UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

DEPURAÇÃO DO PROCESSO DE INICIALIZAÇÃO DE UM SISTEMA OPERACIONAL

TRABALHO DE GRADUAÇÃO

Bruno Rezende Laranjeira

Santa Maria, RS, Brasil

2011

DEPURAÇÃO DO PROCESSO DE INICIALIZAÇÃO DE UM SISTEMA OPERACIONAL

Por

Bruno Rezende Laranjeira

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção de grau de
Bacharel em Ciência da Computação

Orientadora: Profa. Dra. Patrícia Pitthan de Araújo Barcelos

Trabalho de Graduação 323
Santa Maria, RS, Brasil
2011

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

DEPURAÇÃO DO PROCESSO DE INICIALIZAÇÃO DE SISTEMAS OPERACIONAIS

elaborado por
Bruno Rezende Laranjeira

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Profa. Dra. Patrícia Pitthan de Araújo Barcelos
(Presidente/Orientador)

Prof. Dr. Benhur de Oliveira Stein (UFSM)

Profa. Dra. Marcia Pasin (UFSM)

Santa Maria, 20 de dezembro de 2011

“Nearly all men can stand adversity, but if you want to test a man’s character, give him power.”

—ABRAHAM LINCOLN

AGRADECIMENTOS

Agradeço, primeiramente, a Deus, por ter me dado uma família como a que tenho.

Agradeço, também, à minha família pelo suporte em todos os momentos e pelo apoio em minhas decisões, certas ou erradas, e por ter feito eu me tornar quem sou hoje.

Aos meus amigos, por terem se mostrado sempre verdadeiros e presentes tanto nos momentos bons quanto nos ruins. Pelos conselhos, muitas vezes furados, mas sempre de boa intenção. Também pelas inúmeras risadas, festas, churrascos, jogos de truco, futebol e de videogame.

Aos colegas de curso, que certamente também já são amigos há um bom tempo, agradeço além de tudo que citei acima, também, pelas várias horas de estudo, trabalhos em cima da hora e alguns desesperos.

A alguns professores e funcionários do curso, pelos ensinamentos passados, por terem se estado disponíveis para ajudar em vários momentos, e por mostrarem que ainda há pessoas que se preocupam em fazer as coisas bem feitas.

RESUMO

Trabalho de Graduação

Curso de Ciência da Computação

Universidade Federal de Santa Maria

DEPURAÇÃO DO PROCESSO DE INICIALIZAÇÃO DE SISTEMAS OPERACIONAIS

Autor: Bruno Rezende Laranjeira
Orientadora: Profa. Dra. Patrícia Pitthan de Araújo Barcelos
Local e data da defesa: Santa Maria, 20 de dezembro de 2011.

Torna-se cada vez mais comum encontrar pessoas que desenvolvem programas de computador. Isso se deve a vários fatores. Entre eles, estão as facilidades oferecidas por linguagens de programação de alto nível e, principalmente, pelos sistemas operacionais modernos e por *software* auxiliar para a programação. O desenvolvimento desses sistemas, no entanto, é bastante complexo, devido ao elevado grau de conhecimento necessário e pela escassez de ferramentas auxiliares para a realização dessa tarefa. Nesse contexto, este trabalho aparece com a intenção de ajudar no desenvolvimento de sistemas operacionais, através de uma ferramenta de depuração para uma etapa essencial para todo e qualquer sistema operacional: a fase de inicialização. A ferramenta foi desenvolvida utilizando-se a linguagem Java e o RSP, um protocolo de comunicação usado pelo depurador GDB, presente na maioria das distribuições Linux, para troca de mensagens com processos remotos.

ABSTRACT

Undergraduate Final Work

Graduation in Computer Science

Federal University of Santa Maria

DEBUGGING THE OPERATING SYSTEMS INITIALIZATION PROCESS

Author: Bruno Rezende Laranjeira
Adviser: Profa. Dra.Patrícia Pitthan de Araújo Barcelos
Date and Local: December 20, 2011, Santa Maria

It is becoming increasingly common to find people who develop computer programs. This is due to several factors. Among them, are the facilities provided by high level programming languages and, mainly, by modern operating systems and programming supporting tools. The development of these systems, however, is quite complex due to the high degree of knowledge required and the lack of supporting tools for this task. In this context, this work comes out intended to assist in the development of operating systems, using a debugging tool to an essential stage for any operating system: the initialization phase. The tool was developed using the Java language and RSP, a communicating protocol used by the GDB debugger, present in most Linux distributions, for exchanging messages with remote processes.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
BIOS	Basic Input Output System
IDE	Integrated Development Environment
MBR	Master Boot Record
POST	Power On Self-Test
RAM	Random Access Memory
RSP	Remote Serial Protocol

SUMÁRIO

1 INTRODUÇÃO

Com o desenvolvimento de linguagens de programação com sintaxes mais próximas da linguagem do ser humano e com inúmeras facilidades providenciadas por sistemas operacionais, o desenvolvimento de *software* torna-se cada vez mais popular. Outro fator que influencia nessa popularização é o surgimento de uma grande quantidade de ferramentas, que auxiliam em alguns aspectos do desenvolvimento, como gerenciamento de controle de versões, refatoração de código-fonte e depuração de processos, que é o foco deste texto.

O desenvolvimento de sistemas operacionais, no entanto, não é englobado por essa popularização, haja vista que é um processo bastante complexo e que carece de muitas dessas ferramentas citadas acima, como um depurador para o processo de inicialização do *kernel*, por exemplo.

Nesse contexto, surgem problemas na implementação desses sistemas, pois o programador sente falta do auxílio de um depurador que permita entender o porquê da ocorrência de algum eventual defeito em seu código-fonte, a fim de que possa elaborar alguma estratégia para solucioná-lo.

Para se elaborar uma metodologia de implementação do depurador que melhor se encaixe às necessidades dos desenvolvedores, deve-se analisar as estruturas de outras ferramentas de depuração existentes na literatura, como os depuradores integrados em ambientes de desenvolvimento, estudando a maneira como eles oferecem suas principais funcionalidades ao usuário. Também é de extrema importância o estudo do processo de *boot* dos sistemas operacionais, já que o objetivo deste trabalho é auxiliar no desenvolvimento de sistemas operacionais através da depuração da etapa de inicialização.

1.1 Contextualização

Ao contrário do desenvolvimento de aplicações comuns, que se mostra cada vez mais massificado, o estudo de sistemas operacionais não aparenta se encontrar em um estado muito popular. Isso se deve, em grande parte, à alta complexidade de um projeto de um sistema operacional e ao elevado grau de conhecimento necessário para o seu desenvolvimento.

Nesse contexto, este trabalho oferece uma maneira para que tanto desenvolvedores quanto estudantes de sistemas operacionais consigam observar um pouco do comportamento do processo de inicialização do sistema operacional *Android*, direcionado para dispositivos móveis, tornando, assim, o estudo da área menos assustador.

1.2 Objetivo do trabalho

O trabalho tem como objetivo auxiliar a tarefa de desenvolvimento de sistemas operacionais por usuários convencionais. O processo de *boot* de um sistema operacional é bastante complexo, sendo de extrema conveniência a existência de um mecanismo que possibilite a sua depuração. A fim de depurar a etapa de inicialização de sistemas operacionais, o foco deste trabalho é a implementação desse mecanismo.

Em uma arquitetura IBM PC, o processo de *boot* é dividido em várias etapas, sendo uma delas, a fase de *bootloader* primário, constituída basicamente pela MBR (Master Boot Record). O *bootloader* primário é chamado após o término do código pertencente à *BIOS* e tem como função o carregamento do *bootloader* secundário, o qual deveráser responsável pelo carregamento da imagem do *kernel* para a RAM.

O trabalho tem como objetivo específico a construção de um depurador de nível de máquina, independente de qualquer IDE e capaz de possibilitar aos desenvolvedores de sistemas operacionais a depuração do estágio de *bootloader* primário, com as funcionalidades de definição e tratamento de *breakpoints* e *watchpoints*. O *software* deverá exibir as informações consideradas relevantes para esse processo em uma interface gráfica e armazená-las em um arquivo de *log*.

O estudo de caso pelo qual se optou foi o sistema operacional *Android*, por representar um sistema de código aberto e relativamente novo. Por essas razões, encontra-se em constante desenvolvimento, necessitando, portanto, do auxílio de ferramentas como a proposta por este trabalho.

1.3 Estrutura da monografia

Este texto está organizado da seguinte maneira: O capítulo 2 descreve toda fundamentação teórica necessária para o desenvolvimento do trabalho, abordando temas como depuradores, sistemas operacionais, assim como o seu processo de inicialização e o RSP (Remote Serial Protocol). No capítulo 3 é descrito o desenvolvimento do projeto, detalhando o modelo conceitual da ferramenta, juntamente com as ferramentas que auxiliaram o trabalho, como o software de virtualização VMware Workstation, por exemplo. Finalmente, o capítulo 4 conclui o trabalho, apresentando os resultados atingidos.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo trata dos conceitos de sistemas operacionais, do processo de *boot*, de processos em geral, máquinas virtuais, depuradores e do RSP. Através da apresentação dessas definições, será possível ter a devida compreensão dos fundamentos utilizados na criação da ferramenta aqui proposta.

2.1 Sistemas operacionais

Este item dedica-se à explicação de conceitos básicos sobre sistemas operacionais, os quais são imprescindíveis para que ocorra um bom entendimento do restante do texto.

De acordo com Tanenbaum (2001), um sistema operacional é um software o qual possui duas funções principais: funcionar como uma máquina estendida e como um gerenciador de recursos.

Funcionar como máquina estendida significa que permite a abstração de vários detalhes dos processos realizados diretamente no hardware. O gerenciamento de recursos é definido como a administração dos recursos de hardware, como, por exemplo, as memórias voláteis e não voláteis, processadores e dispositivos de entrada e saída.

Uma das principais funções delegadas ao sistema operacional é referente à execução e ao controle de processos. É importante que os que possuem capacidade de executar múltiplos processos concorrentemente impeçam que um processo tenha acesso às áreas de memória referentes a processos pertencentes a outros usuários, ou que estejam protegidos por qualquer outro motivo.

Sharma (2011) cita que um processo tem seu espaço de endereçamento dividido em três grandes segmentos, com diferentes finalidades. São eles: o

segmento de código, o de dados e o de pilha. O segmento de dados é composto pelas divisões *heap*, *gvar* e BSS. A *heap* é um setor que não possui tamanho fixo, sendo responsável pelo espaço alocado dinamicamente durante o programa. A região *gvar* é onde ficam armazenadas as variáveis globais e estáticas que foram inicializadas com valores diferentes de zero. A BSS, por sua vez, guarda as variáveis globais e estáticas que tiverem seu valor inicializado explicitamente com o valor zero ou que não forem inicializadas no código-fonte. Quando o processo é iniciado, todas as variáveis contidas na BSS são zeradas. A figura 2-1 ilustra a divisão das áreas de endereçamento de um processo.

O subitem seguinte trata sobre a etapa de inicialização de um processo operacional, chamado de processo de *boot*.

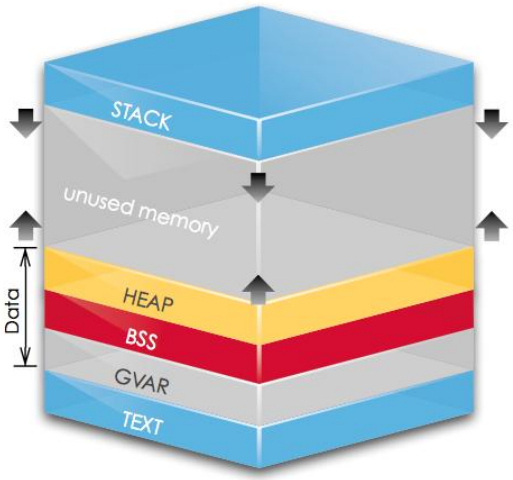


Figura 2-1: Representação dos segmentos de memória de um processo (SHARMA, 2011).

2.1.1 *Boot*

O processo de inicialização de um sistema operacional é chamado de processo de *boot*, nome dado por uma abreviação da palavra em inglês *bootstrap*, tradução para cadarço de bota. De acordo com Quinion (2002), esse nome se deve a uma lenda em que o Barão de Munchausen conseguia se erguer apenas levantando os próprios cadarços. A comparação com o processo de carregar a imagem do *kernel* para RAM é estabelecida porque é um mecanismo onde o sistema operacional se “ergue” com seu próprio esforço. As etapas desse, em uma

arquitetura IBM PC, estão representadas na figura 2-2 e explicadas nos itens seguintes.

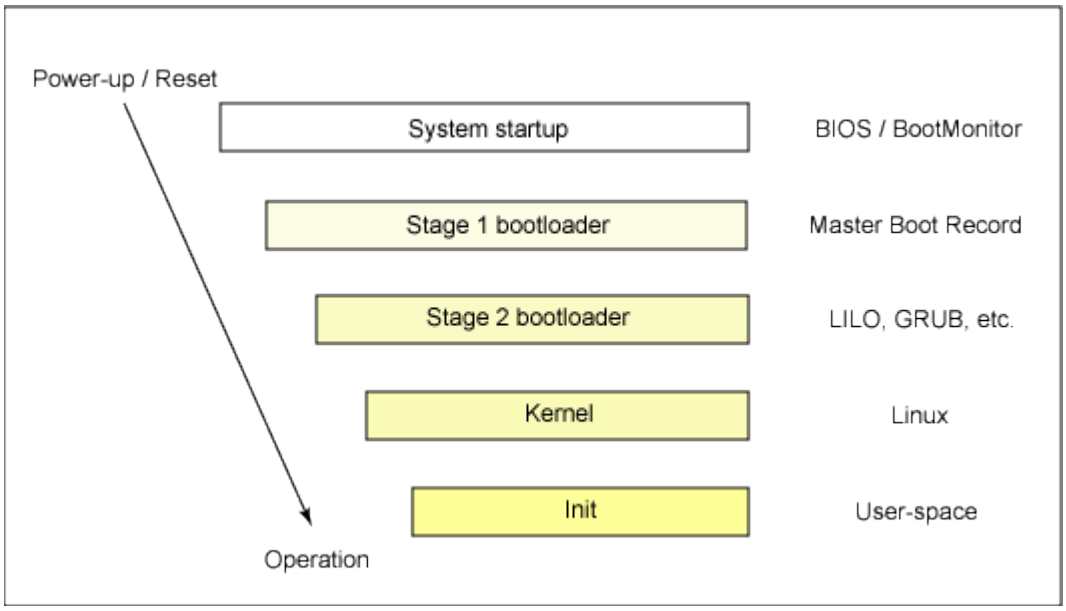


Figura 2-2: Representação dos estágios do processo de *boot*. (JONES, 2006).

2.1.1.1 Carregamento do kernel

Segundo Jones (2006), em um sistema com arquitetura x86, o *startup* é o primeiro estágio do processo de *boot*. É completamente dependente dos circuitos existentes na *BIOS* que são atingidos pelos elétrons no momento em que a máquina é ligada. Sua primeira função é iniciar o POST, que, por sua vez, é responsável pelo teste de todos os componentes de hardware essenciais para o funcionamento do computador. Dependendo do resultado obtido no teste, diferentes combinações de sinais sonoros são emitidas pela placa-mãe. O significado de cada combinação pode ser facilmente encontrado no manual do fabricante.

O próximo papel a ser exercido pelo *startup* é procurar por dispositivos ativáveis e *inicializáveis*, a fim de encontrar e carregar o MBR para a RAM. O MBR é uma seção de 512 bytes, localizada no primeiro setor do disco. No momento em que ele é totalmente carregado, a *BIOS* cede o controle das ações ao MBR. O estudo da MBR é de bastante importância para este trabalho, uma vez que constitui a etapa do processo de inicialização que é depurada.

A figura 2-3 ilustra a forma como os 512 bytes do MBR são divididos. Os primeiros 446 bytes sendo destinados a guardar as instruções que serão executadas. Os próximos 64 compõem quatro partições, de 16 bytes cada uma, que poderão ser utilizadas na sequência do processo e os últimos dois bytes possuem uma constante hexadecimal, 0xAA55, que funciona como se fosse uma assinatura, sinalizando que aquele segmento se trata de um MBR.

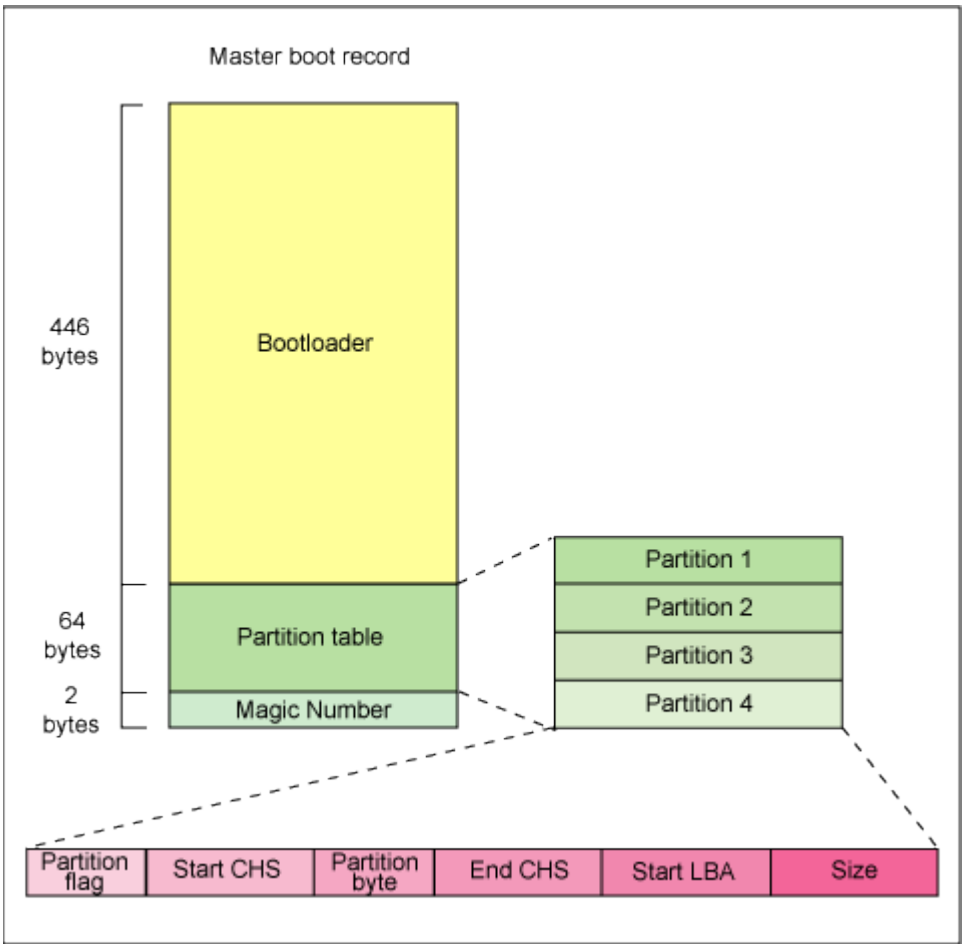


Figura 2-3: Estrutura do MBR (JONES, 2006).

Sedory (2009) explica que cada uma das partições contidas no MBR possui uma estrutura predefinida, como pode ser visto na tabela 2.1. O primeiro byte serve para indicar se a partição é inicializável ou não, contendo o valor hexadecimal 0x80, em caso afirmativo. Os três bytes seguintes tem a função de apontar os números do setor, da trilha e do cilindro correspondentes ao início da partição no disco. O quinto byte indica qual é o sistema de arquivos utilizado na partição, de acordo com uma tabela de valores pré-estabelecida. Os três próximos bytes também remetem a um

endereço no disco, apontando o final da partição. Após, há a informação do setor que inicia a partição, representado de maneira absoluta. Finalmente, os últimos quatro bytes são destinados a informar o tamanho da partição representada.

O MBR também é chamado de *bootloader* primário e sua finalidade é encontrar e carregar o *bootloader* secundário. O *bootloader* secundário possui a função de procurar por imagens disponíveis de *kernel* e executar uma delas, levando ao seguinte estágio do processo de *boot.*

Jones (2006) também explica que, em uma arquitetura x86, o estágio de *bootloader* é combinado em um processo denominado *GRUB* (do inglês *Grand Unified Bootloader*). No *GRUB* é possível realizar o carregamento do *kernel* através de um sistema de arquivos específico, como *ext2* ou *ext3*, pois possui uma etapa que intermedeia os estágios primário e secundário do *bootloader*, que compreende o sistema de arquivos utilizado.

Tabela 2.1: Estrutura de um item da tabela de partições do MBR (SEDORY, 2009).

Relative Offsets (within entry)	Length (bytes)	Contents
0	1	Boot Indicator (80h = active)
1 – 3	3	Starting CHS values
4	1	Partition-type Descriptor
5 – 7	3	Ending CHS values
8 – 11	4	Starting Sector
12 - 15	4	Partition Size (in sectors)

2.1.1.2 Kernel e Init

A etapa do *kernel* inicia quando o *bootloader* secundário finalmente termina de carregar a imagem do *kernel* para a RAM e cede o controle a ela. Jones (2006) explica que a imagem do *kernel* não é apenas um código executável. A imagem consiste em uma combinação de um pequeno trecho de código e uma compactação do que realmente será o *kernel*. O trecho de código é escrito na linguagem C e serve

para realizar algumas configurações de hardware e descompactar a imagem do *kernel*, a fim de que possa ser executada.

Quando sua execução inicia, o *kernel* passa a gerenciar paginações de memória e, finalmente, começa a etapa que independe da arquitetura utilizada pelo sistema. Nesse momento, há a inicialização de vários serviços. Entre eles, encontram-se as habilitações de interrupções, o carregamento do *initrd* (*Initial RAM Disk*) e a execução da função *init*, a qual representa o primeiro processo em nível de usuário. Finalmente, a função *cpu_idle* é invocada e, em sistemas operacionais multiprocessadores, o escalonador passa a controlar a execução dos processos.

O *initrd* carregado servirá como um sistema de arquivos temporário, para que seja possível que o processo de *boot* ocorra sem a necessidade de montar previamente um sistema de arquivos em disco. Segundo Jones (2006), também é possível utilizar o *initrd* como o sistema de arquivos definitivo, em sistemas embarcados que não possuem disco.

2.1.2 Máquina Virtual

De acordo com Tanenbaum (2001), máquinas virtuais são sistemas que permitem a simulação de múltiplos computadores executando concorrentemente no mesmo computador. Para tanto, cada uma das máquinas simuladas cria uma cópia exata do hardware. Dessa forma, as interações de um software que executa dentro da máquina virtual não ocorrem com o sistema operacional nativo, apenas com o que há dentro da máquina virtual, conforme é ilustrado na figura 2-4.

O autor também afirma que, atualmente, as máquinas virtuais são bastante utilizadas para resolver problemas de compatibilidade entre softwares antigos e os sistemas operacionais novos, os quais não oferecem suporte para muitas das aplicações desenvolvidas para outras arquiteturas.

Segundo Laureano (2006), também há máquinas virtuais que são denominadas máquinas virtuais de aplicação. Sua função é apenas executar algum processo, tendo, assim, seu tempo de vida limitado apenas ao momento em que a aplicação é executada. Assim como em qualquer máquina virtual, o sistema operacional nativo não toma conhecimento da aplicação que roda dentro dela,

tratando-a como um processo qualquer. Um bom exemplo de máquina virtual de aplicação é a JVM (Java Virtual Machine), utilizada para executar programas escritos na linguagem de programação Java em qualquer sistema operacional.

Figura 2-4: Representação das camadas de *software* das máquinas virtuais.

2.2 Depuradores

Além dos conceitos sobre sistemas operacionais, tratados no item 2.1, a compreensão de teorias referentes a depuradores também são de fundamental importância para o entendimento do texto. Essa seção trata sobre conceitos de depuradores e de processo de depuração.

O desenvolvimento de novas tecnologias ou de novos sistemas está bastante propenso a erros de vários tipos. Muitas vezes a diferença entre o que acontece e o que se espera que ocorra possui uma causa relativamente difícil de ser encontrada. O processo de depuração consiste na busca por esses motivos. Grötker, Holtmann *et al* (2008) definem treze regras para serem utilizadas no desenvolvimento de um software, as quais devem ser seguidas durante o seu processo de depuração. Entre

elas, podemos destacar algumas. Forçar a presença das condições necessárias para a ocorrência dos bugs, a fim de entender seu comportamento, leitura a priori da documentação da tecnologia utilizada, para saber o que esperar de cada ação realizada e a divisão de um problema em partes.

Quanto maior for a complexidade do software desenvolvido, maior é a necessidade de se utilizar de técnicas e ferramentas auxiliares ao desenvolvimento. Entre essas ferramentas estão, por exemplo, IDEs e controladores de versões, que mostram-se bastante eficientes. Os depuradores também entram nesse contexto, ajudando na detecção de falhas, as quais poderiam ser bastante difíceis de serem encontradas de outra maneira.

Rosenberg (1996) descreve a utilização de ferramentas de depuração. Primeiramente, são utilizadas pela equipe de desenvolvimento, ao criarem um sistema novo. Quando esse sistema passa por alguma mudança, o depurador revela-se útil novamente, tanto para ajudar na compreensão do funcionamento do programa original quanto para identificar os erros presentes nas modificações. Durante a etapa de testes do software também é conveniente o uso de depuradores.

Para o desenvolvimento de qualquer depurador, é necessário que se tenha conhecimento de alguns conceitos considerados básicos. Um deles é a existência de um tipo de *bug* que ocorre a partir da utilização de um depurador. Eles são chamados de Heisenbugs, pois remetem ao Princípio da Incerteza de Heisenberg, enunciado por Heisenberg (1930), que afirma que não se pode mensurar com precisão a velocidade nem o momento de nenhuma partícula, porque durante a medição de tais valores, eles são inevitavelmente alterados.

Os Heisenbugs podem modificar seu comportamento, ou até mesmo surgir ou desaparecer, quando o processo é depurado. Por esse motivo, esses *bugs* possuem causas e efeitos importantes para qualquer processo de depuração.

Grötker, Holtmann *et al* (2008), por exemplo, citam um exemplo de ocorrência desse tipo de *bug* que pode ser percebido simplesmente ao executar um depurador. Apenas a presença dele na RAM provoca a necessidade de que seja gerenciado pelo escalonador do sistema operacional, implicando uma mudança no comportamento do processo depurado, pois, provavelmente, ocorrerá uma redução

no tempo que ele terá os recursos computacionais à sua disposição. Dessa maneira, vários erros podem ocorrer principalmente em sistemas *multithread*, haja vista que a sincronização entre as *threads*, em alguns casos, funciona apenas pela ordem casual em que os eventos independentes costumam ocorrer. Se, por interferência do depurador, essa ordem for modificada, pode haver problemas relacionados a condições de corrida.

Os autores também expõem outro tipo de problema que um depurador pode ter que enfrentar. Ele está ligado à otimização de código realizada pelo compilador. Em muitos casos, o código gerado não reproduz o código-fonte de maneira completamente fiel. Assim, impossibilita que o depurador realize um mapeamento correto entre as linhas do código-fonte e as instruções de máquinas carregadas na RAM. Consequentemente, quando a ferramenta de depuração age, há uma grande chance de estar realizando ações que não deseja, pois existe a possibilidade de estar lidando com trechos errados do código, ou até mesmo acessando regiões não pertencentes ao processo.

A subseção seguinte cita algumas das principais funcionalidades oferecidas pela maioria dos depuradores disponíveis atualmente na literatura. São explicadas as maneiras como geralmente são implementadas e como devem ser utilizadas.

2.2.1 Principais Funcionalidades de um depurador

Há algumas funcionalidades que são oferecidas, para os desenvolvedores, pela maioria dos depuradores disponíveis atualmente na literatura. O modo como devem ser utilizadas e como geralmente são implementadas estão explicados nas seções que seguem.

2.2.1.1 Detecção de falhas

Rosenberg (1996) afirma que na execução de um processo qualquer, o processo de detecção de falhas é essencial para que ocorra o tratamento adequado para a realização de certas instruções as quais não deveriam ser executadas, como divisões por zero, acessos a endereços pertencentes a outros processos ou operações ilegais ou desconhecidas pelo processador.

O autor também explica que durante o processo de depuração, é importante que, quando o sistema operacional detectar a ocorrência de uma falha durante a execução do processo depurado, o tratamento da interrupção seja feito diretamente pelo depurador. Isso possibilita que o programador que o utiliza consiga tratar falhas inesperadas e usufruir as vantagens de outras funcionalidades da ferramenta como, por exemplo, a definição de *breakpoints*, conforme será explicado mais adiante.

Na maioria dos casos, quando uma falha é detectada, o depurador passa a controlar a execução do processo depurado, para que o usuário saiba por que e onde a execução do programa foi interrompida.

Para isso, alguns depuradores enviam, ao sistema operacional, um sinal, indicando que será o responsável pelo tratamento das falhas as quais acontecerem no processo depurado.

2.2.1.2 Criação de breakpoints

Paxson (1990) observa que durante a utilização de uma ferramenta de depuração, muitas vezes, pode ser conveniente que o desenvolvedor observe as informações contidas no bloco de controle de processo em momentos específicos que ele julgar serem importantes. Essas informações podem ser valor das variáveis, pilha de chamada de funções ou valor contido nos registradores.

Para isso, praticamente todos os depuradores oferecem a funcionalidade de definição de *breakpoints*. O programador sinaliza a linha do código-fonte na qual deseja que a execução do processo depurado seja interrompida. Então, antes da instrução selecionada ser executada, o processo depurado cede o controle do processador ao depurador, para que ele possa mostrar suas informações relevantes ao usuário, o qual, por sua vez, tem o poder de fazer o processo depurado retomar sua execução normalmente, quando achar conveniente.

Em algumas ferramentas, há também a possibilidade de anexar uma expressão booleana à instrução correspondente ao breakpoint. O programa depurado continua se comportando da mesma forma descrita acima. A diferença está na maneira com que o depurador trata o evento. Antes de sinalizar ao usuário sobre a interrupção, a expressão é interpretada e avaliada, utilizando valores contidos no contexto de execução do processo depurado.

A fim de oferecer essa funcionalidade, o autor afirma que o depurador substitui a instrução de máquina correspondente à linha de código selecionada por uma instrução especial, que é detectada como *breakpoint*. Para que seja dada sequência ao andamento do processo depurado, há duas possibilidades: o depurador pode ou simular a execução da instrução original ou repô-la no lugar da instrução de *breakpoint*.

2.2.1.3 Criação de watchpoints

Os *watchpoints* constituem uma funcionalidade bastante útil em depuradores, principalmente quando o desenvolvedor não tem ideia de onde pode estar localizada a raiz da falha que ocorre em seu software.

Em alguns casos, tudo o que se sabe é que, em determinado momento do processo, alguma variável apresenta um valor indesejado e, portanto, o software se comporta de maneira inesperada. No entanto, o momento em que o defeito se manifesta não está necessariamente próximo ao momento em que a causa dele ocorre.

Para ocasiões como essa, a utilização de *watchpoints* mostra-se bastante eficiente no processo de depuração. Cada vez que o valor de determinada variável, a ser determinada pelo programador, é modificado, o depurador passa a ter controle sobre o processo depurado e informa ao usuário que uma modificação ocorreu, exibindo os valores antigo e novo.

Um ponto de observação pode ser construído de uma forma relativamente simples, mas totalmente dependente da arquitetura do sistema operacional. A partir de uma análise da tabela de símbolos do programa, determinam-se quais as posições de memória correspondentes à variável. Após isso, mudam-se as permissões de acesso a esses endereços, a fim de proibir determinados acessos a eles. Assim, quando alguma instrução do processo depurado realizar uma tentativa de acesso a algum desses endereços, o processo depurado gera uma *trap* e o depurador toma controle do processador para realizar o tratamento da interrupção.

Há uma maneira alternativa e ainda mais simplificada para que essa funcionalidade seja implementada. Nessa abordagem, utilizam-se alguns registradores específicos para isso. O valor de cada um deles deve ser definido com

o endereço da variável a ser observada. Porém, infelizmente, ela é bastante limitada, pois além de depender bastante da arquitetura do processador, ela restringe o número de pontos de observação. A restrição do número de variáveis observadas se deve pela quantidade de registradores designados para essa função que forem oferecidos pela arquitetura do processador utilizado.

Há três formas diferentes nas quais os *watchpoints* podem ser oferecidos, dependendo dos tipos de acesso que gerarão *traps*. Os *watchpoints* de leitura geram interrupções quando ocorre uma tentativa de leitura no endereço especificado. Os de escrita cedem o controle do processo depurado ao depurador quando acontece uma operação de escrita nesse endereço. Finalmente, os *watchpoints* de acesso são ativados quando ocorre qualquer tipo de acesso na posição de memoria especificada.

2.2.1.4 Visualização passo-a-passo

Outra funcionalidade bastante útil para a depuração de processos é a execução de instruções passo-a-passo. Em qualquer momento em que o usuário, através do depurador, tenha controle sobre a execução do processo depurado, é possível que ele permita que seja executada apenas uma instrução e, após isso, o depurador ganha controle sobre o processo depurado novamente.

Paxson (1990) explica que essa operação pode ser oferecida de duas maneiras distintas. Uma delas, chamada de passo-a-passo em nível de máquina, é completamente dependente da arquitetura do processador utilizado. Seu mecanismo é simplificado, bastando apenas definir o valor de um registrador, que sinaliza ao processador que, após o término de cada instrução, o controle do processo deve voltar a ser do depurador. A outra maneira de implementar essa funcionalidade é simular a inserção de sucessivos *breakpoints* ao longo das linhas do código fonte.

2.2.1.5 Visualização do contexto do processo

Segundo Rosenberg (1996), no momento em que o depurador possui controle sobre o processo depurado, é importante que o usuário tenha acesso a algumas informações, que constituem o contexto do processo. Entre elas, podem-se destacar duas como principais: as informações contidas nas variáveis e a pilha de chamada de funções.

A visualização das variáveis revela-se de bastante importância, pois a partir dela, o utilizador do depurador tem a possibilidade de identificar valores que se encontram diferentes do que deveriam estar, em determinado ponto do programa e, assim, determinar mais facilmente o ponto onde se encontra o erro na lógica do programa.

Ter acesso às informações da pilha de chamada de funções também pode ser muito favorável à tarefa de encontrar as causas de algum comportamento inesperado do software depurado. A partir dessa ajuda, o desenvolvedor pode saber de que modo o programa atingiu o ponto do código em que se encontra. Haja vista que um método pode ser invocado em diferentes trechos do código de um programa, muitas vezes é possível cometer um equívoco sobre onde qual trecho do programa realizou uma chamada à função que executa no momento da preempção.

Assim, essa funcionalidade pode ajudar a mostrar erros no fluxo de instruções do programa, quando, por exemplo, percebe-se que aconteceu algo que nunca deveria ter ocorrido. Outra utilidade da visualização da pilha de chamada de funções é perceber que há algum erro no estabelecimento do critério de parada de alguma função recursiva, como a busca ou a inserção de um nó em alguma estrutura de dados em forma de árvore, por exemplo. Nesse caso, a pilha poderá exibir inúmeras vezes a mesma função. A figura 2-5 mostra um exemplo desse tipo de erro em um código escrito na linguagem de programação Java, na IDE NetBeans. Na execução do código exibido pela figura, pode-se perceber claramente que ocorreu um erro do tipo StackOverflowError, que, indica que a região de memória reservada para o empilhamento de chamada de funções do processo não foi suficiente para as inúmeras invocações sucessivas do mesmo método.

Durante a utilização de alguns tipos de depuradores, geralmente apropriados para linguagens de baixo nível (como *Assembly*) é essencial que o usuário saiba o valor contido em alguns registradores. Conforme explicado na seção 2.2.2.1 deste texto, isso é oferecido por um tipo específico de depuradores: os de nível de máquina. Os depuradores simbólicos, os quais serão tratados na seção 2.2.2.2, também possibilitam a visualização de conteúdos de variáveis e pilha de chamada de funções.

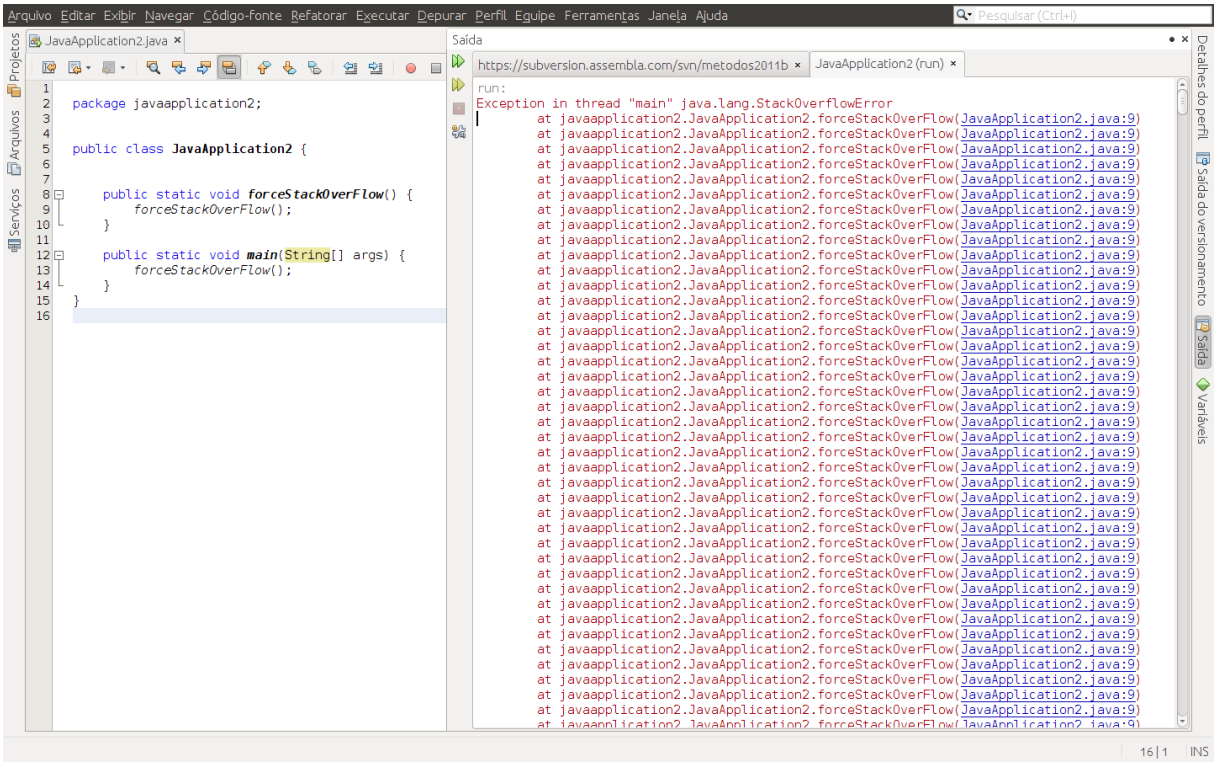


Figura 2-5: Depurador para a linguagem Java da IDE NetBeans.

2.2.2 Tipos de depuradores

Existem diferentes tipos de softwares de depuração presentes na literatura, dependendo das suas características. Rosenberg (1996) diferencia-os de acordo com os subitens a seguir.

2.2.2.1 Depuradores de nível de máquina

Os depuradores de nível de máquina são específicos para utilização em projetos desenvolvidos em linguagens de programação de baixo nível. Apresentam, ao programador, informações como o valor contido nos registradores e nas posições de memória pertencentes ao programa. Não realiza as abstrações feitas por um depurador simbólico, conforme visto no item seguinte.

2.2.2.2 Depuradores simbólicos

Os depuradores simbólicos são úteis principalmente para auxiliar no desenvolvimento de projetos que utilizam linguagens de programação de alto nível. Sua vantagem é que permitem ao usuário abstrair vários processos executados pelo processador. É como se cada linha do código-fonte correspondesse a apenas uma instrução de máquina e as variáveis e funções utilizadas fossem realmente tratadas

pelo processador com o nome que o usuário enxerga. É mais complexo do que um depurador de nível de máquina, pois para oferecer essas funcionalidades, ele precisa, também, realizar os procedimentos feitos pelo outro tipo.

2.2.2.3 Depuradores isolados

Alguns depuradores são construídos de maneira independente do ambiente de desenvolvimento utilizado para a construção do programa. Esses constituem a classe dos depuradores isolados. Possuem a desvantagem de não terem acesso a algumas informações geradas pelo compilador, a fim de oferecerem informações mais precisas e detalhadas sobre o processo depurado.

2.2.2.4 Depuradores integrados

Os depuradores capazes de proporcionar aos usuários mais informações precisam ter acesso a dados adicionais gerados por outros processos, como o compilador. Esses são chamados de depuradores integrados, pois estão junto com as outras ferramentas presentes em um ambiente de desenvolvimento, como gerenciadores de versão e refatoradores de código-fonte.

2.2.3 O estado da arte

Para se obter uma definição aproximada do que atualmente constitui o estado da arte em depuradores, observaram-se alguns depuradores bastante usados no desenvolvimento de aplicações. As ferramentas analisadas foram o Firebug, depurador para a linguagem de programação interpretada Javascript, o depurador para a linguagem Java, integrado na IDE NetBeans e o depurador para C++, que acompanha o ambiente de desenvolvimento Microsoft Visual Studio.

Percebeu-se que assim como o recurso para visualização dos valores das variáveis e da pilha de chamada de funções do processo, as funcionalidades de *breakpoint* e passo-a-passo de nível simbólico são oferecidas em todos os softwares observados, fazendo parte da definição de estado da arte.

Também foi procurado o recurso de *profiling* nos depuradores. Esse recurso permite a realização de uma análise de desempenho nos programas observados, a partir de informações sobre tempo de execução e frequência de uso de cada método do programa. Foi observado que os três depuradores oferecem essa funcionalidade.

Os ambientes em que os depuradores são oferecidos mostraram-se distintos. O Firebug, o qual tem a finalidade de depurar sistemas desenvolvidos em uma linguagem interpretada, é integrado ao navegador, ambiente de execução do programa. Os depuradores para as linguagens compiladas Java e C++, por sua vez, estão associados aos ambientes de desenvolvimento, que disponibilizam recursos tanto para desenvolver quanto para executar os programas. Também foi constatado que os três depuradores avaliados apresentam uma interface gráfica. A figura 2-6 mostra o Firebug, observando um processo que foi preemptado pela ocorrência de uma instrução de *breakpoint*.

A partir dessas observações, concluiu-se que o conceito de estado da arte em depuradores pode ser aproximado como um depurador simbólico, com suporte a funcionalidades de *breakpoints*, passo-a-passo, visualização do contexto do processo e *profiling*, que apresentam uma interface gráfica e podem estar integrados ao ambiente de desenvolvimento ou não.

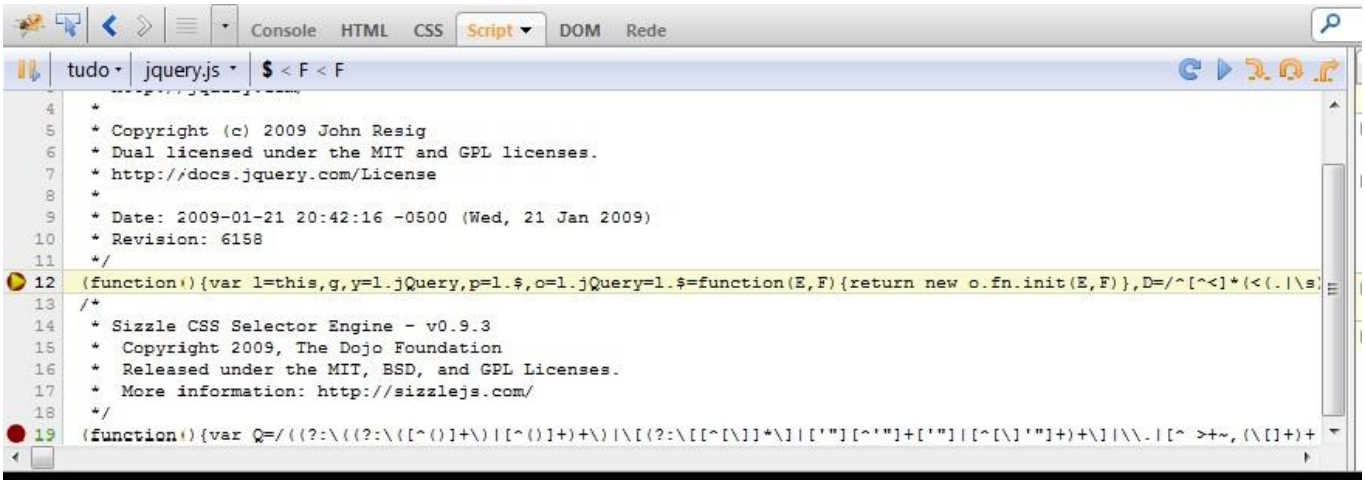


Figura 2-6: Captura de tela do Firebug.

2.3 RSP

Conforme explicado por Tan (2002), a depuração de processos em ambientes que possuem pouco suporte de funcionalidades oferecidas por sistemas operacionais, como o ambiente do processo de *boot* de um sistema operacional ou de sistemas embarcados, é bastante complexa. Para isso, a equipe do GDB desenvolveu um meio para depuração de processos remotamente, necessitando,

apenas, da execução de um processo capaz de realizar algumas operações simples juntamente ao processo depurado e de se comunicar com um depurador mais elaborado, executando em algum local livre das restrições impostas pela falta de recursos como paginação de memória e suporte a multiprogramação.

Gatliff (1999) explica que a solução utilizada pela equipe do GDB para esse problema foi a criação do RSP. Este protocolo define um modelo de comunicação entre um processo depurador, chamado de processo cliente, e um pequeno servidor.

O RSP define uma série de modelos para requisições realizadas pelo cliente e as respectivas respostas enviadas pelo servidor. Entre esses modelos, podem-se citar requisições para obter e definir o conteúdo da memória principal do processo depurado em um determinado endereço, para definição e remoção de *breakpoints* e *watchpoints*.

É bastante importante que seja compreendido o modelo de um pacote do RSP.

De acordo com EMBECOSM (2008), os pacotes do RSP possuem uma estrutura bem definida. No início deles, pode haver um sinal ‘+’, indicando o reconhecimento do último pacote recebido, ou um sinal ‘-‘, sinalizando o contrário. Após esse sinal, há o caractere ‘$’, indicando o início dos dados enviados. Há, então, o setor do pacote o qual contém os dados que devem ser interpretados pelo processo receptor. Ao final desse segmento, encontra-se o caractere ‘#’, que possui a finalidade de indicar que os dados do pacote terminaram. Os dois caracteres seguintes tem a função de certificar a integridade do pacote enviado, através do mecanismo de *checksum*. Apenas alguns modelos de requisição exigem que a resposta recebida tenha um caractere inicial indicando se houve o reconhecimento do pacote enviado. A figura 2-7 representa a estrutura de um pacote do RSP.

Em alguns casos, pode-se desejar que a região que armazena os dados contenha alguns dos caracteres ‘$’ e ‘#’, destinados, respectivamente, para marcação de início e final do segmento. Quando isso ocorrer, deve-se aplicar a operação *xor* entre o sinal que se deseja enviar e a constante hexadecimal 0x20 e inserir o caractere ‘}’ antes dele. Essa regra também se aplica para as ocorrências do caractere ‘}’ no segmento de dados do pacote.

Entre os modelos de requisição definidos pelo RSP, encontram-se, por exemplo, os pacotes ‘m’ e ‘M’ responsáveis, respectivamente, por leitura e escrita na memória principal do processo depurado e os pacotes ‘z’ e ‘Z’, com as funções de remoção e definição de *breakpoints* e *watchpoints*. O pacote ‘c’ é enviado pelo cliente, a fim de indicar que o processo deve ser continuado até que encontre algum *breakpoint* ou *watchpoint*, enquanto o ‘s’ sinaliza que o processo depurado deve executar uma única instrução de nível de máquina e retornar o controle de sua execução ao depurador.

A fim de evitar o envio de dados repetidos, o protocolo permite que qualquer caractere tenha sua repetição explicitada quando for seguido pelo caractere ‘*’. O valor na tabela ASCII do caractere que aparece após o asterisco subtraído de 29 indica quantas vezes o sinal será repetido. Esse padrão pode ser usado para caracteres que possuem um número de repetições entre 4 e 97. No entanto, não se podem sinalizar repetições de 6 nem de 7 caracteres, já que estes números, se somados a 28 correspondem, respectivamente aos caracteres ‘#’ e ‘$’, o que atrapalharia uma futura interpretação do pacote.

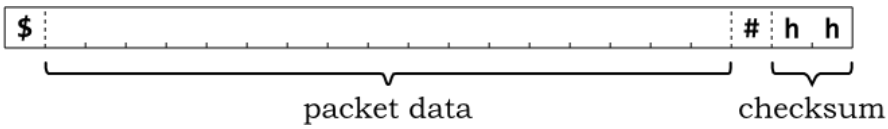


Figura 2-7: Estrutura de um pacote do RSP (EMBECOSM, 2008).

.

3 DESENVOLVIMENTO

Este capítulo esclarecerá todo o processo de desenvolvimento do projeto. Apresentar-se-ão as ferramentas e a metodologia utilizadas durante a realização deste projeto.

3.1 Ferramentas utilizadas

Para que a construção de um depurador se tornasse viável, foram necessários o estudo e a utilização de algumas ferramentas, as quais estão descritas nos itens seguintes.

3.1.1 Ptrace

Padala (2002) afirma que a maioria das distribuições Linux oferece aos usuários um mecanismo para depurar suas aplicações. O meio para a realização dessa tarefa é através de uma chamada de sistema, denominada *ptrace*, abreviação da expressão, em inglês, *process trace*.

A *ptrace* possibilita que um processo tenha total controle sobre a execução dos seus processos-filhos, através de uma chamada PTRACE_ATTACH no processo depurador ou de uma chamada PTRACE_TRACEME, no processo depurado. Apesar dessas duas chamadas terem uma utilidade aparentemente similares, há uma importante diferença entre elas. A PTRACE_ATTACH é usada pelo depurador, para se anexar a um processo que já está sendo executado, enquanto a PTRACE_TRACEME envia um sinal ao sistema operacional, permitindo que outro processo o depure.

Entre as principais chamadas do *ptrace*, encontram-se: PTRACE_PEEKTEXT, PTRACE_POKETEXT, PTRACE_PEEKDATA, PTRACE_POKEDATA, PTRACE_GETREGS e PTRACE_SETREGS. Utilizando-as é

possível examinar e modificar o contexto do processo depurado, pois fornecem meios para examinar e modificar o conteúdo dos segmentos de código e de dados e também dos registradores dele. A chamada PTRACE_CONT sinaliza que o processo depurado deve continuar sua execução normalmente, enquanto a PTRACE_SINGLESTEP permite que ele execute apenas uma instrução de máquina antes de ceder o controle ao depurador novamente.

3.1.2 A ferramenta nm

Nandu310 (2009) afirma que há diversas ferramentas que podem ser utilizadas para verificar em que endereço alguma variável se encontra. Porém, ele destaca a *nm* como sendo uma das mais simples de serem usadas.

O *nm* analisa um arquivo executável e fornece informações sobre os endereços de memória que são definidos em tempo de compilação. Entre os dados informados, encontram-se os locais onde são armazenadas as variáveis globais e estáticas, o início e o final dos segmentos de dados e de código. Também são geradas informações contendo o endereço de cada uma das funções utilizadas e codificadas no programa.

Juntamente com o endereço, a ferramenta comunica qual é o tipo do símbolo analisado, podendo informar que ele referencia um trecho de código, que encontra-se no segmento de dados inicializados, que está no segmento BSS, ou que a ferramenta não foi capaz de interpretá-lo adequadamente. O BSS corresponde à região que armazena todas as variáveis acessíveis em qualquer parte do programa e que foram inicializadas com zero, ou que não tiveram seu valor inicial explicitado no código-fonte, mas que também terão seu valor inicial zerado. A ferramenta informa, ainda, se o símbolo pode ser acessado em algum trecho de código de fora do arquivo no qual está declarado ou não.

3.1.3 VMware Workstation

Conforme será explicado na seção 3.2.3, para o desenvolvimento deste trabalho, foi planejada a utilização de uma máquina virtual para a execução do

processo de *boot* de um sistema operacional, devida a precariedade do ambiente no qual ele é executado.

A ferramenta de virtualização escolhida foi o VMware Wokstation, pois apresenta um meio de comunicação, através do protocolo TCP, entre o processo virtualizado e um depurador, utilizando o RSP. A ferramenta oferece, também, uma opção para que a máquina virtual interrompa sua execução antes da execução da primeira instrução da *BIOS*, possibilitando, assim, que o depurador tenha controle sobre a máquina virtual o tempo todo, o que é uma condição imprescindível para a depuração do processo de inicialização.

Dessa forma, o desenvolvimento de uma ferramenta de depuração remota que se comunique com o processo virtualizado pelo VMware Workstation mostrou-se uma boa opção, uma vez que a implementação de um cliente simplificado do RSP é um projeto viável.

A figura 3-1 mostra um exemplo de uso do VMware Workstation. A ferramenta está sendo executada no sistema operacional Windows e inicializando o *Android*.

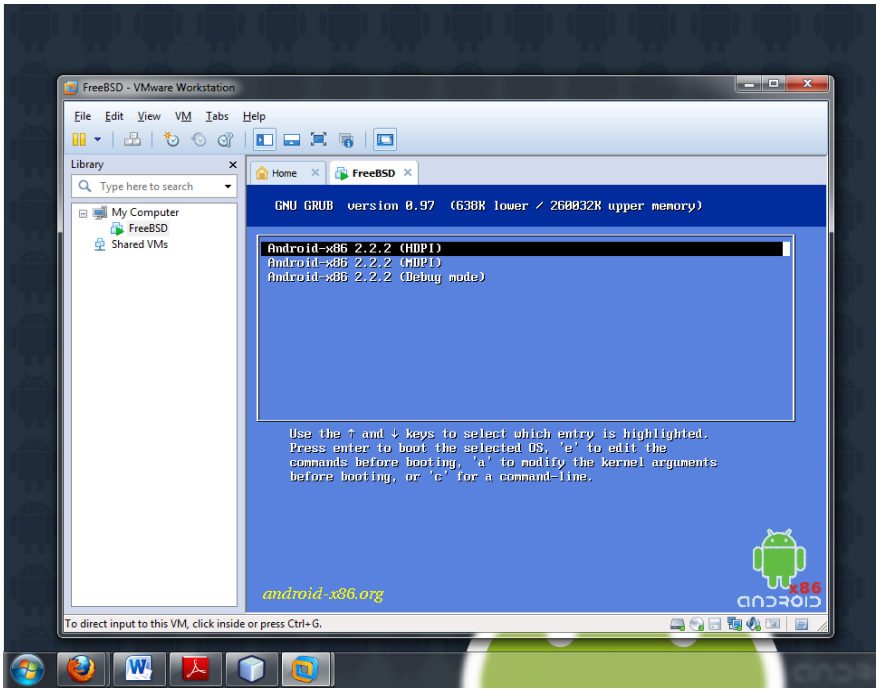


Figura 3-1: Captura de tela durante a inicialização de um sistema operacional em uma máquina virtual.

3.2 Metodologia

Este item dedica-se a apresentar a maneira como ocorreu o processo de desenvolvimento do depurador proposto por este trabalho. Serão descritas a etapa de definição de escopo, as duas abordagens necessárias para a obtenção do resultado final e o cenário de testes utilizado.

3.2.1 Definição do escopo do projeto

Uma etapa bastante importante para a execução do projeto correspondeu a uma análise das necessidades do usuário e da viabilidade da implementação de cada uma delas.

A etapa do processo de *boot* que se optou por depurar foi o *bootloader* primário, porque representa o primeiro estágio da inicialização do sistema operacional.

A partir do estudo de parte da literatura de depuradores, foi feita a decisão de quais das principais funcionalidades de um depurador, explicadas no item 2.2.1 deste texto, eram consideradas de implementação viável.

A presença das funcionalidades de inserção e remoção de *breakpoints* e *watchpoints* são consideradas essenciais para qualquer ferramenta de depuração. Além disso, a implementação desses itens não apresenta uma complexidade que a torne inviável. As funções de detecção de falhas e passo-a-passo, as quais possuem o papel de controle de fluxo de execução do processo, também se mostram bastante importantes.

A principal necessidade identificada foi a inspeção do contexto do processo, que, no caso do *bootloader* primário, corresponde a exibição das informações contidas na MBR. Os principais dados que devem ser visualizados são as instruções contidas na área reservada para código, as informações de cada uma das quatro partições as quais compõem a tabela de partições e o estado dos dois bytes reservados para a assinatura da MBR.

Optou-se, portanto, pela implementação de um depurador com capacidade de controlar o fluxo de execução do processo de *boot* através da manipulação de

breakpoints e *watchpoints*, e das operações de detecção de falhas e passo-a-passo. Foi decidido, também, que o depurador desenvolvido deveria ser de nível de máquina e independente de qualquer IDE utilizada. Para exibição das informações optou-se pela utilização de uma interface gráfica e para armazenamento dessas informações para análises futuras, um arquivo de log.

3.2.2 Abordagem inicial

Inicialmente, o problema de depuração do processo de inicialização de um sistema operacional foi analisado a partir de algoritmos de depuração de processos comuns.

Para o desenvolvimento da ferramenta, foi importante a realização de um modelo conceitual minimamente dependente de questões como arquitetura do processador e particularidades de cada sistema operacional.

Esse modelo foi elaborado utilizando algumas técnicas presentes na literatura e teve o objetivo de guiar a etapa de implementação. Ele está descrito nas subseções que seguem.

3.2.2.1 Laço principal do depurador

Segundo Rosenberg (2006), o comportamento de uma ferramenta de depuração executada em um ambiente com suporte a múltiplas threads pode ser dividido em duas threads com funções distintas. Uma tem o papel de realizar a detecção de erros e de eventos gerados pelo usuário, enquanto a outra se encarrega de tratá-los.

Essa foi a estratégia utilizada para a elaboração da estrutura do laço principal do depurador: inicialmente, deve-se criar um processo filho do depurador o qual executa o processo que se deseja depurar. Após isso, as ações do depurador são divididas nas duas *threads*, sugeridas pelo autor. Uma delas ficará em estado de espera, até que aconteça algum evento no processo depurado. Quando isso ocorrer, ela deverá, através de alguma estrutura de dados compartilhada, notificar a outra *thread,* considerada a principal, da ocorrência do evento, a fim de que ela possa tratá-lo de maneira adequada.

3.2.2.2 Definição e tratamento de breakpoints

A fim de oferecer ao usuário do depurador a funcionalidade de definição de *breakpoints*, planejou-se, para este trabalho, a definição de uma estrutura de dados para possibilitar a sua implementação. A estrutura contém um endereço, especificado pelo usuário, e a instrução de máquina correspondente. Uma lista encadeada deverá ser responsável por armazenar várias dessas estruturas.

Antes de o processo depurado começar sua execução, a lista de estruturas deverá ser inicializada com os devidos ponteiros e as respectivas instruções. Após, para cada uma das entradas da lista, deve-se substituir, no endereço apontado, a instrução original por uma instrução especial para breakpoints, definida pela arquitetura do processador.

Quando o processo depurado é preemptado por ter encontrado uma instrução de *breakpoint*, o depurador passa a ter controle sobre ele. Após o usuário decidir prosseguir a execução do processo depurado, o depurador deverá procurar qual entrada da sua lista de estruturas de *breakpoint* corresponde ao evento que está tratando. Então, a instrução original deverá ser reposta e o valor contido no registrador de *program counter* deverá ser decrementado para o início da instrução. Após isso, a execução do processo é retomada normalmente.

3.2.3 Abordagem implementada

Apesar do sucesso obtido com a abordagem inicial para utilização em processos comuns, havia, ainda, o problema de adaptá-la para que funcionasse em conjunto com uma ferramenta de virtualização. Mesmo não tendo sido completamente implementada, a abordagem inicial foi bastante importante no amadurecimento do projeto, para que se chegasse à abordagem relatada nesta seção.

Tan (2002) cita o problema de depuração de processos executando em ambientes com restrições impostas pela falta de suporte oferecida por um sistema operacional, o que é o caso do processo de *boot* de um sistema operacional, pois funcionalidades como suporte a múltiplas threads, por exemplo, ainda não foram carregadas. Para possibilitar a depuração desses sistemas, uma excelente alternativa é a utilização de uma abordagem de depuração remota, utilizando o RSP.

Para possibilitar a depuração de processos remotos, juntamente com o processo depurado, deve haver um processo capaz de examinar e modificar o seu contexto. A partir disso, percebeu-se a ausência de necessidade de se implementar uma solução para o problema utilizando a abordagem inicial, pois conforme foi explicado no item 3.1.3 o VMware Workstation fornece um meio para controle do processo virtualizado.

As subseções seguintes descrevem a arquitetura e a implementação do sistema desenvolvido.

3.2.3.1 Arquitetura do depurador

Inicialmente, o sistema para depuração remota foi planejado de uma maneira genérica, independente de quaisquer linguagens de programação ou detalhes de implementação de protocolos.

O sistema foi modelado com duas grandes camadas, as quais se comunicam entre si. A camada inferior é representada pelo pacote RemoteSerialProtocol, enquanto o pacote BootDebugger representa a superior, conforme será explicado na seção 3.2.3.2 deste texto. A camada inferior tem as funções de exercer comunicação diretamente com a máquina virtual, codificar e decodificar mensagens para o padrão do RSP, enquanto a superior tem o papel de, através da comunicação com a outra camada, realizar procedimentos específicos para a depuração do processo de *boot* do sistema operacional.

A camada inferior foi dividida em duas subcamadas. Uma delas é responsável pela troca direta de mensagens com o canal de comunicação da VMware Workstation, realizando as codificações e decodificações necessárias. A outra camada tem o papel de, através da primeira, enviar os devidos comandos de depuração do RSP para a máquina virtual.

A camada superior é responsável por se comunicar com a camada inferior, a fim de executar procedimentos específicos para a depuração do estágio de *bootloader* primário, como interpretação dos dados contidos na tabela de partições e controle de *breakpoints* e *watchpoints*. É com essa camada que os métodos responsáveis pela interface gráfica deverão se comunicar.

A figura 3-2 ilustra as camadas componentes do depurador, juntamente com as principais classes as quais compõem cada uma das camadas. O arquivo de log gerado possui as mesmas informações exibidas pela interface gráfica durante a utilização do depurador, a fim de possibilitar a realização de análises posteriores.

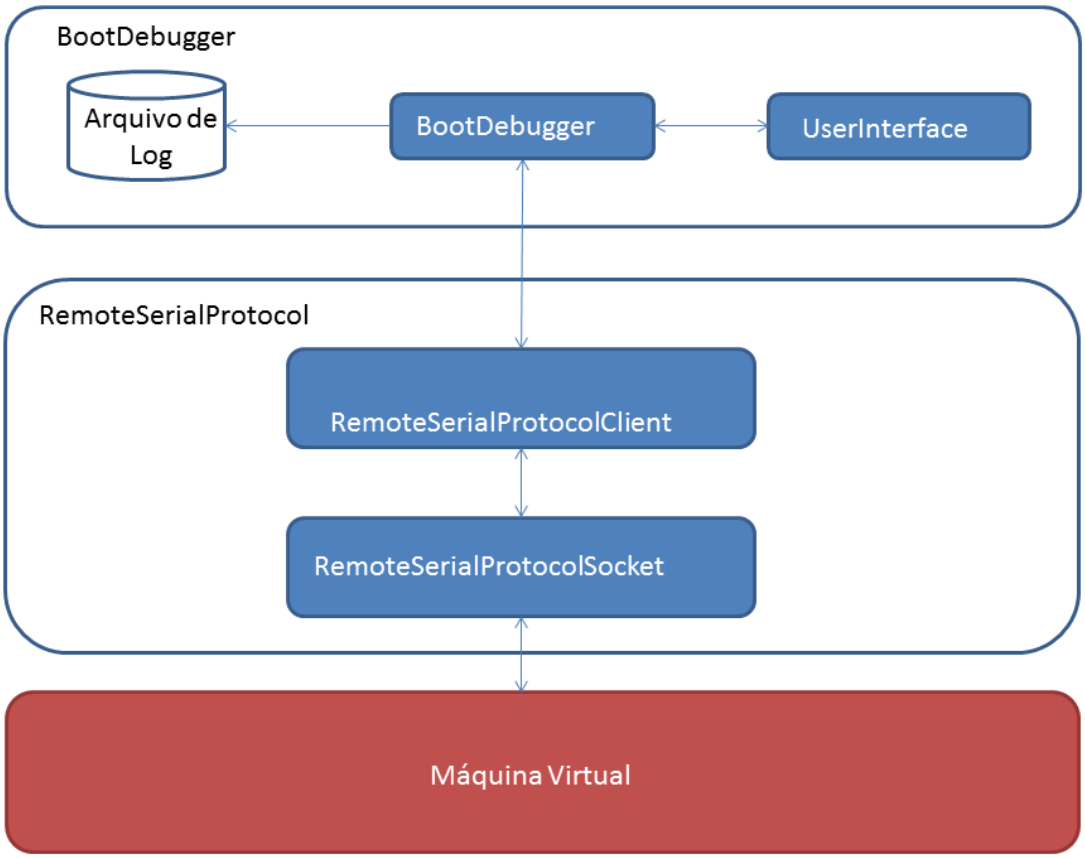


Figura 3-2: Representação das camadas e classes do depurador.

3.2.3.2 Implementação

Esta seção abordará aspectos da implementação do software, detalhando questões como linguagem de programação escolhida e estratégias utilizadas para a realização de procedimentos específicos, como, por exemplo, codificação e decodificação das mensagens para o padrão do RSP.

A linguagem de programação escolhida para implementação desse *software* foi linguagem Java versão 1.7, por apresentar boa biblioteca padrão, facilitando o uso de métodos de comunicação direta com o VMware Workstation através de *sockets*.

As camadas inferior e superior do sistema, citadas no item 3.2.3.1 deste texto, foram implementadas, respectivamente, nos pacotes RemoteSerialProtocol e BootDebugger. Há também, a interação entre as classes BootDebugger e UserInterface, responsável pela exibição das informações em uma interface gráfica e o armazenamento das informações no arquivo de log. A figura 3-3 mostra os passos necessários para comunicação entre o depurador e a máquina virtual, quando o usuário insere um *breakpoint* no endereço 0x7c08.

A classe UserInterface envia uma mensagem à classe BootDebugger, sinalizando que o usuário adicionou o *breakpoint*. A BootDebugger, por sua vez, invoca o método responsável pela definição de *breakpoints* na classe RemoteSerialProtocolClient, a qual envia a mensagem não codificada para a RemoteSerialProtocolSocket, que realiza a codificação e envia a mensagem final para a máquina virtual. Após isso, a mensagem de resposta da máquina virtual é decodificada pela classe RemoteSerialProtocolSocket, passada para a RemoteSerialProtocolClient, a qual interpreta a mensagem como uma resposta positiva e envia um sinal indicando o sucesso da operação para a classe BootDebugger, que envia o mesmo sinal para a UserInterface. Ao final desse procedimento, o usuário é informado que o *breakpoint* foi definido com sucesso.

As seções a seguir explicam a implementação dos pacotes

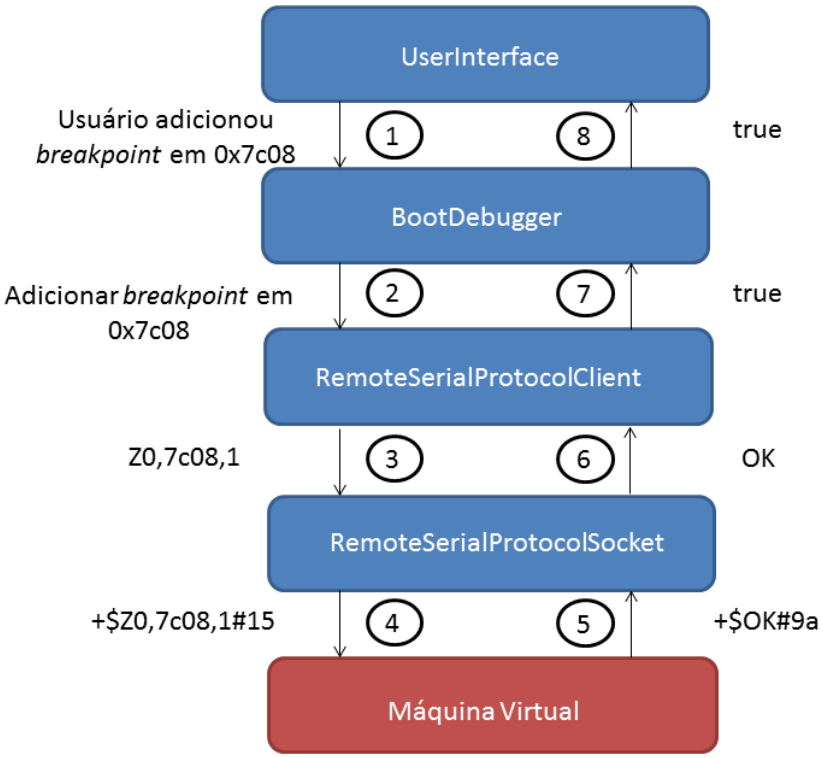


Figura 3-3: Exemplo de comunicação entre as classes do depurador para definição de um breakpoint no endereço 0x7c08.

RemoteSerialProtocol e BootDebugger.

3.2.3.2.1 O pacote RemoteSerialProtocol

Este item dedica-se a detalhar os mecanismos implementados nas classes RemoteSerialProtocolSocket e RemoteSerialProtocolClient, as quais compõem o pacote RemoteSerialProtocol.

A classe RemoteSerialProtocolSocket consiste, basicamente, em um *socket* especializado em trocas de mensagens no padrão do RSP. A classe contém um objeto do tipo java.net.Socket, o qual é responsável por enviar mensagens, que passam, necessariamente, pelo método de codificação, cujo algoritmo está exibido na figura 3-4. O método de codificação cria uma String, concatenando os sinais de reconhecimento (se houver) e de início do segmento de dados, juntamente com o segmento de dados e o sinal de término do segmento de dados e os dois caracteres correspondentes ao *checksum* dos caracteres do segmento de dados aplicando-se a operação de módulo com o valor 256.

```
1   private final static String closeBracketsReplacement = "}[";
2   private final static String sharpReplacement = "}" + (char)3;
3   private final static String sifraoReplacement = "}" + (char) 4;
4
5   private String encodeStr(Boolean ack, String packetData) {
6       String ret = (ack == null) ? "" : (ack) ? "+" : "-";
7
8       packetData = packetData.replace("}", closeBracketsReplacement)
9               .replace("#", sharpReplacement)
10              .replace("$", sifraoReplacement);
11
12      Integer checkSum = 0;
13      for (char currentChar : packetData.toCharArray()) {
14          checkSum += (int)currentChar;
15      }
16      checkSum %= 256;
17      String hexCheckSum = Integer.toHexString(checkSum);
18      if (hexCheckSum.length() == 1) {
19          hexCheckSum = "0" + hexCheckSum;
20      }
21
22      ret += "$" + packetData + "#" + hexCheckSum;
23      return ret;
24  }
```

Figura 3-4: Método de codificação das mensagens no padrão do RSP.

O processo inverso ocorre para realizar o recebimento de mensagens do canal de comunicação da VMware Workstation. Para receber uma mensagem da máquina virtual, há um método bloqueante, o qual permanece em estado de espera até que operação de recepção da mensagem seja concluída.

O pacote RemoteSerialProtocol também conta com a classe RemoteSerialProtocolClient. Esta classe é responsável por efetuar os comandos específicos do RSP, abstraindo-se de quaisquer mecanismos de codificação ou decodificação. A classe oferece métodos para continuar normalmente a execução do processo da máquina virtual, para realizar a operação de passo-a-passo de nível de máquina, manipular *breakpoints* e *watchpoints*, efetuar leitura e escrita na memória principal da máquina virtual e desanexar o depurador da máquina virtual.

Durante a implementação desta classe, encontraram-se grandes dificuldades, haja vista que a documentação localizada sobre os pacotes do RSP está bastante incompleta. Por esse motivo, muitos dos mecanismos programados foram baseados em métodos de tentativa e erro, até que fosse encontrada uma solução funcional.

O algoritmo de tratamento de *breakpoints* e *watchpoints*, realizado através do método público HandleBreak, invoca um método privado de mesmo nome, passando, como parâmetro adicional, uma variável do tipo java.lang.Bolean, indicando sinal de reconhecimento positivo e é exibido na figura 3-5. Nas linhas 21, 23 e 25 da imagem, é possível perceber que o método getStr, da classe RemoteSerialProtocolSocket dispara três tipos de exceções: InvalidPackageException, InvalidCheckSumException e ACKException. Essas são três classes que também compõem o pacote RemoteSerialProtocol, e servem para um objeto da classe RemoteSerialProtocolSocket sinalizar eventuais problemas durante a decodificação. Uma exceção do tipo InvalidPackageException significa que o pacote que se tentou decodificar não obedece aos padrões do RSP. Uma InvalidCheckSumException significa que os dois caracteres fornecidos como *checksum* no pacote não são iguais aos que foram calculados pelo método de decodificação. Quando uma ACKException for disparada, quer dizer que a VMware enviou um pacote com sinal de reconhecimento negativo, sendo necessário o reenvio do último pacote enviado pelo depurador.

```
1  public Boolean handleBreak(String operation, String type, Long address) throws IOException {
2      if ((!operation.equals(setBreak) && !operation.equals(removeBreak))
3              ||  (!type.equals(breakPoint) && !type.equals(writeWatchPoint)
4              && !type.equals(readWatchPoint) && !type.equals(accessWatchPoint))) {
5          return false;
6      }
7      return handleBreak(Boolean.TRUE, operation, type, address);
8  }
9
10 private Boolean handleBreak (Boolean ack, String operation, String type, Long address) throws IOException {
11
12     String request = operation + type + "," + Long.toHexString(address) + ",1";
13
14     rspSocket.sendStr(ack,request);
15     try {
16         String response = rspSocket.getStr();
17         if (response.equals("OK")) {
18             return Boolean.TRUE;
19         }
20         return Boolean.FALSE;
21     } catch (InvalidPackageException ex) {
22         return handleBreak(Boolean.FALSE, operation, type, address);
23     } catch (InvalidCheckSumException ex) {
24         return handleBreak(Boolean.FALSE, operation, type, address);
25     } catch (ACKException ex) {
26         return handleBreak(Boolean.TRUE, operation, type, address);
27     }
28 }
```

Figura 3-5: Método de manipulação de *breakpoints* e *watchpoints*.

As funcionalidades de permitir a retomada de execução normal do processo e passo-a-passo de nível de máquina foram implementadas na classe RemoteSerialProtocolClient através dos métodos continueExecution e singleStep, como pode ser visto no código-fonte exibido na figura 3-6.

```
1   private static final String continueCommand = "c";
2   private static final String singleStepCommand = "s";
3
4   public String continueExecution() throws IOException {
5       return continueExecution(Boolean.TRUE, Boolean.FALSE);
6   }
7
8   public String singleStep() throws IOException {
9       return continueExecution(Boolean.TRUE, Boolean.TRUE);
10  }
11
12  private String continueExecution(Boolean ack, Boolean singleStep) throws IOException {
13      rspSocket.sendStr(ack,(singleStep)? singleStepCommand : continueCommand);
14
15      String pureString = "";
16      while (Boolean.TRUE) {
17          try {
18              pureString += rspSocket.getPureStr();
19              return rspSocket.decodeStr(pureString);
20          } catch (InvalidPackageException keepIterating) {
21          } catch (InvalidCheckSumException ex) {
22              return continueExecution(Boolean.FALSE, singleStep);
23          } catch (ACKException ex) {
24              return continueExecution(Boolean.TRUE, singleStep);
25          }
26      }
27      String operation = (singleStep) ? "single step" : "continue execution";
28      throw new IOException("IO error while performing " + operation + " operation");
29  }
```

Figura 3-6: Métodos para continuar normalmente a execução do processo e passo-a-passo.

3.2.3.2.2 O pacote BootDebugger

Após o término da implementação das classes componentes do pacote RemoteSerialProtocol, ainda era necessária a implementação dos métodos específicos para a depuração do processo de *boot* de um sistema operacional.

A principal classe que realiza estes procedimentos se encontra no pacote BootDebugger e é homônima a ele. As interações dela com a máquina virtual são realizadas através de uma instância de um objeto da classe RemoteSerialProtocolClient.

Entre os métodos implementados nela, podem-se citar os mecanismos para armazenamento das informações geradas em um arquivo de log e a interpretação dos dados contidos na memória principal da máquina virtual. Essa análise permite que os dados das regiões da MBR sejam observados de maneira apropriada pelo usuário.

O método dumpPartitionTable, cujo código pode ser visualizado na figura 3-7, realiza a análise dos dados da MBR, no espaço reservado para a tabela de partições. Sabendo que cada uma das partições tem um endereço fixo para iniciar e que todas elas possuem uma estrutura padronizada para armazenar seus dados, torna-se trivial a implementação de um método para exibir tais informações ao usuário e armazená-las em um arquivo de log.

Na implementação desse procedimento foi utilizado o método estático getPartitionType, pertencente à classe PartitionTypes, criada apenas com esse método, a fim de mapear os tipos de sistemas de partições, codificados em hexadecimal, para seus nomes usuais.

Os métodos de manipulação de *breakpoints* e *watchpoints* contam com o auxílio da classe BreakPoint, utilizada para facilitar o controle deles. Apesar do nome da classe, uma instância dela pode representar tanto um *breakpoint* quanto um *watchpoint* de leitura, escrita ou acesso.

A classe UserInterface possui o encargo de se comunicar com a classe BootDebugger e exibir as informações necessárias na tela. Ela também recebe os eventos gerados pela interação do usuário, como inserção e remoção de

breakpoints e *watchpoints*, assim como os comandos para autorizar a sequência da execução do processo de maneira normal ou passo-a-passo.

```
1  private static final Long partitionTableStart = bootLoaderEnding + 1;
2  private static final Long partitionTableEnding = partitionTableStart + partitionTableSize - 1;
3
4  private void dumpPartitionTable() throws IOException {
5      addToLog("\tPartition Table (from 0x" + Long.toHexString(partitionTableStart) +
6              " to 0x" + Long.toHexString(partitionTableEnding) + ")");
7      String ptDump = client.readMemory(partitionTableStart, partitionTableSize);
8      String[] partitions = new String[4];
9      partitions[0] = ptDump.substring(0, ptDump.length()/4);
10     partitions[1] = ptDump.substring(ptDump.length()/4, ptDump.length()/2);
11     partitions[2] = ptDump.substring(ptDump.length()/2, 3*ptDump.length()/4);
12     partitions[3] = ptDump.substring(3*ptDump.length()/4, ptDump.length());
13     for (int i = 0; i < partitions.length; i++) {
14         String current = partitions[i];
15         Boolean bootable = current.startsWith("80");
16         Integer partitionNumber = i+1;
17         if (bootable) {
18             UI.appendToPartTableText(partitionNumber, "BOOTABLE");
19             addToLog("\tPartition " + (partitionNumber) + " - BOOTABLE");
20             String startingCHS = current.substring(2, 8);
21             String endingCHS = current.substring(10, 16);
22             String startingSector = current.substring(16, 24);
23             String partitionSize = current.substring(24, 32);
24             String endingSector = Long.toHexString(Long.parseLong(startingSector,16) + Long.parseLong(partitionSize, 16));
25             String partitionTypeNumber = current.substring(8, 10);
26             String partitionTypeName = PartitionTypes.getPartitionType(Integer.parseInt(partitionTypeNumber,16));
27             addToLog("\t\tPartition Type Descriptor: 0x" + partitionTypeNumber + " - " + partitionTypeName);//4
28             UI.appendToPartTableText(partitionNumber, "\nPartition type: " + partitionTypeNumber);
29             UI.appendToPartTableText(partitionNumber, "\n(" + partitionTypeName + ")");
30             addToLog("\t\tFrom sector 0x" + startingSector + " to 0x" + endingSector);
31             UI.appendToPartTableText(partitionNumber, "\nStarts at sector 0x" + startingSector);
32             UI.appendToPartTableText(partitionNumber, "\nEnds at sector 0x" + endingSector);
33             addToLog("\t\tCylinder, head and sector: from 0x" + startingCHS + " to 0x" + endingCHS);
34             UI.appendToPartTableText(partitionNumber, "\nCylinder, head and sector:");
35             UI.appendToPartTableText(partitionNumber, "\nfrom 0x" + startingCHS + " to 0x" + endingCHS);
36         } else {
37             UI.appendToPartTableText(partitionNumber, "NOT BOOTABLE");
38             addToLog("\tPartition " + partitionNumber + " - NOT BOOTABLE");
39         }
40     }
41 }
```

Figura 3-7: Método responsável pela análise dos dados da tabela de partições.

A figura 3-8 exibe uma captura de tela da interface gráfica do depurador. No topo esquerdo, pode-se enxergar uma área para controle de *breakpoints* e *watchpoints*. Na parte central superior, encontra-se um componente exibindo os caracteres hexadecimais correspondentes às instruções de nível de máquina presentes na área de código da MBR. No topo direito há uma área para mostrar o estado da assinatura da MBR e, abaixo dela, os botões para controle da execução do processo da máquina virtualizada pelo VMware Workstation. A parte inferior da interface é destinada a cada uma das quatro partições da tabela de partições.

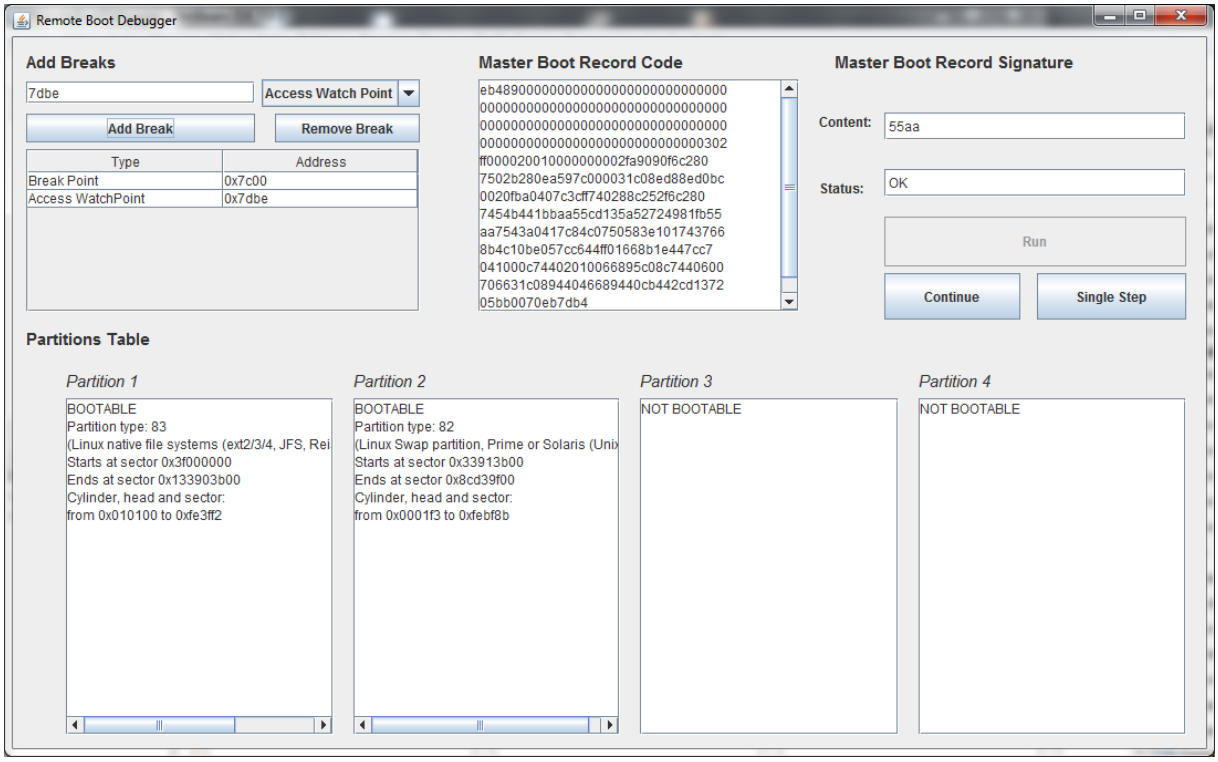


Figura 3-8: Captura de tela do depurador.

Além dessa interface principal, também foi feita uma pequena interface para a realização da conexão com o canal de comunicação do VMware Workstation, conforme pode ser observado na figura 3-9.

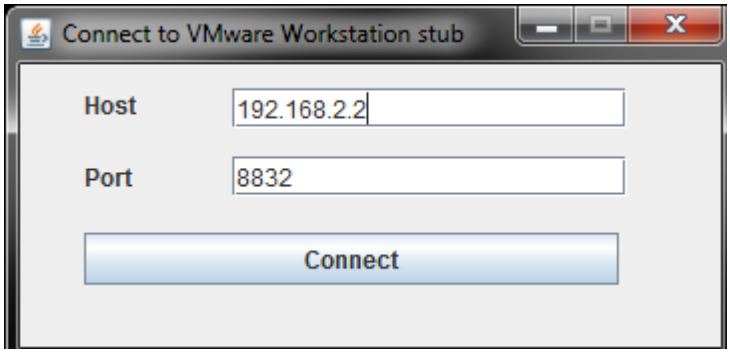


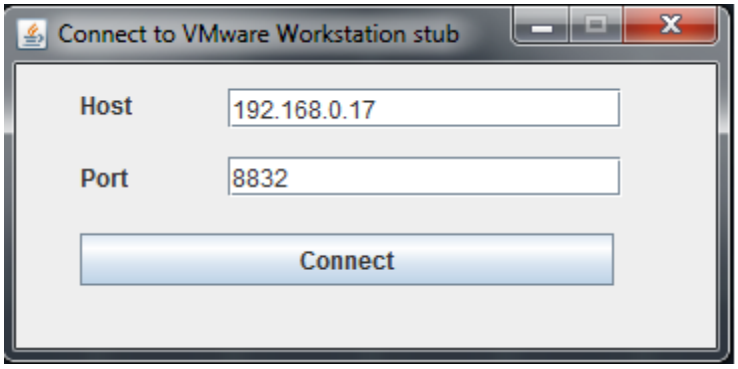
Figura 3-9: Captura de tela da interface de conexão do depurador com o canal de comunicação do VMware Workstation.

3.2.4 Cenário de testes

Após o término da etapa de implementação, relatada no item 3.2.3.2, foi necessário realizar uma série testes, a fim de verificar a corretude do depurador. Esta seção mostrará os detalhes de uma utilização do depurador.

O teste foi realizado no Laboratório de Computação Aplicada (LaCA) da UFSM, utilizando-se dois computadores distintos, através da rede cabeada. O endereço IPv4 do computador que executou o depurador é 192.168.0.12 e o do que executou a máquina virtual é 192.168.0.17.

Primeiramente, na máquina responsável pela execução da máquina virtual, foi iniciado o VMware Workstation e executada a máquina virtual com o sistema operacional *Android*. Após isso, no outro computador, foi iniciado o depurador, indicando o endereço IPv4 correspondente à outra máquina, como pode ser visualizado na captura de tela registrada na figura 3-10.



.

Figura 3-10: Captura de tela da interface para conexão com o VMware Workstation durante os testes.

Assim, abriu-se a interface principal do depurador e percebeu-se que a região da memória principal da máquina virtual que corresponde à MBR estava preenchida com valores nulos, como mostra a figura 3-11.

Adicionou-se, então, um *watchpoint* de leitura no endereço 0x7dbe, que é o primeiro endereço da tabela de partições, como é exibido na figura 3-12. Após a retomada da execução do processo de *boot*, percebeu-se que, quando o VMware Workstation cedeu novamente o controle do processo ao depurador, as posições de memória da MBR da máquina virtual permaneciam com valores nulos, indicando que a MBR ainda não havia sido carregada para a RAM.

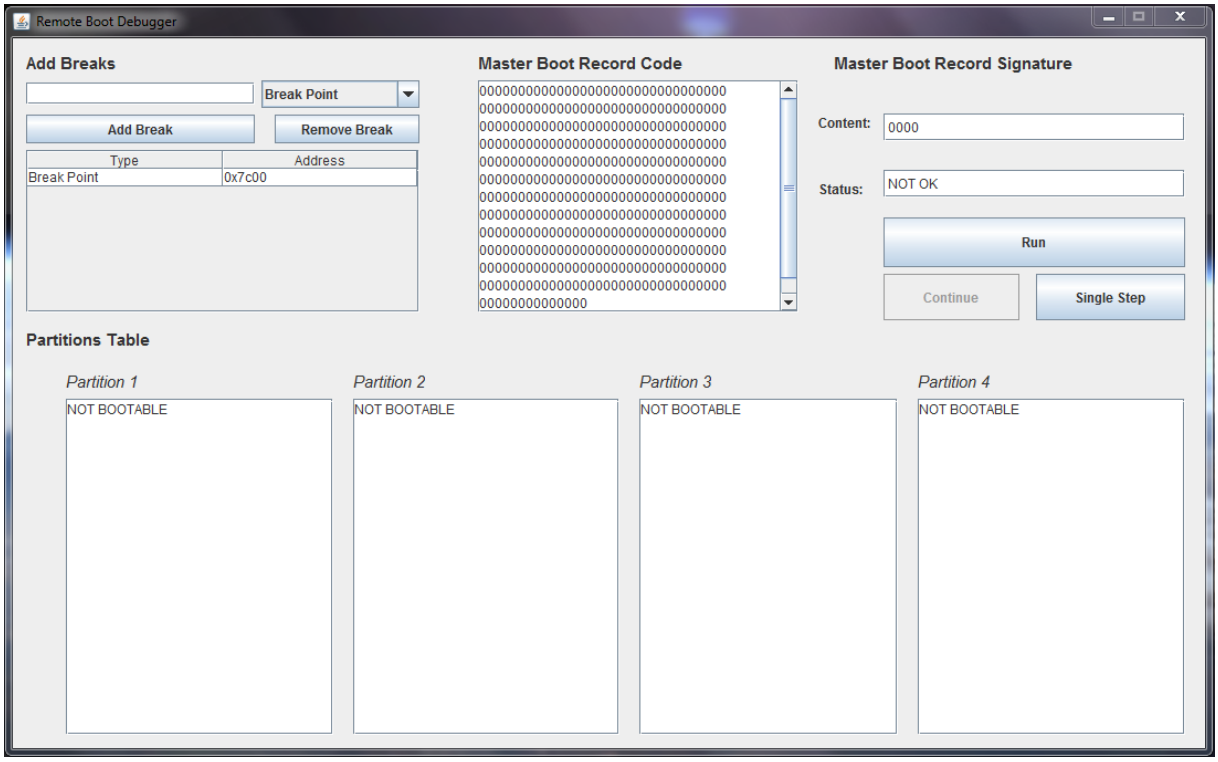


Figura 3-11: Captura de tela do estado inicial da interface principal do depurador.

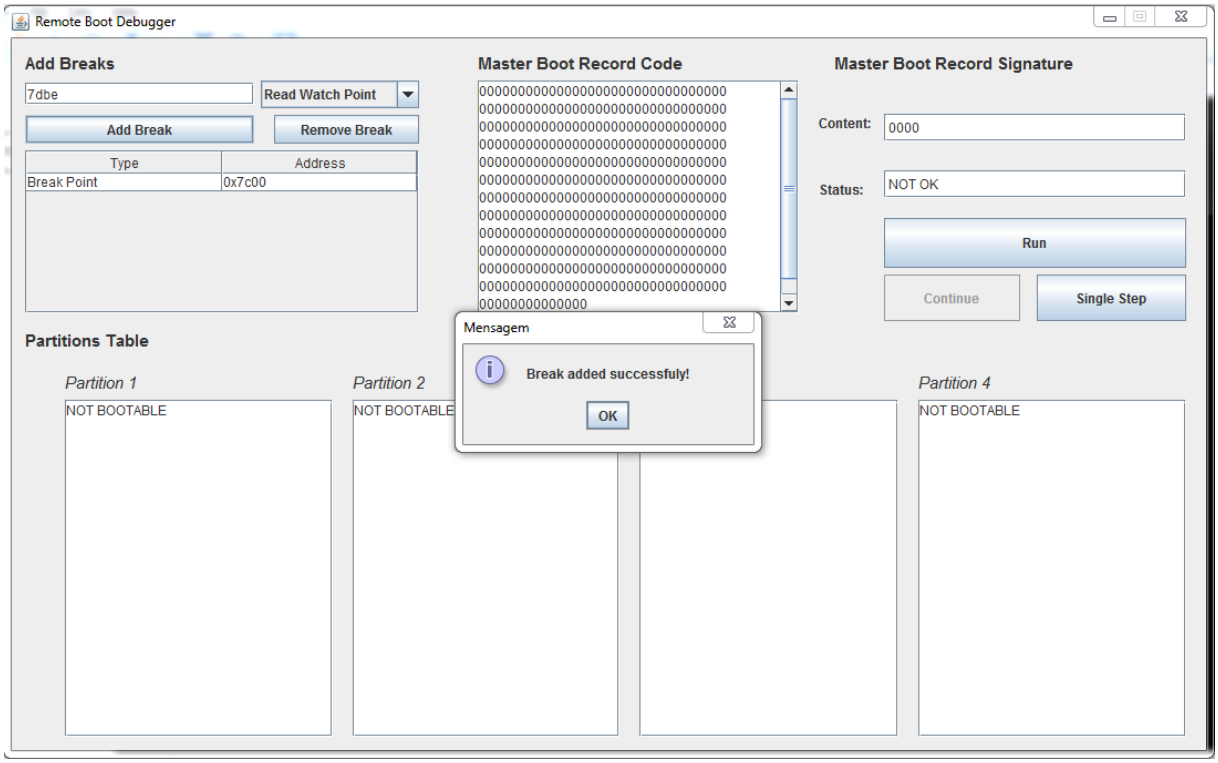


Figura 3-12: Captura de tela no momento da confirmação da adição de um *watchpoint* de leitura.

Após utilizar a operação de passo-a-passo algumas vezes, não houve alteração em quaisquer informações exibidas na interface. As instruções realizadas pelo processador da máquina virtual eram, portanto, anteriores ao estágio de *bootloader* primário.

A fim de avançar para a etapa de *bootloader* primário, foi removido o *watchpoint* adicionado no início deste teste, como se pode visualizar na figura 3-13.

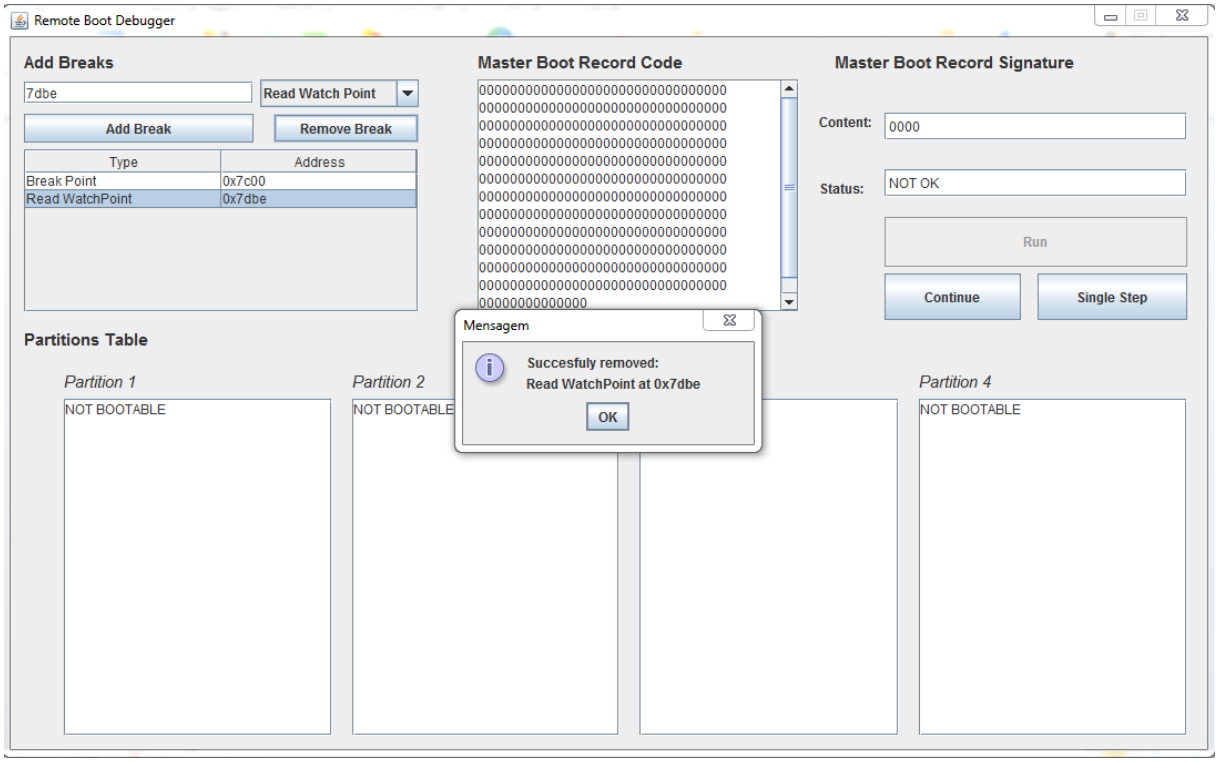


Figura 3-13: Captura de tela no momento da confirmação da remoção de um *watchpoint* de leitura.

Prosseguiu-se, então, com a execução do processo de inicialização do *Android* na máquina virtual. O processo foi interrompido novamente quando atingiu o endereço 0x7c00, pois é o mesmo endereço apontado pelo *breakpoint* que o depurador adiciona automaticamente. Nesse ponto, os dados contidos na MBR, como informações das partições e assinatura da MBR, já eram devidamente interpretados pelo depurador, como é exibido na figura 3-14.

Após isso, a operação de passo-a-passo foi executada mais algumas vezes, mas sem nenhuma mudança perceptível no estado do processo de *boot*. Permitiu-se, então, que o processo fosse continuado normalmente, encerrando a participação do depurador.

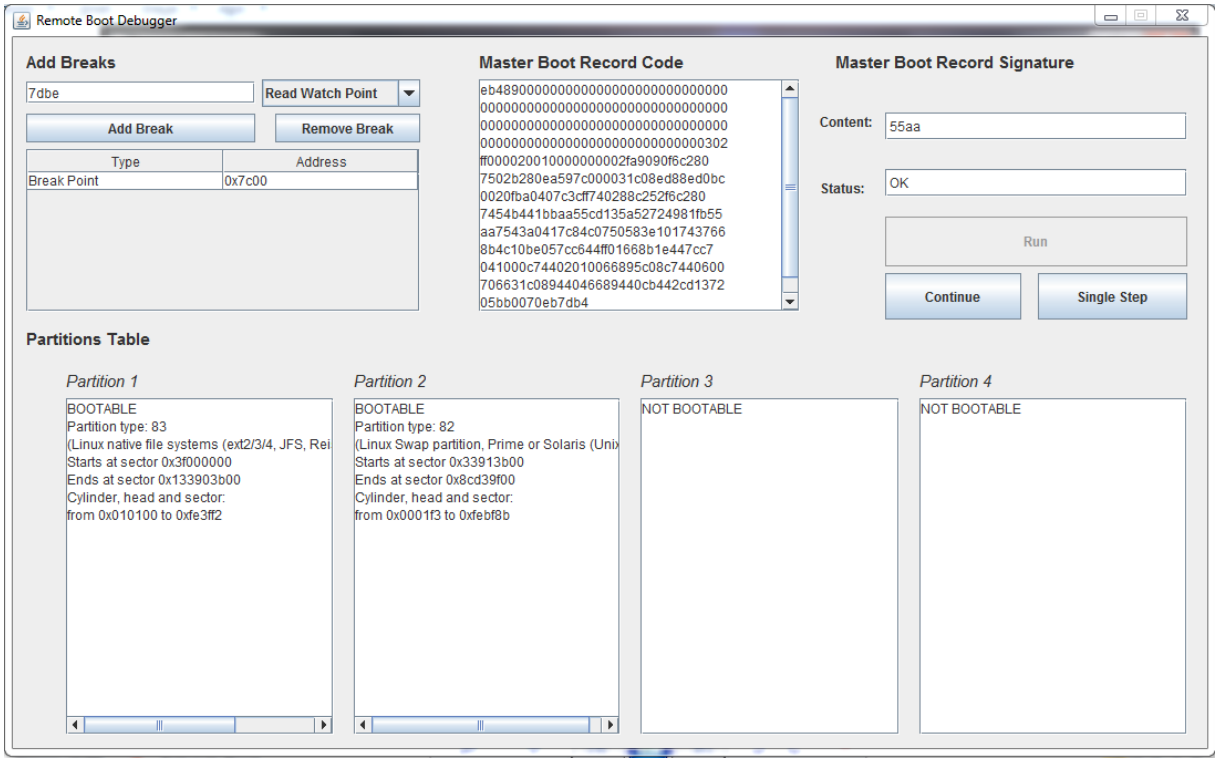


Figura 3-14: Captura de tela no momento em que o depurador pode interpretar adequadamente as informações da MBR.

A figura 3-15 exibe o cabeçalho do arquivo de log gerado pelo depurador no teste relatado, enquanto a figura 3-16 mostra um dos estados da MBR registrados no arquivo durante o processo de depuração. As duas primeiras partições exibem o resultado da criação prévia das duas partições no disco da máquina virtual, ao instalar o sistema operacional *Android*. A primeira partição é do tipo de sistema de arquivos nativo do Linux (ext2, ext3 ou ext4) e a segunda é uma partição de *swap*.

```
################################################
######## LOG FILE FOR DEBUG INFORMATIONS ########
########     09/12/2011 - 15:32:17:651     ########
################################################
```

Figura 3-15: Exemplo de cabeçalho do arquivo de log gerado pelo depurador.

```
 === STATE DUMP NUMBER 8 ===
Current time: 09/12/2011 - 15:35:11:666
Curremt Master Boot Record Breaks
    Type: Break Point - Address: 0x7c00
Master Boot Record dump:
    Boot Loader (from 0x7c00 to 0x7dbd)
        eb4890000000000000000000000000000
        000000000000000000000000000000000
        000000000000000000000000000000000
        000000000000000000000000000000302
        ff000020010000000002fa9090f6c280
        7502b280ea597c000031c08ed88ed0bc
        0020fba0407c3cff740288c252f6c280
        7454b441bbaa55cd135a52724981fb55
        aa7543a0417c84c0750583e101743766
        8b4c10be057cc644ff01668b1e447cc7
        041000c74402010066895c08c7440600
        706631c08944046689440cb442cd1372
        05bb0070eb7db4
    Partition Table (from 0x7dbe to 0x7dfd)
    Partition 1 - BOOTABLE
        Partition Type Descriptor: 0x83 - Linux native file systems (ext2/3/4, JFS, Reiser, xiafs, and others).
        From sector 0x3f000000 to 0x133903b00
        Cylinder, head and sector: from 0x010100 to 0xfe3ff2
    Partition 2 - BOOTABLE
        Partition Type Descriptor: 0x82 - Linux Swap partition, Prime or Solaris (Unix).
        From sector 0x33913b00 to 0x8cd39f00
        Cylinder, head and sector: from 0x0001f3 to 0xfebf8b
    Partition 3 - NOT BOOTABLE
    Partition 4 - NOT BOOTABLE
    Signature (from 0x7dfe to 0x7dff)
        Status: OK! (0x55aa)
```

Figura 3-16: Exemplo de informações do estado da MBR no log gerado pelo depurador.

4 CONCLUSÃO

O objetivo deste trabalho foi proporcionar a estudantes e desenvolvedores de sistemas operacionais um meio para depuração do estágio de *bootloader* primário, etapa componente do processo de inicialização de qualquer sistema operacional.

Durante a realização da revisão bibliográfica e do desenvolvimento, foi possível adquirir uma quantidade significativa de conhecimento acerca de sistemas operacionais e, mais especificamente, sobre o processo de *boot* do sistema operacional *Android*.

O uso da ferramenta, durante a etapa de testes, permitiu que alguns detalhes importantes da etapa inicial do processo de *boot* do *Android* fossem percebidos, como, por exemplo, as informações contidas na tabela de partições e a assinatura da MBR. Também foram realizados testes com o sistema operacional Fedora 16. O depurador obteve, com sucesso, as informações referentes ao trecho de código da MBR e da assinatura. Os dados da tabela de partições, no entanto, não puderam ser interpretados corretamente.

Com o GDB, ferramenta já existente para depuração, é necessário que o usuário tenha pleno conhecimento da estrutura da MBR e dos comandos de utilização da ferramenta. A ferramenta desenvolvida durante este trabalho permite a sua utilização com uma exigência bastante menor de conhecimento prévio sobre o assunto. Além disso, para obter, com o GDB, as informações coletadas e interpretadas pelo depurador construído neste trabalho, seria necessário um trabalho bastante demorado e repetitivo, sujeito, também, a erros humanos durante essa tarefa.

Apesar de o caso de testes ter sido, durante todo o trabalho, o *Android*, a ferramenta de depuração do estágio de *bootloader* primário funciona em outros

sistemas operacionais, haja vista que a etapa de *bootloader* primário é uma fase comum no carregamento de qualquer sistema operacional.

Uma das maiores dificuldades encontradas durante a etapa de implementação foi a precariedade das documentações encontradas. A fim de reduzir esse problema em trabalhos futuros, as classes criadas para o desenvolvimento do depurador foram devidamente documentadas.

Este trabalho, apesar de ter apresentado resultados funcionais, ainda necessita de alguns aperfeiçoamentos. Uma sugestão viável, por exemplo, é a realização de operações que revertem o código hexadecimal do segmento de código da MBR para transformá-lo em operações em *assembly*, legíveis por uma pessoa. Outras opções de trabalho são a execução de procedimentos para leitura e definição dos valores dos registradores de usuário em tempo de execução e injeção de informações nas áreas de memória da MBR.

As ideias citadas acima se referem ao processo de *bootloader* primário, sendo utilizáveis em qualquer sistema operacional. No entanto, também se pode, a partir do estudo das etapas seguintes do processo de inicialização do *Android*, desenvolver um depurador para outros estágios ou, até mesmo, genérico para todo o mecanismo de *boot*.

REFERÊNCIAS

EMBECOSM. Howto: GDB Remote Serial Protocol - Writing a RSP Server , 2008. Disponivel em: <http://www.embecosm.com/appnotes/ean4/embecosm-howto-rsp-server-ean4-issue-2.html>. Acesso em: 18 Setembro 2011.

GATLIFF, B. Embedding with GNU: The gdb Remote Serial Protocol. **Embedded Systems Programming**, Novembro 1999. 108-113.

GRÖTKER, T. et al. **The Developer's Guide to Debugging**. [S.l.]: Springer, 2008.

HEISENBERG, W. **The Physical Principles of the Quantum Theory**. [S.l.]: Dover Pubns, 1930.

JONES, M. T. IBM: developerWorks: Inside the Linux Boot Process - Take a guided tour from the Master Boot Record to the first user-space application, 2006. Disponivel em: <http://www.ibm.com/developerworks/linux/library/l-linuxboot/>. Acesso em: 15 Agosto 2011.

LAUREANO, M. **Máquinas Virtuais e Emuladores - Conceitos, Técnicas e Aplicações**. 1. ed. [S.l.]: Novatec, 2006.

NANDU310. Nandu310′s Blog - Memory Areas in C Language, 2009. Disponivel em: <http://nandu310.wordpress.com/2009/11/09/memory-areas-in-c-language/>. Acesso em: 12 Setembro 2011.

PADALA, P. Playing with ptrace, Part I, 2002. Disponivel em: <http://www.linuxjournal.com/article/6100>. Acesso em: 20 Setembro 2011.

PAXSON, V. A Survey of Support for Implementing Debuggers, Outubro 1990.

QUINION, M. World Wide Words: Boot, 2002. Disponivel em: <http://www.worldwidewords.org/qa/qa-boo2.htm>. Acesso em: 17 Outubro 2011.

ROSENBERG, J. B. **How Debuggers Work:** Algorithms, Data Structures, and Architecture. [S.l.]: [s.n.], 1996.

SEDORY, D. B. MBR/EBR Partition Tables, 2009. Disponivel em: <http://thestarman.pcministry.com/asm/mbr/PartTables.htm>. Acesso em: 22 Outubro 2011.

SHARMA, A. Minnor Adition: this, that and the other of technology, art and architecture - Process Address Space – Code, gvar, BSS, Heap & Stack, 23 Março 2011. Disponivel em: <http://www.minoraddition.com/2011/03/23/where-does-code-execute-process-address-space-code-gvar-bss-heap-stack/>. Acesso em: 17 Outubro 2011.

TAN, M. A minimal GDB stub for embedded remote debugging., 12 Dezembro 2002.

TANENBAUM, A. S. **Modern Operating Systems (2nd Edition) (GOAL Series)**. [S.l.]: Prentice Hall, 2001.