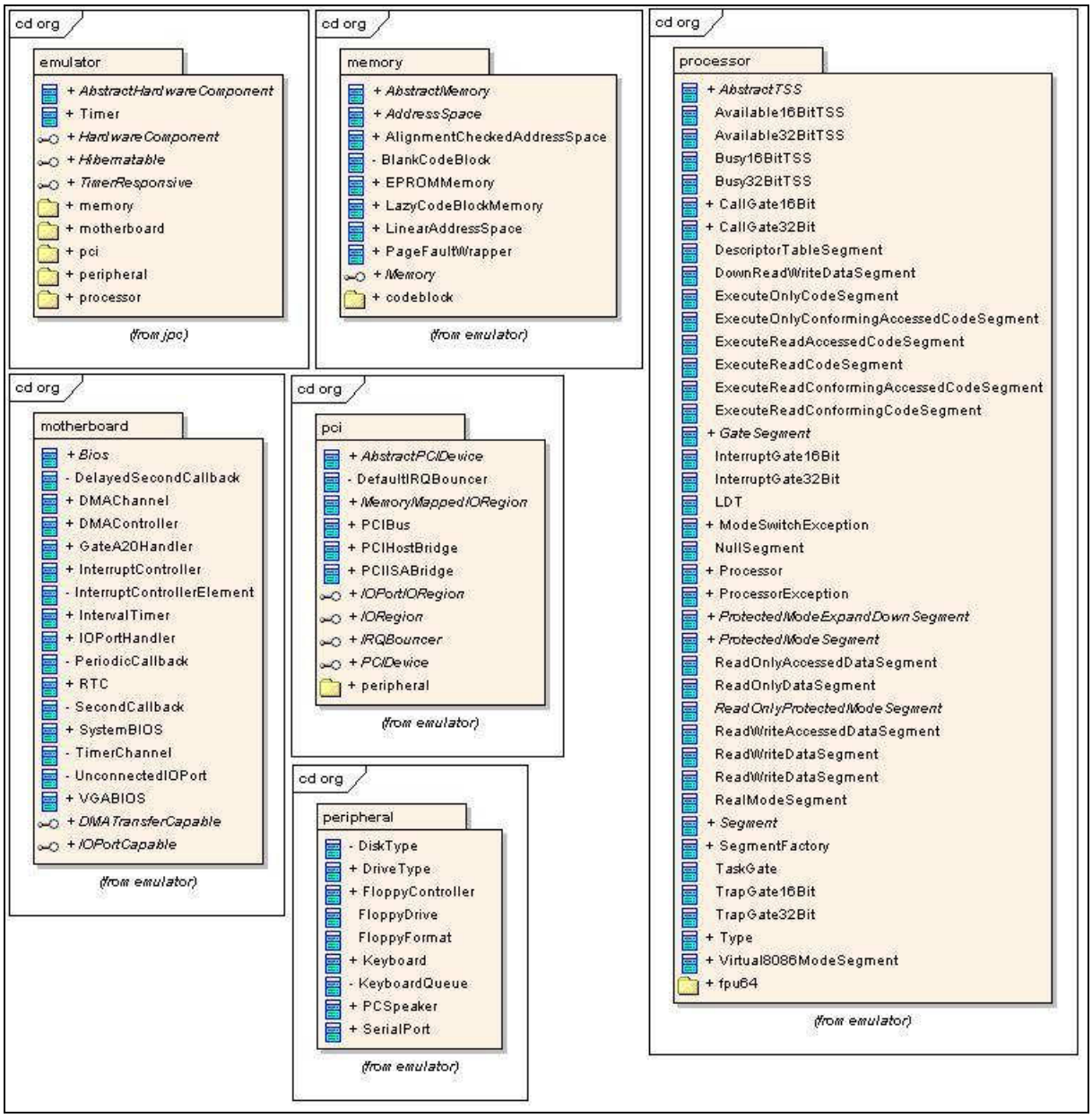


Figura 10- Estrutura de classes da JPC – parte 2

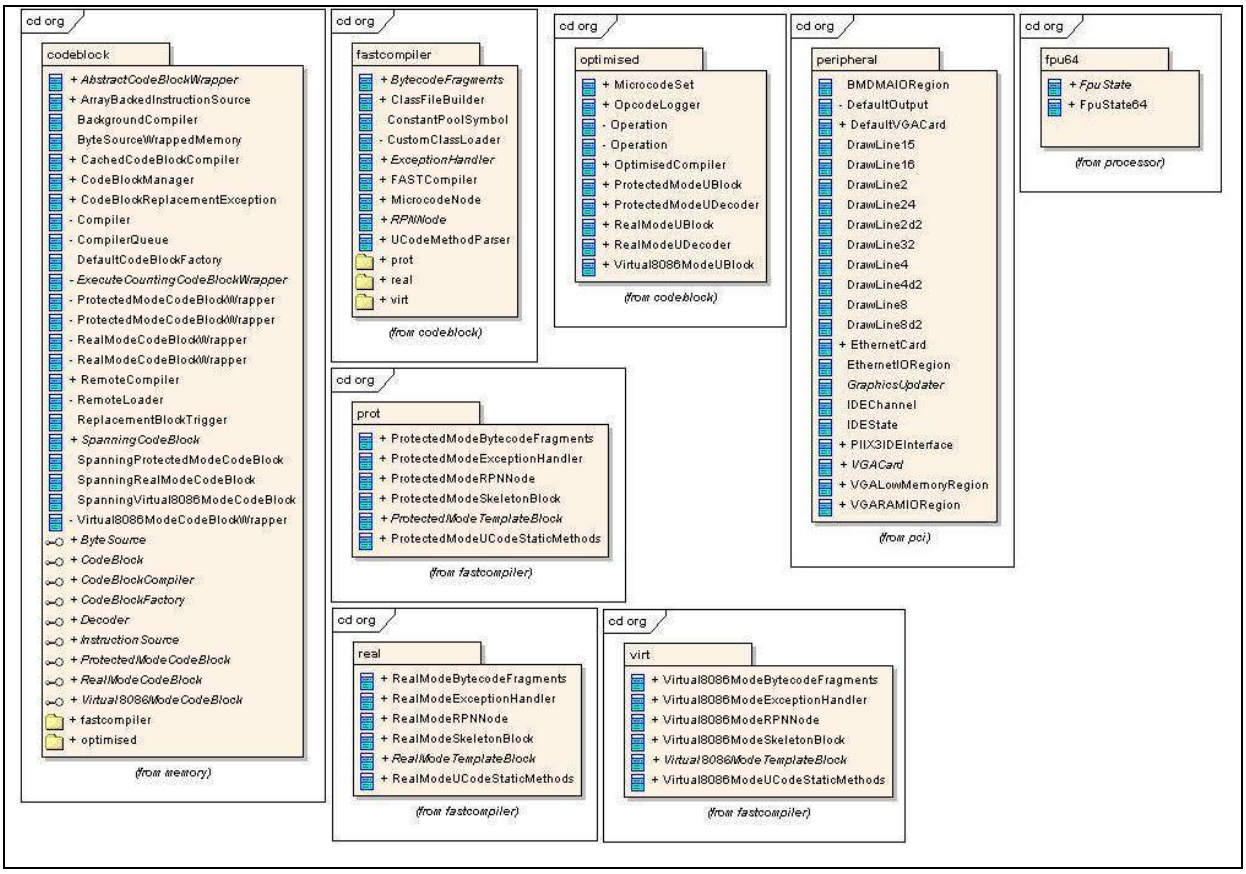


Figura 11 – Estrutura de classes da JPC – parte 3

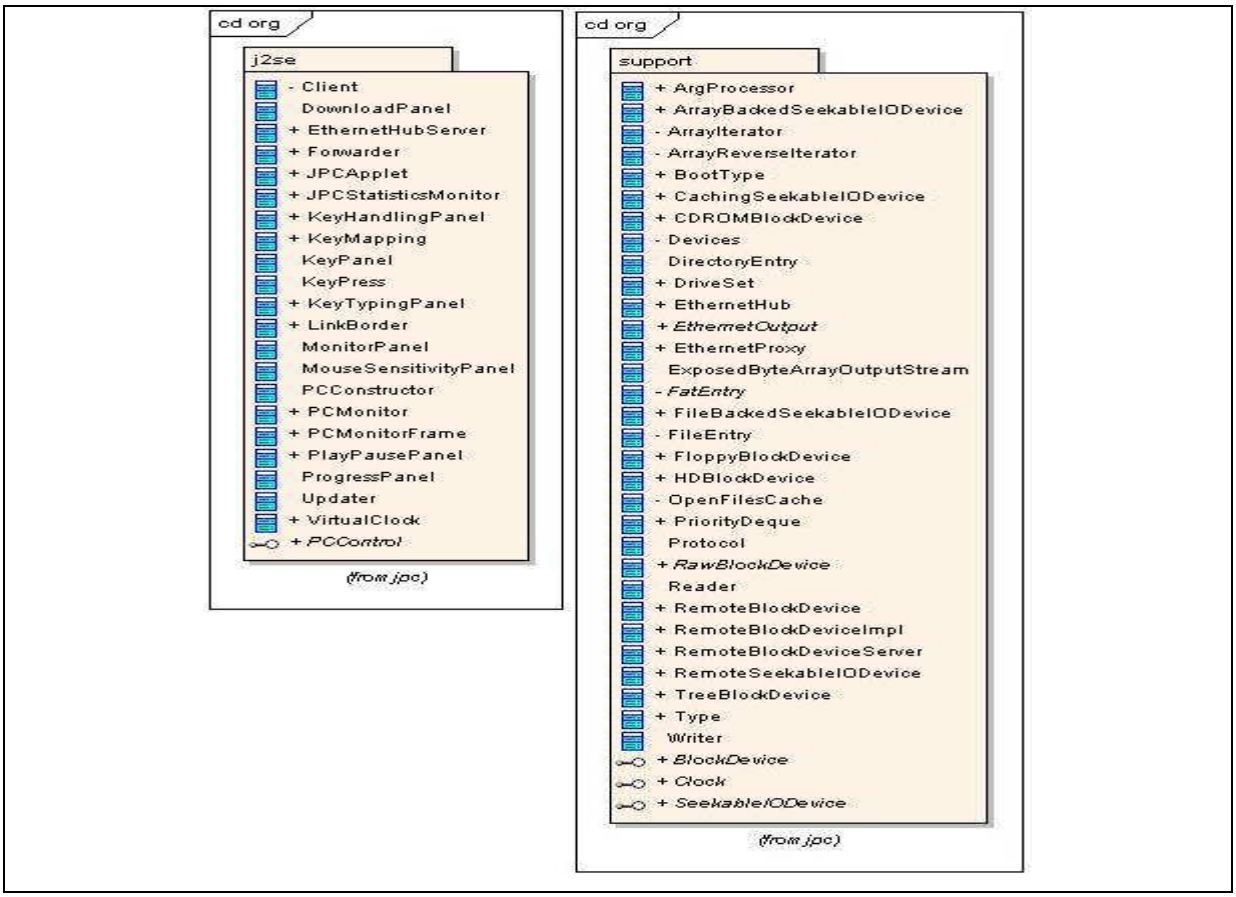


Figura 12 – Estrutura de classes da JPC – parte 4

3 GERÊNCIA DE MEMÓRIA

Uma vez que diversos processos podem ocupar a memória ao mesmo tempo, torna-se necessário um gerenciamento adequado de modo a otimizar a utilização da mesma. O componente do sistema operacional responsável por esta tarefa é o gerente de memória, que entre outras atribuições deve controlar quais partes da memória estão sendo usadas, alocar memória de acordo com as necessidades de cada processo, liberar memória alocada a um processo quando este termina sua execução, transferência de processos ou partes destes entre a memória principal e secundária (MORITZ, 2004, p. 19).

3.1 MEMÓRIA REAL

Conforme Maia (2001), as diversas abordagens propostas para alocação e restrição do espaço de endereçamento envolvem a utilização de registradores que definem os limites dos endereços de memória que um processo poderá acessar. O significado do valor armazenado nos registradores pode variar um pouco de uma técnica para outra, mas a proteção é sempre feita através de comparações do valor do endereço desejado pelo processo para acesso com o endereço armazenado no(s) registrador(es), e caso o endereço esteja fora do espaço autorizado para o processo, uma interrupção é gerada indicando uma violação da memória e o processo pode ter sua execução abortada. As técnicas de gerenciamento de memória bastante conhecidas: partições fixas, partições variáveis, paginação e memória virtual.

Segundo Oliveira, Carissimi e Toscani (2001), partições fixas são a forma mais simples de gerenciamento de memória para ambientes de multiprogramação. A memória é dividida em um número fixo de blocos, eventualmente de diversos tamanhos. À medida que os processos são submetidos à execução (processos no estado pronto) o gerenciador de memória localiza uma partição livre cujo tamanho atenda às necessidades do processo.

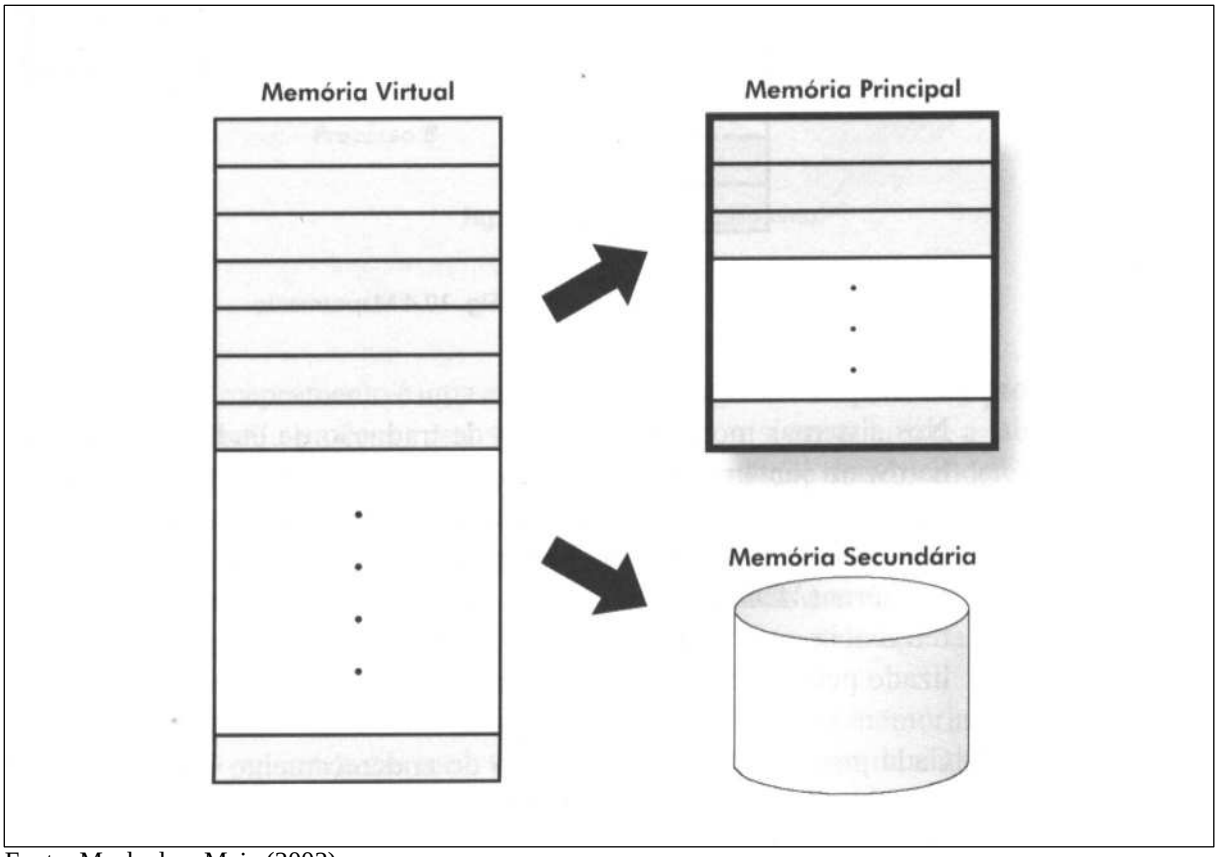
Quando partições variáveis são empregadas, o tamanho das partições é ajustado dinamicamente as necessidades exatas dos processos. Essa é uma técnica de gerência mais flexível que partição fixa. (OLIVEIRA; CARISSIMI; TOSCANI, 2001, p. 101).

3.2 MEMÓRIA VIRTUAL

A memória virtual é a separação da memória lógica do usuário da memória física. Essa separação permite que uma memória virtual extremamente grande seja fornecida para os programadores quando apenas uma pequena quantidade de memória física esteja disponível. (SILBERSHATZ; GALVIN; GAGNE, 2004, p. 211).

O conceito de memória virtual está baseado em desvincular o endereçamento feito pelo programa dos endereços físicos da memória principal. Assim, os programas e suas estruturas de dados deixam de estar limitados ao tamanho da memória primária disponível. Para permitir que apenas partes realmente necessárias à execução do processo estejam na memória, o código deve ser dividido em blocos e mapeado na memória principal, a partir do espaço de endereçamento virtual. O espaço de endereçamento virtual representa o conjunto de endereços virtuais que os processos podem endereçar. Analogamente, o conjunto de endereços reais é chamado espaço de endereçamento real (MAIA, 2001, p. 45).

Conforme Maia (2001, p. 47), um programa pode fazer referência a endereços virtuais que estejam fora dos limites do espaço real, ou seja, os programas e suas estruturas de dados não estão mais limitados ao tamanho da memória física disponível. Como os programas podem ser muito maiores que a memória física, apenas parte deles pode estar residente na memória em um determinado instante. O sistema operacional utiliza a memória secundária como extensão da memória principal e o transporte de programas entre uma e outra dá-se de maneira dinâmica e transparente ao usuário. Quando um programa é executado, só uma parte do código fica residente na memória principal, permanecendo o restante na memória secundária até o momento de ser referenciado (figura 13).



Fonte: Machado e Maia (2002).

Figura 13 - Espaço de endereçamento virtual

Outra vantagem da memória virtual é permitir um número maior de processos compartilhando a memória, já que apenas algumas partes de cada processo estarão residentes. Isto leva a uma utilização mais eficiente também do processador, permitindo um maior número de processos no estado de pronto (OLIVEIRA; CARISSIMI; TOSCANI, 2001).

3.2.1 Políticas de busca de páginas

O mecanismo de memória virtual permite a execução de um programa sem que esteja completamente residente na memória. A política de busca de páginas determina quando uma página deve ser trazida para a memória principal. Existem, basicamente, duas alternativas: paginação por demanda e paginação antecipada. (MACHADO; MAIA, 2002).

Na paginação por demanda, as páginas dos processos são transferidas da memória secundária para a principal apenas quando são referenciadas. Este mecanismo é conveniente, na medida em que leva para a memória principal apenas as páginas realmente necessárias à execução do programa. Desse modo, é possível que partes do programa, como rotinas de

tratamento de erros, nunca sejam carregadas para a memória. (OLIVEIRA; CARISSIMI; TOSCANI, 2001).

Na paginação antecipada, o sistema traz para a memória, além das páginas referenciadas, outras páginas que podem ou não ser necessárias ao processo no futuro.

3.2.2 Política de alocação de páginas

Segundo Maia (2001) a política de alocação de páginas determina quantos *frames* cada processo pode alocar na memória principal. Existem, basicamente, duas alternativas: a alocação fixa e alocação variável:

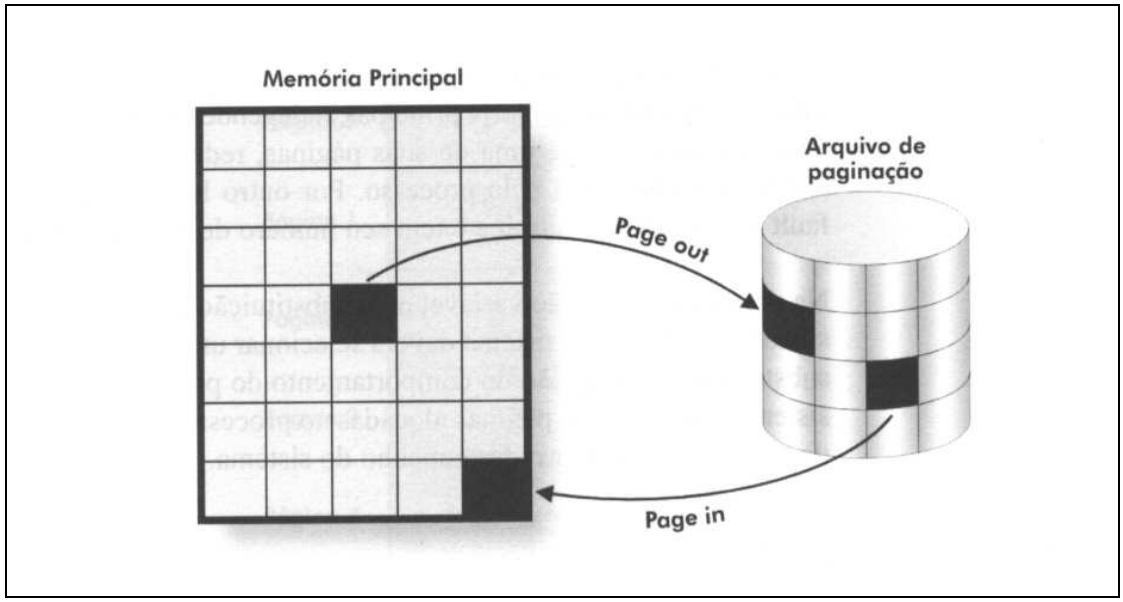
a) alocação fixa: cada processo recebe um número máximo de páginas que pode ser utilizado. Se o número de páginas for insuficiente, o processo gera uma exceção de *page fault* e cede uma página para obter uma nova. O número máximo de páginas pode ser igual para todos os processos ou ser definido individualmente. Alocar o mesmo número de páginas para todos os processos, apesar de justo, em princípio não funciona, caso os processos tenham necessidades diferentes de memória, como geralmente acontece. Se cada processo pode ter um número máximo de páginas, o limite pode ser definido com base no tipo da aplicação, no início da sua execução;
b) alocação variável: o número máximo de páginas alocadas ao processo pode variar durante sua execução, em função de sua taxa de paginação, por exemplo. A taxa de paginação é o número de *page faults* por unidade de tempo de um processo. Processos com elevadas taxas de paginação podem receber *frames* adicionais a fim de reduzí-las, ao mesmo tempo em que processos com taxas baixas de paginação podem cedê-las. Este mecanismo, apesar de mais flexível, exige que o sistema operacional monitore o comportamento dos processos, provocando maior *overhead*.

3.2.3 Políticas de substituição de páginas

O maior problema na gerência de memória virtual por paginação não é decidir que página carregar para a memória, mas quais páginas remover. Quando não existem páginas livres disponíveis na memória e novos *frames* devem ser alocados, a política de substituição (*replacement policy*) de páginas determina, dentre as diversas páginas residentes, quais devem ser realocadas (SILBERSHATZ; GALVIN; GAGNE, 2004).

Qualquer estratégia de substituição de páginas deve considerar se uma página foi ou não modificada, antes de liberá-la, caso contrário, possíveis dados armazenados na página serão perdidos. Sempre que o sistema liberar uma página desse tipo, ele antes deverá gravá-la na memória secundária, preservando seu conteúdo e manter um arquivo de paginação onde as páginas modificadas são armazenadas. Sempre que uma destas páginas for novamente referenciada, ela será trazida novamente para a memória principal (MACHADO; MAIA, 2002).

A política de substituição pode ser classificada conforme seu escopo, ou seja, local ou global. Na política local, apenas as páginas do processo que gerou o *page fault* (figura 14) são candidatas a realocação. Já na política global, todas as páginas residentes são avaliadas, independente do processo que gerou o *page fault* (MAIA, 2001).



Fonte: Machado e Maia (2002).

Figura 14 – Substituição de páginas

Independente se a política seja local ou global, os algoritmos de substituição de páginas devem ter o objetivo de selecionar aquelas que tenham poucas chances de serem

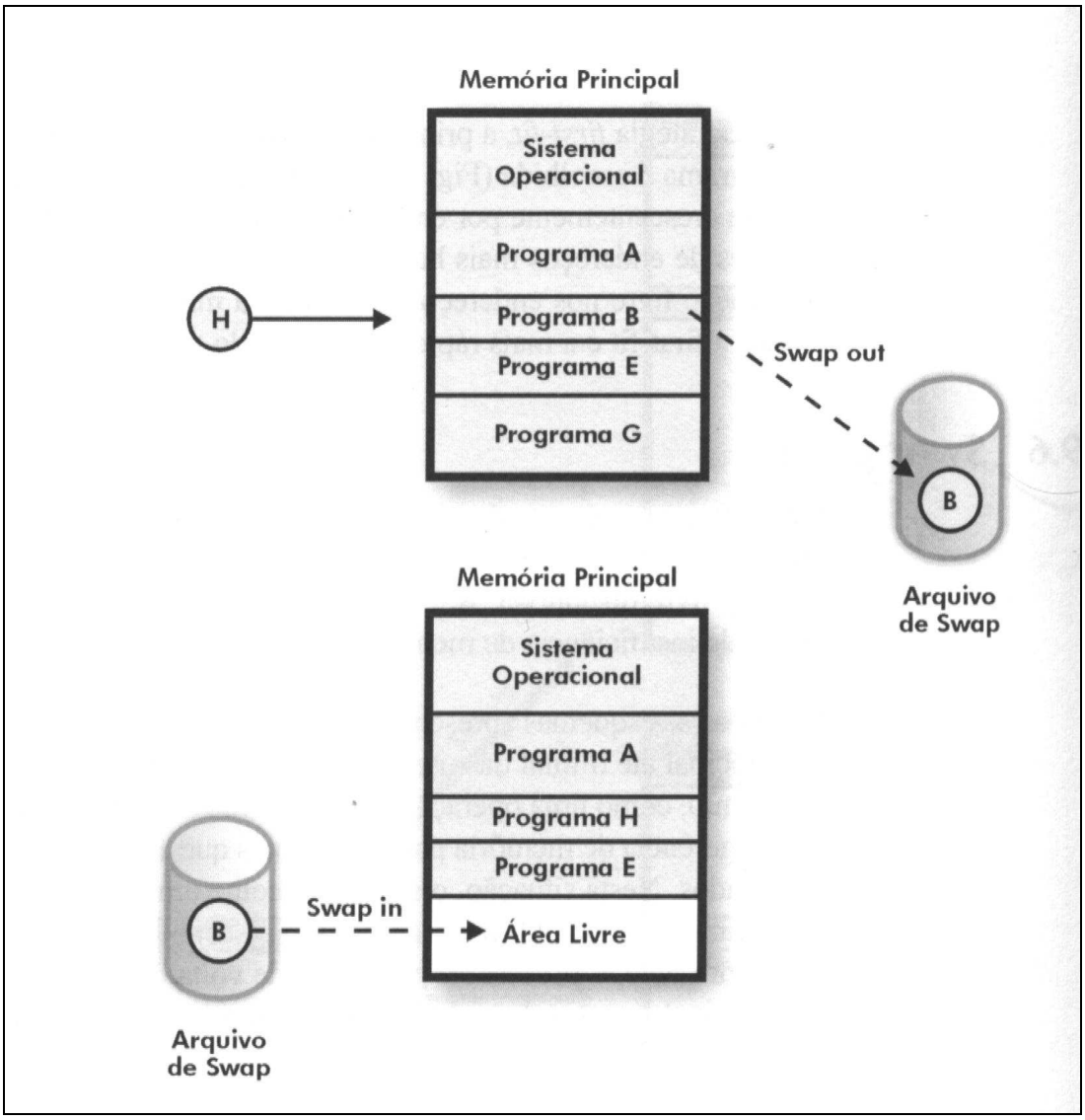
utilizadas novamente num futuro próximo. Quanto mais elaborado e sofisticado é o algoritmo, maior também é o *overhead* para o sistema. (MAIA, 2001)

Existem diversos algoritmos na literatura voltados para a implementação da política de substituição de páginas, como FIFO (*First-In-First-Out*), *buffer* de páginas, LRU (*Least-Recently-Used*) e NRU (*Not-Recently-Used*).

3.2.4 Swapping

A técnica de *swapping* permite aumentar o número de processos compartilhando a memória principal e, conseqüentemente, o sistema. Em sistemas que implementam essa técnica, quando existem novos processos que desejam ser executados e não existe memória real suficiente, o sistema seleciona um ou mais processos que deverão sair da memória para ceder espaço aos novos processos (MAIA, 2001).

Segundo Maia (2001) há vários critérios que podem ser aplicados na escolha do(s) processo(s) que deve(m) sair da memória. Os mais utilizados são a prioridade e o estado do processo. O critério de estado seleciona os processos que estão no estado de espera, ou seja, aguardando por algum evento. O critério de prioridade escolhe, entre os processos, os de menor prioridade de execução (figura 15).



Fonte: Machado e Maia (2002).

Figura 15 – *Swapping*

Depois de escolhidos o(s) processo(s), o sistema intervém e ativa uma rotina do sistema responsável por retirar (*swap out*) e trazer (*swap in*) os processos da memória principal para a memória secundária, onde são gravados em um arquivo de *swapping* (MAIA, 2001).

Swapping impõe aos programadores um grande custo em termos de tempo de execução. Copiar todo o processo da memória para disco e mais tarde de volta para memória é uma operação demorada. É necessário deixar o processo em tempo razoável no disco para justificar tal operação (OLIVEIRA; CARISSIMI; TOSCANI, 2000, p. 103).

3.2.5 Trashing

Thrashing pode ser definido como sendo a excessiva transferência de páginas entre a memória principal e a memória secundária podendo ocorrer em dois níveis: em nível do próprio processo e em nível do sistema (MACHADO; MAIA, 2002).

Sob a ótica de um processo, a excessiva paginação ocorre devido ao elevado número de *page faults*, gerado pelo programa em execução. Esse problema faz com que o processo passe mais tempo esperando por páginas do que realmente está sendo executado e ocorre devido ao mau dimensionamento no tamanho do *working set* do mesmo, pequeno demais para acomodar as páginas constantemente referenciadas por ele (MAIA, 2001).

Sob a ótica do sistema, o *thrashing* ocorre quando existem mais processos competindo por memória real que espaço disponível. Neste caso, o sistema tenta administrar a memória de forma que todos os processos sejam atendidos, descarregando processos para a memória secundária e carregando processos para a memória principal. Se esse mecanismo for levado ao extremo, o sistema passará mais tempo fazendo *swapping* do que executando processos (MAIA, 2001).

De qualquer forma, se persistem mais processos para serem executados que memória real disponível, a solução que realmente restaura os níveis de desempenho adequados é a expansão da memória principal. É importante ressaltar que este problema não ocorre apenas em sistemas que implementam memória virtual, mas também em sistemas com outros mecanismos de gerência de memória (MACHADO; MAIA, 2002).

3.3 LOCALIDADE DE REFERÊNCIA

O mecanismo de memória virtual apesar de suas vantagens, introduz um grande problema. Sempre que um processo faz referência a uma de suas páginas e esta não se encontra na memória (*page fault*), exige do sistema operacional pelo menos uma operação de E/S, que, quando possível, deve ser evitada (MAIA, 2001).

Conforme Machado e Maia (2002) quando um processo é criado, todas as suas páginas estão na memória secundária. À medida que acontecem referências às páginas virtuais, elas são transferidas para o *working set* do processo na memória principal (*page in*). Sempre que

um processo faz referência a uma página, o sistema verifica se a página já se encontra no *working set* do processo. Caso a página não se encontre no *working set*, ocorrerá o *page fault*. O *working set* do processo deve ter um limite máximo de páginas permitidas. Quanto maior o *working set*, menor a chance de ocorrer uma referência a uma página que não esteja na memória principal (*page fault*).

Localidade de referência é uma propriedade que os programas apresentam de realizarem acessos a posições consecutivas da memória em regiões de endereços próximos. Isso deve-se ao fato de que geralmente os programas são concebidos na forma de métodos ou funções os quais possuem laços de repetição que realizam acessos a um pequeno conjunto de variáveis temporárias denominadas variáveis locais e aos parâmetros recebidos. Ao conjunto de acessos constatados pela execução da seqüência de instruções de um método ou função denomina-se localidade temporal. Ao padrão de acessos à variáveis locais em determinado momento denomina-se localidade espacial (OLIVEIRA; CARISSIMI; TOSCANI, 2001, p. 97).

Genericamente, em sistemas de memória virtual paginada, processos computacionais tendem a explorar, com maior ou menor grau de intensidade, a propriedade conhecida como localidade de referências. Esta propriedade afirma que parte das páginas que compõem o espaço de endereçamento virtual de um programa são efetivamente necessárias a execução num certo intervalo de tempo. Em outras palavras, os dados processados em um determinado momento de execução normalmente se concentram em algumas poucas páginas. Esta propriedade é o que fundamenta e valoriza a importância da chamada hierarquia de memória, ou seja, a forma como os componentes físicos de memória são organizados na arquitetura de um computador. Dispositivos rápidos, porém dimensionalmente limitados, deve armazenar temporariamente os dados mais requisitados por um processo. Por outro lado, dispositivos de memória secundária lentos, mas com grande capacidade de armazenamento – alocam a totalidade do espaço de endereçamento utilizado pelo processo (CASSETTARI; MIDORIKAWA, 2004).

3.4 WORKING-SET

O *working set* de um processo é o conjunto de páginas referenciadas por ele durante determinado intervalo de tempo. Uma outra definição seria que o *working set* é o

conjunto de páginas constantemente referenciadas pelo processo, devendo permanecer na memória principal para que ele execute de forma eficiente. Caso contrário, o processo poderá sofrer com a elevada taxa de paginação (*thrashing*), comprometendo seu desempenho. O *working set* do processo deve ter um limite máximo de páginas permitidas. Quanto maior o *working set*, menor a chance de ocorrer uma referência a uma página que não esteja na memória principal (*page fault*) (MACHADO; MAIA, 2002).

Portanto, ao conjunto de páginas referenciadas em um determinado período por um processo denomina-se *working set*. Para otimizar o tempo de resposta os sistemas operacionais possuem algoritmos que analisam o conjunto de páginas a que um processo está fazendo acessos para evitar a remoção de páginas que estejam neste conjunto. Com isso há a probabilidade de que um próximo acesso a memória encontre a página, minimizando a ocorrências de *page-faults* (OLIVEIRA; CARISSIMI; TOSCANI, 2001, p. 103).

O conceito de *working set* também foi formalizado com base na propriedade de localidade de referências. O *working set* de um processo em execução é o conjunto das páginas requeridas para o seu processamento em um dado intervalo de tempo. Ele representa, de certa forma, a caracterização da propriedade de localidade: as páginas que compõem o conjunto retratam, por si só, uma localidade inerente ao momento de execução do processo (CASSETTARI; MIDORIKAWA, 2004).

A importância desta propriedade decorre do seguinte fato: o grau de localidade apresentado por um programa pode afetar diretamente o seu tempo de processamento, haja vista a influência que exerce na reutilização das páginas residentes e, conseqüentemente, no desempenho do sistema de memória. Logo, as políticas de substituição de páginas mais eficientes procuram explorar, cada uma ao seu modo, as características de localidade encontradas no processos, sobretudo localidade temporal. Um caso de localidade temporal é observado quando algumas poucas páginas de memória são continuamente referenciadas durante o intervalo de tempo considerado. Apesar disso, se os acessos a memória se restringem a uma região específica do espaço de endereçamento virtual, isto é, se as páginas referenciadas possuem endereços próximos entre si, observamos um caso de localidade espacial (CASSETTARI; MIDORIKAWA, 2004).

A localidade espacial é positiva quando está restrita a um conjunto de páginas reduzido, como um *loop* no qual poucas páginas são referenciadas – em relação ao tamanho de memória disponível. Um padrão de acessos a memória desse tipo faz com que o *working set* do processo também permaneça reduzido, induzindo à desejada reutilização das páginas residentes. Acompanhada de forte localidade temporal, portanto, a localidade espacial é

interessante. Caso contrário, a presença ou não de localidade espacial é diferente à grande maioria das políticas tradicionais de gerenciamento da memória principal. Ainda assim, pode ser muito importante para o gerenciamento eficiente de memórias *cache*, o que a torna relevante (CASSETTARI; MIDORIKAWA, 2004).

A presença de localidade temporal, por outro lado, é invariavelmente benéfica por que implica na reutilização pontual de páginas residentes. O LRU é um exemplo de algoritmo de substituição que procura prover e explorar tal propriedade no intuito de diminuir a ocorrência de faltas de páginas: seu critério é sempre substituir páginas de memória que não tenham sido recentemente referenciadas, ou seja, páginas que não demonstrem a tendência de acessos contínuos no momento de execução observado.

A construção da tabela de páginas se divide em dois tipos que são: as modificáveis e as não modificáveis. As páginas não modificáveis, geralmente possuem código executável, podem ser removidas da memória numa situação em que a substituição de páginas é necessária. Já as páginas modificáveis (geralmente áreas de variáveis cujo conteúdo altera a cada operação de atribuição) demandam uma operação de *page-out* (ou seja, necessitam ser gravadas em disco antes de serem substituídas por uma operação de *page-in*). Portanto, quanto mais informações um sistema operacional possuir a respeito do *working set* de um processo, tanto mais eficientemente ele poderá aplicar os algoritmos de substituição de páginas no módulo de gerenciamento de memória virtual (OLIVEIRA; CARISSIMI; TOSCANI, 2001, p. 103).

3.5 TRABALHOS CORRELATOS

Existem algumas pesquisas relacionadas ao estudo do uso de memória em sistemas operacionais, dentre elas pode-se citar: o simulador de mecanismos de gerência de memória real e virtual, o *Virtual XT* (VXT), o *Address Translation Simulator* (ATS), o Elephantools e visualização de gráficos sobre a análise e evolução de sistemas de memórias.

3.5.1 Simulador de mecanismos de gerência de memória real e virtual

Este trabalho consiste em uma ferramenta visual com simulador gráfico destinado ao

ensino e aprendizado dos conceitos e técnicas de gerência de memória real e virtual implementados nos sistemas operacionais atuais (MORITZ, 2004).

O Software mostra como internamente o sistema operacional controla a memória ao simular vários modelos de carga (Figura 16).

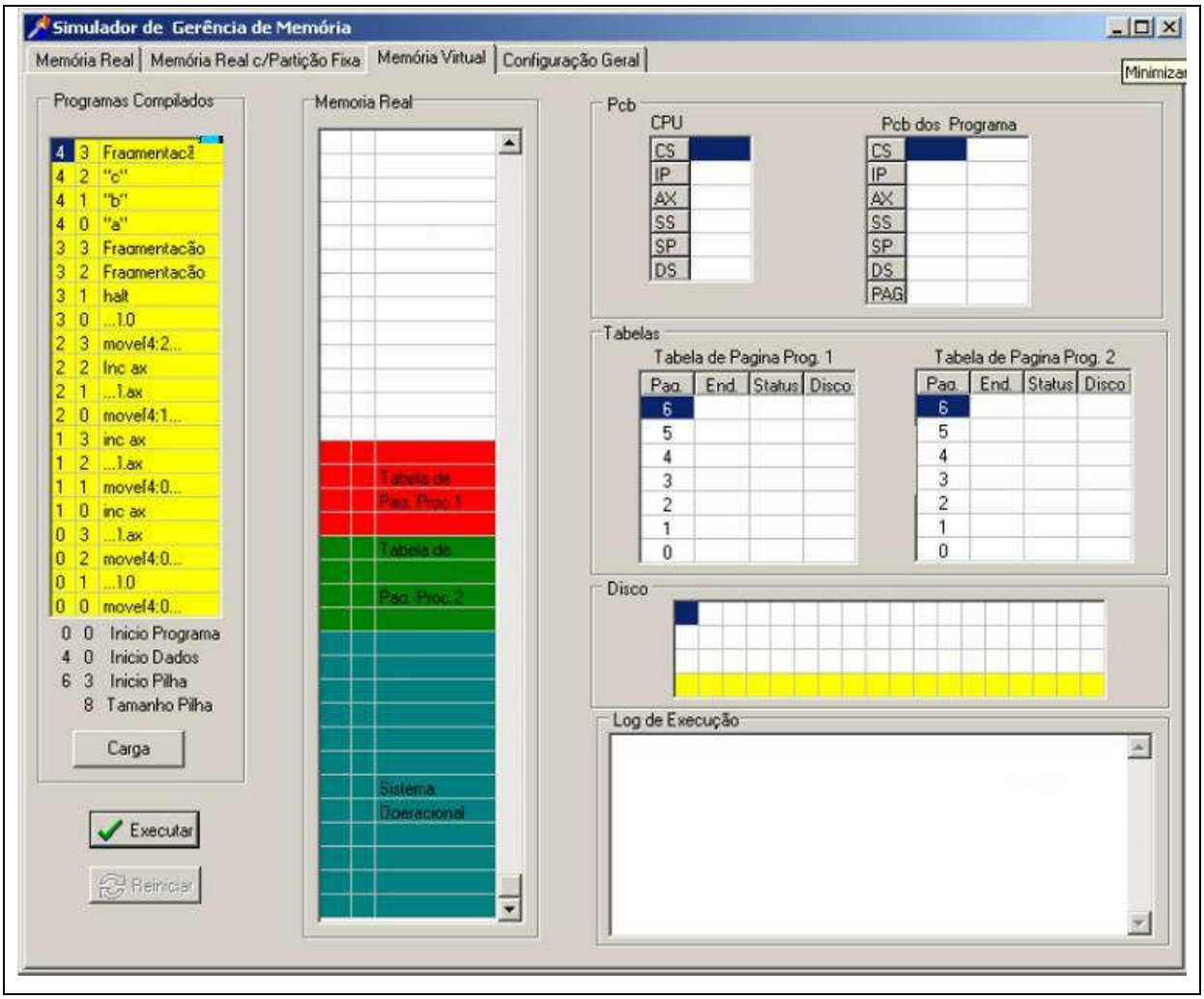


Figura 16 – Simulador de gerência de memória

3.5.2 VXT

O VXT é um projeto que simula uma máquina baseada no processador 8086, desenvolvido através atividades de sala de aula, trabalhos de conclusões de curso e projetos de iniciação científica (MATTOS; TAVARES; FARIAS, 2004).

O projeto VXT possui uma interface gráfica que interage com o usuário com a finalidade de mostrar passo a passo a execução de um programa em modo de instruções de máquina. Segundo Mattos, Tavares e Farias (2004), este projeto tem um grande valor

didático, por auxiliar no ensino de sistemas operacionais. Além disso, o VXT possui outras características importantes como:

a) trabalha no conceito de cliente-servidor;
b) tem interface completamente isolada da implementação do processador;
c) permite que sejam executados vários simuladores em paralelo;
d) permite que seja salvo ou carregado o contexto de execução de uma determinada simulação;
e) permite salvar *log* de uma simulação, possibilitando assim efetuar uma análise destas informações.

3.5.3 ATS

O ATS é uma ferramenta desenvolvida na linguagem de programação Java. Seu objetivo principal é ser um explorador de endereços de memória e de converter o conteúdo de um endereço de memória virtual em memória real utilizando um ou dois níveis de tabelas de páginas, contribuindo assim para fins didáticos ou de testes de uso de memória. É possível selecionar um determinado segmento de endereço lógico e mover para o endereço físico quantas vezes desejar, tendo assim várias simulações, obtendo uma análise do comportamento de endereçamentos (NATIONAL, 2005).

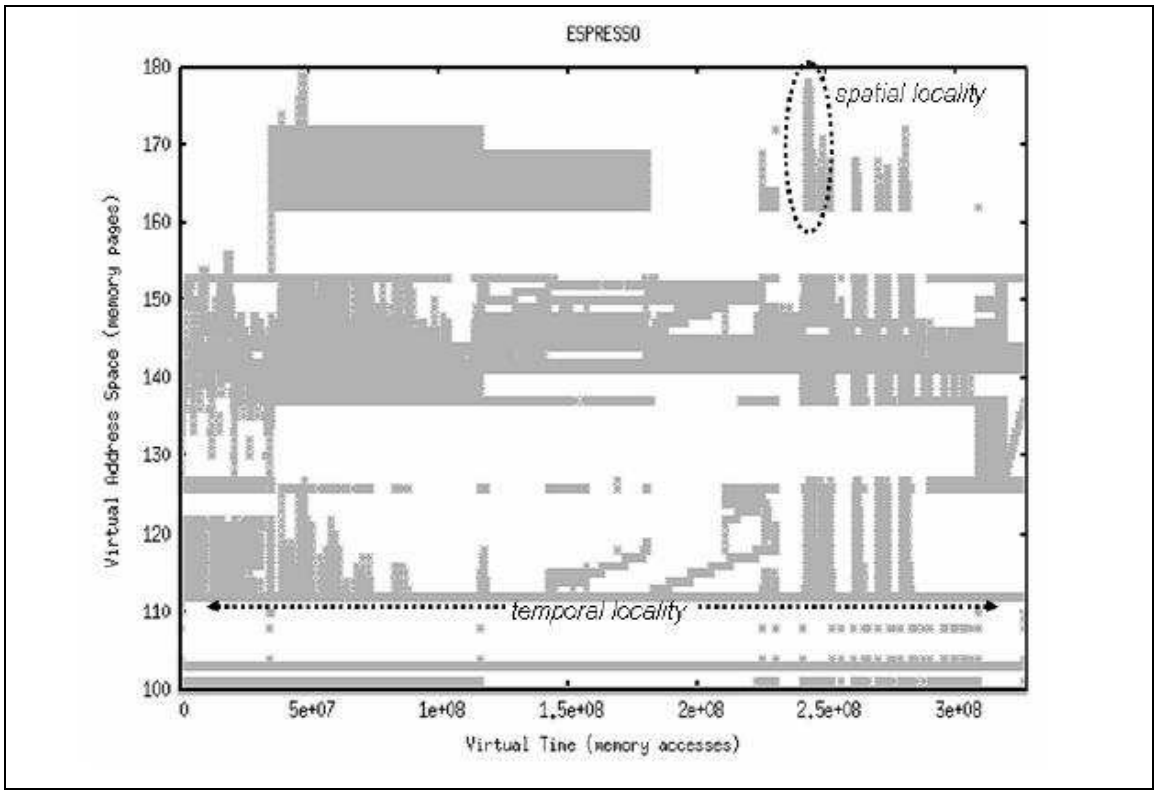
3.5.4 Elephantools

O Elephantools é um software composto por várias ferramentas com a finalidade de análises sobre gerência de memória que contribui para estudos sobre localidade de referências, além de possuir recursos visuais e estatísticas para identificar padrões de acesso a memória.

Segundo Cassettari e Midorikawa (2004), esta ferramenta é subdividida em três módulos que são:

a) tela *trace*: tem como finalidade apresentar graficamente os padrões de acesso a regiões de memória;
b) mapa 3D: apresenta os padrões de acesso destacando informações sobre a distribuição dos mesmos entre as diversas posições de memória referenciadas;

c) *trace explorer*: com esta ferramenta é possível gerar um mapa, um histograma, um mapa de variações, um mapa de distâncias e um relatório analítico de uso de acessos a localidades da memória. Este relatório contém informações como número total de acessos e de páginas referenciadas, maior e menor página referenciada, posição de acessos na fila LRU e a variação média de acessos na fila LRU. Um exemplo de gráfico que pode ser gerado com esta técnica pode ser visto na figura 17.



Fonte: Cassettari e Midorikawa (2004).

Figura 17 - Gráfico de acessos a memória

O gráfico de acesso a memória consiste em representar de forma visual uma matriz de pontos contendo informações correspondentes a quais páginas de memória estão referenciando os processos que estão em execução. As informações apresentadas no gráfico são simples, onde o eixo horizontal corresponde ao tempo de processo virtual e o eixo vertical corresponde ao espaço de endereçamento virtual (CASSETTARI; MIDORIKAWA, 2004).

3.6 JCCKIT

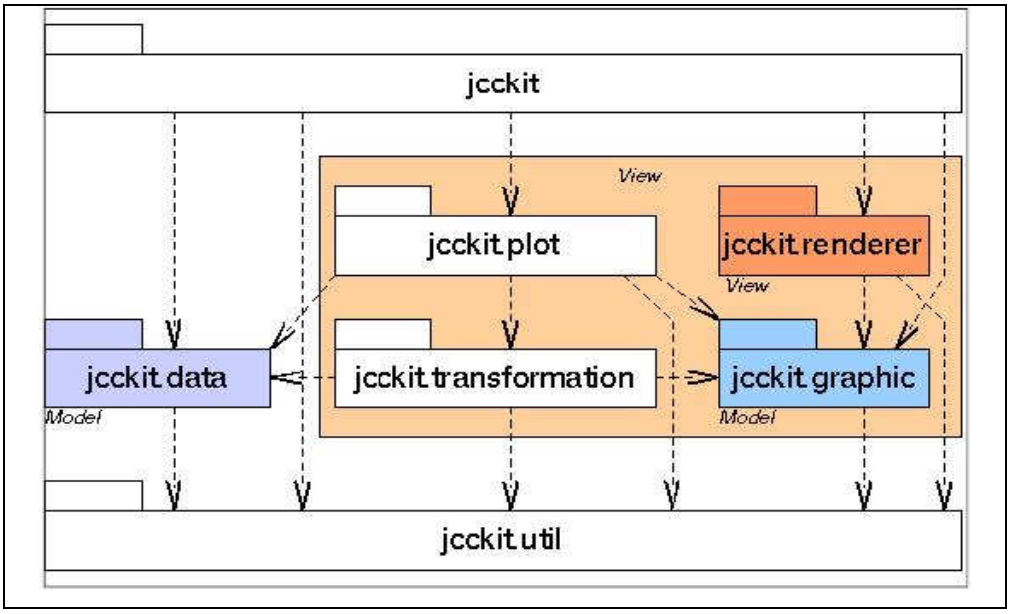
Java Chart Construction Kit (JCCKit) é uma biblioteca desenvolvida na linguagem de

programação Java, que possui uma estrutura flexível para a construção de gráficos e diagramas científicos. As classes da JCCKit são agrupadas em sete pacotes constituindo quatro camadas. O quadro 1 mostra os sete pacotes com a descrição de cada uma das classes:

PACOTE	DESCRIÇÃO
Jcckit	É um pacote composto por classes responsáveis pela montagem de gráficos para uso imediato.
jcckit.plot	É um pacote com classes responsáveis pela geração de gráficos e diagramas baseados em dados modelados por classes do pacote jcckit.data.
jcckit.renderer	É um pacote com classes responsáveis em renderizar um gráfico (composto por objetos gráficos de jcckit.graphic) para um determinado dispositivo de saída.
jcckit.data	É um pacote com classes responsáveis em guardar e controlar os dados de coordenadas de gráficos.
jcckit.transformation	É um pacote com classes responsáveis em transformar os dados de coordenadas em dispositivos de coordenadas.
jcckit.graphic	É um pacote com classes responsáveis em representar um gráfico ou diagrama no dispositivo de objetos gráficos, ou seja, numa área de desenho.
jcckit.util	É um pacote com classes que contém uma coleção de métodos utilitários estáticos.

Quadro 1 – Pacotes do JCCKit

Na figura 18 é apresentado um diagrama de pacotes JCCKit mostrando suas dependências e suas divisões em camadas.



Fonte: Elmer (2005).

Figura 18 – Diagrama dos pacotes JCCKit

Os cinco pacotes centrais podem ser agrupados dentro de um modelo que represente o pacote de gráficos e diagramas, sendo que quatro destes pacotes são para visualização. Dentro deste modelo de visualização existem dois pacotes abstratos, sendo um para modelo (jcckit.graphic) e outro para exibição (jcckit.renderer).

A figura 19 mostra alguns tipos de gráficos que podem ser construídos com a biblioteca JCCKit.

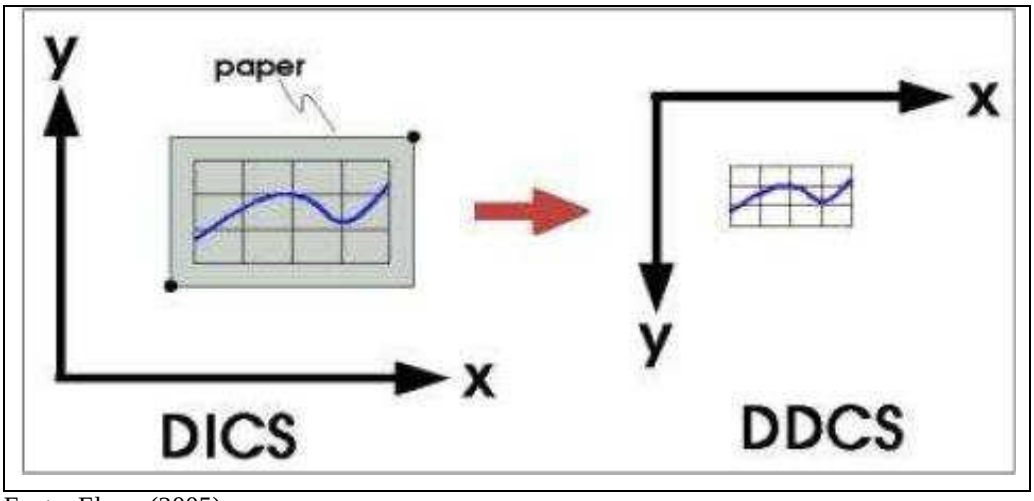
Figura 19 – Exemplo de gráficos construídos com a biblioteca JCCKit

Existem três sistemas de coordenadas no jcckit, descritos no quadro 2.

SISTMA DE COORDENADA	DESCRIÇÃO
Data Coordinate System (DCS)	Este é o sistema de coordenadas dos dados a serem traçados.
Device-Independent Coordinate System (DICS)	Um sistema de coordenadas cartesianas independentes do dispositivo.
Device-Dependent Coordinate System (DDCS)	Sistema de coordenadas do dispositivo que apresenta a renderização de diagramas e gráficos.

Quadro 2 – Sistema de coordenadas do JCCKit

Pontos mapeados no sistema de coordenadas DCS (representado por instâncias da classe `DataPoint`) podem ser mapeados das seguintes formas: DCS para DICS e DICS para DDCS, conforme necessidade da implementação. A figura 20 mostra uma comparação entre os sistemas de coordenadas DICS e DDCS.



Fonte: Elmer (2005).

Figura 20 – Representação do mapeamento DICS para DDCS

A figura 21 mostra um exemplo de código fonte definindo coordenadas x e y, tipos e formas e tamanhos para um gráfico.

```
private void setCoordinateSystem(double xMin, double yMin,
                                 double xMax, double yMax)
{
  Properties props = new Properties();
  props.put("xAxis/minimum", Double.toString(xMin));
  props.put("xAxis/maximum", Double.toString(xMax));
  props.put("xAxis/ticLabelFormat", "%1.3f");
  props.put("xAxis/axisLabel", "");
  props.put("yAxis/minimum", Double.toString(yMin));
  props.put("yAxis/maximum", Double.toString(yMax));
  props.put("yAxis/ticLabelFormat", "%1.3f");
  props.put("yAxis/axisLabel", "");
  props.put("yAxis/axisLength", "0.8");
  CartesianCoordinateSystem cs = new CartesianCoordinateSystem(
                new ConfigParameters(new PropertiesBasedConfigData(props)));
  _plotCanvas.getPlot().setCoordinateSystem(cs);
}

private void drawMarker(GraphPoint point)
{
  double x0 = _anchor.getX();
  double y0 = _anchor.getY();
  double x1 = point.getX();
  double y1 = point.getY();
  GraphPoint center = new GraphPoint(0.5 * (x0 + x1), 0.5 * (y0 + y1));
  setMarker(new Rectangle(center, Math.abs(x1 - x0), Math.abs(y1 - y0),
                          MARKER_ATTRIBUTES));
}

private void setMarker(GraphicalElement marker)
{
  _plotCanvas.setMarker(marker);
  _plotCanvas.getGraphicsCanvas().repaint();
}

private GraphPoint getPosition(MouseEvent event)
{
  return _plotCanvas.mapCursorPosition(event.getX(), event.getY());
}

private void changeViewingWindow(GraphPoint point)
{
  DataPoint p0 = _plotCanvas.getPlot().transform(_anchor);
  DataPoint p1 = _plotCanvas.getPlot().transform(point);
  ...
}
```

Figura 21 – Código fonte para configuração de coordenadas e de desenho para o gráfico

No quadro 3 apresenta-se a descrição de cada método.

MÉTODO	FUNCIONALIDADE
setCoordinateSystem	Define as coordenadas de tela e o intervalo para as coordenadas x e y para o gráfico que será usado para a montagem deste.
drawMarker	Define um novo intervalo para as coordenadas x e y conforme a área do gráfico selecionada, ou seja, gera um zoom de uma parte do gráfico e o redesenha.
getPosition	Pega a coordenada x e y a partir de um ponto selecionado pelo *mouse* na tela.
changeViewingWindow	Pega o ponto inicial e final de uma área selecionada para ser redesenhada.

Quadro 3 – Funcionalidade dos métodos de construção de gráficos

4 DESENVOLVIMENTO

Neste capítulo descreve-se os requisitos principais do trabalho, a especificação do módulo de análise de localidade de referência e os aspectos relacionados a sua implementação.

4.1 REQUISITOS PRINCIPAIS DO PROBLEMA

A ferramenta deverá:

a) implementar um contador de acesso a posições de memória real da JPC (Requisito Funcional - RF);
b) implementar um modelo de *log* dos acessos a memória real da JPC (RF);
c) disponibilizar ao final da execução a possibilidade de visualização das expressões coletadas (RF);
d) ser implementada no ambiente de desenvolvimento NetBeans (Requisito Não-Funcional - RNF);
e) ser implementada na linguagem de programação Java (RNF).

4.2 ESPECIFICAÇÃO

Nesta seção são apresentados os diagramas de casos de uso, de atividades e de classes. Os diagramas foram desenvolvidos utilizando a ferramenta Enterprise Architect versão 4.5 da empresa Sparx Systems.

4.2.1 Diagrama de casos de uso

O diagrama de casos de uso apresentado nesta seção mostra o usuário como ator principal e as ações efetuadas por este na máquina JPC. A figura 22 mostra essa relação entre

usuário e a aplicação.

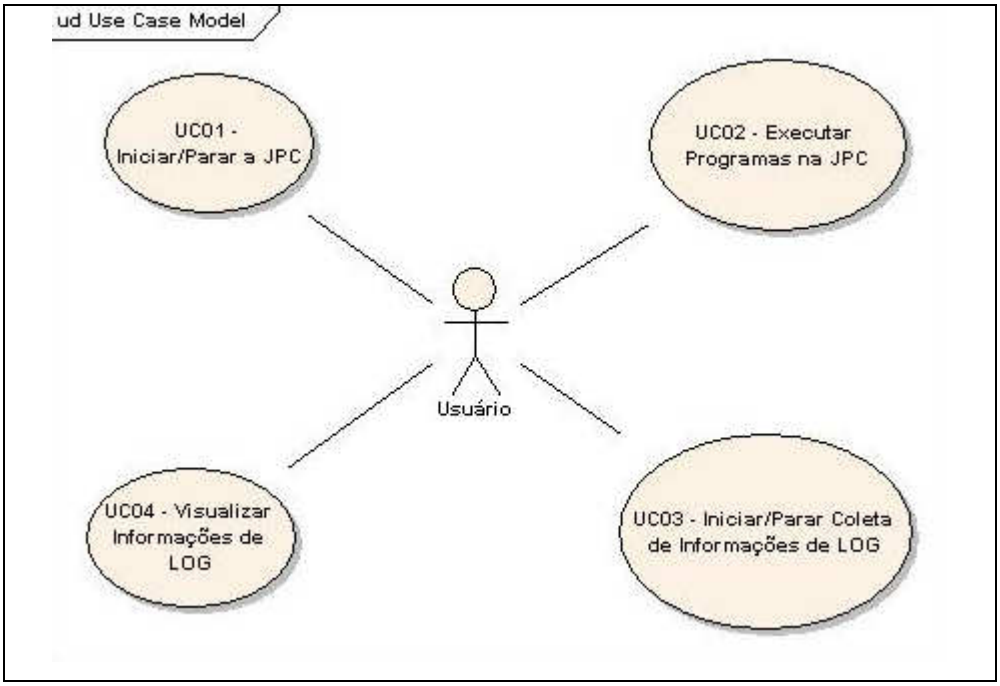


Figura 22 – Diagrama de casos de uso entre usuário e JPC

Como pode ser visto, o usuário pode efetuar as seguintes ações:

a) iniciar ou parar a JPC: o usuário pode interagir com a JPC clicando em um botão para iniciar ou parar a sua execução;
b) executar programas na JPC: o usuário tem permissão para poder efetuar a execução de qualquer programa que seja compatível;
c) iniciar ou parar coleta de informações de *log*: o usuário pode definir diretamente a opção desejada de iniciar ou para a coleta de informações de *log*;
d) visualizar informações de *log*: é permitido ao usuário efetuar visualizações das informações geradas pela coleta efetuada no caso UC03.

4.2.2 Diagrama de atividades

A figura 23 descreve a seqüência de atividades do processo de controle de *log*. Inicialmente a máquina deve ser configurada (pode ser carregada uma imagem rodando em Windows ou *Linux* conforme desejado). Em seguida é iniciada a máquina e enquanto a máquina estiver ligada verifica se foi desligada. Se afirmativo, ou seja, se a máquina foi desligada então é finalizado o processo. Caso contrário, verifica se o controle de *log* foi ativado e caso afirmativo o sistema realiza o processo de gravação das informações, se não

volta na verificação de máquina ligada ou desligada e sempre testando em seguida a verificação de *log* ativo.

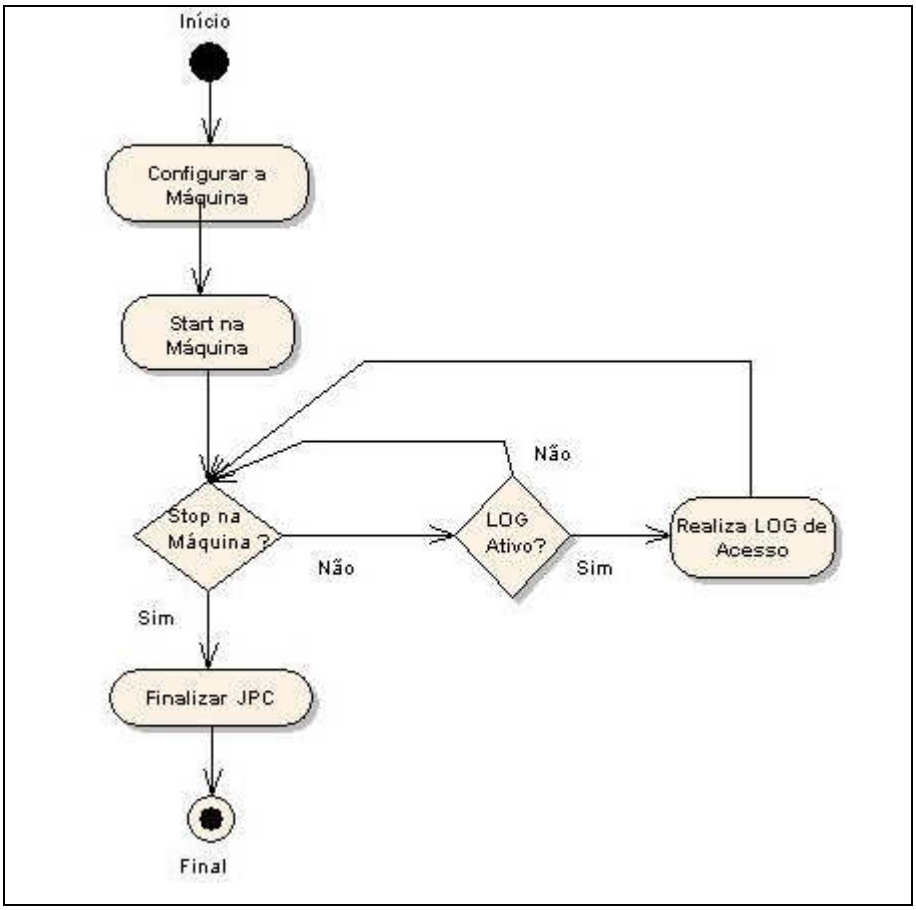


Figura 23 - Diagrama de atividades do processo de aquisição de dados

4.2.3 Diagrama de classes

O diagrama de classes apresentado nesta seção mostra a estrutura da aplicação de gerenciamento de *log* desenvolvida no trabalho (figura 24).

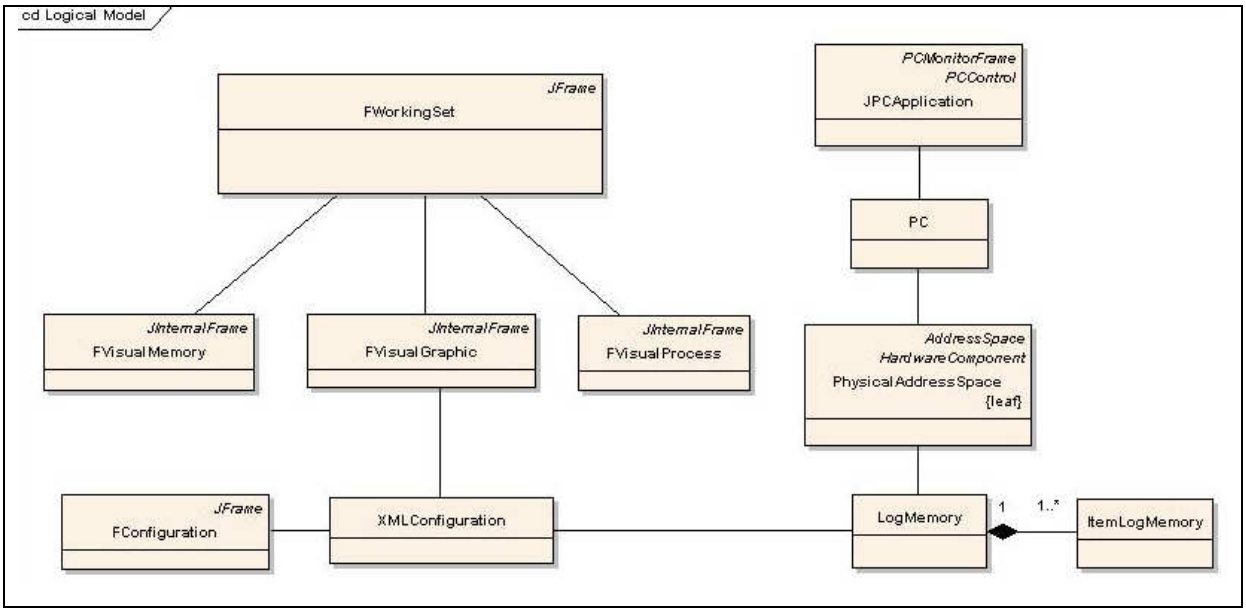


Figura 24 - Diagrama de classes da estrutura desenvolvida no trabalho

O quadro 4 detalha os métodos existentes na classe `FWorkingSet`.

MÉTODO	FUNCIONALIDADE
FWorkingSet ()	Construtor da classe que inicializa a aplicação com um formulário de visualização de *log*.
createGraphicPanel()	Cria um painel dentro do formulário principal onde será acoplado os gráficos gerados.
createButtonsPanel()	Cria um painel dentro do formulário principal com os botões de controle da aplicação que são: abrir arquivo; gerar gráfico; visualizar memória; configurações e sair.
createControlPanel()	Cria um painel dentro do formulário principal que mostra detalhes de todas as ações efetuados pela aplicação.
btnOpeFileClique()	Disponibiliza uma tela com os diretórios disponíveis do seu computador e solicita ao usuário selecionar um arquivo de informações de *log* (arquivos com extensão log).
btnExecuteClique()	Executa a leitura do arquivo selecionado e efetua a geração do gráfico percorrendo todos os registros gravados no arquivo, mostrando ainda em forma de percentual o ponto que esta o processo.
btnMemoryViewClique()	Executa a leitura do arquivo selecionado e efetua a geração de uma listagem com tempo e endereço acessado de todos os registros gravados no arquivo, mostrando ainda em forma de percentual em que ponto está o processo.
btnConfigClique()	Mostra ao usuário uma tela com opções de configurações para a aplicação como: gerar gráfico em forma de linhas ou pontos; definir o tamanho dos pontos do gráfico e tempo máximo para cada arquivo de log guardar.
btnSairClique()	Finaliza a aplicação.

Quadro 4 – Especificação da classe `FWorkingSet`

O quadro 5 detalha os métodos existentes na classe `FVisualMemory`.

MÉTODO	FUNCIONALIDADE
FVisualMemory (String title)	Construtor da classe que inicializa a visualização de uma listagem das informações de acesso a memória em um formulário interno. Possui um parâmetro que define o título da janela.
add(ArrayList list)	Associa as informações contidas em um ArrayList com o formulário interno.
execute()	Executa a leitura do arquivo selecionado conforme descrito no quadro 3 e efetua a geração das informações percorrendo todos os registros gravados no arquivo, mostrando ainda em forma de percentual em que ponto está o processo.
getMemoryView()	Retorna um formulário interno com os elementos da classe para que possa ser acoplado ao formulário principal.

Quadro 5 – Especificação da classe `FVisualMemory`

O quadro 6 detalha os métodos existentes na `FVisualGraphic`.

MÉTODO	FUNCIONALIDADE
FVisualGraphic(String title)	Construtor da classe que inicializa a visualização de um gráfico das informações de acesso a memória em um formulário interno. Possui um parâmetro que define o título da janela.
createPlotCanvas(double xMin, double xMax, double yMin, double yMax)	Define as propriedades da área onde será impresso o gráfico como coordenadas, tamanhos, cores, etc...
addPoint(double x, double y)	Adiciona uma informação de coordenadas x e y ao gráfico definindo a posição correspondente a ser desenhado na tela.
execute()	Executa a chamada de dois métodos da jcckit onde um deles adiciona a curva gerada pelos pontos adicionados através do método addPoint e um outro método connect que efetua a conexão de todos os elementos envolvidos.
save()	Efetua a gravação de uma imagem do gráfico em disco no formato *jpg*. Solicita ao usuário o diretório a ser gravado.
getGraphicView()	Retorna um formulário interno com os elementos da classe para que possa ser acoplado ao formulário principal.
getPosition(MouseEvent event)	Converte a coordenada x e y de tela (enviada por parâmetro através de um clique do mouse) em coordenada de gráfico.
drawMarker(GraphPoint point)	Cria um ponto no gráfico conforme as coordenadas definidas no parâmetro.
setMarker(GraphicalElement marker)	Insere o ponto construído no método acima na área de desenho e pinta os pontos na tela.
changeViewingWindow(GraphPoint point)	Desenha uma parte do gráfico conforme uma parte selecionada, ou seja, efetua um *zoom* de uma determinada área selecionada.
setCoordinateSystem(double xMin, double yMin, double xMax, double yMax)	Define as coordenadas x e y da área de desenho do gráfico. Utilizado para *zoom* onde diminui a distância x e y para ampliar a imagem.
resetViewWindow()	Desenha o gráfico completo conforme as coordenadas iniciais. Utilizado para voltar ao estado inicial quando aplicado *zoom*.

Quadro 6 – Especificação da classe `FVisualGraphic`

O quadro 7 detalha os métodos existentes na classe `FVisualProcess`.

MÉTODO	FUNCIONALIDADE
FVisualProcess(boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)	Construtor da classe que inicializa a visualização das informações processadas pela aplicação em um formulário interno. Possui parâmetros padrões de controle de um formulário característicos da classe `JInternalFrame` do Java.
showWindow()	Define o formulário como visível.
clearInformation()	Limpa todo texto informativo da classe.
addInformation(String sText)	Adiciona um texto informativo para a classe.
replaceLastLine(String sText)	Substitui o texto da última linha adicionada por um novo texto.
refresh()	Atualiza os componentes envolvidos na classe.

Quadro 7 – Especificação da classe `FVisualProcess`

O quadro 8 detalha os métodos existentes na classe `FConfiguration`.

MÉTODO	FUNCIONALIDADE
FConfiguration()	Construtor da classe que inicializa a visualização de um formulário com opções para configuração de propriedades para a aplicação. Efetua leitura e escrita em arquivo XML com as informações necessárias.
createRadioGroupGraphic(JPanel parent)	Cria um grupo de seleção dentro do formulário com as seguintes opções: Gerar Gráfico em Pontos; Gerar Gráfico em Linhas.
createPanelPoints(JPanel parent)	Cria um campo no formulário para ser informado o tamanho das linhas de cada gráfico a ser gerado.
createPanelMemory(JPanel parent)	Cria um campo no formulário para ser informado limite de tempo que será armazenado nos arquivos de *log*.
createControlButtons(JPanel parent)	Cria um painel dentro do formulário com os botões de controle que são: salvar e sair.
btnSalvarClique(ActionEvent evt)	Salva as informações do formulário em um arquivo XML.
btnSairClique(ActionEvent evt)	Fecha o formulário de configurações.
showWindow()	Define o formulário como visível.

Quadro 8 – Especificação da classe `FConfiguration`

O quadro 9 detalha os métodos existentes na classe `XMLConfiguration`.

MÉTODO	FUNCIONALIDADE
getSizeFile()	Retorna o valor para o campo tamanho do arquivo.
setSizeFile(int sizeFile)	Define o valor para o campo tamanho do arquivo.
getSizePoints()	Retorna o valor para o campo tamanho dos pontos.
setSizePoints(double sizePoints)	Define o valor para o campo tamanho dos pontos.
getUsePoints()	Retorna o valor para o campo usa pontos.
setPropUsePoints(boolean flag)	Define o valor para o campo usa pontos
loadXML()	Executa a leitura do arquivo XML.
saveXML()	Executa a gravação do arquivo XML.

Quadro 9 – Especificação da classe `XMLConfiguration`

O quadro 10 detalha os métodos existentes na classe `LogMemory`.

MÉTODO	FUNCIONALIDADE
LogMemory(String fileName)	Construtor da classe que efetua leitura do arquivo XML a partir do arquivo definido pelo seu parâmetro. Este parâmetro é atribuído como um atributo da classe que pode ser usado a qualquer momento por toda a classe.
getLogDir()	Retorna o diretório do *log* que esta definido na classe instânciada.
setFileName(String sFileName)	Define qual será o arquivo de *log* gerado.
getMaxTimeFile()	Retorna o tempo máximo definido no arquivo XML.
getTimeCount()	Retorna o contador de tempo que é incrementado a cada uso, ou seja, a cada posição de memória lida este método é chamado e utilizado como tempo de acesso.
resetTimeCout()	Inicializa o contador de tempo em zero. É chamado cada vez que se deseja iniciar um arquivo de *log*.
getSizeRecord()	Retorna o tamanho de cada registro do arquivo de log. Esta definido como padrão de tamanho 10, pois é o máximo permitido pela aplicação ter 10 dígitos para cada bloco de registro de informações no arquivo de *log*. A figura 26 mostra a divisão dos blocos.
formatRecord(String) sValue)	Formata um bloco de informação com tamanho de 10 dígitos, onde se for menor complementa com zeros a esquerda.
saveToFile(String sConteudo, boolean bAdicionar)	Executa a gravação em disco conforme especificado o arquivo pelo método setFileName. Possui um parâmetro com o conteúdo de *log* e outro que define se concatena ou não as informações no arquivo.
loadBuffer()	Executa a leitura do arquivo de *log* definido no método setFileName. Atribui estas informações em uma lista que pode ser manipulada conforme descrito nos métodos descritos abaixo.
clear()	Executa a limpeza das informações da lista.
first()	Posiciona a lista no seu primeiro registro.
hasValue()	Verifica se existem registros a serem lidos na lista.
next()	Posiciona a lista no próximo registro.
getIndex()	Retorna o índice posicionado da lista.
getAdress()	Retorna o valor do atributo endereço da classe.
getTime()	Retorna o valor do atributo tempo da classe.
getSize()	Retorna o valor do atributo tamanho da classe.
getLogList()	Retorna a listagem completa com todas as informações do arquivo lido.

Quadro 10 - Especificação da classe `LogMemory`

O quadro 11 detalha os métodos existentes na classe `ItemLogMemory`.

MÉTODO	FUNCIONALIDADE
setAdress(int iAdress)	Define o valor para o atributo endereço conforme seu parâmetro.
getAdress()	Retorna o valor do atributo endereço.
setTime(int iTime)	Define o valor para o atributo tempo conforme seu parâmetro.
getTime()	Retorna o valor do atributo tempo.
setType(registerType tType)	Define o valor para o atributo tipo de registro conforme seu parâmetro. Este tipo esta definido como enumeração e pode assumir os seguintes tipos: rtNull; rtRead; rtWrite;
getType()	Retorna o valor do atributo tipo de registro.

Quadro 11 - Especificação da classe `ItemLogMemory`

4.3 IMPLEMENTAÇÃO

Esta seção descreve as técnicas e a operacionalidade da implementação.

4.3.1 Classe `PhysicalAddressSpace`

A classe `PhysicalAddressSpace` implementa o sistema de endereçamento de memória física. Abaixo detalha-se os métodos implementados para definição do diretório de gravação dos arquivos e controles por um indicador que definido pelo usuário gera ou não o *log*, permitindo assim efetuar a geração de *log* a partir de qualquer ponto de execução da JPC. No quadro 12 detalha-se os principais métodos da classe para controle de *log*.

MÉTODO	FUNCIONALIDADE
createFileName()	Define um novo arquivo de *log* a ser gerado.
isLogActived()	Verifica se está ativada a geração de *log*.
setLogActived(boolean flag)	Define a ativação do *log* conforme o valor do parâmetro.
resetLog()	Inicializa um novo arquivo de *log*.
getReadMemoryBlockAt(int offset)	Executa a leitura de um endereço de memória e gera a informação de *log* conforme o endereço acessado.
getWriteMemoryBlockAt(int offset)	Executa a escrita de um endereço de memória e gera a informação de *log* conforme o endereço acessado.

Quadro 32 - Especificação da classe `PhysicalAddressSpace`

A solução para a captura dos acessos a memória deu-se através da gravação das informações de leitura em endereços (figura 28) e escrita em endereços (figura 29) que será descrito mais adiante sua implementação.

4.3.2 Classe `PC`

A classe `PC` efetua todo o controle de hardware da máquina virtual adicionando todos os periféricos configurando e iniciando a execução. Sendo assim, foram implementados alguns controles de manipulação de *log*, pois é nesta classe que se efetua a comunicação entre a interface da aplicação com acesso ao hardware. Desta forma, no quadro 13 detalha-se os principais métodos da classe para controle de *log*.

MÉTODO	FUNCIONALIDADE
startLog()	Ativa o indicador de ativação de *log* contido na classe `PhysicalAddressSpace`.
stopLog()	Desativa o indicador de ativação de *log* contido na classe `PhysicalAddressSpace`.
isLogStarted()	Verifica se o indicador de *log* esta ativo.

Quadro 13 - Especificação da classe `PC`

4.3.3 Classe `PCApplication`

Esta classe monta a interface com o usuário e a JPC. Entretanto, foi necessário implementar um novo menu (figura 25) com os seguintes sub-menus:

a) *start*: cria um menu onde ativa um indicador que começa a gravar as informações de *log*;

b) *stop*: cria um menu onde desativa um indicador que para de gravar as informações de *log;*

c) *view*: abre a tela de visualização das informações de *log* conforme detalhado no item 3.2.3.1.

```
//*************************************************************************************************
// willian.begin of block in 2009 Sep 19
//*************************************************************************************************
// Adicionar novo menu de geração de log
JMenu jmnLog = new JMenu("Log");
jmnLog.setForeground(java.awt.Color.blue);

jmnLog.add("Start").addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent evt)
    {
        pc.startLog();
        JOptionPane.showMessageDialog(null, "Started");
    }
});
jmnLog.add("Stop").addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt)
    {
        pc.stopLog();
        JOptionPane.showMessageDialog(null, "Stoped");
    }
});
jmnLog.add("View").addActionListener(new ActionListener() {

    public void actionPerformed(ActionEvent e) {
        FWorkingSet ws = new FWorkingSet();
    }
});
bar.add(jmnLog);
//*************************************************************************************************
// willian.end of block in 2009 Sep 19
//*************************************************************************************************
```

Figura 25 – Código fonte para construção dos menus para controle de *log*

4.3.4 Implementação do modelo de *log*

Foi definido um modelo de arquivo no formato texto contendo a seguinte estrutura:

a) o arquivo apenas terá uma única linha continua de informação;
b) cada informação arquivada em blocos de tamanho fixo pode conter informações de leitura, escrita ou de tempo de acesso a uma determinada posição de memória;
c) cada bloco é representando por uma caractere w (quando for de escrita), ou um caractere r (quando for de leitura), ou t (quando for de tempo) mais a quantidade de caracteres configuráveis entre 1 a 10 dígitos para contagem de tempos.

A figura 26 mostra um exemplo de como fica o arquivo de *log* gerado com um dígito para informar o tipo do registro (escrita, leitura ou tempo), e nove dígitos para informar o tempo.

```
w001048560t000000001w001040475t000000002r001040530t000000003
```

Figura 26 – Modelo de *log* gerado pela ferramenta

A figura 27 mostra a tela de configuração da aplicação onde pode ser manipulado o limite máximo de tempos que cada arquivo deve conter. Como pode ser visto, no campo tempo máximo pode ser configurado este controle de tempo.

Figura 27 – Tela de configuração de tempos de *log*

Observa-se que o tempo ao qual referido, é o da resposta ao tempo da máquina virtual.

4.3.5 Implementação do coletor de dados de *log*

Para poder coletar as informações, efetuou-se um estudo sobre a funcionalidade da JPC. A seguir indica-se uma instância da classe `LogMemory` na classe `PhysicalAddressSpace` aplicando os métodos de salvamento de *log* quando um endereço da memória é acessado. A figura 28 mostra a implementação do código fonte para geração de acesso a memória de leitura e a figura 29 para acesso a memória de escrita.

```
protected Memory getReadMemoryBlockAt(int offset) {
//*************************************************************************************************
// willian.begin of block in 2009 Sep 19
//*************************************************************************************************
    // Verificar se o flag de geração de log esta ativo
    if  ((offset > 0) && (isLogActived())){
        try {
            // Adicionar 'w' (writer) - Informação de escrita no endereço memória
            String logBlock = "w";
            // Pegar posição da memória acessada e formatá-la
            logBlock += logMem.formatRecord(String.valueOf(offset));
            // Adicionar 't' (timer) - Informação do tempo que foi acessado
            logBlock += "t";
            // Pegar tempo de acesso da memória e formatá-la
            int time = logMem.getTimeCount();
            logBlock += logMem.formatRecord(String.valueOf(time));

            // geração de arquivos de log com tamanho parametrizado
            if  (time > logMem.getMaxTimeFile())
                resetLog();

            // Salvar log no arquivo
            logMem.saveToFile(logBlock, true);
        } catch (IOException ex) {
            Logger.getLogger(PhysicalAddressSpace.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
//*************************************************************************************************
// willian.end of block in 2009 Sep 19
//*************************************************************************************************
```

Figura 28 – Código fonte para geração de *log* de acesso a memória de leitura

```
protected Memory getWriteMemoryBlockAt(int offset) {
//*************************************************************************************************
// willian.begin of block in 2009 Sep 19
//*************************************************************************************************
    // Verificar se o flag de geração de log esta ativo
    if  ((offset > 0) && (isLogActived())){
        try {
            // Adicionar 'r' (reader) - Informação de leitura no endereço memória
            String logBlock = "r";
            // Pegar posição da memória acessada e formatá-la
            logBlock += logMem.formatRecord(String.valueOf(offset));
            // Adicionar 't' (timer) - Informação do tempo que foi acessado
            logBlock += "t";
            // Pegar tempo de acesso da memória e formatá-la
            int time = logMem.getTimeCount();
            logBlock += logMem.formatRecord(String.valueOf(time));

            // geração de arquivos de log com tamanho parametrizado
            if  (time > logMem.getMaxTimeFile())
                resetLog();

            // Salvar log no arquivo
            logMem.saveToFile(logBlock, true);
        } catch (IOException ex) {
            Logger.getLogger(PhysicalAddressSpace.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
//*************************************************************************************************
// willian.end of block in 2009 Sep 19
//*************************************************************************************************
```

Figura 29 – Código fonte para geração de *log* de acesso a memória de escrita

Além disso, implementou-se uma parametrização onde se define quando se deseja iniciar e finalizar a coleta das informações. Assim, podem ser geradas informações de inicialização da JPC, e/ou informações de programas específicos sendo executados dentro da máquina virtual.

Na figura 30, pode ser visto o menu incluído na aplicação onde o usuário inicia, e finaliza a geração do *log* ou até mesmo visualizá-lo. Para incluir esta funcionalidade, foi modificada a classe `JPCAplication` incluindo o menu *log* e seus respectivos sub-menus (*start*, *stop* e *view*) juntamente com um evento associado a cada sub-item que efetua a seguinte tarefa ao ser executado:

a) *start*: ativa um indicador que salva as informações a cada acesso efetuado na memória;
b) *stop*: desativa um indicador onde não salva mais as informações a cada acesso efetuado na memória;
c) *view*: abre uma janela de visualização das informações coletadas.

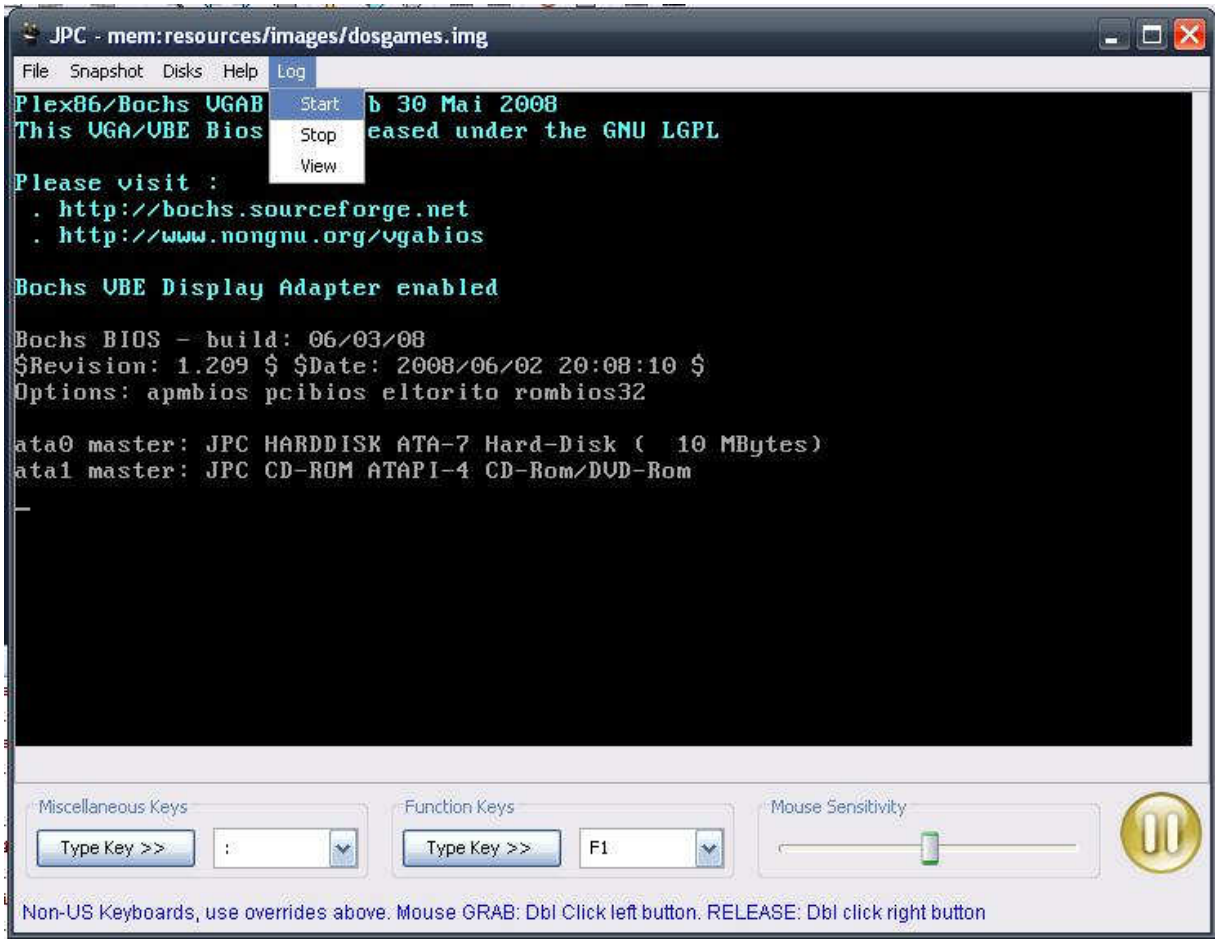


Figura 30 – Menu de controle de informações de *log*

4.3.6 Funcionalidade de implementação

Nesta seção apresenta-se a tela principal do visualizar mostrando seus processos com suas respectivas configurações, a tela para visualização de uma listagem e a tela de visualização do gráfico dos dados coletados.

4.3.6.1 Tela de visualização de processos

A tela `ProcessViwer` contém as seguintes características (figura 31):

a) um botão para selecionar o arquivo de *log* que se deseja efetuar a visualização;
b) um botão para efetuar a geração e visualização dos gráficos a partir do arquivo selecionado;
c) um botão para visualizar em forma de lista os endereços acessados e seus

respectivos tempos a partir do arquivo selecionado;

d) um botão para efetuar configurações para tratamento durante a geração dos gráficos que abre uma tela conforme mostrado na figura 27;
e) um botão para sair da aplicação;
f) um visualizador para mostrar passo a passo cada processo executado pela aplicação a cada intervenção que o usuário efetua com o visualizador;
g) uma área disponível para acoplar os gráficos após gerados.

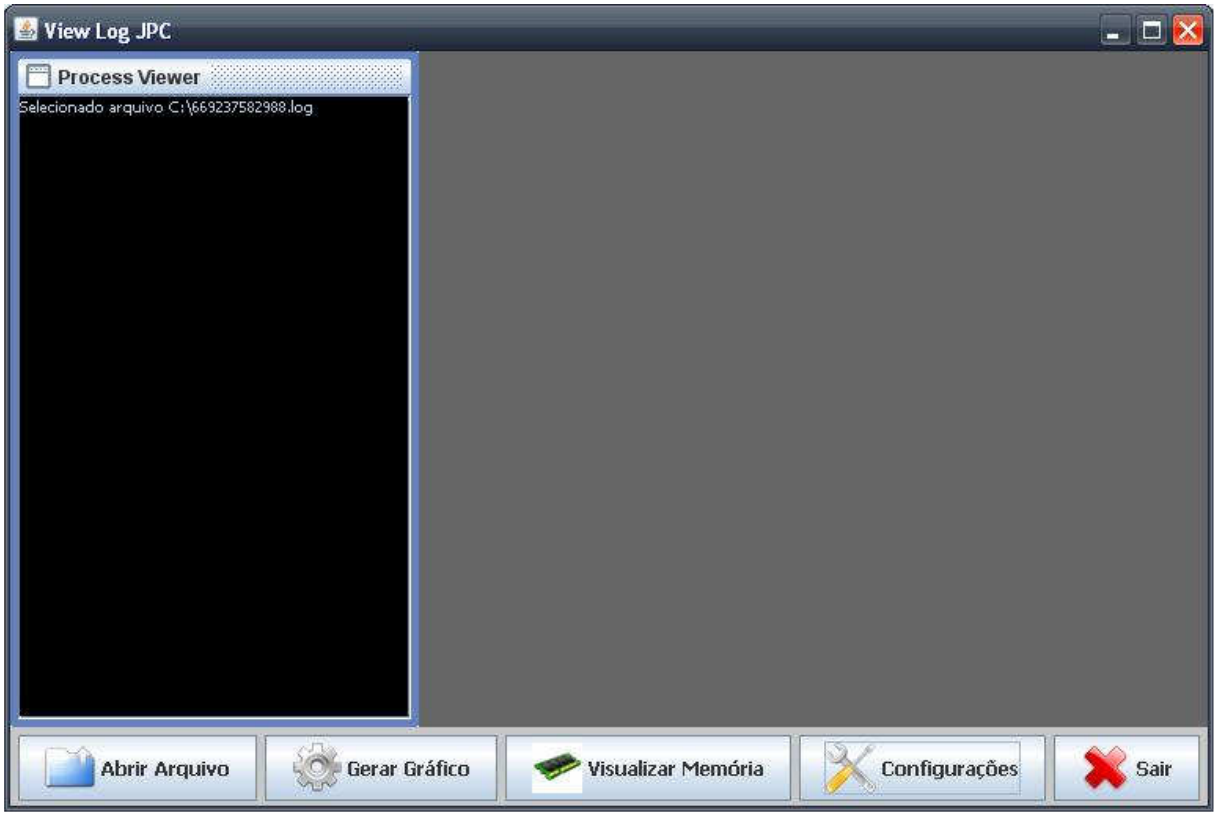


Figura 31 – Tela principal de visualização das informações de *log*

4.3.6.2 Tela de visualização em lista do arquivo de *log*

A figura 32 apresenta o visualizador que mostra, para cada tempo, o endereço acessado. Isto contribui para saber se a contagem total de endereços analisados (requisito do trabalho). Outra utilidade é efetuar uma visualização do gráfico e identificar na lista o momento exato da posição acessada em determinado tempo de determinados pontos.

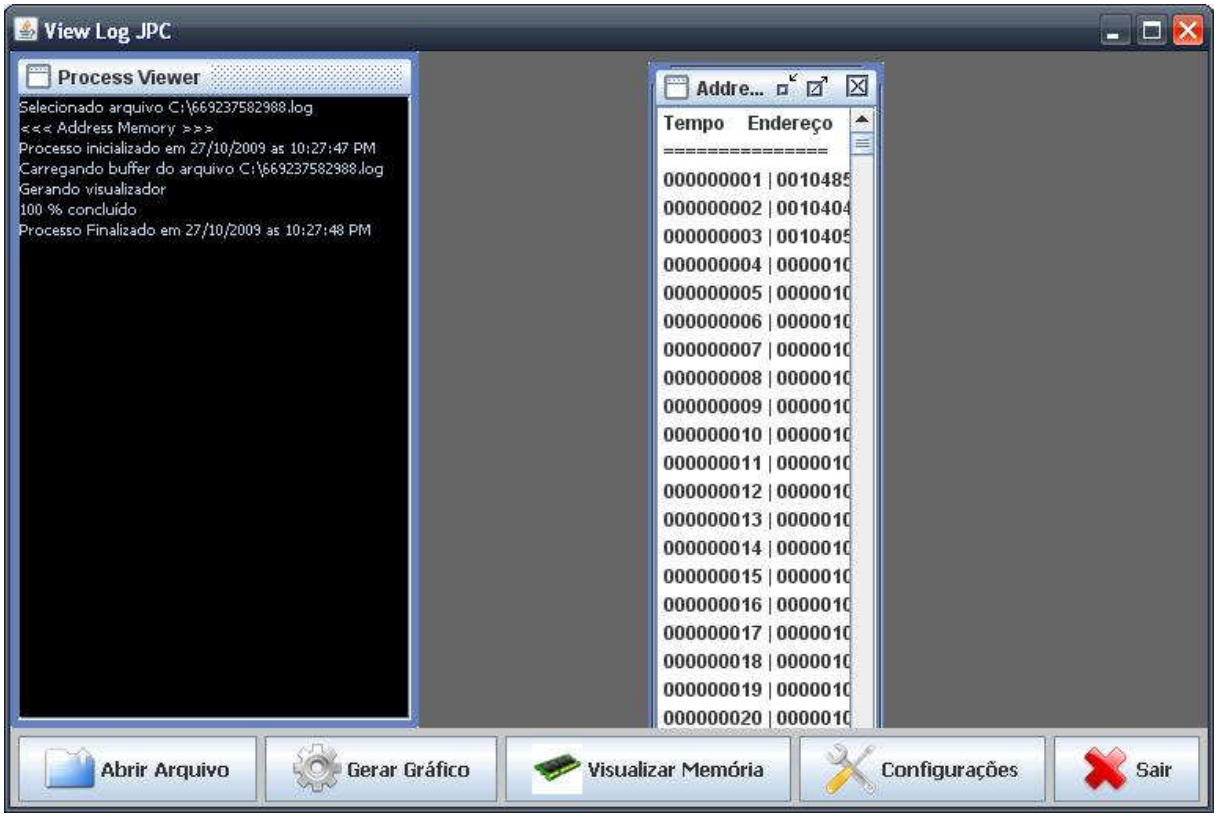


Figura 32 – Tela de visualização das informações de *log* em modo listagem

4.3.6.3 Tela de visualização em gráfico do arquivo de *log*

A figura 33 apresenta na janela `ProcessViewer` cada etapa interna processada quando é selecionado o botão gerar gráfico. Selecionando-se um arquivo de *log*, com o botão abrir arquivo, onde contém as informações de acessos a endereços da memória acessados em determinado momento e executa-se a geração do gráfico através do botão gerar gráfico apresentado a direita e visualizado em forma de linhas (onde cada ponto é interligado ao seu próximo). Este gráfico possui no eixo y a posição de memória acessada e no eixo x o tempo de cada acesso. Para efetuar um *zoom* de uma determinada área, apenas deve-se selecionar a área para visualização. Para retornar a posição inicial, excute-se um duplo clique em qualquer área do gráfico.

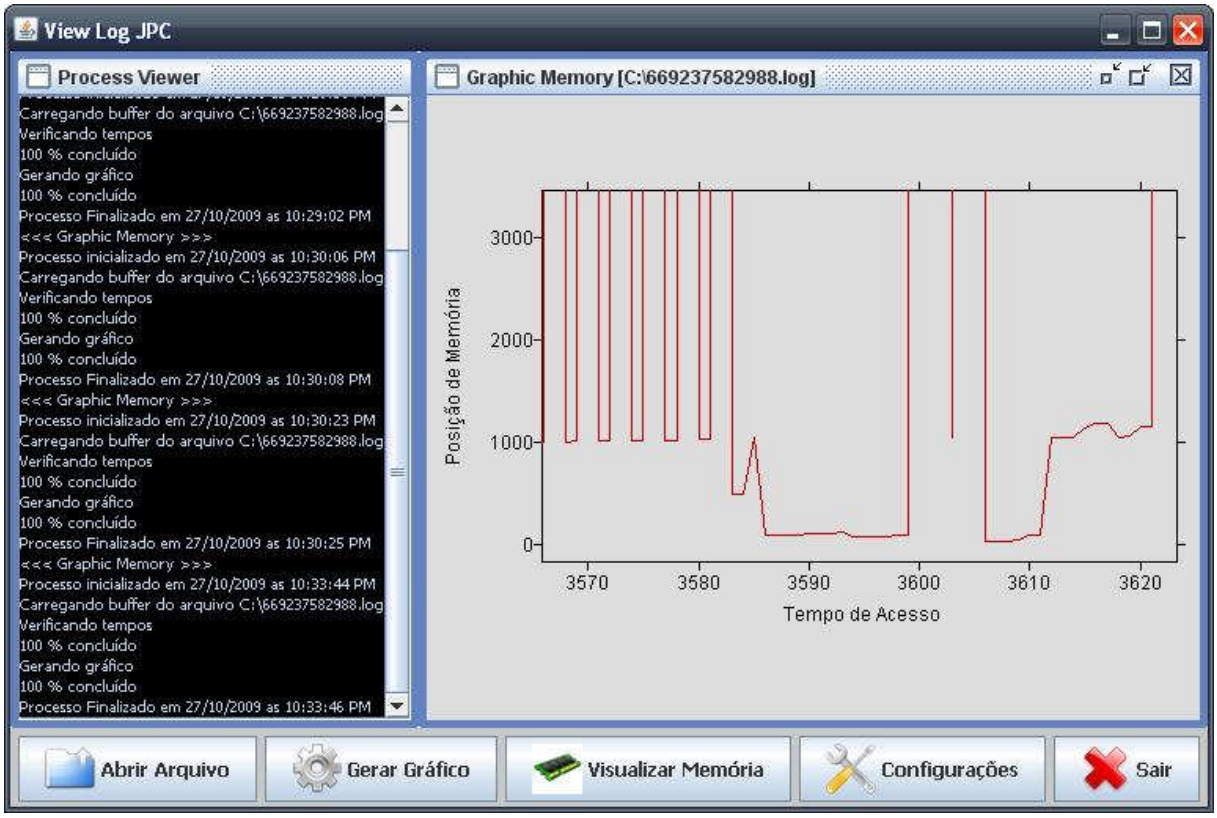


Figura 33 – Tela de visualização das informações de *log* em modo de gráfico por linhas

Conforme apresentado na figura acima, entende-se que no intervalo de tempo de 0 (zero) a 3000 foram acessados os endereços entre 3570 e 3620 cuja seqüência de acessos é o percurso da linha que interliga os pontos do gráfico.

Como em alguns tipos de análise para encontrar padrões de acesso a memória fica difícil na forma apresentada acima, implementou-se uma parametrização (figura 27) onde é possível selecionar a opção de gerar o gráfico em linhas ou em pontos a critério do usuário.

Na figura 34 pode ser visto o mesmo gráfico da figura 33 na forma de pontos. Para os tratamentos de *zoom* aplicam-se as mesmas regras descritas para a forma em linhas. Para este tipo de gráfico, destaca-se a opção de configurar o tamanho para cada ponto desenhado na tela (configuração que pode ser vista na figura 27).

Outra funcionalidade incluída no projeto foi a opção de salvar uma imagem do gráfico gerado, bastando para isso clicar com o botão direito sobre o gráfico e selecionando-se o diretório que se deseja salvar a imagem.

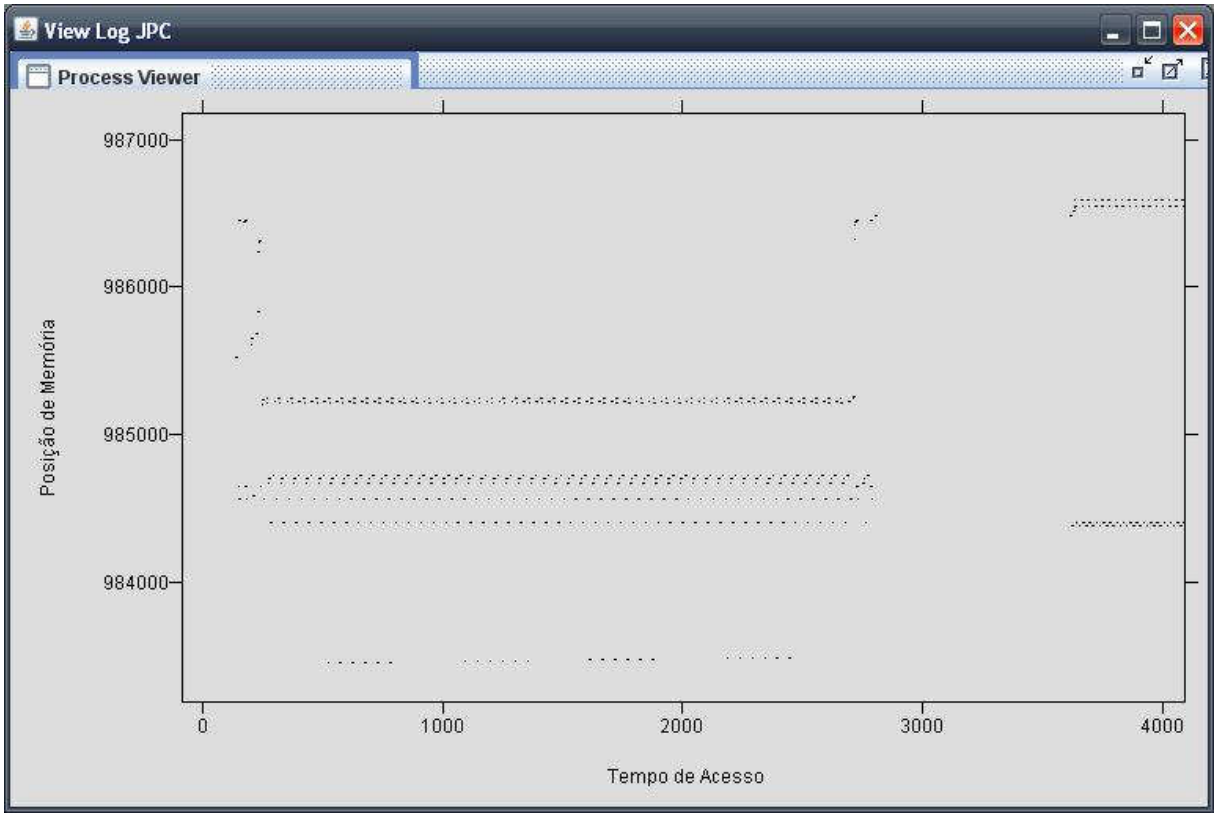


Figura 34 – Tela de visualização das informações de *log* em modo gráfico gerado por pontos

4.4 RESULTADOS E DISCUSSÃO

Foram apresentados alguns trabalhos correlatos. A relação entre eles e o trabalho proposto são:

a) o trabalho de Gustavo foi utilizado durante a fase de apropriação de conhecimento para embasar a compreensão da implementação na JPC;
b) o VXT facilitou o entendimento da implementação da JPC para implementar uma arquitetura 8086;
c) no ATS, o trabalho proposto apresenta características semelhantes como de permitir rodar em qualquer plataforma que contenha a JVM instalada e de explorar endereços de memória contribuindo para fins didáticos de acessos a memória. A diferença para este projeto é que apresenta estes estudos em forma de gráficos e listas e no ATS relação de endereços que podem ser manipulados modificando a tabela de páginas pelo usuário;
d) o Elephantools caracterizou o desenvolvimento da aplicação uma vez que ambos

possuem a mesma finalidade. A diferença é que o Elephantools executa em ambiente real e o sistema desenvolvido executa em ambiente simulado.

5 CONCLUSÕES

O presente trabalho descreveu o desenvolvimento de um módulo de análise de localidade de referência o qual foi acoplado a máquina JPC.

Foram apresentados cinco trabalhos correlatos. O trabalho do Gustavo serviu para o entendimento do funcionamento do processo de gerência de memória. O projeto VXT serviu como base conceitual, uma vez que implementa sua versão simplificada da JPC. O projeto ATS e Midorikawa serviram como apoio ao entendimento de funcionamento da MMU da máquina JPC. Já o trabalho Elephantools contribuiu para estudos sobre padrões de acesso que pode ser encontrado na gerência de memória. Por fim, o trabalho de Cassettari e Midorikawa (2004) contribuiu com idéias de como gerar os gráficos para análise da gerência de memória que será desenvolvido no projeto.

Sendo assim, o maior desafio do trabalho foi entender o código fonte da JPC por não possuir documentação, além de ser complexa e possuir 5 pacotes compostos por mais de 200 classes. Um segundo grande desafio foi entender a funcionalidade do gerenciador de memória que envolve em grande parte do trabalho.

5.1 EXTENSÕES

Este trabalho constitui numa primeira iniciativa de desenvolvimento de uma ferramenta para inspeção didática de usabilidade da JPC.

Para trabalhos futuros, podem ser acrescentados os seguintes complementos:

a) destacar no gráfico gerado os padrões de acesso a memória;
b) implementar a geração e visualização das instruções executadas para cada *log* gerado pela JPC;
c) Destacar os pontos relativos a acessos de leitura e escrita com cores diferentes;
d) implementar a geração de *log* e um contador de endereços que geram *page-fault*.

REFERÊNCIAS BIBLIOGRÁFICAS

ADAMS, K.; AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND SYSTEMS, 12th, 2006, California. **Proceedings…** California: [s.n.], 2006. Paginação irregular.

CASSETTARI, H. H.; MIDORIKAWA, E. T. Caracterização de cargas de trabalho em estudos sobre gerência de memória virtual. In: WORKSHOP EM DESEMPENHO DE SISTEMAS COMPUTACIONAIS E DE COMUNICAÇÃO, 1., 2004, Salvador. **Anais...** Salvador: SBC, 2004. Paginação irregular.

ELMER, F. **JCCkit:** chart contruction kit for the Java plataform. [S.l.], 2005. Disponível em: <http://jcckit.sourceforge.net/>. Acesso em: 10 out. 2009.

_____. New visual characterization graphs for memory system analysis and evaluation. In: WORKSHOP EM DESEMPENHO DE SISTEMAS COMPUTACIONAIS E DE COMUNICAÇÃO, 3., 2006, Campo Grande. **Anais...** Campo Grande: SBC, 2006. Paginação irregular.

MACHADO, F. B.; MAIA, Luiz Paulo. **Arquitetura de sistemas operacionais.** Rio de Janeiro: LTC, 2002.

MAIA, L. P. **Simulador para ensino de sistemas operacionais**, Rio de Janeiro, [2001]**.** Disponível em: <http://www.training.com.br/sosim>. Acesso em: 15 nov. 2009.

MATTOS, M. M. **Linguagens para programação de sistemas.** Blumenau, 2005. Paginação irregular. Notas de Aula. Disciplina Linguagens para Programação de Sistemas, Ciências da Computação da Universidade Regional de Blumenau.

MATTOS, M. M.; TAVARES, A. C.; FARIAS, J. S. VXt: um ambiente didático para ensino de conceitos básicos de sistemas operacionais e arquiteturas de computadores. In: WORKSHOP DE COMPUTAÇÃO DA REGIÃO SUL, 1., 2004, Florianópolis. **Anais...** Florianópolis: UNISUL, 2004. Paginação irregular.

MORITZ, G. **Simulador de mecanismos de gerência de memória real e virtual.** Blumenau, 2004. 81 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

NATIONAL, S. F. **Using the address translation simulator.** [S.l.], 2005. Disponível em: <http://vip.cs.utsa.edu/simulators/guides/addres/addres_doc.html/>. Acesso em: 12 set. 2009.

OLIVEIRA, R. S.; CARISSIMI, A. S.; TOSCANI, S. S. **Sistemas operacionais.** 2. ed. Porto Alegre: Sagra Luzzatto, 2001.

OXFORD UNIVERSITY. **The JPC project**: computer virtualization in Java. Oxford, 2008. Disponível em: <http://www-jpc.physics.ox.ac.uk/intro.html>. Acesso em: 12 set. 2009.

PIANTOLA, R. L.; MIDORIKAWA, E. T. Ajustando o LRU-WAR para uma política de gerência de memória global. In: WORKSHOP DE SISTEMAS OPERACIONAIS, 5., 2008, Belém. **Anais...** Belém: SBC, 2008. p. 1-11.

SANTOS, R. S. **Guia de estruturação e administração do ambiente de cluster e grid.** Brasília, 2006. Disponível em: <http://guialivre.governoeletronico.gov.br/guiaonline/guiacluster/guiacluster.php>. Acesso em: 12 set. 2009.

SCOPE. **Virtualization**: state of the art. [S.l.], 2008. Disponível em: <http://www.scope-alliance.org/pr/SCOPE-Virtualization-StateofTheArt-Version-1.0.pdf >. Acesso em: 12 set. 2009.

SILBERSCHATZ, A.; GALVIN, P.; GAGNE, G. **Sistemas operacionais**: conceitos e aplicações. Rio de Janeiro: Campus, 2004.

TANENBAUM, A. S. **Sistemas operacionais modernos.** 2. ed. São Paulo: Prentice Hall, 2003.