Projeto 2 - Administração de Processos no Minix

Giovanna Vendramini (173304), Luiz Eduardo T. C. Cartolano (183012) e Rafael Figueiredo Prudencio (186145)

I. Resumo

Através deste projeto, busca-se compreender como é feito o gerenciamento de processos nos sistemas operacionais. O problema diz respeito à criação de um programa em nível usuário denominado *padmon* para gerenciar os processos do sistema através de um serviço *spadmon* que implementa as funcionalidades requeridas com acesso mais privilegiado às estruturas de dados do sistema. O *padmon* deve ser capaz de imprimir o estado dos processos e mudar o estado de um processo dado o seu PID. A solução encontrada envolve apenas a mudança das flags responsáveis pelo estado de cada processo para que o estado impresso pelo padmon seja compatível com o estado impresso pelo programa *ps* do UNIX. Para evitar problemas de acesso compartilhado, optou-se por não escrever na tabela de processos do PM e do kernel por meio do *spadmon*. Portanto, quando a funcionalidade requer uma alteração nas flags do sistema, apenas encaminha-se a chamada para o PM ou kernel, dependendo de qual tem acesso à tabela que deve ser alterada. Em geral, a mudança de estado de um processo funciona como esperado, e o resultado impresso pelo padmon é compatível com o *ps* do UNIX. Entretanto, a mudança de estado de um processo crítico pode corromper o sistema e isso não foi tratado. Além do mais, se rodarmos o MINIX por muito tempo após mudar os estados dos processos, o sistema pode tornar-se instável, pois a mudança de estado não é feita de fato, apenas as flags são setadas. Conclui-se que o *padmon* é capaz de gerenciar corretamente os estados dos processos, considerando-se como correto o estado impresso pelo *ps* do UNIX ser igual ao impresso pelo *padmon*.

II. Introdução

Chamadas de sistema permitem que programas em nível de usuário usem os serviços do sistema operacional. Os usuários não devem ter acesso às estruturas de dados do kernel, pois elas são de suma importância para o funcionamento correto do sistema operacional e haveria uma grande chance de programas criados pelo usuário corromperem estruturas de dados importantes [1].

Para o projeto, foi pedido que implementasse um programa em nível de usuário que gerencia os processos do sistema operacional através de chamadas de sistema para um servidor com acesso privilegiado às estruturas de dados do sistema. Pediu-se um programa em nível de usuário capaz de imprimir os estados e PIDs dos processos em uso, mudar o estado de um processo para zumbi, dormindo, executável ou detido e terminar um processo dado seu PID.

Para as seis funcionalidades do programa foram criadas seis chamadas de sistema, cada uma capaz de realizar uma das tarefas. A função get_pstate realiza uma chamada de sistema para o spadmon que percorre as estruturas de dados privilegiadas e retorna ao usuário as informações relevantes. As funções set_runnable, set_sleep, set_zombie, set_stopped e set_exit são responsáveis por mudar os estados de cada processo ou terminá-los. Vale notar que todos os setters são chamadas de sistema para o spadmon que, por sua vez, encaminha as chamadas para o PM ou o kernel, onde é feita a mudança das flags responsáveis pela impressão do estado de cada processo.

III. Tarefas

A. Projeto e implementação do padmon

A principal escolha de design na implementação do *padmon* foi que o usuário deveria ter acesso ao mínimo de informações necessárias para implementar as funcionalidades requeridas. Essa escolha tornou o padmon um programa extremamente simples, já que seu trabalho principal é a análise dos argumentos da linha de comandos e o tratamento de erros.

A primeira parte do programa consiste na análise dos argumentos da função *main*, onde é determinado se o usuário usou uma sintaxe válida para a execução do programa. A ideia do algoritmo usado é verificar todas as flags passadas como argumentos e armazenar em um vetor de inteiros as opções que foram selecionadas. No caso de haver um argumento numérico, verifica-se a validade da posição do argumento, isso é, se vier depois de uma das opções e (exit), s (sleep), r (running), t (stopped), ou z (zombie), e então o armazenamos como o PID. Em seguida, basta verificar se o conjunto de opções selecionadas formam um subconjunto válido, ou seja, se as opções -ps e -help não estão acompanhadas da opção verbose ou de alguma opção para mudança do estado do processo e se não foram selecionadas mais de uma opção para mudança do estado de um processo. No caso de uma sintaxe inválida, é impressa uma mensagem de ajuda.

A segunda parte do programa consiste na implementação das funcionalidades. Como discutido no início da seção, buscou-se dar o mínimo de informação necessária para o usuário. Portanto, as funcionalidades que envolvem a mudança do estado de um processo resumem-se a uma chamada de sistema que realiza essa operação. A única funcionalidade que requer um pouco mais de esforço por parte do usuário é a de imprimir os PIDs e estados dos processos em uso.

A chamada de sistema get_pstate admite como parâmetro um endereço de memória do mesmo tamanho do vetor do tipo *struct pstate* declarado dentro do *spadmon* e preenche nesse vetor os campos *int pid*, *char state* e *int blocked*. Para verificar se um estado está bloqueado é verificado se o campo blocked é diferente de 0, 1, 31743 e 31744 e, em caso afirmativo, o valor de state é trocado para o caractere 'D'. Essa verificação é necessária para que a impressão dos estados dos processos no padmon seja compatível com a do ps. Em seguida, basta

percorrer o vetor do tipo *struct pstate* e imprimir na saída padrão o PID e estado de todos os processos cujo estado é diferente de 'U' (undefined).

Para os *setters*, primeiro foi verificado se o PID passado como argumento era de um processo existente. Caso não fosse, imprime-se um erro na saída padrão exceto pela função set_exit, que apenas retorna. Além disso, é feita a verificação para determinar se o processo já se encontra no estado especificado nos argumentos e, em caso afirmativo, é impressa uma mensagem se a flag verbose estiver setada ou apenas retorna caso contrário. Também foi feita uma verificação do valor de retorno de todas as chamadas de sistema para o *spadmon*, que retorna 0 no caso de sucesso e um valor positivo em caso de falha.

É importante notar que as chamadas de sistema fazem uma simples alteração nas flags das estruturas de dados do sistema e isso pode levar a um comportamento imprevisível e indesejável por parte do sistema operacional na hora de gerenciar esses processos. Portanto, dentro do *padmon* foi tomado um cuidado especial para não alterarmos os estados de processos que têm grandes chances de causar problemas ao PM ou o kernel. Nas funções *setters*, é impressa uma mensagem de erro na tentativa de mudar um processo que encontra-se no estado zumbi ou bloqueado para os estados dormindo, executável ou detido. No caso da função *exit*, verificamos que o processo existe e não é um zumbi antes de fazer a chamada de sistema e é impresso um erro caso contrário. A manipulação do estado de um processo zumbi cujo pai é o processo *init* é imprevisível, já que está sujeito a descarte pelo sistema operacional devido à chamada de *wait* em loop dentro do *init*.

Para que o *padmon* faça parte da varíavel *PATH* do MINIX e seja executado como um programa do UNIX convencional (e.g. *ls*), adicionou-se uma pasta no diretório *bin/* denominada *padmon*. Dentro dessa pasta, copiou-se o Makefile do programa *rm*, escrevendo *padmon* no lugar de *rm*, foi feitauma cópia o arquivo *ps.1* do diretório *bin/ps* renomeando-o para *padmon.1* e foi colocado o código fonte do *padmon*. Por fim, alterou-se o *Makefile* do diretório *bin*, adicionando o *padmon* na lista de diretórios que devem ser compilados. Dessa forma, o executável do *padmon* estará situado na pasta */bin* e ao executar um comando no terminal essa pasta sempre será buscada, possibilitando que o usuário rode o *padmon* de qualquer diretório do sistema.

B. Projeto e implementação do spadmon

No desenvolvimento do spadmon, a principal preocupação foi evitar problemas de concorrência com as estruturas de dados privilegiadas do sistema operacional, em particular, as tabelas de processos do gerenciador de processos (PM) e do kernel. Portanto, todas as chamadas de sistema que requerem uma alteração nessas tabelas foram redirecionadas para o PM ou o kernel, dependendo de quem possui acesso às flags necessárias.

Para implementar chamadas de sistema no *spadmon* capazes de retornar mensagens ao nível usuário, foi necessário contornar as checagens de segurança do kernel. Em particular, é feita uma checagem no arquivo *proc.c* no diretório *minix/kernel*, onde é verificado se os processos têm autorização para comunicar-se entre si através da função *may_send_to* definida no arquivo *priv.h* do mesmo diretório. Para permitir a livre comunicação com o *spadmon*, basta ignorar essa verificação quando o endpoint de quem manda a mensagem e quem recebe a mensagem vale 11, ou seja, quando algum processo deseja comunicar-se com o *spadmon*. Isso pode ser feito verificando que *caller_ptr->p_endpoint* e *src_dst_e* são diferentes de 11 antes de realizar a chamada para *may_send_to*.

Para implementar a chamada de sistema *get_pstate*, não foi necessário redirecionar a chamada, já que ela requer apenas a leitura do estado das tabelas. Dentro do diretório *spadmon*, no diretório de servidores do MINIX, implementou-se a função *do_getpstate* no arquivo *get_table.c* que é chamada sempre que houver uma chamada de sistema com número 1 para o servidor *spadmon*. Para que o padmon tenha um comportamento análogo ao ps do UNIX, usou-se o mesmo código do servidor de base de informações gerenciais (MIB) para determinar o estado de cada processo. Dessa forma, é necessário fazer a leitura da tabela de processos do kernel e do PM para que possamos ler os campos *mp_pid* e *mp_flags* pertencentes a *struct mproc* e *p_rts_flags* da *struct proc*. Para obter uma cópia da tabela de processos do kernel foi utilizada a chamada de sistema *get_sysproctab* e para a do PM usou-se a chamada *get_sysinfo*, onde é retornado um erro no caso de alguma falha na obtenção das tabelas. Com as duas tabelas em mãos, basta percorrer a tabela de processos do PM (a do kernel tem mais processos que não nos interessam e.g. drivers) e para todo processo que tiver a flag *IN_USE* setada, devemos determinar o seu estado atual. O estado de maior precedência é o zombie, que é verificado primeiro através das flags *ZOMBIE* e *TRACE_ZOMBIE* no campo *mp_flags* da tabela do PM. Em seguida é feita a verificação se a flag *TRACE_STOPPED* do campo *mp_flags* ou *RTS_P_STOP* do campo *p_rts_flags* está setada para determinar se o processo está detido. Depois, é verificado se o campo *p_rts_flags* vale 0, indicativo que o processo está executando-se ou é executável. Caso o processo não satisfaça alguma das condições anteriores, considera-se que ele se encontra dormindo. Os processos que não têm a flag *IN_USE* setadas, têm seu estado como 'U' de undefined. Dentro do *spadmon* foi feita a declaração de um vetor de escopo global no arquivo *sproc.h* do tipo *struct pstate* com os campos *char state*, *int blocked* e *int pid*. O campo *state* é atualizado através das verificações descritas acima e os campos *blocked* e *pid* são atualizados a partir do retorno da função *P_BLOCKEDON* e do valor do campo *mp_flags* da tabela de processos do PM. Finalmente, copia-se o vetor do tipo *struct pstate* para o endereço passado como argumento através da chamada ao kernel *sys_datacopy*.

As demais chamadas de sistema foram implementadas no arquivo *forward.c* do servidor *spadmon*. O valor de retorno de todas as chamadas é 0 caso um processo com o PID passado como argumento seja encontrado na tabela de processos e um valor positivo caso contrário. Não são feitas verificações adicionais, pois assume-se que o usuário as fará, como no caso do *padmon*. As funções *do_setzombie* e *do_setstopped* realizam uma nova chamada de sistema para o servidor PM, pois ambas realizam uma escrita na tabela de processos do PM. Foram criadas outras duas chamadas de sistema dentro

do PM no arquivo *spadmon.c* com o mesmo nome e que realizam esse trabalho. A função *do_setzombie* seta ambas as flags *ZOMBIE* e *TRACE_ZOMBIE* do campo *mp_flags* do processo com o pid passado como argumento através do campo *m_lc_pm_sig.pid* da mensagem. A função *do_setstopped* apenas seta a flag *TRACE_STOPPED* do campo *mp_flags*. A função *do_setexit* do servidor *spadmon* também encontra-se no arquivo *forward.c* e realiza uma simples chamada de sistema de kill para o PM. Optou-se por essa simplificação pois implementar uma chamada de sistema que fizesse o mesmo trabalho do *SIG_KILL* seria muito complexo e foge da motivação do trabalho, que busca entender o gerenciamento de processos de forma geral, sem preocupar-se necessariamente com o gerenciamento de memória do sistema.

As funções *do_setrunning* e *do_setsleep* do spadmon são primeiro redirecionadas para o PM e depois para o kernel, onde são implementadas com o mesmo nome no arquivo *spadmon.c* dentro do diretório *minix/servers/pm* e *minix/kernel/system*, respectivamente. As chamadas para o PM são necessárias para removermos as flags do campo *mp_flags* referentes ao estado zumbi e detido, que podem estar setadas e têm uma precedência maior na determinação do estado de cada processo. A chamada para o PM deve também encontrar o índice no vetor de processos que corresponde ao PID do processo passado como parâmetro. Isso pode ser feito percorrendo a lista de processos do PM em busca de um processo que satisfaz a condição de ter a flag *IN_USE* setada do campo *mp_flags* e o campo *mp_pid* igual ao PID passado como parâmetro. Feito isso, o PM realiza uma chamada de sistema para o kernel preenchendo o campo *m_lc_pm_sig.pid* da mensagem com o índice do processo na tabela de processos. As chamadas para o kernel requerem um cuidado especial, pois elas esperam indefinidamente por um retorno para progredir dentro de um loop da forma *while(1)*. Portanto, a implementação da função *do_setrunnable* consiste em setar a o campo *p_rts_flags* para 0 e retornar OK (vale 0). A função *do_setsleep* primeiro verifica se a flag *p_rts_flags* vale 0 ou se a flag *RTS_P_STOP* está setada e só caso uma das condições sejam satisfeitas, zeramos a o campo *p_rts_flags* e depois setamos a flag para *RTS_SLOT_FREE* antes de retornar OK.

Seguindo o padrão de chamadas de sistema do UNIX, o protótipo das seis chamadas de sistema foram inclusos no arquivo *unistd.h* dentro do diretório *include*. O objetivo era ocultar ao usuário o protocolo de comunicação entre processo no MINIX, que é feito através de mensagens. As funções foram declaradas no arquivo *lib/libc/sys/spadmon.c*, onde a chamada para *get_pstate* cria uma mensagem com o ponteiro para o vetor de *struct pstate* e as demais chamadas de sistema criam mensagens com apenas o PID do processo que deseja-se alterar o estado. Essas mensagens são encaminhadas para o *spadmon* através de uma chamada à função *_syscall* com a constante *SPADMON_PROC_NR*, o número da chamada de sistema e um ponteiro para a mensagem passados como parâmetros.

C. Comandos para se realizar os serviços

Para criar um serviço no spadmon foram seguidos os passos indicados em [2]. O primeiro passo é copiar um dos servidores do MINIX na pasta *minix/servers* e renomeá-lo para *spadmon*. Escolheu-se copiar o PM, pois este já havia uma estrutura para retornar mensagens ao nível usuário, devido a chamadas de sistema como *fork* e *getpid*. Manteve-se apenas o arquivo *main.c*, *Makefile*, *glo.h* e *proto.h* do PM, onde foram removidas as linhas de código que não eram pertinentes à implementação de um novo servidor, como o corpo da função *sef_cb_init_fresh* que trata-se da inicialização da tabela de processos do PM. Em seguida, foi necessário adicionar o servidor spadmon à lista de diretórios que devem ser compilados no Makefile da pasta *minix/servers*. O próximo passo foi adicionar o serviço ao arquivo de comunicações do sistema *minix/include/minix/com.h*, onde definiu-se o *endpoint* do *spadmon* denominado *SPADMON_PROC_NR* como 11 e deslocou-se o *LAST_SPECIAL_PROC_NR* para o 12. Em seguida, foi necessário escolher um intervalo de números para as chamadas de sistema do *spadmon*. Escolheu-se o menor intervalo desocupado, sendo este os números entre 0x1B00 e 0x1BFF. Esse intervalo deve então ser usado para definir a constante *SPADMON_BASE* no arquivo *minix/include/minix/callnr.h* como 0x1B00. Nesse mesmo arquivo, basta seguir o mesmo padrão do PM e VFS para definir constantes mapeadas para inteiros no intervalo de números escolhido que serão usadas para realizar chamadas de sistema para o *spadmon* [3].

O próximo passo é adicionar o spadmon à sequência de boot. Para isso, é necessário adicionar a linha *{SPADMON_PROC_NR, "spadmon"}* na última posição do vetor do tipo *struct boot_image* no arquivo *minix/kernel/table.c*. Depois, adicionou-se no arquivo *minix/servers/rs/table.c* a linha *{SPADMON_PROC_NR, "spadmon", SRV_F}* depois da linha onde está o servidor MIB no vetor *boot_image_priv_table* e a linha *{SPADMON_PROC_NR, SRVR_SF}* após a linha do VFS no vetor *boot_image_sys_table*. Finalmente, foi necessário adicionar o servidor ao arquivo *etc/system.conf*, onde foi feita uma cópia exata da declaração do serviço DS, apenas alterando o nome para *spadmon*. Por fim, adicionou-se a linha *PROGRAMS+= ${PROGROOT}/minix/servers/spadmon/spadmon* ao arquivo *releasetools/Makefile* para que o sistema compile o novo servidor ao executar um *make hdboot*.

D. Validação do projeto

A validação do projeto consiste na comparação dos estados dos processos com o programa *ps* do UNIX. Não foram feitos testes para validar exclusivamente o *spadmon*, já que essa avaliação está contemplada na do *padmon*. Criou-se um conjunto de oito testes que avaliam cada uma das funcionalidades do programa. sete dos testes foram feitos em *Python* usando a biblioteca *unittest* com o intuito deixá-los mais modularizados. Um dos testes foi manual, através da manipulação do estado de processos como uma shell conectada via *ssh*.

O teste mais simples envolve uma comparação da saída do programa ao executar *padmon -help* na linha de comando. O teste para avaliar a o comando *padmon -ps* envolve a comparação dos estados dos processos com os obtidos pelo *ps*. É importante notar que inevitavelmente, haverá diferenças. Primeiro, o *ps* comunica-se com o MIB para obter suas informações e, portanto, esse serviço aparece como executando-se

na saída do *ps*. Por sua vez, o *padmon* puxa as informações do *spadmon* e esse servidor é o que aparece como executando-se na saída do programa. Além do mais, o próprio *ps* aparece como um processo bloqueado na saída do programa, da mesma forma que o *padmon* aparece como bloqueado quando executa-se o programa com a flag *-ps*. Obviamente, o *ps* não aparece quando executa-se o *padmon* e vice-versa. Uma outra diferença é a representação dos estados no *ps*, que oferece informações mais detalhadas a respeito de cada processo além de novos estados. Considerando-se o estado I do *ps* como equivalente ao estado S do *padmon* e ignorando os caracteres com informações adicionais do processo, somos capazes de comparar as saidas dos dois programas. Levando essas diferenças em consideração, o teste do *-ps* compara todos os estados dos processos do comando *ps aux* com os estados impressos ao executar-se *padmon -ps*. Em caso de sucesso, esse teste mostra-nos que o *padmon* é capaz de fazer uma leitura coerente dos estados dos processos no sistema. Note que não podemos dizer que é a leitura correta, pois essa está sujeita a interpretação e às vezes muda com a versão do *ps*.

Os demais testes envolvem a criação de um processo para em seguida manipulá-lo como desejarmos. Isso pode ser feito em *Python* através da função *Popen()* da biblioteca *subprocess* que permite rodar um processo no background. Dessa forma, optou-se por rodar um simples programa em C que apenas coloca-se para dormir com a linha *sleep(100)* dentro da função *main* e então troca-se o estado desse programa para fazer os testes do *padmon*. Os testes de cada funcionalidade são simples: pegou-se o PID do processo recém-criado e executou-se o comando da forma *padmon -t <PID>*. Para que o estado do *ps* atualize-se é necessário esperar um tempo após a execução do comando antes de verificar se o estado foi alterado no *ps*, que pode ser feito com a função *sleep* da biblioteca *time*. O teste descrito acima foi feito para as opções exit, run e stop. O teste para a funcionalidade sleep não pode ser feito da mesma forma, já que o processo já encontra-se adormecido. Primeiro mudou-se o programa para o estado detido usando o comando *padmon -t <PID>*, verificou-se caso o estado impresso no *ps* do UNIX é o esperado para depois executar *padmon -s <PID>*. O teste é considerado bem sucedido caso o *ps* imprima que o processo encontra-se adormecido.

O teste da opção verbose foi feito colocando um processo no estado detido com um comando da forma *padmon -tv <PID>*, onde verificou-se a impressão de uma mensagem de sucesso na saída padrão.

Até agora, todos os testes descritos funcionaram normalmente no ambiente de integração contínua do *GitLab*, já que o script em *Python* permite a execução de um processo simples como um programa em C no *background*. Entretanto, não foi possível testar a funcionalidade zombie no ambiente de integração contínua, já que ao mudar o estado para zumbi de um processo rodando em background faz com que o sistema operacional imediatamente termine-o. Dessa forma, a funcionalidade do zombie foi testada apenas localmente, manipulando processos como o *grep* ou shells conectadas por *ssh* (processos que não rodam em backgound). Nesses casos, executar o comando *padmon -z <PID>* faz com que o estado do processo impresso pelo ps seja alterado para zumbi, como pode ser visto na Figura 1.

```
service 272  0.0  0.1  1304  1304 ?  S    10:02PM 0:00.02 /service/lwip
root    288  0.0  0.0   476   288 ?  S    10:02PM 0:00.02 /service/uds
root    292  0.0  0.0   208   208 ?  I    10:02PM 0:00.00 /service/ipc
service 296  0.0  0.0   164   164 ?  S    10:02PM 0:00.00 /service/log
root    305  0.0  0.0   112   112 ?  S    10:02PM 0:00.00 /service/vbox
root    307  0.0  0.0   184    76 ?  S    10:02PM 0:00.00 update
root    309  0.0  0.0   528   288 ?  I    10:02PM 0:00.00 cron
root    482  0.0  0.0   896   656 ?  Is   10:02PM 0:00.08 dhcpcd -qM le0
root    562  0.0  0.0  2232   840 ?  Ss   10:02PM 0:00.00 /usr/sbin/syslogd -s
root    641  0.0  0.1  4396  1904 ?  Ss   10:02PM 0:00.02 /usr/pkg/sbin/sshd
root    650  0.0  0.0   544   220 ?  Is   10:02PM 0:00.00 /usr/sbin/inetd -l
root    764  0.0  0.1  4420  2304 ?  Ss   10:15PM 0:00.05 sshd: root@pts/0 (ssh
root    765  0.0  0.1  4420  2304 ?  Ss   10:15PM 0:00.07 sshd: root@pts/3 (ssh
root    766  0.0  0.1  4420  2304 ?  Ss   10:15PM 0:00.08 sshd: root@pts/2 (ssh
root    767  0.0  0.1  4420  2304 ?  Ss   10:15PM 0:00.08 sshd: root@pts/1 (ssh
root    768  0.0  0.0     0     0 ?  Zs   10:15PM 0:00.00 (sh)
root    666  0.0  0.0   828   576 co Ss+  10:02PM 0:00.10 -sh
root    786  0.0  0.0   476   368 co D+   10:15PM 0:00.00 ps -aux
root    667  0.0  0.0   420   244 c1 Is+  10:02PM 0:00.00 /usr/libexec/getty de
root    668  0.0  0.0   420   244 c2 Is+  10:02PM 0:00.00 /usr/libexec/getty de
root    669  0.0  0.0   420   244 c3 Is+  10:02PM 0:00.00 /usr/libexec/getty de
root    772  0.0  0.0   836   572 1  Ss+  10:15PM 0:00.02 -sh
root    773  0.0  0.0   836   572 2  Ss+  10:15PM 0:00.00 -sh
root    774  0.0  0.0   836   572 3  Ss+  10:15PM 0:00.00 -sh
minix# _
```

Fig. 1. Saída após a execução do comando *ps aux* depois de mudar o estado de uma shell conectada por *ssh* para o estado zumbi através do *padmon*. O processo com PID 768 encontra-se no estado zumbi.

Esses testes mostram que as funcionalidades implementadas no *padmon* são capazes de mudar a aparência do estado dos processos, já que um programa de terceiros tem a mesma interpretação do *padmon* a respeito do estado do processo após uma mudança.

IV. Conclusão

Foi possível construir um programa *padmon* em nível de usuário capaz de gerenciar os processos do sistema operacional através de chamadas de sistema para um serviço *spadmon* com acesso mais privilegiado às estruturas de dados do sistema. O *padmon* é capaz da analisar os argumentos passados na linha de comandos e detectar erros de sintaxe e execução, fazendo apenas chamadas de sistema quando as mesmas fazem sentido dado o estado atual dos processos. O serviço *spadmon* foi criado com a capacidade de fazer uma leitura das tabelas de processos do PM e do kernel e realizar chamadas de sistema para o PM ou kernel, alterando as duas tabelas de processos quando necessário. Os testes unitários demonstram a corretude tanto do *padmon* como do *spadmon*, que funcionam em conjunto. Os testes mostram que a interpretação dos estados tanto como a mudança de estado dos processos do sistema é coerente com a interpretação do programa *ps* do UNIX. Como a implementação escolhida para o *spadmon* realiza apenas a mudança das flags que caracterizam o estado do processo, o sistema pode tratar um processo de forma indevida e entrar em pane. Por exemplo, um processo crítico cujo pai é o processo init com PID 1 e que está executando-se, pode ser alterado para o estado zumbi e descartado da lista de processos. Para um projeto futuro, seria interessante tratar erros associados à mudança de flags de processos críticos do sistema e de fato implementar a alteração de estado, desenfileirando um processo e enfileirando-o corretamente.

References

[1] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[2] Creating a service in minix. Accessed 18-04-2018. [Online]. Available: https://blog.heabuh.com/jekyll/update/2016/11/13/minix-service-tutorial.html

[3] Minix 3 wiki. Accessed 16-03-2018. [Online]. Available: https://wiki.minix3.org