UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MATEUS CANALLE HANEIKO

UM ESTUDO SOBRE A PARALELIZAÇÃO DO SISTEMA DE INICIALIZAÇÃO DO FREEBSD

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA
2017

MATEUS CANALLE HANEIKO

UM ESTUDO SOBRE A PARALELIZAÇÃO DO SISTEMA DE INICIALIZAÇÃO DO FREEBSD

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. MSc. Saulo Jorge Beltrão de Queiroz

PONTA GROSSA
2017

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Ponta Grossa
Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática
Bacharelado em Ciência da Computação

TERMO DE APROVAÇÃO

UM ESTUDO SOBRE A PARALELIZAÇÃO DO SISTEMA DE INICIALIZAÇÃO DO FREEBSD

por

MATEUS CANALLE HANEIKO

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 08 de novembro de 2017 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Prof. MSc. Saulo Jorge Beltrão de Queiroz
Orientador

_____
Prof. Dr. Erikson Freitas De Morais
Membro titular

_____
Prof. Dr. Andre Koscianski
Membro titular

_____
Prof.ª Dr.ª Helyane Bronoski Borges
Responsável pelo Trabalho de Conclusão de Curso

_____
Prof. MSc. Saulo Jorge Beltrão de Queiroz
Coordenador do curso

RESUMO

Haneiko, Mateus Canalle. Um Estudo Sobre a Paralelização do Sistema de Inicialização do FreeBSD. 2017. 75 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) — Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2017.

O sistema operacional FreeBSD possui suporte a vários processadores. Mais do que dar suporte básico, ele tenta usar as funcionalidades específicas de cada um visando melhorar seu desempenho. Com o advento de processadores com múltiplos núcleos surgiu a oportunidade de otimizar várias tarefas do sistema operacional. Sistemas operacionais, como o Debian/Linux, começaram a tirar vantagem dos múltiplos núcleos em várias áreas, uma dessas áreas foi a inicialização dos serviços. A estratégia adotada, no Systemd usado no Debian/Linux por exemplo, foi paralelizar a inicialização do serviços, assim conseguindo diminuir o tempo de inicialização. O FreeBSD é um dos sistemas operacionais que não explorou essa vantagem, continuando com uma inicialização de serviços sequencial. O objetivo deste trabalho é propor alterações no sistema de inicialização de serviços do FreeBSD a fim de assegurar a sua paralelização, e avaliar o desempenho do sistema proposto em comparação com o atual sistema utilizado. Os resultados mostram que a solução proposta consegue diminuir o tempo de inicialização quando usada em conjunto com processador de 4 núcleos. Também mostram que não é somente paralelismo que difere o FreeBSD de outros sistemas.

Palavras-chaves: FreeBSD. *Boot*. Sistema operacional. Inicialização de serviços. Paralelização de processos.

ABSTRACT

Haneiko, Mateus Canalle. A Study on Parallelization of the FreeBSD Boot System. 2017. 75 p. Undergraduate thesis (Bachelor of Computer Science) — Federal University of Technology - Paraná. Ponta Grossa, 2017.

The FreeBSD operating system supports many platforms. It also tries to use any specific behavior given by the platform that might improve its performance. With the introduction of processors with multi-core capabilities many opportunities to optimize various operating system's tasks became known. Operating systems like Debian/Linux have started taking advantage of the multi-core in many ways. One of those was the system initialization sub-system. Systemd, used on Debian/Linux, paralleled the service initialization, therefore decreasing the time taken for it to be done. FreeBSD is one of the systems that didn't change, keeping a sequential service initialization. The objective of this work is to propose changes to the FreeBSD service initialization scheme making it parallel and evaluate those changes in comparison to the currently used system. The results show a decrease in initialization time when the proposed solution is combined with a quad-core processor. It also shows that the diference between FreeBSD and others systems goes beyond parallelism.

Keywords: FreeBSD. *Boot*. Operating system. Service initialization. Parallelization of processes.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

BSD	Berkeley Software Distribution
DARPA	Defense Advanced Research Projects Agency
AT&T	American Telephone and Telegraph
TCP	Transmission Control Protocol
IP	Internet Protocol
BIOS	Basic Input Output System
PID	Process Identification
POSIX	Portable Operating System Interface
XML	eXtensible Markup Language
UDP	User Datagram Protocol
GPL	General Public License
LGPL	Lesser General Public License
D-BUS	Desktop BUS
CDDL	Common Development and Distribution License
CPU	Central Processing Unit
GB	Gigabyte
MB	Megabyte
RAM	Random Access Memory

SUMÁRIO

1 INTRODUÇÃO

O FreeBSD é um sistema operacional de propósito geral. Como servidor é usado pela empresa Netflix, onde em 2015 foi responsável por um terço do tráfego de internet nos Estados Unidos. Essa empresa emprega servidores baseados no FreeBSD 9 como seu principal componente de infraestrutura (FULLAGAR, 2015). Também é usado como servidor pela WhatsApp, onde segundo Koum (2015) mantém entre 2 e 3 milhões de conexões concorrentes por servidor.

Apesar dos exemplos citados, a relevância desse sistema operacional é difícil de ser estabelecida devido a liberdade irrestrita de uso associada à sua licença permissiva. A licença BSD (*Berkeley Software Distribution*) permite que o código fonte seja alterado e usado em soluções comerciais sem necessidade de retorno, seja monetário ou em forma de contribuição do código alterado. Por causa disso a Sony, uma empresa produtora de jogos, pôde usar o código em seu video game Playstation 4 sem divulgar tal uso (SONY, 2016). Até que tal fato fosse divulgado não era possível contabilizar os usuários do Playstation 4 como usuários do FreeBSD - ainda que indiretos.

Com a popularização de processadores de múltiplos núcleos, sistemas operacionais tais como o Debian/Linux começaram a tirar vantagem da possibilidade de executar tarefas independentes em paralelo. Uma dessas possibilidades consiste em paralelizar os serviços executados durante a fase de inicialização do sistema, fase essa também conhecida como *boot*. A adoção dessa estratégia pelo Systemd (sistema de inicialização usado no Debian/Linux), por exemplo, permitiu diminuir o tempo total de inicialização (POETTERING; SIEVERS; LEEMHUIS, 2012).

Não obstante tal vantagem, o sistema base do FreeBSD ainda não realiza a paralelização de serviços de inicialização, permanecendo sob uma abordagem sequencial (MCKUSICK; NEVILLE-NEIL; WATSON, 2014). O presente trabalho propõe um estudo, implementação e avaliação de desempenho de um sistema de inicialização de serviços em paralelo para o FreeBSD.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Propor alterações no sistema de inicialização de serviços do FreeBSD que assegurem a inicialização em paralelo e que mantenham o formato de arquivo usado nos *scripts* de inicialização.

1.1.2 Objetivos específicos

- Estudar o sistema de inicialização do FreeBSD;
- Pesquisar por soluções de inicialização em paralelo de serviços em sistemas operacionais variados;
- Avaliar o desempenho do sistema proposto em comparação com o atual sistema do FreeBSD.

1.2 JUSTIFICATIVA

O FreeBSD possui um sistema de inicialização sequencial, portanto não tira vantagem de processadores com múltiplos núcleos. Como observado em outros sistemas operacionais essa paralelização pode diminuir o tempo de inicialização do sistema.

Muito embora seja possível encontrar sistemas operacionais de código aberto capazes de paralelizar o sistema de inicialização, adaptar tais soluções ao FreeBSD não é uma tarefa trivial.

Além do fato supramencionado, algumas opções de paralelização existentes possuem licenças que conflitam com a licença do FreeBSD, portanto não podem ser usadas no mesmo. Esse trabalho propõe alterações no código-fonte do FreeBSD sem alteração de licença.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados um pouco da origem do FreeBSD, uma descrição do funcionamento do sistema de inicialização do FreeBSD e de alguns sistemas alternativos, uma síntese das diferenças entre as alternativas e por fim uma descrição sobre processos e *threads*.

2.1 FREEBSD

O FreeBSD é um sistema operacional de código fonte aberto descendente do 4.3BSD que por sua vez é descendente do UNIX. O BSD surgiu em 1979, inicialmente financiado pela DARPA (*Defense Advanced Research Projects Agency*) com o objetivo de prover suporte aos seus protocolos de rede TCP/IP (*Transmission Control Protocol/Internet Protocol*). O seu uso dependia da licença de código da AT&T (*American Telephone and Telegraph*). Até o lançamento do 4.3BSD, todos os usuários do BSD deveriam ter uma licença de código da AT&T. Com o aumento do custo dessa licença a utilização do código BSD começou a se tornar proibitiva. A última versão requerendo essa licença foi a 4.4BSD em junho de 1993 (MCKUSICK; NEVILLE-NEIL; WATSON, 2014).

A primeira versão redistribuível gratuitamente foi lançada em 1994, o que veio a ser conhecido como FreeBSD. Sua licença de uso é a BSD, uma licença que permite a redistribuição do FreeBSD em formato de código fonte ou binário sem custo e sem necessidade de publicação das modificações do código (MCKUSICK; NEVILLE-NEIL; WATSON, 2014). O último lançamento foi a versão 11.1 em julho de 2017 (FREEBSD, 2017).

2.1.1 Inicialização do sistema

Inicializar o sistema operacional é uma tarefa complexa e de múltiplas etapas que iniciam com o BIOS (*Basic Input Output System*) da plataforma, o qual carrega um *kernel*[1] e seus módulos. Após carregar, o *kernel* do FreeBSD inicia execução e após inicialização ele chama o primeiro processo de usuário, **init**. O processo **init** é responsável por começar os passos de inicialização do espaço de usuário.

O *kernel* começa inicializando uma variedade de subsistemas internos como o alocador de memória e o escalonador de processos. Ele usa métodos de enumeração de *hardware* para identificar recursos de *hardware* disponíveis e anexar os *drivers* apropriados. Ele também inicializa suas próprias estruturas de dados, como a estrutura de memória virtual que descreve a memória física. Eventualmente um dispositivo de armazenamento adequado a ser usado como

[1] Núcleo do sistema operacional.

sistema de arquivos *root* será descoberto e o sistema de arquivos será montado. O passo final é a criação do primeiro processo de usuário com PID [2] 1 para executar o binário **/sbin/init**. O processo **init** é responsável por executar os *scripts* de inicialização que fazem checagem do sistema de arquivos, configuram as interfaces de rede, entre outros, e trazer o sistema para operação multiusuário (MCKUSICK; NEVILLE-NEIL; WATSON, 2014).

2.2 SISTEMAS DE INICIALIZAÇÃO

Os sistemas de *boot* variam em implementação e funcionalidade. Nesta seção serão analisados o sistema presente no FreeBSD, OS X, Android e outros sistemas populares do Linux como o Launchd e OpenRC.

2.2.1 FreeBSD RC

No FreeBSD o sistema de inicialização de espaço de usuário começa com o processo **/sbin/init**. Ao iniciar esse programa o *kernel* já está operando e funcional, mas existem vários passos adicionais que devem ser feitos antes que os usuários possam logar no sistema.

O programa **init** é chamado e recebe alguns parâmetros do *kernel*. O programa usa estes valores para determinar se deve levar o sistema ao modo multiusuário ou modo de usuário único, e também determinar se deve realizar a checagem de consistência dos discos rígidos com o programa **fsck**. Em modo de usuário único, **init** apenas invoca o *shell* padrão **/bin/sh** e serve apenas para executar operações de manutenção. Em modo multiusuário, **init** invoca o *shell* padrão para interpretar os comandos presentes no *script* **/etc/rc**, o qual é a raiz de um conjunto de *scripts* que fazem a inicialização do espaço de usuário do sistema (MCKUSICK; NEVILLE-NEIL; WATSON, 2014).

O arquivo **/etc/rc** serve para ordenar e executar os vários *scripts* presentes no diretório **/etc/rc.d**. Para ordenar os *scripts* é chamado o programa **rcorder** que recebe um conjunto de nomes de *scripts* como entrada, descobre suas interdependências e devolve como saída uma lista topologicamente ordenada de seus nomes (MCKUSICK; NEVILLE-NEIL; WATSON, 2014). A saída pode ser vista no Quadro 8 (apêndice C). Outro *script* importante é o **/etc/rc.subr** que possui várias funções auxiliares usadas pelo **/etc/rc**, como por exemplo a função *run_rc_script* que será vista mais adiante.

Cada *script* declara suas dependências usando uma série de expressões no início do arquivo. Cada expressão fica em um comentário em sua própria linha e é declarada usando o formato **# EXPRESSÃO: valores**. As expressões e seus significados estão descritas no Quadro

[2] *Process Identification*: número que identifica o processo.

1.

Quadro 1 – Lista de expressões

Expressão	Descrição
PROVIDE	Nomes que o *script* provê.
REQUIRE	Nomes que o *script* requer/depende.
BEFORE	Nomes que dependem do *script*.
KEYWORD	Palavras usadas para representar casos específicos.

Fonte: Autoria própria

Código 1 – Cabeçalho do script yppasswdd

```
1 #!/bin/sh
2 #
3 # $FreeBSD: releng/11.0/etc/rc.d/yppasswdd 298514 2016-04-23 16:10:54
    Z lme $
4 #
5
6 # PROVIDE: yppasswdd
7 # REQUIRE: ypserv ypset
8 # BEFORE:  LOGIN
9 # KEYWORD: shutdown
```

O Código 1 mostra um exemplo de *script* contendo essas expressões: o *script* **yppasswdd** provê o nome *yppasswdd*; os *scripts* que provêm os nomes *ypserv* e *ypset* devem ser executados antes dele (*yppasswdd* depende ou requer *ypserv* e *ypset*); e ele deve ser executado antes do *script* que provê *LOGIN*. A expressão *KEYWORD* contém valores para casos particulares. O Quadro 2 mostra os valores mais comuns que aparecem em *KEYWORD*.

Quadro 2 – Valores comuns para KEYWORD

Valor	Descrição
nostart	Indica que o *script* não deve ser executado na inicialização.
shutdown	Indica que o *script* deve ser executado no desligamento do sistema.
firstboot	Indica que o *script* deve ser executado somente se for o primeiro boot.
nojail	Indica que o *script* não deve ser executado quando estiver dentro de uma *jail*[a].
nojailvnet	Indica que o *script* não deve ser executado quando a *jail* possui um VNET[b].

Fonte: Autoria própria

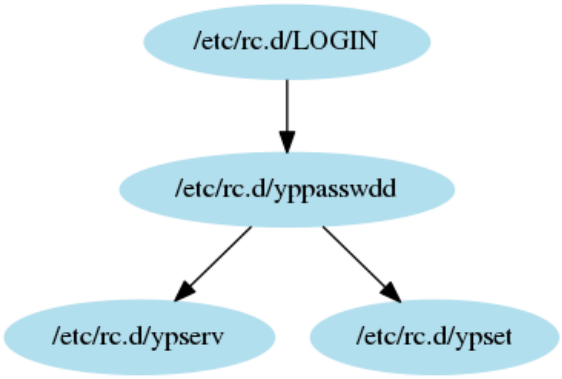
[a] Um tipo de máquina virtual leve.
[b] Uma placa de rede virtual usada em *jails*.

O controle de quais serviços são executados é realizado de duas maneiras. A primeira é usando as expressões do Quadro 2. Por exemplo, se o *script* declara a *KEYWORD nostart* então ele não será executado. A segunda maneira é a partir de variáveis declaradas nos arquivos de configuração **/etc/rc.conf** e **/etc/default/rc.conf**. O arquivo **/etc/defaults/rc.conf** contém valores padrão para uma série de variáveis e não deve ser editado, ele é carregado antes do arquivo **/etc/rc.conf** servindo como uma fonte de configuração padrão. Já o **/etc/rc.conf** contém as configurações do administrador do sistema que se sobrepõem ao valores de **/etc/defaults/rc.conf**

(MCKUSICK; NEVILLE-NEIL; WATSON, 2014). Por exemplo, o serviço de conexão remota por *ssh*[3] vem desabilitado por padrão, o que significa que no arquivo **/etc/defaults/rc.conf** ele está desabilitado: *sshd_enable="NO"*. Para habilitar esse serviço é necessário sobrescrever essa variável no arquivo **/etc/rc.conf** com o valor: *sshd_enable="YES"*.

Usando as expressões do Quadro 1 e os valores do Quadro 2 o programa **rcorder** pode determinar a ordem, usando ordenação topológica, na qual os *scripts* devem ser executados (MCKUSICK; NEVILLE-NEIL; WATSON, 2014). A ordenação topológica segundo Cormen et al. (2009, p. 612) pode ser realizada por uma busca em profundidade em um grafo acíclico direcionado. No caso dos *scripts* de inicialização do FreeBSD, cada *script* é um vértice do grafo e cada dependência é uma aresta que sai do módulo requerente em direção ao módulo requerido.

Figura 1 – Exemplo de grafo de dependência



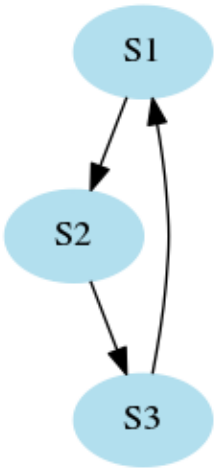
Fonte: Autoria própria

A Figura 1 mostra que **LOGIN** depende de **yppasswdd**, e este depende de **ypserv** e **ypset**. A ordem de execução desses *scripts* seria: **ypserv** e **ypset** são executados por primeiro em seguida vem **yppasswdd** e por último **LOGIN**.

Como o grafo deve ser acíclico então não pode existir dependência circular nos *scripts*. A Figura 2 mostra um exemplo com três *scripts* (**S1**, **S2** e **S3**) onde **S1** depende de **S2**, **S2** depende de **S3** e **S3** depende de **S1** formando um ciclo. Nesse exemplo, devido à dependência circular, não há como decidir qual *script* deve ser executado primeiro.

[3] Secure shell.

Figura 2 – Exemplo de grafo com dependência circular



Fonte: Autoria própria

O grafo de dependência com todos os *scripts* presentes por padrão no FreeBSD pode ser visto no apêndice B.

2.2.1.1 Funcionamento do **rc**

Esta seção descreve o funcionamento do *script* **/etc/rc** mostrando as partes que possuem importância para a ordenação e inicialização dos *scripts*.

No Código 2 observa-se que o **rc** inicia determinando quais *KEYWORDS* do Quadro 2 serão usadas e armazenadas na variável *skip*, em especial as *KEYWORDS* que tem a ver com ignorar *scripts*. O valor *nostart* é sempre usado (linha 79) mas os valores *nojail* (linha 81) e *nojailvnet* (linha 83) estão condicionados ao fato de o sistema operacional estar executando dentro de uma *jail* (linha 80) e possuir uma *vnet* (linha 82), respectivamente.

Código 2 – Arquivo /etc/rc: KEYWORDS

```
79 skip="-s nostart"
80 if [ `/sbin/sysctl -n security.jail.jailed` -eq 1 ]; then
81         skip="$skip -s nojail"
82         if [ `/sbin/sysctl -n security.jail.vnet` -ne 1 ]; then
83                 skip="$skip -s nojailvnet"
84         fi
85 fi
```

No Código 3 observa-se que a próxima *KEYWORD* a ser vista é *firstboot* (linha 89), que está condicionada ao fato de ser o primeiro *boot* da máquina (linha 88). Assim como pode ser visto pelo comentário na linha 87, os *scripts* com o valor *firstboot* serão ignorados se não for o primeiro *boot*.

Código 3 – Arquivo /etc/rc: KEYWORDS

```
87 # If the firstboot sentinel doesn't exist, we want to skip firstboot
       scripts.
```

```
88 if ! [ -e ${firstboot_sentinel} ]; then
89         skip_firstboot="-s firstboot"
90 fi
```

No Código 4, na linha 95 há uma chamada ao programa **rcorder** para ordenar todos os *scripts* da pasta **/etc/rc.d**. Esse programa faz a ordenação topológica dos *scripts* ignorando todos os que possuírem as *KEYWORDS* determinadas anteriormente. O resultado da ordenação é armazenado na variável *files*.

Código 4 – Arquivo /etc/rc: ordenação dos scripts

```
95 files=`rcorder ${skip} ${skip_firstboot} /etc/rc.d/* 2>/dev/null`
```

No Código 5 observa-se a iteração sobre a lista contida em *files* (linha 98), onde cada elemento é executado (linha 99) e armazenado na variável *_rc_elem_done* (linha 100). A iteração pára quando o elemento executado for igual ao valor da variável *early_late_divider* (linhas 102 e 103). Por padrão essa variável está definida no arquivo **/etc/defaults/rc.conf** e contém o valor *FILESYSTEMS*, o que quer dizer que os *scripts* serão executados até encontrar o *script* **FILESYSTEMS**.

Código 5 – Arquivo /etc/rc: execução dos scripts

```
97  _rc_elem_done=' '
98  for _rc_elem in ${files}; do
99          run_rc_script ${_rc_elem} ${_boot}
100         _rc_elem_done="${_rc_elem_done}${_rc_elem} "
101
102         case "$_rc_elem" in
103         */${early_late_divider})        break ;;
104         esac
105 done
```

A execução do *script* **FILESYSTEMS** sinaliza que outros sistemas de arquivos (outras partições) já foram montadas, portanto podem haver mais pastas contendo *scripts* de inicialização. O Código 6 mostra a chamada da função *find_local_scripts_new* que serve para detectar essas pastas (linha 114), a chamada dessa função é condicionada pela variável *local_startup* (linha 112). Por padrão, essa variável está definida no arquivo **/etc/defaults/rc.conf** e contém o valor */usr/local/etc/rc.d*, o que significa que novos *scripts* serão pesquisado na pasta **/usr/local/etc/rc.d**. A função *find_local_scripts_new* (linha 114) está definida no arquivo **/etc/rc.subr** e serve para verificar se a pasta existe e declarar a variável *local_rc* contendo o nome de todos os *scripts* encontrados nela.

Código 6 – Arquivo /etc/rc: pesquisa por novos repositórios de scripts

```
112 case ${local_startup} in
113 [Nn][Oo] | '') ;;
114 *)      find_local_scripts_new ;;
115 esac
```

Como visto no Código 5, cada *script* é executado pela função *run_rc_script* (linha 99). Esta função pode ser vista no Código 7, ela está definida em **/etc/rc.subr** e serve para

executar os *scripts*. A função começa capturando o primeiro e segundo parâmetros nas variáveis *_file* e *_arg* (linhas 1324 e 1325), respectivamente. Em seguida é verificado se essas variáveis possuem algum valor (linhas 1326 à 1328), mostrando um erro em caso negativo (linha 1327). Nas linhas 1330 à 1334, várias variáveis globais são desalocadas para não interferir na execução do *script*. Na linha 1336 inicia-se uma estrutura condicional *case* (estrutura equivalente ao *switch* da linguagem C) onde são feitas algumas verificações sobre o valor de *_file*: se o valor casa com o formato */etc/rc.d/*.sh* (linha 1337), em caso positivo é mostrado um aviso (linha 1338) já que não é mais suportado; se o valor termina em *#*, *.OLD*, *.bak*, *.orig* ou *,v*, em caso positivo também é mostrado um aviso já que são apenas arquivos temporários; todos os outros entram no terceiro caso (linha 1343). Verifica-se que o arquivo é executável (linha 1344), e se a variável *rc_fast_and_loose* existe, caso exista então o *script* é executado dentro do *shell* atual. Por padrão essa variável não existe, portanto o *else* (linha 1347) é executado. Nas linhas 1348 à 1351 é preparado um processo filho para executar (linha 1351) o *script*.

Código 7 – Arquivo /etc/rc.subr: função run_rc_script

```
1322 run_rc_script()
1323 {
1324         _file=$1
1325         _arg=$2
1326         if [ -z "$_file" -o -z "$_arg" ]; then
1327                 err 3 'USAGE: run_rc_script file arg'
1328         fi
1329
1330         unset    name command command_args command_interpreter \
1331                 extra_commands pidfile procname \
1332                 rcvar rcvars rcvars_obsolete required_dirs
                        required_files \
1333                 required_vars
1334         eval unset ${_arg}_cmd ${_arg}_precmd ${_arg}_postcmd
1335
1336         case "$_file" in
1337         /etc/rc.d/*.sh)                 # no longer allowed in the
                base
1338                 warn "Ignoring old-style startup script $_file"
1339                 ;;
1340         *[~#]|*.OLD|*.bak|*.orig|*,v)   # scratch file; skip
1341                 warn "Ignoring scratch file $_file"
1342                 ;;
1343         *)                              # run in subshell
1344                 if [ -x $_file ]; then
1345                         if [ -n "$rc_fast_and_loose" ]; then
1346                                 set $_arg; . $_file
1347                         else
1348                                 ( trap "echo Script $_file
                                        interrupted >&2 ; kill -QUIT $$" 3
1349                                   trap "echo Script $_file
                                        interrupted >&2 ; exit 1" 2
1350                                   trap "echo Script $_file running
                                        >&2" 29
1351                                   set $_arg; . $_file )
1352                         fi
1353                 fi
1354                 ;;
1355         esac
1356 }
```

Voltando ao **rc**, no Código 8 é feita uma nova checagem para a *KEYWORD firstboot* (linha 119), mas dessa vez ela é retirada da variável *skip_firstboot* (linha 120).

Código 8 – Arquivo /etc/rc: firstboot

```
117 # The firstboot sentinel might be on a newly mounted filesystem; look
        for it
118 # again and unset skip_firstboot if we find it.
119 if [ -e ${firstboot_sentinel} ]; then
120         skip_firstboot=""
121 fi
```

No Código 9 observa-se que uma segunda ordenação é realizada usando todos os *scripts*, incluindo os que estão na nova pasta contidos na variável *local_rc*. E a lista ordenada resultante é armazenada na variável *files*.

Código 9 – Arquivo /etc/rc: reordenação dos scripts

```
123 files=`rcorder ${skip} ${skip_firstboot} /etc/rc.d/* ${local_rc} 2>/
    dev/null`
```

No Código 10 observa-se que há uma iteração sobre os valores de *files* (linha 124) onde todos os *scripts* são executados (linha 129) exceto aqueles que já tinham sido executados antes, ou seja aqueles que estão na variável *_rc_elem_done* (linhas 125 e 126).

Código 10 – Arquivo /etc/rc: execução dos scripts restantes

```
124 for _rc_elem in ${files}; do
125         case "$_rc_elem_done" in
126         *" $_rc_elem "*)        continue ;;
127         esac
128
129         run_rc_script ${_rc_elem} ${_boot}
130 done
```

Ao terminar a execução do **rc** o sistema mostra a tela de login.

2.2.1.2 Tipos de script de inicialização

Existem três tipos de *script* de inicialização: aqueles que criam um *daemon*[4], aqueles que não criam *daemons* mas executam instruções necessárias apenas na hora do *boot* e aqueles que não executam nenhum comando servindo apenas como dependência (chamados de *dummy dependencies*). O Quadro 3 mostra todas as *dummy dependencies* presentes por padrão.

Quadro 3 – Dummy dependencies

Script	Descrição
DAEMON	Usado para garantir que *daemons* de propósito geral iniciem após esse *script*.
FILESYSTEMS	Usado por *scripts* que requerem que os sistemas de arquivos estejam montados antes de sua execução.
LOGIN	Usado pelos *scripts* que não necessitam iniciar antes de algum *script* padrão.
NETWORKING	Usado por *scripts* que requerem conexão de rede funcional antes de sua execução.
SERVERS	Usado por *scripts* padrão que iniciam cedo na sequência de inicialização.

Fonte: descrições retiradas do código fonte de cada *script*.

[4] Também chamado de serviço, é um programa que executa em segundo plano e não possui interface gráfica de usuário.

2.2.2 OpenRC

OpenRC é um sistema de inicialização que funciona em conjunto com o programa **init** provido pelo sistema operacional. Possui licença *2-clause BSD* e tem o objetivo de ser compatível com várias plataformas. É o sistema de inicialização padrão do *Gentoo Linux* (OPENRC, 2016).

OpenRC é baseado em *runlevels*. Um *runlevel* é um estado no qual o sistema pode se encontrar. Cada *runlevel* possui um conjunto de *shell scripts* que devem ser executados ao entrar ou sair do mesmo. Por padrão existem sete *runlevels*, três são internos: *sysinit*, *shutdown* e *reboot*; e quatro definidos pelo usuário: *boot*, *default*, *nonetwork* e *single*. O nome de cada *runlevel* é usado como subdiretório de **/etc/runlevels** (GENTOO FOUNDATION, 2016).

O repositório de *shell scripts* de inicialização localiza-se em **/etc/init.d**. OpenRC não executa todos os *scripts*, somente aqueles que possuem um atalho[5] em um dos subdiretórios de **/etc/runlevels**. Os *scripts* são executados em ordem alfabética, mas havendo informação de dependência, ela é executada antes do *script* corrente (GENTOO FOUNDATION, 2016).

Os *scripts* de serviço são programados na linguagem *shell script*, e usam apenas as funcionalidades definidas no padrão POSIX (*Portable Operating System Interface*). Cada *script* deve possuir as funções *depend* e *start*. Outra função que pode existir é a *stop*. A função *start* serve para iniciar o serviço, a função *stop* serve para parar o serviço e a função *depend* declara as dependências do serviço. Os *scripts* podem declarar novos comandos caso necessário (GENTOO FOUNDATION, 2016).

Dentro da função *depend* são usadas palavras reservadas para declarar as dependências do serviço, como visto no Quadro 4.

Quadro 4 – OpenRC: palavras reservadas e seus efeitos

Palavra reservada	Efeito	Implica dependência?
need P	Inicie P antes do serviço atual.	Sim
use P	Inicie P antes do serviço atual, mas somente se ele existe na *runlevel* atual.	Não
want P	Inicie P antes do serviço atual, independente de qual *runlevel* ele pertence.	Não
before P	Inicie o serviço atual antes de P.	Sim
after P	Inicie o serviço atual após P.	Não
provide P	Permite que várias implementações forneçam um mesmo serviço.	-
keyword P	Permite sobreposições específicas de plataforma.	-

Fonte: Autoria própria

No Quadro 4 a coluna “Implica dependência” indica se um serviço passa a depender do outro.

[5] Também conhecido como *symbolic link*.

O Código 11 mostra a estrutura de um *script* de serviço. No Código 11 o *script* requer a dependência *net*, usa as dependências *logger* e *dns* e provê a dependência *mta* (GENTOO FOUNDATION, 2016).

Código 11 – Exemplo de script do OpenRC

```
#!/sbin/openrc-run

depend() {
    #  (Informacao de dependencia)
    need net
    use logger dns
    provide mta
}

start() {
    #  (Comandos necessarios para iniciar o servico)

    if [ "${RC_CMD}" = "restart" ];
    then
        # Fazer algo em caso de reinicializacao do servico
    fi

    ebegin "Iniciando my_service"
    start-stop-daemon --start --exec /path/to/my_service \
                      --pidfile /path/to/my_pidfile
    eend $?
}

stop() {
    #  (Comandos necessarios para parar o servico)
    ebegin "Parando my_service"
    start-stop-daemon --stop --exec /path/to/my_service \
                      --pidfile /path/to/my_pidfile
    eend $?
}
```

Para melhor performance o OpenRC mantém um cache dos metadados de dependência e o atualiza apenas quando um *script* é modificado.

2.2.3 Launchd

Launchd é para o OS X[6] o que o **init** é para o FreeBSD. É o primeiro processo a ser criado em modo de usuário e é responsável por iniciar (direta e indiretamente) todos os outros processos do sistema. Apareceu pela primeira vez no Mac OS X versão 10.4 possuindo licença *Apple Public Source License* mais tarde foi re-licenciado usando a licença *Apache* versão 2.0. Launchd é iniciado diretamente pelo *kernel* e não pode ser iniciado pelo usuário, mas pode ser controlado pelo programa **launchctl** para iniciar e parar outros serviços. Ele também é iniciado uma vez para cada usuário logado no sistema (LEVIN, 2012).

Cada usuário pode controlar apenas seus serviços e o usuário *root* pode controlar todos os serviços. Existem dois tipos de serviços: o tipo *daemon*, que é um serviço que trabalha em

[6] Sistema operacional da Apple Inc

segundo plano sem interação com o usuário; e o *agent* que é um serviço iniciado apenas quando um usuário loga no sistema. Ao contrário do *daemon* esse pode ter interação com o usuário, podendo até ter uma interface gráfica de usuário. Ambos *daemons* e *agents* são declarados em seus próprios arquivos de configuração usando um formato do tipo XML (*eXtensible Markup Language*). O arquivo deve conter palavras chaves específicas que são reconhecidas pelo Launchd, o formato de arquivo é conhecido como listas de propriedades ou *plists* (LEVIN, 2012).

O Launchd substitui vários serviços conhecidos do UNIX: **init**, assim como visto anteriormente; **crond**, permitindo a execução de programas em intervalos constante; **atd**, permitindo a execução de programas em horários específicos; **inetd** e **xinetd**, permitindo a execução de serviços por demanda; serviços de *log*, monitoramento, entre outros (LEVIN, 2012).

O Código 12 mostra um exemplo de arquivo de configuração (MILLER; ZOVI, 2009).

Código 12 – com.apple.PreferenceSyncAgent.plist

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http:
       //
3  www.apple.com/DTDs/PropertyList-1.0.dtd">
4  <plist version="1.0">
5  <dict>
6          <key>Label</key>
7          <string>com.apple.PreferenceSyncAgent</string>
8          <key>ProgramArguments</key>
9          <array>
10                 <string>/System/Library/CoreServices/
11 PreferenceSyncClient.app/Contents/MacOS/PreferenceSyncClient</string>
12                 <string>--sync</string>
13                 <string>--periodic</string>
14         </array>
15         <key>StartInterval</key>
16         <integer>3599</integer>
17 </dict>
18 </plist>
```

No Código 12 observa-se o arquivo de configuração **PreferenceSyncAgent.plist**. Esse arquivo usa 3 chaves, a chave *Label* (linhas 6 e 7) identifica o serviço para o Launchd (cada *Label* deve ser única), a chave *ProgramArguments* (linhas 8 á 14) é um vetor consistindo da aplicação à ser executada (linhas 10 e 11) assim como os argumentos à serem passados por linha de comando (linhas 12 e 13), a última chave é *StartInterval* (linhas 15 e 16) que indica o tempo de espera entre as execuções do programa. Além dessas chaves exitem outras como *UserName* que indica com qual usuário o serviço deve ser executado, *OnDemand* que indica se o serviço deve ser executado por demanda ou deve ser mantido em execução e *Program* que indica qual programa deve ser executado. Várias outras chaves são suportadas além dessas (MILLER; ZOVI, 2009).

Um serviço pode ser especificado apenas com *Label*, *Program* e *ProgramArguments*, mas apenas isso não é suficiente para executá-lo. Ele deve também especificar quando deve ser executado usando as chaves *RunAtLoad*, *StartInterval*, *StartCalendarInterval*, *StartOnMount*,

WatchPaths, *QueueDirectories*, *KeepAlive*, entre outros. Cada uma dessas chaves designa um tipo de ativação de serviço (MAC DEVELOPER LIBRARY, 2016).

Para programas cliente, a porta (TCP ou UDP) que representa o serviço está sempre disponível e pronta para receber requisições, estando o serviço em execução ou não. Quando um cliente envia uma requisição para a porta, o Launchd pode ter que executar o serviço para que ele possa responder à requisição. Uma vez executado, o serviço pode continuar executando ou pode finalizar-se para liberar os recursos que utiliza. Se o serviço termina, o Launchd pode reiniciá-lo quando uma nova requisição aparecer (MAC DEVELOPER LIBRARY, 2016).

Esse modo de ação simplifica o gerenciamento de dependências entre serviços porque como os serviços são iniciados por demanda, requisições de comunicação não falham se o serviço ainda não foi iniciado. Elas são apenas atrasadas até que o serviço possa processá-las (MAC DEVELOPER LIBRARY, 2016). Isso permite que no momento da inicialização do sistema vários serviços possam ser executados ao mesmo tempo. O gerenciamento de dependências é implícito ao contrário do presente em outros sistemas UNIX, onde cada serviço explicitamente descreve quais outros serviços ele depende.

2.2.4 Systemd

Systemd é um gerenciador de serviços e sistema usado em sistemas operacionais Linux, foi criado pela Red Hat e pode ser distribuído nas licenças GPL (*General Public License*) ou LGPL (*Lesser General Public* License). Como um gerenciar de sistema ele introduz um *daemon* para controle de logging, utilitários para configuração de *hostname*, data, localização, mantém as listas de usuários logados, *containers* e máquinas virtuais, faz resolução de nomes, entre outras funcionalidades (SVISTUNOV et al., 2016).

A configuração é feita usando arquivos chamados *systemd units* onde a extensão do arquivo mostra qual é o seu tipo de ativação. O Quadro 5 mostra as extensões usadas.

Os tipos de ativação de serviços variam conforme a extensão de seu arquivo de configuração. O Systemd suporta ativação à partir de dispositivos (*Device Unit*), onde um serviço é ativado quando um dispositivo em particular torna-se disponível. Suporta ativação por barramento, usando D-BUS (*Desktop BUS*, uma forma de comunicação entre processos). Quando um cliente usa um canal D-BUS o serviço que disponibiliza este canal é acionado. Suporta ativação por caminho do sistema de arquivos (*Path Unit*), onde o serviço é acionado quando um arquivo ou um diretório em específico é alterado. Como visto no Quadro 5 existem vários meios de ativação de serviços (SVISTUNOV et al., 2016).

O utilitário **systemctl** é usado para controlar os serviços, podendo receber os comandos de *start*, *stop*, *status*, *enable*, *disable*, *restart*, *reload*, entre outros.

Os arquivos de configuração contém tipicamente 3 seções: seção [*Unit*] contém opções

Quadro 5 – Extensão das systemd units

Tipo da unidade	Extensão do arquivo	Descrição
Service unit	.service	Um serviço do sistema.
Target unit	.target	Um grupo de systemd units.
Automount unit	.automount	Um ponto de automontagem do sistema de arquivos.
Device unit	.device	Um arquivo de dispositivo reconhecido pelo kernel.
Mount unit	.mount	Um ponto de montagem do sistema de arquivos.
Path unit	.path	Um arquivo ou diretório do sistema de arquivos.
Scope unit	.scope	Um processo criado externamente.
Slice unit	.slice	Um grupo de units organizado hierarquicamente que gerencia processos do sistema.
Snapshot unit	.snapshot	Um estado salvo do gerenciador Systemd.
Socket unit	.socket	Um socket de comunição entre processos.
Swap unit	.swap	Um arquivo ou dispositivo de swap.
Timer unit	.timer	Um temporizador do Systemd.

Fonte: Svistunov et al. (2016).

genéricas independentes do tipo de arquivo; seção [*unit type*] contém as diretivas específicas à unidade usada, onde *unit type* é o nome de um tipo de Unit, exemplo [*Service*]; seção [*Install*] contém informações sobre instalação usadas pelos comandos *enable* e *disable* do **systemctl** (SVISTUNOV et al., 2016). O Quadro 6 mostra as opões da seção *Unit*.

Quadro 6 – Opções importantes da seção [Unit]

Opção	Descrição
Description	Descrição textual da Unit.
Documentation	Provê referências externas de documentação da Unit.
After	Define a ordem na qual as Units são iniciadas. A Unit é iniciada após a Unit especificada em After ser ativada. Ao contrário de Requires, After não ativa a Unit especificada.
Before	Define a ordem na qual as Units são iniciadas. A Unit é iniciada antes que a Unit especificada em Before é ativada. Ao contrário de Requires, Before não ativa a Unit especificada.
Requires	As Units especificadas aqui são ativadas junto com a esta Unit. Se alguma Unit falhar, esta não é ativada.
Wants	Igual à Requires, mas esta Unit é ativada mesmo se as outras falharem.
Conflicts	Configura dependências negativas, ao contrário de Requires.

Fonte: Svistunov et al. (2016).

O Código 13 mostra um exemplo de arquivo de configuração de serviço retirado de Svistunov et al. (2016). Nesse exemplo a seção [*Unit*] descreve o serviço (linha 2), especifica quais serviços este requer (linha 3) e especifica quais serviços não devem ser ativados junto com este (linha 4). Na seção [*Service*] são especificados o tipo de inicialização do serviço (linha 7), o arquivo de PID (para indicar qual o seu número de PID), comandos que devem ser executados antes de iniciar o serviço (linhas 10 e 11), o comando que inicia o serviço (linha 12), o comando usado para recarregar o serviço (linha 13) e o comando usado para parar o serviço (linha 14).

EnviromentFile (linha 9) aponta para a localização onde variáveis de ambiente são definidas para o serviço. A seção [*Install*] lista os grupos que dependem do serviço.

Código 13 – postfix.service

```
1  [Unit]
2  Description=Postfix Mail Transport Agent
3  After=syslog.target network.target
4  Conflicts=sendmail.service exim.service
5
6  [Service]
7  Type=forking
8  PIDFile=/var/spool/postfix/pid/master.pid
9  EnvironmentFile=-/etc/sysconfig/network
10 ExecStartPre=-/usr/libexec/postfix/aliasesdb
11 ExecStartPre=-/usr/libexec/postfix/chroot-update
12 ExecStart=/usr/sbin/postfix start
13 ExecReload=/usr/sbin/postfix reload
14 ExecStop=/usr/sbin/postfix stop
15
16 [Install]
17 WantedBy=multi-user.target
```

2.2.5 Android **init**

O Android é um sistema operacional de código aberto desenvolvido pela Google Inc e tem como público alvo o mercado móvel. Possui dois sistemas inicialização de serviços: o primeiro inicia processos de nível nativo e seu responsável é o programa **init**; e o segundo inicia os processos dentro da máquina virtual Java e seus responsáveis são os programas **servicemanager**, **system_server** e **zygote**. Ambos possuem uma arquitetura específica para o Android. O primeiro pode executar qualquer programa (na sua maioria escritos em C/C++), e inicia apenas as funcionalidades que dão suporte ao próprio sistema operacional para que no fim inicialize o segundo sistema. O segundo sistema é responsável por iniciar a máquina virtual Java (chamada Dalvik) e os serviços escritos especificamente com o SDK (Software Development Kit) do Android (LEVIN, 2014). Esta seção descreve o funcionamento apenas do primeiro sistema.

A inicialização de espaço de usuário começa com o programa **/init**. Ele faz a leitura do arquivo de configuração **/init.rc** no qual são declarados quais serviços devem ser inicializados. Geralmente o arquivo **/init.harware.rc** também é lido pois contém serviços específicos do dispositivo (LEVIN, 2014).

Os arquivos **.rc** são compostos por blocos iniciados pelas palavras *service* e *on*. Blocos iniciados por *on* são chamados de *triggers* e contém comandos que são disparados em resposta à uma ação específica. Blocos *service* definem *daemons*, os quais são executados por **init** (LEVIN, 2014). O bloco *service* pode conter várias opções, que podem ser vistas no Quadro 7.

O Código 14 mostra um exemplo de bloco *service*. Nesse exemplo observa-se que o nome do serviço é *bugreport*, qual o comando será executado, que pertence ao grupo de serviços *main*, que ele não deve ser iniciado (linha 4), que o seu término não precisa ser monitorado (linha

Quadro 7 – Opções do bloco *service*

Opção	Descrição
capability	Suporte ao Linux *capabilities*[a].
class	Define qual grupo o serviço pertence.
console	Define que é um serviço de console. Liga *stdin/stdout/stderr* ao arquivo **/dev/console**.
critical	Define como um serviço crítico. São serviços reiniciados automaticamente, se o serviço parar mais que 4 vezes em 240 segundos, o sistema é reiniciado em modo de recuperação.
disabled	O serviço não será iniciado.
group	Define qual grupo do sistema este serviço pertence.
ioprio	Define a prioridade de entrada/saída do serviço.
keycodes	Define uma sequência de teclas para ativar o serviço.
oneshot	O **init** não precisa monitorar seu término.
onrestart	Lista quais comandos executar quando este serviço reiniciar.
seclabel	Define qual nome *SELinux*[b] aplicar ao serviço.
setenv	Define uma variável de ambiente.
socket	Define se o init deve abrir um *UNIX Domain socket* para este processo.
user	Define qual usuário do sistema este serviço pertence.

Fonte: Levin (2014)

[a] Funcionalidade do Kernel que separa os privilégios do super usuário.
[b] Security-Enhanced Linux: Implementação da arquitetura *Mandatory Access Control*.

5) e qual a sequência de teclas para acioná-lo (linha 6).

Código 14 – Serviço bugreport

```
1 service bugreport /system/bin/dumpstate -d -p -B \
2         -o /data/data/com.android.shell/files/bugreports/bugreport
3     class main
4     disabled
5     oneshot
6     keycodes 114 115 116
```

Fonte: Levin (2014)

O Código 15 mostra um exemplo de serviço que utiliza *socket*. Esse exemplo mostra um tipo de dependência implícita de serviços. Qualquer cliente que utilize o *socket* irá ser bloqueado até que esse serviço esteja executando. Portanto cliente e servidor podem ser iniciados em paralelo sem necessidade de esperar que o serviço esteja ativo para iniciar o cliente. O próprio **init** não possui conhecimento de quem são os clientes, apenas sabe que deve criar o *socket* para esse serviço.

Os dois exemplos vistos são serviços que pertencem ao grupo *main*, existem outros grupos, que são: *core* e *late_start*. A execução dos grupos é sequencial e é dada pela ordem *core*, *main* e *late_start*. O grupo *late_start* é usado por serviços que dependem da partição **/data** mas por padrão não existe nenhum serviço nesse grupo (LEVIN, 2014). A existência desses grupos mostra um tipo de dependência explícita onde um grupo é executado antes do outro.

Código 15 – Serviço mdnsd

```
1  service mdnsd /system/bin/mdnsd
2      class main
3      user mdnsr
4      group inet net_raw
5      socket mdnsd stream 0660 mdnsr inet
6      disabled
7      oneshot
```

Fonte: Levin (2014)

Uma outra maneira de expressar dependência entre serviços pode ser vista no Código 16. Nesse exemplo o serviço **servicemanager** indica que ao ser reiniciado, os serviços **healthd**, **zygote**, **media**, **surfaceflinger**, **inputflinger** e **drm** devem ser reiniciados também. Isso mostra que há uma dependência direta entre esses serviços e **servicemanager**.

Código 16 – Serviço servicemanager

```
1  service servicemanager /system/bin/servicemanager
2      class core
3      user system
4      group system
5      critical
6      onrestart restart healthd
7      onrestart restart zygote
8      onrestart restart media
9      onrestart restart surfaceflinger
10     onrestart restart inputflinger
11     onrestart restart drm
```

Fonte: Levin (2014)

2.3 SÍNTESE DAS ALTERNATIVAS DE INICIALIZAÇÃO

O tipo de gerenciamento de dependências e o formato dos arquivos são as principais diferenças entre as soluções descritas. Em relação ao formato, o rc e o OpenRC usam *shell scripts* para a inicialização, e esses arquivos descrevem ações a serem tomadas. No caso do Launchd, Systemd e Android os arquivos são vistos como configuração e descrevem apenas o que devem ser feito e não como. Em relação ao gerenciamento de dependências o rc e o OpenRC usam um tipo explícito onde cada *script* descreve quais outros *scripts* ele requer. No Launchd, Systemd e Android os serviços podem depender de outros serviços, mas nesses sistemas a forma de se expressar uma dependência é expandida, eles podem também depender de *sockets*, canais D-BUS ou de vários outros meios que outros serviços criam.

As soluções descritas resultam de um longo processo de evolução (OpenRC existe desde 2007, Systemd desde 2010, Launchd desde 2005 e Android desde 2008). Esse processo implica em mudanças estruturais no sistema correspondente, mudanças essas realizadas em parceria com uma equipe de mantenedores. O processo de adequação estrutural do FreeBSD para

um desses sistemas demanda um tempo que foge ao escopo desse trabalho.

Uma outra barreira para a adoção dessas soluções são as licenças usadas. Systemd pode ser redistribuído com as licenças GPL ou LGPL mas possui partes do sistema com licenças incompatíveis com a licença BSD como é o caso da biblioteca udev que só é distribuída com a licença GPL.

O Launchd tem o mesmo problema. Seu código é distribuído com a licença Apache mas partes de seu código possui a licença CDDL (*Common Development and Distribution License*). Android é distribuído na licença Apache 2.0 com partes de seu código na licença GPL.

Usando uma licença BSD, o OpenRC é o único com licença compatível ao FreeBSD.

2.4 PROCESSOS E THREADS

Em sistemas operacionais de usuário único e processo único, os programas são executados um após o outro. Neste ambiente, a execução de um programa não pode ser quebrada por outro programa e voltar a ser executada sem problemas, exceto por interrupções do sistema. Isso quer dizer que o programa ocupa a CPU e a memória sem compartilhar com outro programa até que sua execução termina (LIU; YUE; GUO, 2011).

Em sistemas operacionais multiusuário e multiprocesso, não só a execução de vários programas podem compartilhar a CPU, como também vários usuários podem interagir com o sistema (LIU; YUE; GUO, 2011). Esse é o caso do FreeBSD.

No FreeBSD um processo é um programa em execução. Um processo tem um espaço de endereçamento contendo um mapeamento do código e das variáveis globais do programa. Também possui um conjunto de recursos do *kernel* que são controlados por chamadas de sistema. Cada processo tem ao menos uma e possivelmente várias *threads* que executam seu código (MCKUSICK; NEVILLE-NEIL; WATSON, 2014).

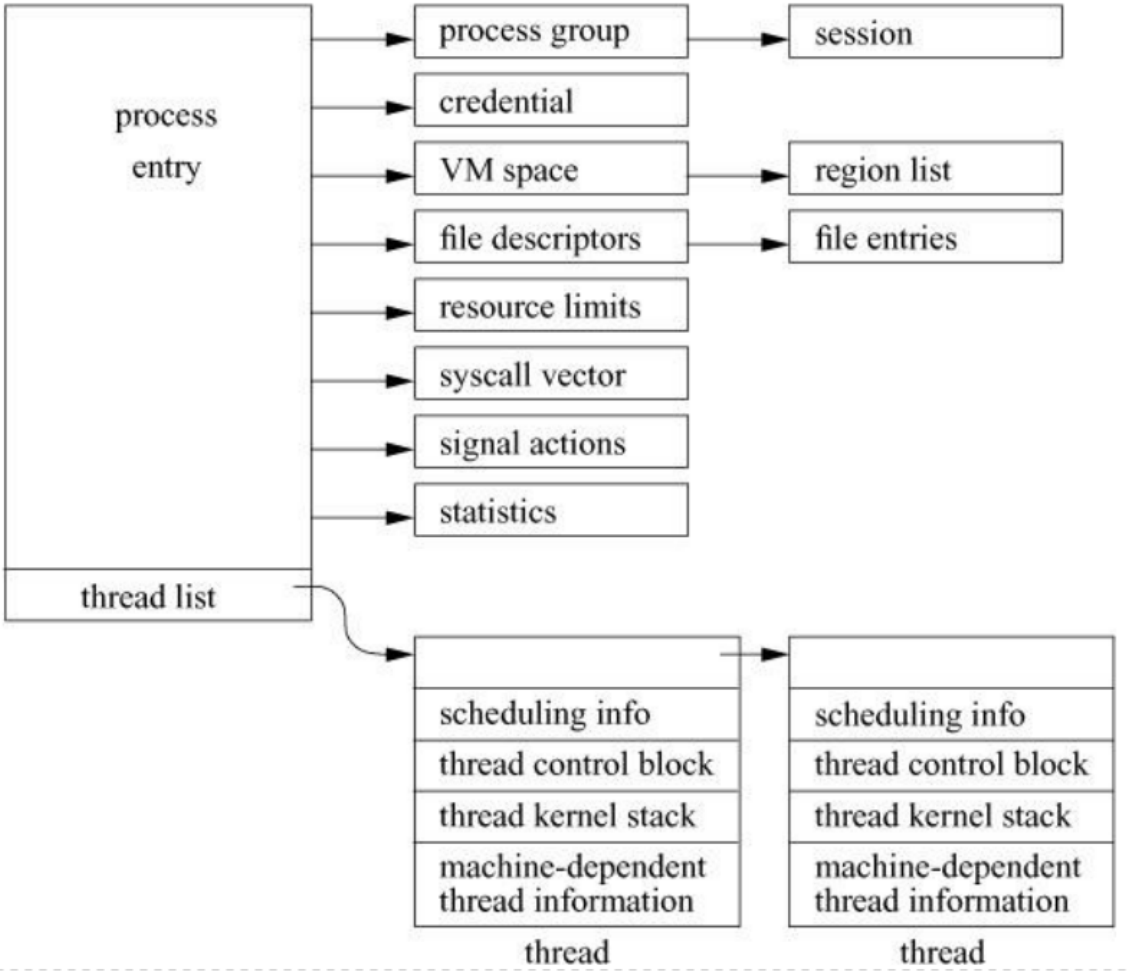
No código fonte do FreeBSD os processos são representados pela estrutura *process_entry* e as *threads* são representadas pela estrutura *thread*. A Figura 3 mostra a relação entre as duas estruturas. Como pode ser visto, cada processo contém uma lista de *threads*.

O *kernel* cria a ilusão de execução concorrente de processos pelo escalonamento de recursos do sistema entre o conjunto de processos que estão prontos para executar. Em sistemas com mais de um núcleo de processamento, múltiplas *threads* do mesmo ou de diferentes processos podem executar concorrentemente (MCKUSICK; NEVILLE-NEIL; WATSON, 2014).

As *threads* de um processo operam em modo usuário ou modo *kernel*. Em modo usuário ela executa com a máquina em um modo de proteção não privilegiado. Quando requisita um serviço do sistema operacional com uma chamada de sistema, ela muda para um modo de proteção privilegiado e opera em modo *kernel* (MCKUSICK; NEVILLE-NEIL; WATSON, 2014).

O mecanismo de troca de contexto ajuda a criar a ilusão de execução concorrente de

Figura 3 – Relação entre processo e thread



Fonte: McKusick, Neville-Neil e Watson (2014)

múltiplos processos ou programas. Ele é definido pela troca de contexto entre *threads* de um mesmo ou de diferentes processos. Um outro mecanismo também é provido para escalonar a execução das *threads*, ou seja, decidir qual será a próxima à ser executada (MCKUSICK; NEVILLE-NEIL; WATSON, 2014).

No escalonador padrão do FreeBSD, a prioridade de um processo é periodicamente recalculada baseado em vários parâmetros, como o tempo total da CPU utilizado, a quantidade de recursos de memória ele requer para execução, entre outros. Esse escalonador tende a favorecer mais os programas interativos do que programas com grandes períodos de execução. Programas interativos tendem a exibir um curto tempo de atividade seguido de períodos de inatividade ou de entrada/saída (MCKUSICK; NEVILLE-NEIL; WATSON, 2014).

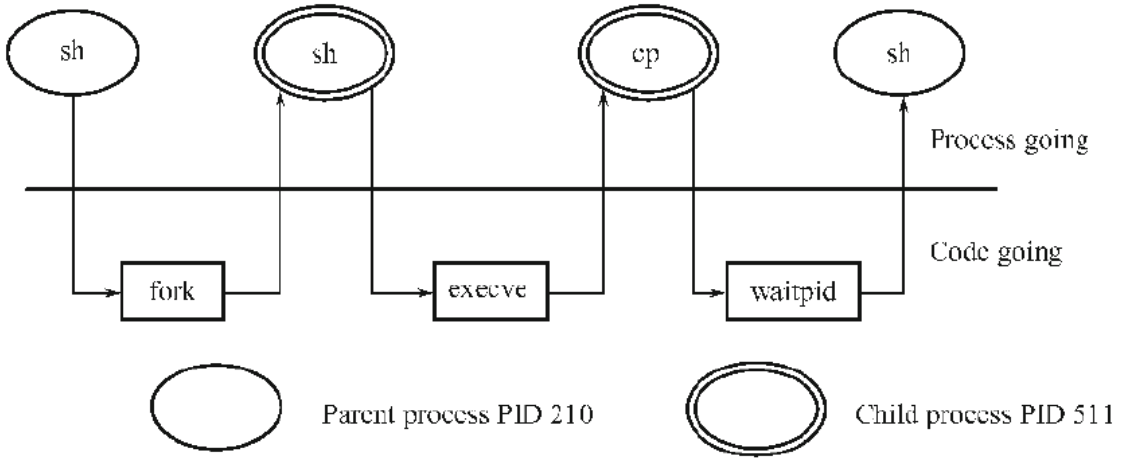
2.4.1 Como os processos são criados?

Novos processos são criados com as chamadas de sistema da família *fork*. *fork* cria uma cópia exata do processo original, incluindo todos os registradores, descritores de arquivos, etc. Todas as variáveis têm valores idênticos no momento do *fork*, mas como o processo pai é copiado para criar o processo filho, alterações subsequentes às variáveis de um não alteram as variáveis do outro. Portanto pai e filho seguem em caminhos separados. Para identificar qual processo é o pai e qual é o filho, o *fork* retorna zero para o filho e um valor igual ao PID do filho para o pai (LIU; YUE; GUO, 2011).

Em casos onde o processo filho deve executar um programa diferente do pai, ele faz a chamada de sistema *execve*. *execve* causa o carregamento de um novo programa sobrepondo aquele sendo atualmente executado pelo processo. Caso o pai precise esperar o filho terminar de executar, ele pode executar a chamada de sistema *waitpid* (LIU; YUE; GUO, 2011).

A Figura 4 mostra um exemplo de chamada das três funções. Quando o usuário digita *cp file1 file4* na linha de comando do *shell* **sh**, o *shell* cria um processo filho usando *fork*. Depois o processo filho sobrepõe seu programa com o programa **cp** usando *execve*. O *shell* espera o processo do **cp** terminar usando *waitpid*. Por fim o processo pai volta a sua execução normal.

Figura 4 – Passos da execução do comando *cp* no *shell*



Fonte: Liu, Yue e Guo (2011)

2.5 MÉTODO DE COMPARAÇÃO DE MÉDIAS

Para realizar a paralelização foi criada uma solução alternativa ao sistema atual de inicialização. Foram criados alguns programas e realizadas algumas alterações no sistema atual. No entanto, é necessário comparar os dois sistemas para saber se houve algum impacto no tempo médio despendido na inicialização.

Para a comparação dos tempos médios foi adotado o teste **z**. Segundo LARSON e FARBER (2010) o teste **z** é usado para testar uma afirmação comparando as médias de duas populações. Para LARSON e FARBER (2010), três condições são necessárias para desempenhar tal teste:

1. As amostras devem ser selecionadas aleatoriamente;
2. As amostras devem ser independentes;
3. Cada tamanho de amostra deve ser pelo menos 30 ou, se não, cada população deve ter uma distribuição normal com desvio padrão conhecido.

Duas amostras são independentes se a amostra selecionada de uma das populações não é relacionada à amostra selecionada da segunda população. Duas amostras são dependentes se cada membro de uma amostra corresponde a um membro da outra amostra (LARSON; FARBER, 2010).

As amostras coletadas neste trabalho são aleatórias e independentes, mas como não se sabe o desvio padrão das populações, foram coletadas amostras com tamanho maior que 30, satisfazendo as três condições do teste.

LARSON e FARBER (2010) definem instruções para usar o teste **z**:

1. Expresse a afirmação matematicamente. Identifique as hipóteses nula e alternativa;
2. Especifique o nível de significância;
3. Faça um esboço da distribuição amostral;
4. Determine o(s) valor(es) crítico(s);
5. Determine a(s) região(ões) de rejeição;
6. Encontre a estatística do teste padronizado;
7. Tome a decisão de rejeitar ou falhar em rejeitar a hipótese nula;
8. Interprete a decisão no contexto da afirmação original.

As hipóteses podem ser expressas de três formas:

1. $H_0 : \mu_1 = \mu_2$

   $H_a : \mu_1 \neq \mu_2$

2. $H_0 : \mu_1 \geq \mu_2$

   $H_a : \mu_1 < \mu_2$

3. $H_0 : \mu_1 \leq \mu_2$

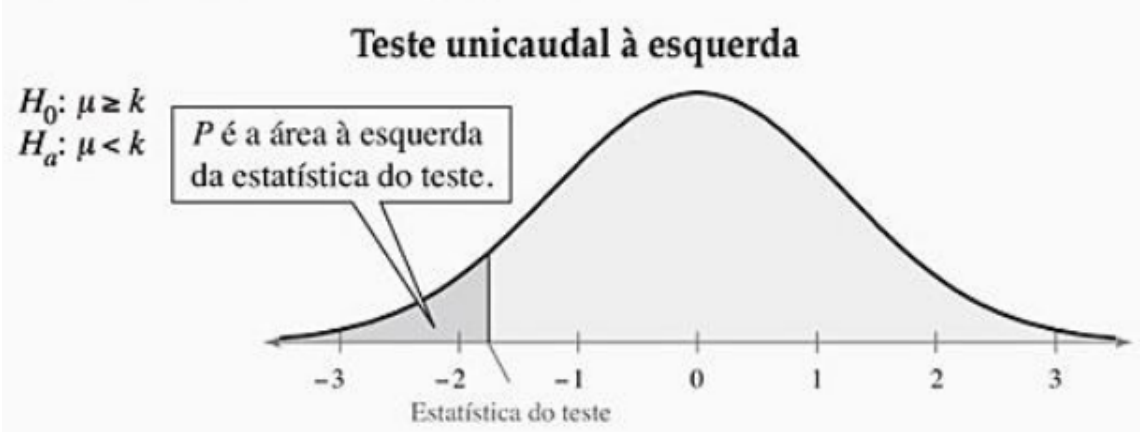
   $H_a : \mu_1 > \mu_2$

A hipótese nula é representada por H_0 e a hipótese alternativa é representada por H_a. Na forma 1 a hipótese nula é de que “as médias são iguais”, na forma 2 é de que “a média da primeira população é maior ou igual à da segunda” e na forma 3 é de que “a média da primeira população é menor ou igual à da segunda”.

Existem três tipos de teste: teste bicaudal, teste unicaudal à esquerda e teste unicaudal à direita. Se a hipótese alternativa H_a contém o símbolo de menos que (<), o teste de hipótese será o teste unicaudal à esquerda, mostrado na Figura 5. Se a hipótese alternativa H_a contém o símbolo maior que (>), o teste de hipótese será o teste unicaudal à direita, mostrado na Figura 6. Se a hipótese alternativa H_a contém o símbolo de não igualdade ($\neq$), o teste de hipótese será o teste bicaudal, mostrado na Figura 7.

O valor P de um teste de hipótese (visto nas figuras) depende da natureza do teste (se é bicaudal, bicaudal à esquerda ou direita), e as áreas apontadas por P são chamadas de áreas de rejeição. A estatística do teste aponta para os valores críticos. Se o valor de z cair na área de rejeição, então a hipótese nula deve ser rejeitada, se não, ela não pode ser rejeitada.

Figura 5 – Teste unicaudal à esquerda

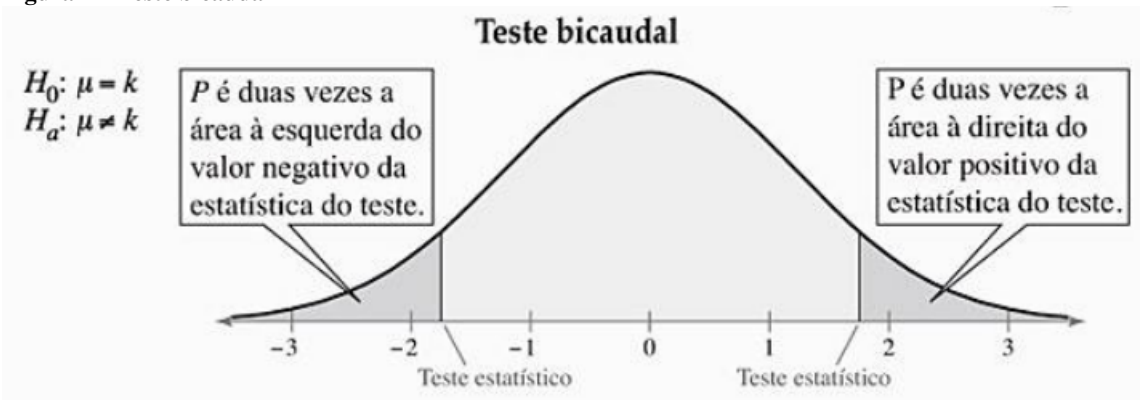


Fonte: (LARSON; FARBER, 2010)

Figura 6 – Teste unicaudal à direita

Fonte: (LARSON; FARBER, 2010)

Figura 7 – Teste bicaudal



Fonte: (LARSON; FARBER, 2010)

A Tabela 1 mostra os valores críticos para os níveis de confiança mais usados.

Tabela 1 – Níveis de confiança mais usados

Nível de confiança	Z_c
90%	1,645
95%	1,96
99%	2,575

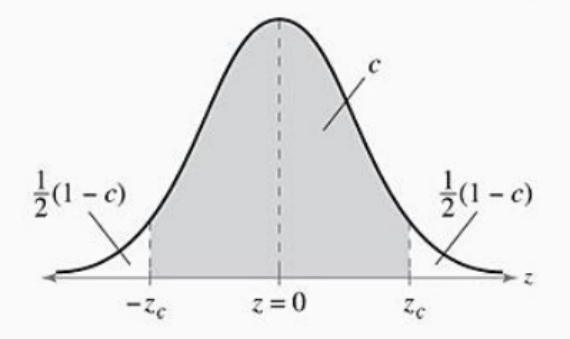
Fonte: (LARSON; FARBER, 2010)

Em um teste de hipótese, o nível de significância é a probabilidade máxima permissível para cometer um erro tipo I. Um erro tipo I ocorre se a hipótese nula for rejeitada quando é verdadeira. Um erro tipo II ocorre se a hipótese nula não for rejeitada quando é falsa. O nível de significância é denotado por α. Configurando-o com um valor pequeno a probabilidade de rejeitar uma hipótese nula verdadeira torna-se pequena. Os três níveis de significância comumente usados são $\alpha = 0,10$, $\alpha = 0,05$ e $\alpha = 0,01$ para níveis de confiança de 90%, 95% e 99%, respectivamente (LARSON; FARBER, 2010).

Pelo Teorema do Limite Central, quando $n > 30$ (amostra maior que trinta), a distribuição de amostragem das médias amostrais é uma distribuição normal. O nível de confiança

c é a área sob a curva normal padrão entre os valores críticos, $-z_c$ e z_c (LARSON; FARBER, 2010). Na Figura 8 pode-se ver que c é a porcentagem da área sob a curva normal entre $-z_c$ e z_c. A área remanescente é $1-c$, então a área em cada cauda é $1/2(1-c)$. Por exemplo, se $c = 90\%$, então 5% da área está à esquerda de $-z_c = 1,645$ e 5% está à direita de $z_c = 1,645$.

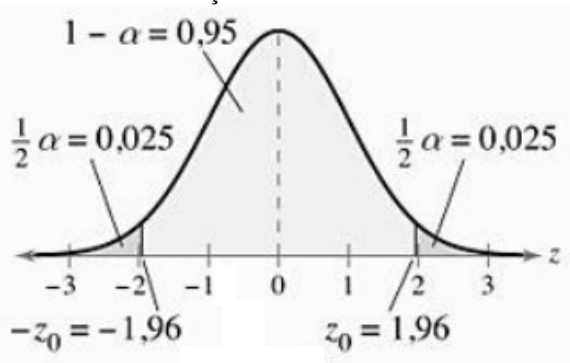
Figura 8 – Gráfico da distribuição normal



Fonte: (LARSON; FARBER, 2010)

A Figura 9 mostra um exemplo de distribuição com $\alpha = 0,05$, os valores críticos são $-z_0 = -1,96$ e $z_0 = 1,96$ e as regiões de rejeição são $z < -1,96$ e $z > 1,96$.

Figura 9 – Áreas de rejeição no gráfico da distribuição normal



Fonte: (LARSON; FARBER, 2010)

A estatística de teste padronizado é:

$z = \frac{(\bar{x}_1 - \bar{x}_2) - (\mu_1 - \mu_2)}{\sigma_{\bar{x}_1 - \bar{x}_2}}$

onde

$\sigma_{\bar{x}_1 - \bar{x}_2} = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}$,

$\bar{x}_1$ e $\bar{x}_2$ são as médias amostrais, s_1^2 e s_2^2 são as variâncias amostrais, n_1 e n_2 são os tamanhos das amostras. μ_1 e μ_2 são as médias populacionais, porém assume-se que $\mu_1 - \mu_2 = 0$ (LARSON; FARBER, 2010).

A tomada de decisão se baseia no valor de z. Se o valor de z estiver na região de rejeição então rejeita-se a hipótese nula, se não, a hipótese nula não pode ser rejeitada. O último passo

é interpretar a decisão conforme o contexto e as hipóteses construídas (LARSON; FARBER, 2010).

Em um exemplo de aplicação do teste z, deseja-se comparar as medias de duas populações. Obtendo uma amostra de cada população descobriu-se na primeira amostra que $\bar{x}_1 = 8$ e $s_1^2 = 0,5$ e na segunda amostra que $\bar{x}_2 = 9$ e $s_2^2 = 0,6$, com amostras de tamanho $n = 40$. Observando os dados surge a hipótese de que $\mu_1 < \mu_2$, portanto a representação seria $H_0 : \mu_1 \geq \mu_2$ e $H_a : \mu_1 < \mu_2$, e o teste de hipótese é o teste unicaudal à esquerda.

Primeiramente, calcula-se o erro padrão:

$\sigma_{\bar{x}_1 - \bar{x}_2} = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}, = \sqrt{\frac{0,5}{40} + \frac{0,6}{40}} = 0,1658.$

Em seguida calcula-se o valor de z:

$z = \frac{(\bar{x}_1 - \bar{x}_2) - (\mu_1 - \mu_2)}{\sigma_{\bar{x}_1 - \bar{x}_2}}$,

assume-se que $\mu_1 - \mu_2 = 0$,

$\frac{(8-9)-(0)}{0,1658} = -6.0313.$

Para o nível de confiança de 99%, o valor crítico é $-z_c = 2,575$. Como o valor $z = -6.0313$ é menor que $-z_c$, ele está na área de rejeição. Com isso, a hipótese nula deve ser rejeitada e a interpretação dessa decisão é dada: com 99% de confiança, há evidencia suficiente para apoiar a afirmação de que a média da primeira população μ_1 é menor que a média da segunda população μ_2.

3 IMPLEMENTAÇÃO

A solução foi implementada utilizando a linguagem de programação C. Foram construídos dois programas **rcpar** e **rcorder2**, com os códigos fonte **rcpar.c** e **rcorder2.c** respectivamente e também houve uma modificação do **/etc/rc**. Os códigos fonte do **rcpar** e do **rcorder2** podem ser vistos no apêndice D. Eles também podem ser vistos no repositório <github.com/haneiko/tcc2>.

O **rcorder2** faz a ordenação e agrupamento de todos os *scripts* ignorando aqueles que não devem ser iniciados. O **rcpar** executa em paralelo todos os *scripts* repassados por linha de comando. Os Quadros 8 e 9 (apêndice C) mostram as saídas dos programas **rcorder2** e **rcorder** (o programa atualmente utilizado para ordenar os *scripts* no FreeBSD). Como pode ser visto, o **rcorder2** faz um agrupamento de *scripts* separando-os com a palavra *END*, e ele garante que todos os *scripts* dentro de um grupo podem ser executados em paralelo. Assim como o **rcorder**, ele também utiliza a ordenação topológica. Cada grupo é repassado para o **rcpar**, e este executa em paralelo todos os *scripts* do grupo.

Em ordem, as ações que o programa **rcorder2** executa:

- Carrega as informações de cada *script* em uma lista.
- Remove os *scripts* que devem ser ignorados.
- Transforma os valores do campo *BEFORE* em valores do campo *REQUIRE*.
- Escreve o nome dos *scripts* que não tem dependência.
- Remove os *scripts* que não tem dependência.
- Retira dos *scripts* restantes a dependência daqueles que foram removidos.
- Repete os três passos anteriores até a lista esgotar.

O **/etc/rc** tem o trabalho de receber do **rcorder2** os agrupamentos e repassá-los para o **rcpar** como argumentos na linha de comando.

Em ordem, as ações que o programa **rcpar** executa:

- Cria um processo para cada *script* repassado pela linha de comando.
- Espera o término dos processos criados.
- Cada processo criado executa um dos *scripts*.

3.1 DETALHES DE IMPLEMENTAÇÃO

Esta seção mostra os detalhes de implementação do **rcorder2**, **rcpar** e das modificações realizadas no **/etc/rc**.

3.1.1 /etc/rc

O Código 17 mostra as modificações feitas no arquivo **/etc/rc**. A primeira mudança é que agora os *scripts* são ordenados pelo **rcorder2** (linha 93). Ele recebe os mesmos parâmetros que o **rcorder**: as variáveis *skip* e *skip_firstboot*, que contém valores usados para ignorar os *scripts* que não devem ser executados; e os nomes de todos os *scripts* das pastas **/etc/rc.d** e **/usr/local/etc/rc.d**.

Após a execução do **rcorder2** a variável *files* contém agora a saída desse programa. Como visto no Quadro 9 (apêndice C), a saída possui elementos que separam os *scripts* em grupos. A linha 95 itera sobre todos os elementos da variável *files* verificando se o elemento é igual à palavra *END* (linhas 96 e 97). Caso não seja igual (linha 101), então o elemento é acumulado na variável *to_exec* (linha 102). Se for igual então o **rcpar** é executado com os argumentos *_boot* e *to_exec* (linha 98), onde *to_exec* contém todos os *scripts* de um dos agrupamentos. Em seguida *to_exec* é esvaziado (linha 99). A variável *_boot* é repassada ao **rcpar** com a opção *-t* e contém um valor que deve ser repassado como argumento para todos os *scripts*, o código que determina esse valor não foi modificado.

Código 17 – Arquivo /etc/rc modificado

```
92  local_=`ls /usr/local/etc/rc.d/* 2>/dev/null`
93  files=`rorder2 ${skip} ${skip_firstboot} /etc/rc.d/* ${local_}`
94
95  for _rc_elem in ${files}; do
96          case ${_rc_elem} in
97          END)
98                  rcpar -t ${_boot} ${to_exec}
99                  to_exec=""
100         ;;
101         *)
102                 to_exec="${to_exec} ${_rc_elem}"
103         ;;
104         esac
105 done
```

3.1.2 rcorder2

O código do **rcorder2** pode ser divido em três partes:

1. Inicialização da lista de scripts;
2. Criação das tabelas; e
3. Execução do algoritmo.

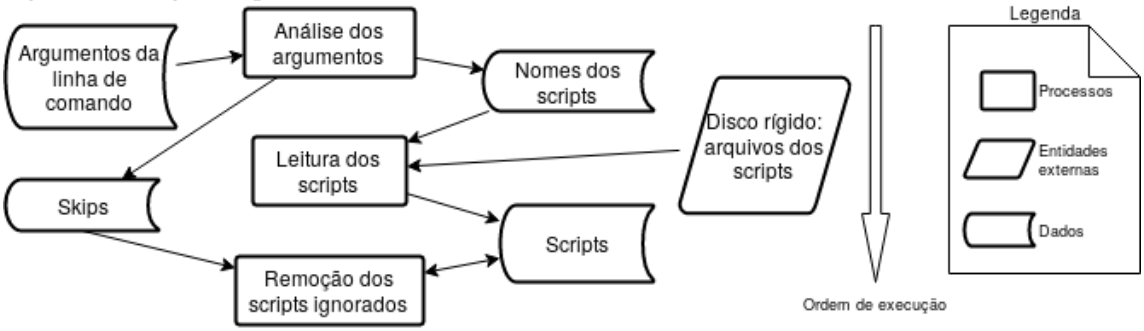
A primeira parte foca em construir a lista de *scripts* que devem ser executados. A Figura 10 mostra o diagrama da parte 1.

O programa recebe como parâmetro da linha de comando o nome dos *scripts* e uma lista de palavras. A lista de palavras é a mesma vista no Quadro 2. O programa inicia fazendo uma análise dos parâmetros para então separá-los em duas listas. No código fonte do **rcorder2**, as palavras são colocadas em uma lista chamada *skips* enquanto os nomes continuam na lista original.

Em seguida é feita a leitura dos *scripts* a partir dos nomes recebidos. Os *scripts* são analisados para que se possa extrair apenas as expressões que indicam as dependências. Essas expressões foram vistas no Quadro 1. No código fonte a leitura é feita pela função *parse_scripts*. O resultado dessa leitura/análise é uma lista de *struct script* de nome *scripts*. Essa *struct* contém o caminho do *script* e todas as expressões (Quadro 1) que ele possui.

Finalizando a primeira parte do programa está a remoção dos *scripts* que devem ser ignorados. A partir da lista *skips* é decidido quais *scripts* devem ser ignorados, e portanto removidos da lista *scripts*. No código fonte a remoção é feita pela função *del_skipped_scripts*.

Figura 10 – Diagrama: parte 1 do rcorder2



Fonte: Autoria própria

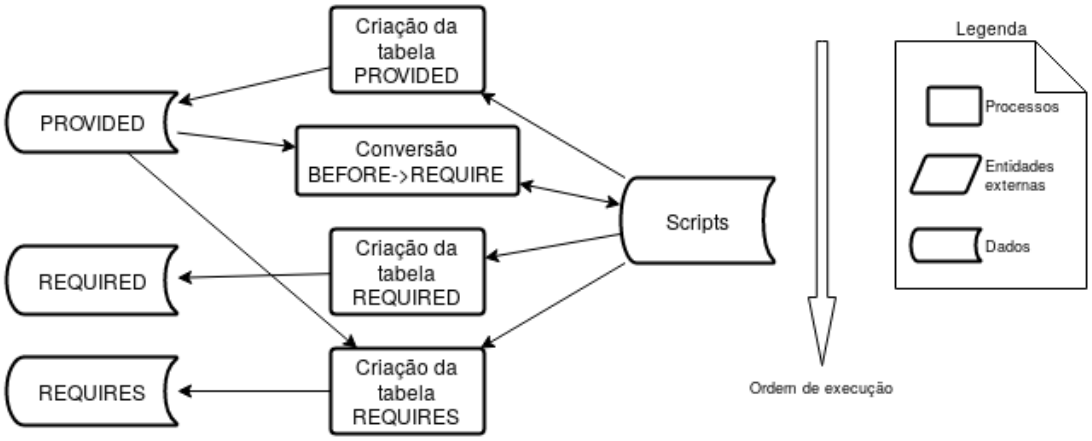
A segunda parte do programa foca em criar tabelas de dispersão para acesso rápido aos *scripts*. A Figura 11 mostra o diagrama da parte 2. Nessa parte são criadas três tabelas de dispersão: *provided*, *required* e *requires*.

A primeira tabela a ser criada é *provided*. Essa tabela serve para acessar todos os *scripts* que proveem um certo nome. O nome é a chave da tabela e os *scripts* que proveem esse nome são os valores. Por ser uma tabela de dispersão, pesquisar por valores tem complexidade O(1), enquanto que pesquisar na lista tem complexidade O(n), onde n é o número de itens na lista. No código fonte a tabela é criada nas linhas 110 à 116.

Após criar a tabela *provided*, o programa a usa para realizar a conversão dos valores de *BEFORE* para os valores de *REQUIRE*. Essas são duas expressões vistas no Quadro 1. Os valores *BEFORE* são nomes de *scripts* que devem ser iniciados depois do *script* corrente. Isso que dizer que o *script* dependido contém a informação de dependência. Eles são transformados em valores *REQUIRE* para que o próprio *script* dependente contenha a informação de dependência. No código fonte, a conversão é feita nas linhas 118 à 129.

Após a conversão são criadas as tabelas *required* e *requires*. A tabela *required* serve para acessar os *scripts* que dependem de um nome. O nome é a chave da tabela e os *scripts* que dependem desse nome são os valores. No código fonte, a tabela *required* é criada nas linhas 131 à 137. A tabela *requires* serve para acessar os *scripts* que um nome requer. O nome é a chave da tabela e os *scripts* que esse nome requer são os valores. No código fonte, a tabela *requires* é criada nas linhas 139 à 154.

Figura 11 – Diagrama: parte 2 do rcorder2



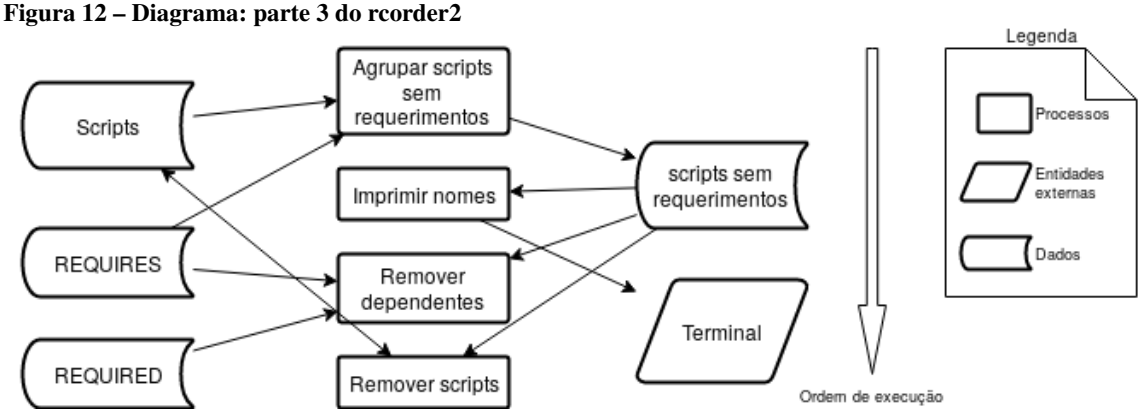
Fonte: Autoria própria

A última parte do programa realiza a execução do algoritmo em si. A Figura 12 mostra o diagrama da parte 3.

O algoritmo realiza o agrupamento e a ordenação dos *scripts*, escrevendo os nomes na saída padrão do programa. Os *scripts* são agrupados de forma que não haja dependências dentro do grupo, permitindo a execução paralela dos mesmos.

O algoritmo inicia em uma estrutura de repetição que só pára quando não houver mais itens na lista *scripts*. Sue primeira tarefa é agrupar os *scripts* que não possuem requerimentos/dependências. Em seguida, o nome desses *scripts* é escrito na saída padrão (por padrão é o terminal, mas como visto anteriormente, o **rc** captura a saída em uma variável de *shell*). Após isso, são removidas as dependências que outros *scripts* possuem em relação ao grupo corrente. Por último, os itens do grupo são removidos da lista *scripts*. Todo o processo descrito continua até que a lista *scripts* esteja vazia. No código fonte, o algoritmo está expresso nas linhas 156 à 168. Após o término do algoritmo, o programa se encerra.

Figura 12 – Diagrama: parte 3 do rcorder2



Fonte: Autoria própria

3.1.3 rcpar

O Código 25 (apêndice D) mostra o código fonte do **rcpar**. O programa recebe vários argumentos por linha de comando. Na linguagem de programação C eles estão na variável *argv* e a quantidade está na variável *argc* (linha 12). O primeiro parâmetro é recebido pela opção *-t* que é capturado pelo código das linhas 20 à 26 e é armazenado na variável *start_type* (linha 23). Em seguida esse parâmetro é retirado das variáveis *argc* e *argv* (linhas 28 e 29). Ele deve ser repassado à todos os *scripts*.

O restante dos parâmetros é formado pela lista de *scripts*. A linha 31 itera sobre todos os parâmetros restantes. A cada iteração, um novo processo é criado usando *fork* (linha 33), caso o processo filho não possa ser criado então o programa é abortado (linha 34). Na linha 35 há uma verificação do valor de retorno de *fork*, ele retorna 0 no processo filho e o valor do PID do filho no processo pai. Portanto o processo pai continua com a próxima iteração, criando um processo para cada *script*, e os filhos entram nessa condicional.

Na linha 36 cada filho executa a função *execl*. Os parâmetros passados à essa função são:

- */bin/sh*: esse é o programa que irá sobrepor o programa do processo atual;
- *sh*: o primeiro parâmetro de um programa é sempre o seu próprio nome;
- *argv[i]*: um dos *scripts* repassados por linha de comando;
- *start_type*: o argumento que deve ser passado ao *script*;
- *(char*)NULL*: como a função *execl* desconhece o número de argumentos, *(char*)NULL* é repassado para indicar o fim da lista.

No término da estrutura de repetição da linha 31, o processo pai espera, com a chamada da função *wait*, todos os filhos terminarem sua execução (linha 41), para em seguida terminar sua própria execução (linha 43).

4 AVALIAÇÃO DE DESEMPENHO

Nesse trabalho a avalização de desempenho realizada foi a comparação dos tempos despendidos na inicialização entre a solução atual do FreeBSD e a solução proposta. A comparação foi realizada em três máquinas distintas, uma com 1 núcleo, uma com 2 núcleos e uma com 4 núcleos de processamento. A versão utilizada do FreeBSD foi a 11.1. Por questão de compatibilidade, a máquina de 1 núcleo foi usada com a versão I386 e as outras utilizaram a versão AMD64.

Em uma etapa preliminar do trabalho houve a utilização de uma máquina virtual para a realização das comparações, mas devido à incerteza dos efeitos que teria nos tempos despendidos, essa medições foram descartadas.

A avaliação foi feita em três etapas. A primeira obteve os tempos de uma instalação padrão do FreeBSD. A segunda obteve os tempos de uma instalação com a solução proposta instalada. Os dois conjuntos de tempos formam as amostras de cada população. Cada amostra teve tamanho $n = 60$. A terceira etapa foi a aplicação das fórmulas do teste z. O método de instalação e coleta automatizada dos tempos pode ser visto no apêndice A.

A instalação padrão possui apenas os *scripts* que já vêm com o sistema operacional, o que quer dizer que não foi instalado nenhum outro *software* que adiciona mais *scripts*.

As duas instalações tiveram o arquivo **/etc/rc** alterado da seguinte forma: duas *timetags* são gravadas nas variáveis *BEGIN* e *END*, antes e depois, respectivamente, da ordenação e execução de todos os *scripts*. A diferença entre as *timetags* denota o tempo decorrido e é armazenada em um arquivo. Na instalação padrão é no arquivo **/root/rc_freebsd.times** e após implantar a solução é no arquivo **/root/rc_rcpar.times**.

O Código 18 mostra o **/etc/rc** modificado para coletar os dados da instalação padrão. Foram adicionadas as linhas 92, 133, 134 e 135. As linhas 92 e 133 gravam *timetags* nas variáveis *BEGIN* e *END*, respectivamente. A linha 134 grava a diferença entre as *timetags* no arquivo **/root/rc_freebsd.times** e a linha 135 desaloca as variáveis *BEGIN* e *END*.

Código 18 – Arquivo /etc/rc: instalação padrão

```
92  BEGIN=`mtime`
93  # Do a first pass to get everything up to $early_late_divider so that
94  # we can do a second pass that includes $local_startup directories
95  #
96  files=`rcorder ${skip} ${skip_firstboot} /etc/rc.d/* 2>/dev/null`
97
98  _rc_elem_done=' '
99  for _rc_elem in ${files}; do
100         run_rc_script ${_rc_elem} ${_boot}
101         _rc_elem_done="${_rc_elem_done}${_rc_elem} "
102
103         case "$_rc_elem" in
104         */${early_late_divider})        break ;;
105         esac
106 done
107
```

```
108 unset files local_rc
109
110 # Now that disks are mounted, for each dir in $local_startup
111 # search for init scripts that use the new rc.d semantics.
112 #
113 case ${local_startup} in
114 [Nn][Oo] | '') ;;
115 *)      find_local_scripts_new ;;
116 esac
117
118 # The firstboot sentinel might be on a newly mounted filesystem; look
        for it
119 # again and unset skip_firstboot if we find it.
120 if [ -e ${firstboot_sentinel} ]; then
121         skip_firstboot=""
122 fi
123
124 files=`rcorder ${skip} ${skip_firstboot} /etc/rc.d/* ${local_rc} 2>/
      dev/null`
125 for _rc_elem in ${files}; do
126         case "$_rc_elem_done" in
127         *" $_rc_elem "*)        continue ;;
128         esac
129
130         run_rc_script ${_rc_elem} ${_boot}
131 done
132
133 END=`mtime`
134 echo "$END-$BEGIN" | bc -l >> "/root/rc_freebsd.times"
135 unset BEGIN END
```

O Código 19 mostra o **/etc/rc** modificado para coletar os dados com a solução implantada. Observa-se que esse é o mesmo Código 17, mas com as linhas 92, 111, 112 e 113 adicionadas para a coleta de tempos e que o arquivo de saída agora é o **/root/rc_rcpar.times** (linha 112).

Código 19 – Arquivo /etc/rc: solução implantada

```
92  BEGIN=`mtime`
93
94  local_=`ls /usr/local/etc/rc.d/* 2>/dev/null`
95  files=`rcorder2 ${skip} ${skip_firstboot} /etc/rc.d/* ${local_}`
96
97  for _rc_elem in ${files}; do
98          case ${_rc_elem} in
99          END)
100                 rcpar -t ${_boot} ${to_exec}
101                 to_exec=""
102         ;;
103         *)
104                 to_exec="${to_exec} ${_rc_elem}"
105         ;;
106         esac
107 done
108
109 unset files local_rc
110
111 END=`mtime`
112 echo "$END-$BEGIN" | bc -l >> "/root/rc_rcpar.times"
113 unset BEGIN END
```

A *timetag* nada mais é do que um tempo medido em microsegundos retirado do programa **mtime**. O Código 20 mostra o código fonte do programa **mtime**. A parte principal do código é a chamada da função *gettimeofday* (linha 8). Ela retorna no primeiro parâmetro o tempo em segundos e microsegundos desde o *Epoch*, uma *timetag* que representa a data de 1 de janeiro e 1970 (também conhecida como *Posix time* ou *Unix time*). O primeiro parâmetro é uma *struct* chamada *timeval*, ela possui dois campos *tv_sec* e *tv_usec*, que são os tempos em segundos e microsegundos respectivamente. O código transforma os microsegundos e os soma aos segundos na linha 9 e escreve o resultado na linha 10.

Código 20 – Código fonte mtime.c

```
1  #include <stdio.h>
2  #include <sys/time.h>
3
4  int main()
5  {
6          struct timeval t1;
7          double time;
8          gettimeofday(&t1, NULL);
9          time = t1.tv_sec + (t1.tv_usec / 1000000.0);
10         printf("%f", time);
11         return 0;
12 }
```

5 RESULTADOS

As Tabelas 2 e 3 mostram todas as amostras coletadas e foram divididas em 3 grupos, um para cada máquina: máquina de 1, 2 e 4 núcleos. Cada grupo foi dividido em outros 2 grupos: *freebsd* representando a amostra coletada na instalação padrão; e *rcpar* representando a amostra coletada na instalação com a solução implantada.

Tabela 2 – Amostras coletadas

Tempos					
1 núcleo		2 núcleos		4 núcleos	
freebsd	rcpar	freebsd	rcpar	freebsd	rcpar
7.891792	9.230133	6.181341	6.043839	3.926863	3.646628
8.033063	9.181810	6.200426	6.208448	3.876815	3.652451
7.958238	9.157099	6.266432	6.199874	3.885086	3.587683
7.986109	9.190245	6.189490	6.221388	3.885106	3.577719
8.041368	9.189510	6.277451	6.155261	3.818463	3.752894
8.000114	9.230649	6.123336	6.176718	3.851830	3.618968
8.004496	9.230737	6.123411	6.232763	3.793371	3.637968
8.177432	9.171761	6.288478	6.209486	3.918552	3.621459
7.924519	9.115077	6.266428	6.186682	3.776804	3.597617
7.991701	9.296820	6.463830	6.286021	3.910055	3.827304
7.424948	9.224072	6.189334	6.208496	3.801908	3.669501
7.367480	9.204800	6.134354	6.230753	3.843623	3.656648
7.425493	9.222956	6.069086	6.186594	3.735134	3.646457
7.450385	9.214181	6.145326	6.197882	3.910144	3.696863
7.486796	9.221507	6.079361	6.219964	4.476514	3.540720
7.417066	9.264335	6.157013	6.285681	3.851820	3.638013
7.499405	9.140718	6.188368	6.264002	3.801671	3.615007
7.450465	9.140536	6.255423	6.110085	3.843411	3.654652
7.475227	9.263727	6.310468	6.220372	3.826756	3.591061
7.491933	9.230520	6.343433	6.470996	3.851745	3.631689
7.358653	9.164524	6.167476	6.285568	3.826793	3.598289
7.384078	9.164819	6.156409	6.208436	3.926734	3.713301
7.475290	9.198718	6.167395	6.131583	3.935114	3.685648
7.458694	9.188919	6.244402	6.274991	3.810002	3.769199
7.400446	9.139870	6.179025	6.314142	3.918543	3.615159
7.375253	9.181231	6.146017	6.329630	3.785139	3.619242
7.433722	9.156909	6.157013	6.286246	3.835134	3.521303
7.410840	9.080669	6.343497	6.330288	3.893394	3.640059
7.367295	9.255792	6.299371	6.331626	3.951809	3.685545
7.425784	9.271021	6.299490	6.253289	3.860050	3.652950

Fonte: Autoria própria

Tabela 3 – Amostras coletadas (continuação)

Tempos					
1 núcleo		2 núcleos		4 núcleos	
freebsd	rcpar	freebsd	rcpar	freebsd	rcpar
7.424804	9.090279	6.167404	6.230810	3.851709	3.544332
7.383772	9.228983	6.068949	6.176938	3.843385	3.585174
7.420084	9.190115	6.171999	6.109272	3.910185	3.674332
7.466441	9.298375	6.277460	6.208817	3.835140	3.644397
7.500301	9.221735	6.222384	6.186793	3.926775	3.623345
7.566343	9.146573	6.146009	6.243615	3.710169	3.521418
7.467022	9.156553	6.157039	6.233011	3.910107	3.686347
7.491408	9.115413	6.425155	6.230881	3.810099	3.730204
7.507890	9.157372	6.277446	6.221300	3.876700	3.694512
7.375524	9.263283	6.266407	6.232840	3.785114	3.661434
7.483997	9.263749	6.222475	6.186840	3.859992	3.529875
7.433823	9.205320	6.233443	6.209063	3.893492	3.679943
7.374719	9.220859	6.190036	6.263984	3.960051	3.684620
7.525441	9.255613	6.244573	6.131937	3.926808	3.571476
7.533811	9.157314	6.234050	6.264152	3.909942	3.736299
7.458657	9.214060	6.211360	6.230949	3.926814	3.648730
7.378661	9.188037	6.342720	6.188468	3.785033	3.623246
7.541772	9.196709	6.404522	6.275817	3.876764	3.677521
7.483625	9.213695	6.430707	6.264450	3.893424	3.761391
7.375313	9.205745	6.299472	6.221798	3.793483	3.602999
7.436612	9.132487	6.156358	6.098709	3.876850	3.611307
7.458848	9.218896	6.090323	6.274712	3.835064	3.602967
7.367245	9.229192	6.145364	6.219879	3.851742	3.657089
7.591558	9.190002	6.101326	6.296961	3.860137	3.703034
7.450441	9.115019	6.167392	6.131918	3.785103	3.579385
7.483612	9.255041	6.101318	6.219582	3.893454	3.588094
7.441396	9.164098	6.266458	6.286023	3.768421	3.678041
7.417159	9.104936	6.338442	6.208936	3.926777	3.679974
7.458816	9.112826	6.244472	6.274981	3.801740	3.613295
7.517306	9.148583	6.288395	6.285860	4.843366	3.522132

Fonte: Autoria própria

As médias das amostras podem ser vistas na Tabela 4. Observando-se as médias surgem as seguintes hipóteses:

1. na máquina de 1 núcleo a média da solução proposta é maior que a do sistema atual;
2. na máquina de 2 núcleos as médias são iguais; e
3. na máquina de 4 núcleos a média da solução proposta é menor que a do sistema atual.

Tabela 4 – Médias das amostras

1 núcleo		2 núcleos		4 núcleos	
freebsd	**rcpar**	**freebsd**	**rcpar**	**freebsd**	**rcpar**
7.540075	9.193075	6.221772	6.227840	3.884270	3.641315

Fonte: Autoria própria

Definindo a média populacional μ_1 como a média dos tempos do sistema atual e a média populacional μ_2 como a média dos tempos da solução proposta, as hipóteses podem ser matematicamente expressas da seguinte forma:

1. $H_0 : \mu_1 \geq \mu_2$

   $H_a : \mu_1 < \mu_2$

2. $H_0 : \mu_1 = \mu_2$

   $H_a : \mu_1 \neq \mu_2$

3. $H_0 : \mu_1 \leq \mu_2$

   $H_a : \mu_1 > \mu_2$

Para aplicar o teste z, foi utilizado um nível de confiança de 99%, portanto $\alpha = 0,01$. O teste de hipótese 1 é o teste unicaudal à esquerda, portanto o valor crítico é $-z_c = -2,575$. O teste de hipótese 2 é o teste bicaudal, portanto os valores críticos são $-z_c = -2,575$ e $z_c = 2,575$. O teste de hipótese 3 é o teste unicaudal à direita, portanto o valor crítico é $z_c = 2,575$.

A Tabela 5 mostra os valores calculados da média amostral ($\bar{x}$), variância (s^2), erro padrão ($\sigma_{\bar{x}_1-\bar{x}_2}$) e z.

Tabela 5 – Valores de z

	1 núcleo		2 núcleos		4 núcleos	
	freebsd	**rcpar**	**freebsd**	**rcpar**	**freebsd**	**rcpar**
Média $\bar{x}$	7.540075	9.193075	6.221772	6.227840	3.884270	3.641315
n	60					
s^2	0.046693	0.002645	0.008425	0.004541	0.025350	0.004024
$\sigma_{\bar{x}_1-\bar{x}_2}$	0.028676		0.014700		0.022126	
z	-57.644559		-0.412762		10.980496	

Fonte: Autoria própria

Observando os valores de z, pode-se tomar a decisão de rejeitar ou não as hipóteses nulas previamente descritas. Para a hipótese 1, o valor $z = -57.6445$ é menor que o valor crítico e, portanto, está na área de rejeição. A interpretação dessa decisão é dada como: com 99% de confiança, há evidência suficiente para apoiar a afirmação de que a média do sistema proposto μ_2 é maior que a média do sistema atual μ_1.

Para a hipótese 2, o valor $z = -0.4127$ está entre os valores críticos e, portanto, não está na área de rejeição. A interpretação dessa decisão é dada como: com 99% de confiança, há

evidência suficiente para apoiar a afirmação de que as médias dos sistemas proposto μ_2 e atual μ_1 são iguais.

Para a hipótese 3, o valor $z = 10.9804$ é maior que o valor crítico e, portanto, está na área de rejeição. A interpretação dessa decisão é dada como: com 99% de confiança, há evidência suficiente para apoiar a afirmação de que a média do sistema proposto μ_2 é menor que a média dos sistema atual μ_1.

6 CONCLUSÃO

Usando os resultados obtidos, pode-se concluir que a solução desenvolvida consegue diminuir o tempo de inicialização de espaço de usuário do FreBSD apenas quando é utilizada uma máquina com processador de 4 núcleos. Mesmo assim, a diferença das médias é pequena. Para as outras máquinas a solução teve média maior ou igual.

Um ponto a ser considerado é que a carga usada nos experimentos (os *scripts* que vêm com o FreeBSD) não condiz com aquilo que acontece em instalações com perfil de servidor Web ou um computador do tipo pessoal por exemplo. O FreeBSD é um sistema operacional completo do ponto de vista de seus desenvolvedores, mas na prática ele não contém os programas instalados nos perfis mencionados. Portanto essa carga escolhida pode não ser suficiente para mostrar diferenças significativas entre o sistema atual e o proposto quando utilizado em processador com 2 núcleos.

Os resultados em máquinas de 1 e 2 núcleos fazem surgir a hipótese de que a execução dos *scripts* pode não constituir um trabalho suficientemente grande ao ponto de ultrapassar a despesa de criação de um processo filho, ou seja, o tempo de criação do processo filho supera o tempo de execução do próprio *script*. Isso não pode ser dito para todos os *scripts*, mas os resultados apontam que é verdade para a maioria. Seria interessante em um trabalho futuro, achar maneiras de mensurar e comparar o tempo de execução de cada *script* (maneiras estas que funcionem tanto no sistema padrão quanto na solução proposta) e o tempo despendido na criação dos processos filho. Para isso, também seria importante descobrir se o *shell* cria processos filho (assim como visto no Código 7 linha 1351) mais rápido que as funções *fork* e *exec* usadas na proposta e se ele realmente cria processos filho.

Na análise das alternativas ao sistema atual do FreeBSD pode ser visto que algumas delas estendem a noção de dependência entre serviços/*scripts*. A extensão mais proeminente foi a de *sockets*, em que um serviço pode dizer que cria um *socket* e outros serviços podem dizer que dependem desse *socket* e não do serviço. Nos casos do Launchd, Systemd e do Android init, isso permite que o sistema crie o *socket* antecipadamente e só depois inicie-os paralelamente, dando oportunidade de encurtar o tempo de inicialização. Como pode ser visto na Figura 18 (apêndice B), o grafo de dependências do FreeBSD possui várias arestas de dependência. Implementar essa extensão poderia diminuir o número de dependências permitindo melhor paralelismo. Além de *sockets* o mesmo pode ser dito para canais D-BUS. Ligada à essa extensão está a funcionalidade de iniciar serviços por demanda, já que o servidor pode ser iniciado apenas quando o cliente faz uma requisição. Outra funcionalidade implementada nas alternativas, e que não está presente no sistema atual, é a de reinicialização automática de serviços caso estes parem inesperadamente.

Um problema encontrado durante a pesquisa foi entender a estruturação do sistema **rc** e a principal causa é o uso de *shell scripting* como linguagem de programação. Chamadas de procedimentos no **rc** são feitas com passagem de parâmetros por variáveis globais. Aliado

ao fato dos procedimentos não poderem retornar valor, isso transforma qualquer chamada de procedimento em um simples **goto** (em alusão à linguagem de programação C) e dificulta o entendimento do código. Apesar de não ser diretamente relacionado a esse trabalho, isso mostra que propor alterações no sistema de inicialização pode não ser suficiente, talvez uma completa reescrita em outra linguagem de programação seja necessária.

Outro problema encontrado foi quando tentou-se utilizar máquinas virtuais para a coleta dos tempos. Nessa tentativa houve dúvidas quanto aos efeitos do programa, gerenciador de máquina virtual, nas medições. O próprio sistema operacional hospedeiro pode influenciar esses valores. Seria interessante, em um trabalho futuro, pesquisar as influências desse tipo de programa nos experimentos.

REFERÊNCIAS

CORMEN, Thomas H. et al. **Introduction to Algorithms, Third Edition**. 3. ed. [S.l.]: The MIT Press, 2009.

FREEBSD. **The FreeBSD Project**. 2017. Disponível em: <https://www.freebsd.org>. Acesso em: 28 set. 2017.

FULLAGAR, David. **Netflix Testimonial**. 2015. Disponível em: <https://www.freebsdfoundation.org/testimonial/netflix>. Acesso em: 05 abr. 2017.

GENTOO FOUNDATION. **Gentoo AMD64 Handbook**. 2016. Disponível em: <https://wiki.gentoo.org/wiki/Handbook:AMD64>. Acesso em: 20 mar. 2017.

KOUM, Jan. **WhatsApp Testimonial**. 2015. Disponível em: <https://www.freebsdfoundation.org/testimonial/whatsapp>. Acesso em: 30 ago. 2017.

LARSON, Ron; FARBER, Betsy. **Estatística aplicada**. 4. ed. [S.l.]: Pearson Prentice Hall, 2010.

LEVIN, Jonathan. **Mac OS X and iOS Internals: To the Apple's Core**. 1. ed. Birmingham, UK, UK: Wrox Press Ltd., 2012.

LEVIN, J. **Android Internals - Volume I: A Confectioner's Cookbook**. [S.l.]: Jonathan Levin, 2014.

LIU, Y.; YUE, Y.; GUO, L. **UNIX Operating System: The Development Tutorial via UNIX Kernel Services**. [S.l.]: Springer Berlin Heidelberg, 2011.

MAC DEVELOPER LIBRARY. **Daemons and Services Programming Guide**. 2016. Disponível em: <https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingLaunchdJobs.html#//apple_ref/doc/uid/10000172i-SW7-BCIEDDBJ>. Acesso em: 30 mai. 2017.

MCKUSICK, Marshall Kirk; NEVILLE-NEIL, George; WATSON, Robert N.M. **The Design and Implementation of the FreeBSD Operating System**. 2. ed. [S.l.]: Addison-Wesley Professional, 2014.

MILLER, Charles; ZOVI, Dino Dai. **The Mac Hacker's Handbook**. [S.l.]: Wiley Publishing, 2009.

OPENRC. **OpenRC**. 2016. Disponível em: <https://wiki.gentoo.org/wiki/OpenRC>. Acesso em: 20 mar. 2017.

POETTERING, Lennart; SIEVERS, Kay; LEEMHUIS, Thorsten. **Control Centre, The systemd Linux init system**. 2012. Disponível em: <http://www.h-online.com/open/features/Control-Centre-The-systemd-Linux-init-system-1565543.html>. Acesso em: 08 abr. 2017.

SONY. **Open Source Software used in PlayStation®4**. 2016. Disponível em: <http://doc.dl.playstation.net/doc/ps4-oss/>. Acesso em: 05 abr. 2017.

SVISTUNOV, Maxim et al. **Red Hat Enterprise Linux 7: System Administrator's Guide**. [S.l.: s.n.], 2016.

APÊNDICE A - Replicação dos Experimentos

Este apêndice mostra os passos para replicar os experimentos realizados neste trabalho e foi dividido em partes:

- A.1 - Instalação do FreeBSD: abrange a seleção de quais componentes devem ser instalados;
- A.2 - Automação da coleta de dados: mostra como realizar a coleta dos tempos de forma automatizada e quais ferramentas foram usadas na automação.

A.1 - INSTALAÇÃO DO FREEBSD

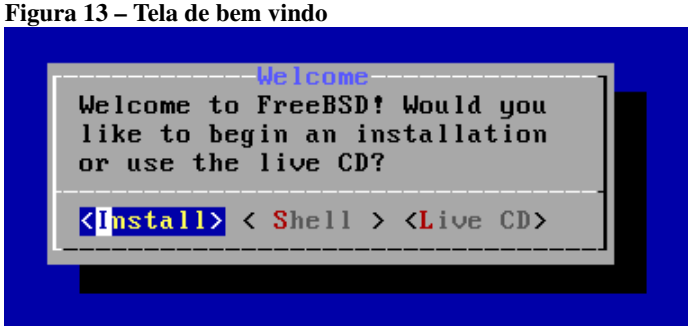
As mídias de instalação do FreeBSD 11.1 i386 e amd64 podem ser obtidas, respectivamente, das URLs:

<https://download.freebsd.org/ftp/releases/i386/i386/ISO-IMAGES/11.1/>

<https://download.freebsd.org/ftp/releases/amd64/amd64/ISO-IMAGES/11.1/>

Esta instalação pressupõe que o computador tem ao menos uma placa de rede funcional. Após preparar e iniciar o computador pela mídia, o processo de instalação irá começar:

Figura 13 – Tela de bem vindo

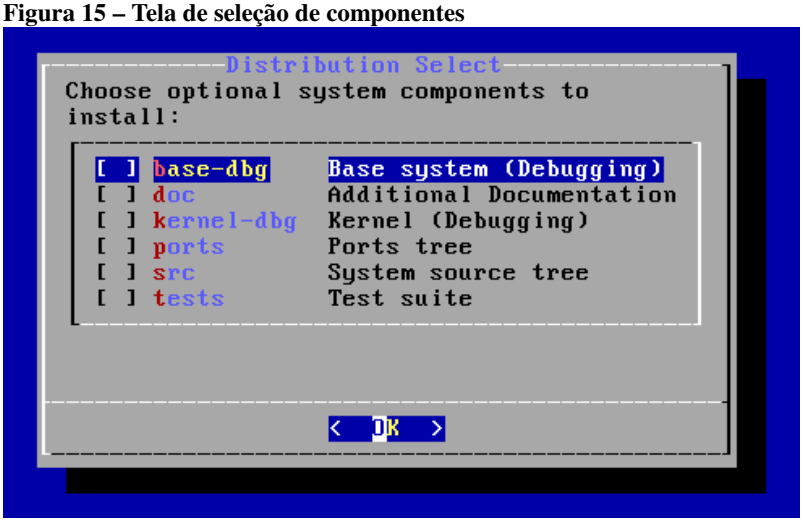


Selecione **install**. Na próxima tela selecione o modelo do teclado, qualquer **Brazilian** é suficiente. Depois selecione a opção **Continue with br275...**:

Figura 14 – Tela de seleção de teclado

Em seguida digite o **hostname** da máquina. Na tela de componentes opcionais desmarque todas as opções (esta tela possui mais opções na versão **amd64** mas o procedimento é o mesmo):

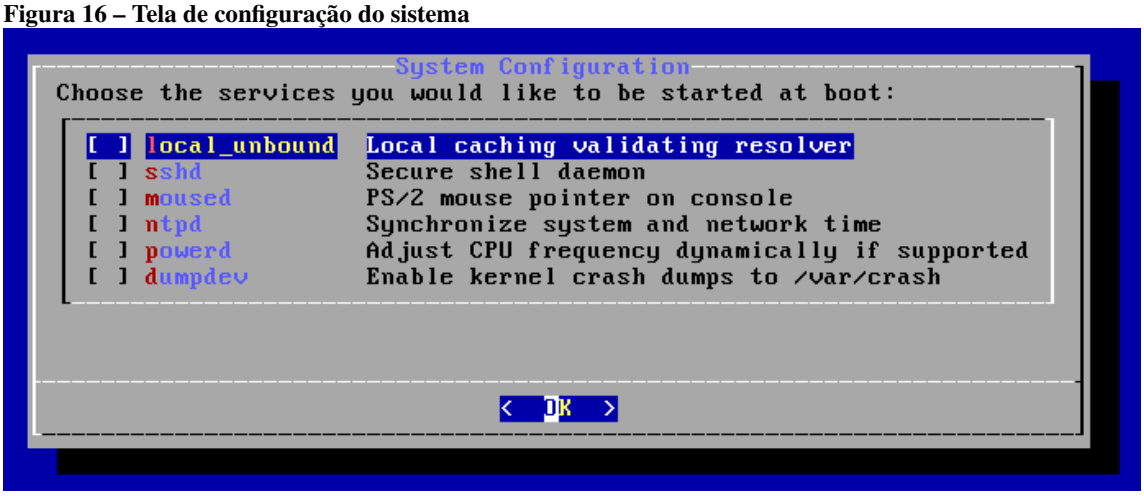
Figura 15 – Tela de seleção de componentes



Na tela de particionamento selecione **Auto (UFS)**. Caso esteja instalando a partir de um pendrive, a próxima tela **Partitioning** perguntará qual o destino da instalação, selecione o disco rígido (opção **ada0**). Em seguida selecione **Entire Disk**, depois confirme com **Yes**. Depois **MBR**. Depois **Finish** e depois **Commit**. Após copiar os dados para o disco rígido, a instalação pedirá a senha do usuário *root*.

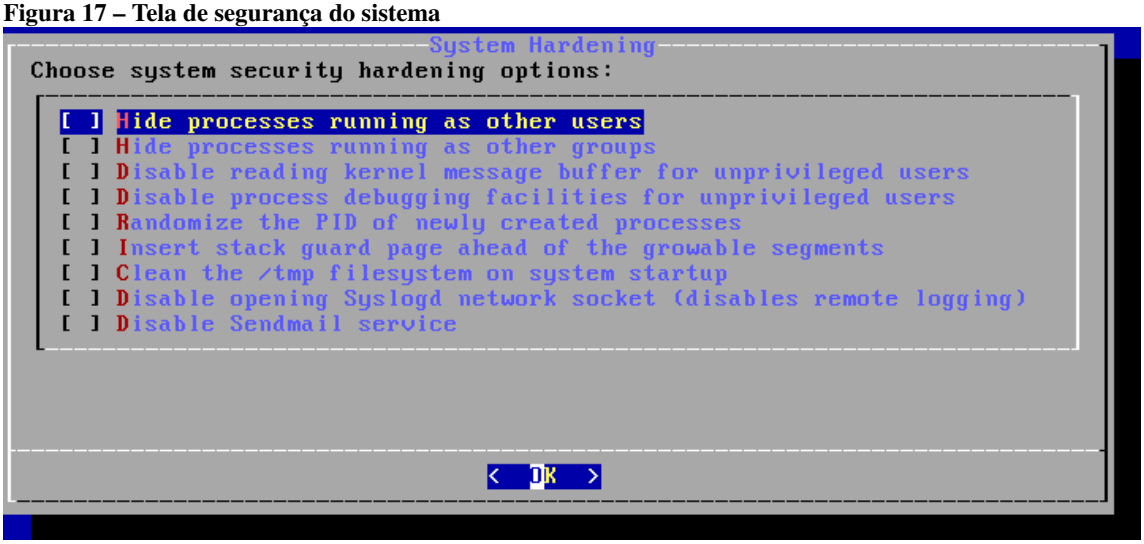
Na tela de configuração de rede, selecione a opção **cancel**. Em seguida configure a *timezone* para **2 America**, depois **10 Brazil**, depois **8 Brazil** e depois confirme com **Yes**. Na tela de **Time & Date** selecione **Skip** duas vezes. Na tela **System Configuration** desmarque todas as opções:

Figura 16 – Tela de configuração do sistema



Na tela **System Hardening** apenas confirme no **OK**:

Figura 17 – Tela de segurança do sistema



Em seguida a instalação perguntará se deseja adicionar mais usuários, selecione **No**. Na tela **Final Configuration** selecione **Exit**. Na tela **Manual Configuration** selecione **No**. Na tela **Complete** selecione **Reboot**, o computador irá reiniciar, assim que o computador desligar retire a mídia de instalação.

A instalação do sistema operacional está terminada, mas como é necessário baixar alguns arquivos, é preciso configurar temporariamente a interface de rede. Primeiro, logue com o usuário *root*. Depois, para saber qual é a interface de rede padrão digite:

```
1  ifconfig
```

Para configurar a interface usando **DHCP** digite:

```
1  dhclient interface
```

trocando **interface** pelo nome da interface de rede. Se a interface é **em0** o comando fica

```
1  dhclient em0
```

Agora a interface de rede está configurada temporariamente, se a máquina for reiniciada essa configuração será perdida. O processo de instalação do FreeBSD está terminado.

A.2 - AUTOMAÇÃO DA COLETA DE DADOS

O Código do **rcpar** pode ser obtido do repositório **github.com/haneiko/tcc2**. O repositório vem preparado para coletar os dados de forma automática. Para iniciar o processo é necessário baixar o arquivo **setup**:

```
1  fetch --no-verify-peer \
2  https://raw.githubusercontent.com/haneiko/tcc2/master/setup
```

Depois dar permissão de execução para o arquivo baixado:

```
1  chmod +x setup
```

Para iniciar o processo digite:

```
1  ./setup
```

O processo realiza a coleta das amostras, primeiramente da instalação padrão e depois da instalação com a solução implantada. Ambas as amostras têm tamanho 60.

Toda a automação é realizada pelos programas: **setup** e **collect**. **setup** serve para baixar, compilar e instalar os programas, habilitar o login automático do usuário *root* e a execução automática do **collect**. **collect** monitora o número de coletas realizadas.

O Código 21 mostra o código fonte do programa **setup**. Nas linhas 5 e 6 são declarados os arquivos que serão baixados. Nas linhas 8 à 12 são baixados os arquivos. Na linha 15 os programas são compilados. Na linha 18 os programas são marcados como executáveis e na 19 são instalados no sistema. Na linha 20 o arquivo **rc_freebsd** é instalado no lugar de **/etc/rc**. Esse é o arquivo que contém o Código 18. Nas linhas 22 à 24 é habilitado o auto login para o usuário *root*. Nas linhas 26 à 28 é habilitada a execução automático do **collect**, assim que o usuário *root* realizar o login, esse programa é executado. Nas linhas 30 e 31 é retirado o atraso de inicialização do *kernel*, que por padrão é de 10 segundos. Nas linhas 33 e 34 é desabilitado o serviço de checagem de disco em segundo plano, esse serviço pode segurar a inicialização por até 1 minuto se não for desabilitado. Na linha 37 o computador é reiniciado.

Código 21 – Código fonte: setup

```
1  #!/bin/sh
2
3  set -e
4
5  files="rcpar.c rcorder2.c mtime.c Makefile gen_graph \
6       rc_freebsd rc_rcpar collect"
7
```

```
 8 echo "Downloading files"
 9 for file in $files; do
10     fetch --no-verify-peer\
11           https://raw.githubusercontent.com/haneiko/tcc2/master/$file
12 done
13
14 echo "Compiling"
15 make
16
17 echo "Installing"
18 chmod +x rcorder2 rcpar mtime collect gen_graph
19 make install
20 cp rc_freebsd /etc/rc
21
22 echo "Enabling autologin for root"
23 sed -i -- 's:^ttyv0.*$:ttyv0 \"/usr/libexec/getty al.Pc\" xterm on
       secure:g'\
24     /etc/ttys
25
26 echo "Setting up autotest"
27 [ -n "`grep "collect" ~/.cshrc`" ]\
28     || echo "~/collect" >> ~/.cshrc
29
30 [ -n "`grep "autoboot_delay" /boot/loader.conf`" ]\
31     || echo "autoboot_delay=\"0\"" >> /boot/loader.conf
32
33 [ -n "`grep "background_fsck_delay" /etc/rc.conf`" ]\
34     || echo "background_fsck_delay=\"0\"" >> /etc/rc.conf
35
36 echo "Rebooting"
37 reboot
```

Após o usuário *root* logar, o programa **collect** será executado. O Código 22 mostra o código fonte do **collect**. O programa começa declarando algumas variáveis (linhas 4 e 5): *files* é a lista de nomes dos arquivos que contém os tempos coletados; *n* é o tamanho das amostras.

Esse programa irá monitorar o número de coletas realizadas. A estrutura de repetição (linhas 7 à 16) itera sobre a lista *files*. O primeiro valor da lista é *rc_freebsd.times* e é armazenado na variável *file*, este valor representa o arquivo de mesmo nome. Na linha 8 o arquivo é criado se não existir. A variável *name* captura (linha 9) o nome do arquivo sem extensão (*rc_freebsd*) e a variável *lines* captura o número de linhas do arquivo (linha 10). A condicional da linha 11 verifica se o número de linhas é menor que *n*, caso seja, então o arquivo **rc_freebsd** é instalado no lugar de **/etc/rc** (linha 13) e o computador é reiniciado (linha 14). Ao final da iteração, o arquivo **rc_freebsd.times** vai estar com as *n* linhas (os *n* tempos coletados). A próxima iteração inicia com próximo valor da lista *files* (*rc_rcpar.times*) e os mesmos procedimentos são executados: o computador é reiniciado até que o arquivo **rc_rcpar.times** contenha os *n* valores.

Quando todos os tempos forem coletados, o processo irá parar e emitir uma mensagem de sucesso.

Código 22 – Código fonte: collect

```
1 #!/bin/sh
2 set -e
3
4 files="rc_freebsd.times rc_rcpar.times"
5 n=60
```

```
6
7  for file in $files; do
8      touch $file
9      name=`echo $file | cut -f1 -d'.'`
10     lines=`cat $file | wc -l`
11     if [ $lines -lt $n ]; then
12         echo "$name $lines"
13         cp $name /etc/rc
14         reboot
15     fi
16 done
17
18 echo "Amostras coletadas com sucesso"
```

APÊNDICE B - Grafo de dependência completo

A Figura 18 mostra o grafo de dependência com todos os *scripts* presentes por padrão no FreeBSD. O grafo foi criado com o programa **gen_graph**, ele está presente no repositório da solução (<github.com/haneiko/tcc2>) e seu código fonte pode ser visto no Código 23. O programa usa o pacote graphviz para gerar uma imagem do grafo. O comando

```
pkg install -y graphviz
```

instala o pacote no FreeBSD. Executando o programa

```
./gen_graph
```

gera-se a imagem **graph.png** vista na Figura 18.

Código 23 – Código fonte: gen_graph

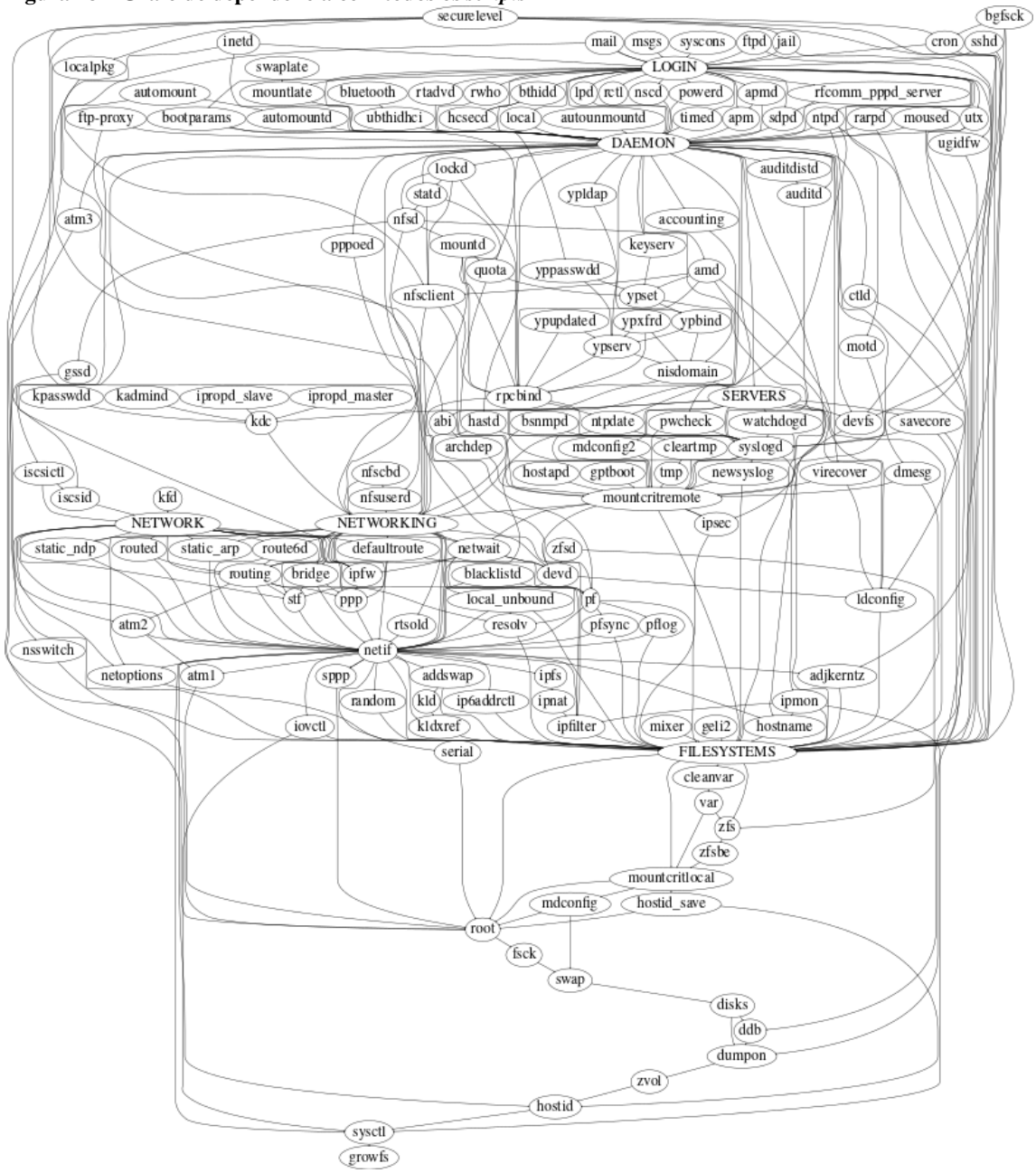
```
#!/bin/sh
set -e

files=`rcorder -s nostart /etc/rc.d/*`
out=graph.dot

echo "digraph G {" > $out
for file in $files; do
    awk '
BEGIN {
        i_pro = 0
        i_req = 0
        i_bef = 0
}
/# PROVIDE/ {
        split($0, arr)
        for(i = 3; i <= length(arr); i++)
                provide[i_pro++] = arr[i]
}
/# REQUIRE/ {
        split($0, arr)
        for(i = 3; i <= length(arr); i++)
                require[i_req++] = arr[i]
}
/# BEFORE/ {
        split($0, arr)
        for(i = 3; i <= length(arr); i++)
                before[i_bef++] = arr[i]
}
END {
        for(j = 0; j < length(provide); j++) {
                for(i = 0; i < length(require); i++)
                        print "\"" provide[j] "\"->\"" require[i] "\"
                            ;"
                for(i = 0; i < length(before); i++)
                        print "\"" before[i] "\"->\"" provide[j] "\";
                            "
        }
}' $file >> $out
done
echo "}" >> $out

dot -Tpng $out -o graph.png
rm -f $out
```

Figura 18 – Grafo de dependência com todos os *scripts*



APÊNDICE C - Saída do rcorder e rcorder2

Os Quadros 8 e 9 mostras as saídas dos programas **rcorder** e **rcorder2**, respectivamente. Ambos foram invocados com os parâmetros: *-s nostart /etc/rc.d/**.

Quadro 8 – Saída do rcorder

Início	Cont...	Cont...	Cont...
/etc/rc.d/growfs	/etc/rc.d/atm2	/etc/rc.d/yppasswdd	/etc/rc.d/nscd
/etc/rc.d/sysctl	/etc/rc.d/pfsync	/etc/rc.d/ypldap	/etc/rc.d/ntpd
/etc/rc.d/hostid	/etc/rc.d/pflog	/etc/rc.d/virecover	/etc/rc.d/powerd
/etc/rc.d/zvol	/etc/rc.d/pf	/etc/rc.d/accounting	/etc/rc.d/rarpd
/etc/rc.d/dumpon	/etc/rc.d/stf	/etc/rc.d/nfsclient	/etc/rc.d/rctl
/etc/rc.d/ddb	/etc/rc.d/ppp	/etc/rc.d/amd	/etc/rc.d/sdpd
/etc/rc.d/geli	/etc/rc.d/routing	/etc/rc.d/atm3	/etc/rc.d/rfcomm_pppd_server
/etc/rc.d/gbde	/etc/rc.d/ipfw	/etc/rc.d/auditd	/etc/rc.d/rtadvd
/etc/rc.d/ccd	/etc/rc.d/netwait	/etc/rc.d/auditdistd	/etc/rc.d/rwho
/etc/rc.d/swap	/etc/rc.d/resolv	/etc/rc.d/tmp	/etc/rc.d/LOGIN
/etc/rc.d/fsck	/etc/rc.d/local_unbound	/etc/rc.d/cleartmp	/etc/rc.d/syscons
/etc/rc.d/root	/etc/rc.d/nsswitch	/etc/rc.d/ctld	/etc/rc.d/swaplate
/etc/rc.d/mdconfig	/etc/rc.d/routed	/etc/rc.d/dmesg	/etc/rc.d/sshd
/etc/rc.d/hostid_save	/etc/rc.d/rtsold	/etc/rc.d/hastd	/etc/rc.d/sendmail
/etc/rc.d/mountcritlocal	/etc/rc.d/static_ndp	/etc/rc.d/iscsid	/etc/rc.d/cron
/etc/rc.d/zfsbe	/etc/rc.d/static_arp	/etc/rc.d/iscsictl	/etc/rc.d/jail
/etc/rc.d/zfs	/etc/rc.d/bridge	/etc/rc.d/keyserv	/etc/rc.d/localpkg
/etc/rc.d/var	/etc/rc.d/route6d	/etc/rc.d/nfsuserd	/etc/rc.d/securelevel
/etc/rc.d/cleanvar	/etc/rc.d/defaultroute	/etc/rc.d/gssd	/etc/rc.d/othermta
/etc/rc.d/FILESYSTEMS	/etc/rc.d/NETWORKING	/etc/rc.d/quota	/etc/rc.d/nfscbd
/etc/rc.d/ldconfig	/etc/rc.d/mountcritremote	/etc/rc.d/mountd	/etc/rc.d/msgs
/etc/rc.d/kldxref	/etc/rc.d/newsyslog	/etc/rc.d/nfsd	/etc/rc.d/moused
/etc/rc.d/kld	/etc/rc.d/syslogd	/etc/rc.d/statd	/etc/rc.d/mixer
/etc/rc.d/addswap	/etc/rc.d/ntpdate	/etc/rc.d/lockd	/etc/rc.d/kpasswdd
/etc/rc.d/adjkerntz	/etc/rc.d/rpcbind	/etc/rc.d/pppoed	/etc/rc.d/kfd
/etc/rc.d/atm1	/etc/rc.d/devfs	/etc/rc.d/pwcheck	/etc/rc.d/kadmind
/etc/rc.d/hostname	/etc/rc.d/ipmon	/etc/rc.d/DAEMON	/etc/rc.d/ipropd_slave
/etc/rc.d/ip6addrctl	/etc/rc.d/kdc	/etc/rc.d/utx	/etc/rc.d/ipropd_master
/etc/rc.d/netoptions	/etc/rc.d/mdconfig2	/etc/rc.d/ugidfw	/etc/rc.d/inetd
/etc/rc.d/random	/etc/rc.d/watchdogd	/etc/rc.d/ubthidhci	/etc/rc.d/hostapd
/etc/rc.d/sppp	/etc/rc.d/savecore	/etc/rc.d/timed	/etc/rc.d/gptboot
/etc/rc.d/ipfilter	/etc/rc.d/archdep	/etc/rc.d/apm	/etc/rc.d/geli2
/etc/rc.d/ipnat	/etc/rc.d/abi	/etc/rc.d/apmd	/etc/rc.d/ftpd
/etc/rc.d/ipfs	/etc/rc.d/SERVERS	/etc/rc.d/bootparams	/etc/rc.d/ftp-proxy
/etc/rc.d/serial	/etc/rc.d/nisdomain	/etc/rc.d/hcsecd	/etc/rc.d/bsnmpd
/etc/rc.d/iovctl	/etc/rc.d/ypserv	/etc/rc.d/bthidd	/etc/rc.d/blacklistd
/etc/rc.d/netif	/etc/rc.d/ypxfrd	/etc/rc.d/local	/etc/rc.d/bgfsck
/etc/rc.d/devd	/etc/rc.d/ypupdated	/etc/rc.d/lpd	/etc/rc.d/autounmountd
/etc/rc.d/zfsd	/etc/rc.d/ypbind	/etc/rc.d/motd	/etc/rc.d/automountd
/etc/rc.d/ipsec	/etc/rc.d/ypset	/etc/rc.d/mountlate	/etc/rc.d/automount

Fonte: autoria própria

Quadro 9 – Saída do rcorder2

Início	Cont...	Cont...	Cont...
/etc/rc.d/growfs	/etc/rc.d/ldconfig	/etc/rc.d/devfs	/etc/rc.d/nfsd
/etc/rc.d/rctl	/etc/rc.d/mixer	/etc/rc.d/dmesg	END
END	/etc/rc.d/netoptions	/etc/rc.d/gptboot	/etc/rc.d/statd
/etc/rc.d/sysctl	/etc/rc.d/random	/etc/rc.d/hostapd	END
END	/etc/rc.d/ugidfw	/etc/rc.d/ipropd_master	/etc/rc.d/lockd
/etc/rc.d/hostid	END	/etc/rc.d/ipropd_slave	END
END	/etc/rc.d/ipmon	/etc/rc.d/iscsictl	/etc/rc.d/DAEMON
/etc/rc.d/zvol	/etc/rc.d/ipnat	/etc/rc.d/kadmind	END
END	/etc/rc.d/kld	/etc/rc.d/kpasswdd	/etc/rc.d/apm
/etc/rc.d/dumpon	END	/etc/rc.d/mdconfig2	/etc/rc.d/automountd
END	/etc/rc.d/addswap	/etc/rc.d/motd	/etc/rc.d/autounmountd
/etc/rc.d/ddb	/etc/rc.d/ipfs	/etc/rc.d/newsyslog	/etc/rc.d/bootparams
END	END	/etc/rc.d/nfscbd	/etc/rc.d/ftp-proxy
/etc/rc.d/ccd	/etc/rc.d/netif	/etc/rc.d/tmp	/etc/rc.d/hcsecd
/etc/rc.d/gbde	END	/etc/rc.d/virecover	/etc/rc.d/local
/etc/rc.d/geli	/etc/rc.d/atm2	END	/etc/rc.d/lpd
END	/etc/rc.d/devd	/etc/rc.d/abi	/etc/rc.d/mountlate
/etc/rc.d/swap	/etc/rc.d/pflog	/etc/rc.d/cleartmp	/etc/rc.d/moused
END	/etc/rc.d/pfsync	/etc/rc.d/syslogd	/etc/rc.d/nscd
/etc/rc.d/fsck	/etc/rc.d/ppp	END	/etc/rc.d/ntpd
END	/etc/rc.d/resolv	/etc/rc.d/auditd	/etc/rc.d/powerd
/etc/rc.d/root	/etc/rc.d/rtsold	/etc/rc.d/bsnmpd	/etc/rc.d/rarpd
END	/etc/rc.d/static_arp	/etc/rc.d/hastd	/etc/rc.d/rtadvd
/etc/rc.d/atm1	/etc/rc.d/static_ndp	/etc/rc.d/localpkg	/etc/rc.d/rwho
/etc/rc.d/gssd	/etc/rc.d/stf	/etc/rc.d/ntpdate	/etc/rc.d/sdpd
/etc/rc.d/hostid_save	END	/etc/rc.d/pwcheck	/etc/rc.d/timed
/etc/rc.d/mdconfig	/etc/rc.d/atm3	/etc/rc.d/savecore	/etc/rc.d/ubthidhci
/etc/rc.d/nsswitch	/etc/rc.d/bridge	/etc/rc.d/watchdogd	/etc/rc.d/utx
/etc/rc.d/serial	/etc/rc.d/defaultroute	END	END
/etc/rc.d/sppp	/etc/rc.d/ipfw	/etc/rc.d/auditdistd	/etc/rc.d/apmd
END	/etc/rc.d/local_unbound	/etc/rc.d/rpcbind	/etc/rc.d/automount
/etc/rc.d/mountcritlocal	/etc/rc.d/pf	/etc/rc.d/SERVERS	/etc/rc.d/bthidd
END	/etc/rc.d/zfsd	END	/etc/rc.d/rfcomm_pppd_server
/etc/rc.d/zfsbe	END	/etc/rc.d/nfsclient	/etc/rc.d/swaplate
END	/etc/rc.d/blacklistd	/etc/rc.d/nisdomain	END
/etc/rc.d/zfs	/etc/rc.d/routing	END	/etc/rc.d/LOGIN
END	END	/etc/rc.d/ypserv	END
/etc/rc.d/var	/etc/rc.d/netwait	END	/etc/rc.d/cron
END	/etc/rc.d/route6d	/etc/rc.d/ypbind	/etc/rc.d/ftpd
/etc/rc.d/cleanvar	/etc/rc.d/routed	/etc/rc.d/ypldap	/etc/rc.d/inetd
END	END	/etc/rc.d/ypupdated	/etc/rc.d/jail
/etc/rc.d/FILESYSTEMS	/etc/rc.d/NETWORKING	/etc/rc.d/ypxfrd	/etc/rc.d/msgs
END	END	END	/etc/rc.d/othermta
/etc/rc.d/adjkerntz	/etc/rc.d/iscsid	/etc/rc.d/ypset	/etc/rc.d/sendmail
/etc/rc.d/ctld	/etc/rc.d/kdc	END	/etc/rc.d/sshd
/etc/rc.d/geli2	/etc/rc.d/kfd	/etc/rc.d/amd	/etc/rc.d/syscons
/etc/rc.d/hostname	/etc/rc.d/mountcritremote	/etc/rc.d/keyserv	END
/etc/rc.d/iovctl	/etc/rc.d/nfsuserd	/etc/rc.d/quota	/etc/rc.d/bgfsck
/etc/rc.d/ip6addrctl	/etc/rc.d/pppoed	/etc/rc.d/yppasswdd	/etc/rc.d/securelevel
/etc/rc.d/ipfilter	END	END	END
/etc/rc.d/ipsec	/etc/rc.d/accounting	/etc/rc.d/mountd	
/etc/rc.d/kldxref	/etc/rc.d/archdep	END	

Fonte: autoria própria

APÊNDICE D - Código fonte: rcorder2.c e rcpar.c

Os Códigos 24 e 25 mostram os códigos fonte dos programas **rcorder2** e **rcpar**, respectivamente.

Código 24 – Código fonte: rcorder2.c

```
1  #include <getopt.h>
2  #include <stdbool.h>
3  #include <stdint.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <sys/hash.h>
8
9  typedef struct list {
10         void *item;
11         struct list *next;
12         struct list *prev;
13 } list;
14
15 typedef struct script {
16         list *provide;
17         list *require;
18         list *before;
19         list *keyword;
20         struct script *next;
21         struct script *prev;
22         bool printed;
23         const char *path;
24 } script;
25
26 typedef struct slot {
27         const uint8_t *key;
28         size_t len;
29         list *items;
30         struct slot *next;
31         struct slot *prev;
32 } slot;
33
34 typedef struct hasht {
35         size_t size;
36         slot **slots;
37 } hasht;
38
39 #define die() do {
              \
40                 fprintf(stderr, "ERROR: %s %d\n", __FILE__, __LINE__)
                       ;  \
41                 exit(EXIT_FAILURE);
                           \
42         } while(0)
43
44 #define PUSH(ITEM, HEAD) do {                    \
45                 ITEM->next = HEAD;               \
46                 if(HEAD)                         \
47                         HEAD->prev = ITEM;       \
48                 HEAD = ITEM;                     \
49         } while(0)
50
51 #define FREE(ITEM, HEAD) do {                              \
52                 typeof (HEAD) tmp;                         \
53                 tmp = ITEM->next;                          \
54                 if(ITEM->next)                             \
55                         ITEM->next->prev = ITEM->prev;     \
56                 if(ITEM->prev)                             \
57                         ITEM->prev->next = ITEM->next;     \
```

```
58                  else                                                \
59                          HEAD = tmp;                                 \
60                  free(ITEM);                                         \
61                  ITEM = tmp;                                         \
62          } while(0)
63
64  #define FOREACH(ITEM, HEAD)                                         \
65          for(ITEM = HEAD; ITEM; ITEM = ITEM->next)
66
67  void list_alloc_and_push(list **head, void *item);
68  hasht *hasht_new(size_t size);
69  void hasht_insert(hasht *h, void *dat, const uint8_t *key, size_t len
        );
70  list *hasht_getlist(hasht *h, const uint8_t *key, size_t len);
71  script *parse_scripts(int argc, char *argv[]);
72  script *del_skipped_scripts(script *head, list *skips);
73  void del_required(hasht *requires, hasht *required, script *cur);
74  bool has_reqs(hasht *requires, script *cur);
75
76  int main(int argc, char *argv[])
77  {
78          list *skips;
79          script *scripts;
80          hasht *provided;
81          hasht *required;
82          hasht *requires;
83
84          /* tmps */
85          int i;
86          list *key;
87          list *key2;
88          list *lst;
89          script *s;
90          script *cur;
91          script *tmp;
92          list *no_reqs;
93
94          skips = NULL;
95
96          while ((i = getopt(argc, argv, "s:")) != -1) {
97                  switch (i) {
98                  case 's':
99                          list_alloc_and_push(&skips, optarg);
100                         break;
101                 }
102         }
103
104         argc -= optind;
105         argv += optind;
106
107         scripts = parse_scripts(argc, argv);
108         scripts = del_skipped_scripts(scripts, skips);
109
110         /* Prepare table: provided */
111         provided = hasht_new(1000);
112         FOREACH(cur, scripts) {
113                 FOREACH(key, cur->provide)
114                         hasht_insert(provided, cur, key->item,
115                                         strlen(key->item));
116         }
117
118         /* Convert BEFORE to REQUIRE */
119         FOREACH(cur, scripts) {
120                 FOREACH(key, cur->before) {
121                         lst = hasht_getlist(provided, key->item,
```

```
122                                             strlen(key->item));
123                         for(; lst; lst = lst->next) {
124                                 s = lst->item;
125                                 list_alloc_and_push(&s->require,
126                                                     cur->provide->
                                                        item);
127                         }
128                 }
129         }
130
131         /* Prepare table: required */
132         required = hasht_new(1000);
133         FOREACH(cur, scripts) {
134                 FOREACH(key, cur->require)
135                         hasht_insert(required, cur, key->item,
136                                      strlen(key->item));
137         }
138
139         /* Prepare table: requires */
140         requires = hasht_new(1000);
141         FOREACH(cur, scripts) {
142                 FOREACH(key, cur->require) {
143                         lst = hasht_getlist(provided, key->item,
144                                             strlen(key->item));
145                         for(; lst; lst = lst->next) {
146                                 tmp = lst->item;
147                                 FOREACH(key2, cur->provide) {
148                                         hasht_insert(requires, tmp,
149                                                      key2->item,
150                                                      strlen(key2->
                                                         item));
151                                 }
152                         }
153                 }
154         }
155
156         while(scripts) {
157                 no_reqs = NULL;
158                 FOREACH(cur, scripts)
159                         if(!has_reqs(requires, cur))
160                                 list_alloc_and_push(&no_reqs, cur);
161                 FOREACH(lst, no_reqs) {
162                         cur = lst->item;
163                         printf("%s\n", cur->path);
164                         del_required(requires, required, cur);
165                         FREE(cur, scripts);
166                 }
167                 printf("END\n");
168         }
169
170         return EXIT_SUCCESS;
171 }
172
173 void list_alloc_and_push(list **head, void *item)
174 {
175         list *tmp;
176         tmp = calloc(1, sizeof(list));
177         tmp->item = item;
178         PUSH(tmp, (*head));
179 }
180
181 char *getline(FILE *file)
182 {
183         char c, *buffer;
184         size_t size;
```

```
185         fpos_t begin;
186
187         buffer = NULL;
188         size = 0;
189
190         if(fgetpos(file, &begin))
191                 die();
192
193         do {
194                 c = fgetc(file);
195                 size++;
196         } while(c != '\n' && c != EOF);
197         size++;
198
199         if(size > 1) {
200                 if(fsetpos(file, &begin))
201                         die();
202                 buffer = malloc(size * sizeof(char));
203                 fgets(buffer, size, file);
204                 buffer[size - 1] = '\0';
205         }
206         return buffer;
207 }
208
209 list *split(const char *buffer, char delim)
210 {
211         list *strings;
212         const char *begin;
213         const char *end;
214         size_t size;
215         char *tmp;
216
217         strings = NULL;
218         begin = end = buffer;
219
220         while((end = strchr(end, delim)) != NULL) {
221                 size = end - begin + 1;
222
223                 if(size > 1) {
224                         tmp = malloc(size * sizeof(char));
225                         strncpy(tmp, begin, size);
226                         tmp[size - 1] = '\0';
227                         list_alloc_and_push(&strings, tmp);
228                 }
229                 ++end;
230                 begin = end;
231         }
232
233         size = strlen(begin);
234         /* size = strlen(begin) + 1; */
235         tmp = malloc(size * sizeof(char));
236         strncpy(tmp, begin, size);
237         tmp[size - 1] = '\0';
238         list_alloc_and_push(&strings, tmp);
239
240         return strings;
241 }
242
243 script *parse_lines(const char *path)
244 {
245         bool parsing;
246         char *buffer;
247         FILE *file;
248         script *cur;
249
```

```
250         cur = calloc(1, sizeof(script));
251         cur->path = path;
252         file = fopen(cur->path, "r");
253         if(!file) {
254                 fprintf(stderr, "Cannot open file %s\n", cur->path);
255                 die();
256         }
257         parsing = false;
258
259         while((buffer = getline(file))) {
260                 if(strstr(buffer, "# PROVIDE: ")) {
261                         cur->provide = split(buffer + 11, ' ');
262                         parsing = true;
263                 } else if(strstr(buffer, "# REQUIRE: ")) {
264                         cur->require = split(buffer + 11, ' ');
265                         parsing = true;
266                 } else if(strstr(buffer, "# BEFORE: ")) {
267                         cur->before = split(buffer + 10, ' ');
268                         parsing = true;
269                 } else if(strstr(buffer, "# KEYWORD: ")) {
270                         cur->keyword = split(buffer + 11, ' ');
271                         parsing = true;
272                 } else if(parsing) {
273                         break;
274                 }
275         }
276         fclose(file);
277         return cur;
278 }
279
280 script *parse_scripts(int argc, char *argv[])
281 {
282         script *s, *scripts;
283         const char *path;
284         int i;
285
286         scripts = NULL;
287         for(i = 0; i < argc; i++) {
288                 path = argv[i];
289
290                 if(strstr(path, ".sh")
291                    || strstr(path, ".OLD")
292                    || strstr(path, ".bak")
293                    || strstr(path, ".orig")
294                    || strstr(path, ",v")
295                    || strstr(path, "#")
296                    || strstr(path, "~")) {
297                         /* printf("ignoring %s\n", path); */
298                         continue;
299                 }
300                 s = parse_lines(path);
301                 PUSH(s, scripts);
302         }
303         return scripts;
304 }
305
306 bool must_skip(script *s, list *skips)
307 {
308         list *key;
309         list *skip;
310
311         FOREACH(key, s->keyword)
312                 FOREACH(skip, skips)
313                 if(strcmp(key->item, skip->item) == 0)
314                         return true;
```

```
315         return false;
316 }
317
318 script *del_skipped_scripts(script *head, list *skips)
319 {
320         script *cur;
321         cur = head;
322
323         while(cur) {
324                 if(must_skip(cur, skips)) {
325                         /* printf("skipping '%s'\n", cur->path); */
326                         FREE(cur, head);
327                         continue;
328                 }
329                 cur = cur->next;
330         }
331         return head;
332 }
333
334 hasht *hasht_new(size_t size)
335 {
336         hasht *h;
337         h = malloc(sizeof(hasht));
338         h->size = size;
339         h->slots = calloc(size, sizeof(slot *));
340         return h;
341 }
342
343 size_t hasht_index(hasht *h, const uint8_t *key)
344 {
345         uint32_t hash;
346         hash = hash32_str(key, 0);
347         return hash % h->size;
348 }
349
350 slot *hasht_getslot(hasht *h, const uint8_t *key, size_t len)
351 {
352         size_t index, i;
353         slot *slot;
354         bool found;
355
356         index = hasht_index(h, key);
357
358         FOREACH(slot, h->slots[index]) {
359                 if(len != slot->len)
360                         continue;
361
362                 found = true;
363                 for(i=0; i<len; i++)
364                         if(key[i] != slot->key[i])
365                                 found = false;
366                 if(found)
367                         break;
368         }
369
370         return slot;
371 }
372
373 list *hasht_getlist(hasht *h, const uint8_t *key, size_t len)
374 {
375         slot *slot;
376         slot = hasht_getslot(h, key, len);
377         if(slot)
378                 return slot->items;
379         return NULL;
```

```
380 }
381
382 void hasht_insert(hasht *h, void *dat, const uint8_t *key, size_t len
    )
383 {
384         slot *slot;
385         size_t index;
386
387         slot = hasht_getslot(h, key, len);
388
389         if(!slot) {
390                 slot = calloc(1, sizeof(*slot));
391                 slot->key = key;
392                 slot->len = len;
393
394                 index = hasht_index(h, key);
395                 PUSH(slot, h->slots[index]);
396         }
397         list_alloc_and_push(&slot->items, dat);
398 }
399
400 void del_required(hasht *requires, hasht *required, script *cur)
401 {
402         script *scr;
403         script *scr2;
404         list *prov;
405         list *prov2;
406         list *lst;
407         list *lst2;
408         slot *s;
409
410         FOREACH(prov, cur->provide) {
411                 lst = hasht_getlist(required, prov->item,
412                                     strlen(prov->item));
413                 for(; lst; lst = lst->next) {
414                         scr = lst->item;
415
416                         FOREACH(prov2, scr->provide) {
417                                 s = hasht_getslot(requires,
418                                                   prov2->item,
419                                                   strlen(prov2->item)
                                                      );
420                                 lst2 = s->items;
421                                 while(lst2) {
422                                         scr2 = lst2->item;
423                                         if(scr2 == cur) {
424                                                 FREE(lst2, s->items);
425                                                 continue;
426                                         }
427                                         lst2 = lst2->next;
428                                 }
429                         }
430                 }
431         }
432 }
433
434 bool has_reqs(hasht *requires, script *cur)
435 {
436         list *lst;
437         list *key;
438
439         FOREACH(key, cur->provide) {
440                 lst = hasht_getlist(requires, key->item,
441                                     strlen(key->item));
442                 if(lst && lst->item)
```

```
443                         return true;
444         }
445         return false;
446 }
```

Código 25 – Código fonte: rcpar.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  #define die()                                                    \
7          do {                                                     \
8            fprintf(stderr, "ERROR: %s %d\n", __FILE__, __LINE__);\
9            exit(EXIT_FAILURE);                                    \
10         } while(0)
11
12 int main(int argc, char *argv[])
13 {
14         int i, status;
15         pid_t pid;
16         const char *start_type;
17
18         start_type = NULL;
19
20         while ((i = getopt(argc, argv, "t:")) != -1) {
21                 switch (i) {
22                 case 't':
23                         start_type = optarg;
24                         break;
25                 }
26         }
27
28         argc -= optind;
29         argv += optind;
30
31         for (i = 0; i < argc; i++) {
32                 /* printf("%s\n", argv[i]); */
33                 if ((pid = fork()) == -1)
34                         die();
35                 if(pid == 0) {
36                         execl("/bin/sh", "sh", argv[i], start_type, (
                               char *)NULL);
37                         die();
38                 }
39         }
40
41         while(wait(&status) > 0);
42
43         return EXIT_SUCCESS;
44 }
```