Isolando Faltas em Sistemas Operacionais: Características e Abordagens

Luís Fernando Friedrich
Universidade Federal de Santa Catarina
PET Computação
fernando@inf.ufsc.br

Resumo

Este artigo explora os princípios e práticas do isolamento de drivers de dispositivos de baixo nível com o propósito de melhorar a confiabilidade do Sistema Operacional. O artigo mostra as mais frequentes causas de falhas em Sistemas Operacionais e apresenta algumas abordagens que buscam a eliminação destas falhas e consequente melhoria da confiabilidade.

1. Introdução

Para muitos usuários de computadores, o maior problema na utilização de computadores é a percepção de que eles são não confiáveis. Pesquisadores e profissionais de computação estão acostumados falhas regulares dos computadores e para eles instalar alterações todos os meses pode ser considerado normal. Entretanto, a maioria dos usuários considera esta falta de confiabilidade não aceitável. Para estes, o modelo de funcionamento de um dispositivo eletronico esta baseado na experiência com utilitários como as Televisões, ou seja, compra-se, liga-se e ela funciona perfeitamente durante os próximos anos. Sem falhas, sem atualizações de software, sem notícias de virus recentes. Para tornar computadores mais parecidos com TVs, a meta seria melhorar a confiabilidade dos sistemas de computação, iniciando com o Sistema Operacional [1].

2. Porque os Sistemas Falham?

A razão principal para a falha (queda) dos sistemas operacionais pode ser associada a questões relativas ao projeto que todos os sistemas compartilham: muitos privilégios e falta de isolamento de falhas adequado. Virtualmente, todos os sistemas operacionais consistem de muitos módulos ligados juntos em um espaço de endereçamento para formar um único programa binário que executa em modo kernel. Um bug em qualquer módulo pode facilmente corromper estruturas de dados críticas em um módulo não relacionado e derrubar o sistema. A razão para fazer isto é: melhor desempenho a custo de mais falhas no sistema.

Porque as falhas (quedas) acontecem? Se cada módulo fosse perfeito, não existiria necessidade de isolamento de falhas entre os módulos porque não existiriam falhas para propagar.Muitos defendem que a maioria das falhas (faltas) são devido a erros de programação(bugs) e são principalmente consequência de muita complexidade e utilização de código externo.

Estudos mostram que em média o software apresenta 1-16 bugs por 1000 linhas de código e que esta faixa é certamente subestimada porque apenas os bugs que forma encontrados são contados. A conclusão óbvia destes estudos é que mais código significa mais bugs. Considerando o próprio desenvolvimento de software, cada nova versão (release) tende a adquirir mais características (e assim mais código) e é frequentemente menos confiável do que sua predecessora. Estudos mostram que o numero de bugs por mil linhas de código tende a estabilizar depois de várias versões.

Alguns destes bugs são explorados por ataques para permitir que virus infectem e causem estragos nos sistemas. Assim, alguns problemas considerados de segurança, não tem nada a ver com as medidas de segurança em princípio (falha de criptografia ou protocolos de autorização quebrados) mas são simplesmente devido a erros no código, como por exemplo: transbordo de buffer, o que

permite execução de código injetado.
O segundo problema é a introdução de código externo no sistema operacional. Usuários mais sofisticados nunca permitiriam a inserção de código, por parte de terceiros, no coração do seu sistema operacional, mesmo que quando comprar um novo dispositivo periférico e instalar o driver, o que é precisamente o que estão fazendo. Drivers de dispositivos são usualmente escritos por programadores que trabalham para os fabricantes dos periféricos, que tipicamente tem menor controle de qualidade do que um fabricante de sistema operacional. Quando o driver é resultado de um esforço "open-source", é frequentemente autoria de voluntários bem intencionados mas não necessariamente experientes e com menor controle de qualidade. No Linux, por exemplo, a taxa de erros em drivers de dispositivos é 3 a 7 vezes maior do que no resto do kernel. Mesmo na Microsoft, que tem a motivação e os recursos para aplicar controle de qualidade rígido, não é muito melhor: 85% de todos falhas (quedas) do XP são devido a erros de código nos drivers de dispositivo.
Consertar erros de código em drivers é uma tarefa praticamente impossível devido as constantes mudanças que as as configurações sofrem. Além disso, a manutenção de drivers existentes é difícil de realizar devido as mudanças nas interfaces do kernel e o crescimento do código. Uma análise do Sistema Linux, kernel versão 2.6, mostrou um crescimento sustentado em Linhas de Código (LoC) de cerca de 5,5% a cada 6 meses, como mostra a Figura 1 [2]. Nos últimos anos, o crescimento do kernel foi de 49,2%, ultrapassando 5M de linhas de código executável, na sua grande maioria devido aos drivers de dispositivos, cerca de 57,6% do kernel ou 3.0M linhas de código.

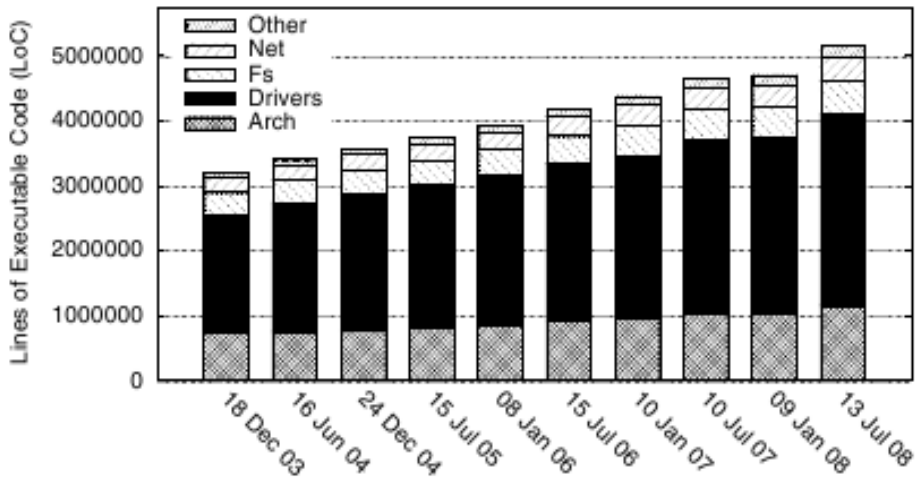

Figura 1. Crescimento do Sistema Operacional Linux, kernel 2.6, desde sua liberação[1]

Por décadas, a técnica aprovada para tratar código não confiável tem sido isolar este código, por exemplo, colocando o mesmo em um processo separado e executando o mesmo em modo usuário. Algumas pesquisas apontam que o aumento da confiabilidade do Sistema Operacional só pode ser alcançada a partir da execução de cada driver de dispositivo como se fosse um processo de usuário com apenas privilégios mínimos necessários. Desta forma, código com bug é isolado, não causando a queda do sistema.

3. Características de Confiabilidade

É importante entender que a confiabilidade de um sistema operacional pode ser afetada de forma direta e imediata em função da escolha da estrutura de sistema operacional a ser adotada. Assim,

para ilustrar esta influência vamos considerar duas estruturas bem conhecidas neste caso: sistemas monolíticos e sistemas baseados em micro-kernel.
No caso de sistemas monolítico padrão, o kernel contém o sistema operacional todo em um único espaço de endereçamento e executando em modo kernel, como mostra a Figura 2 [Tan]. Nesta forma de estruturação do sistema é possível identificar problemas que podem afetar de maneira significativa a confiabilidade do Sistema Operacional, como por exemplo: falta de isolamento de falhas apropriado; todo código executa no mais alto nível de privilégio (kernel); grande quantidade de código, implicando muitos bugs; código não confiável de terceiros no kernel; difícil manutenção devido a complexidade.

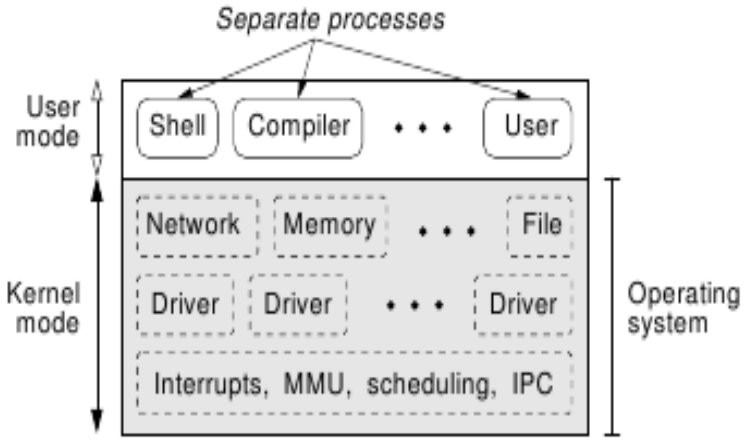

Figura 2. Estrutura de um Sistema Operacional monolítico padrão[2]

No outro extremo esta a estrutura baseada em microkenel, que contém apenas a funcionalidade mínima necessária, ou seja, o kernel mínimo que oferece tratamento de interrupção, um mecanismo para iniciar e parar processos, um escalonador e comunicação entre processos. Funcionalidade padrão de sistema opracional, como leitura e escrita em arquivos, que esta presente em um kernel monolítico é movido para o espaço de usuário, e não executa mais como mais alto nível de privilégio, como mostrado na Figura 3. Assim, é possível executar todos os módulos não confiáveis como processos separados incluíndo servidores e drivers, em modo usuário, fora do kernel permitindo assim a sua recolocação de forma mais independente, tornando o sistema mais modular.
A partir do entendimento da influência que a estrutura do sistema pode ter na questão confiabilidade, podemos apontar algumas questões importantes, como as citadas a seguir, que devem ser consideradas e que certamente trazem melhoramentos na confiabilidade dos sistemas.

Reduzir o numero de faltas críticas - Uma das estratégias muito importantes nesta questão é um kernel muito pequeno. É bem entendido que mais código significa mais bugs, assim significa menos bugs no kernel. De acordo com [2], usando uma estimativa de 6 bugs por 1000 linhas de código executável como limite inferior, em um sistema baseado em microkernel, com 3800 linhas de código executável, como o caso do MINIX3[3], teremos provavelmente pelo menos 22 bugs no kernel. Na comparação com um sistema monolítico como o Linux com 2.5 milhões de linhas de código executável no kernel pode ter pelo menos 6 x 2500 = 15,000 bugs.

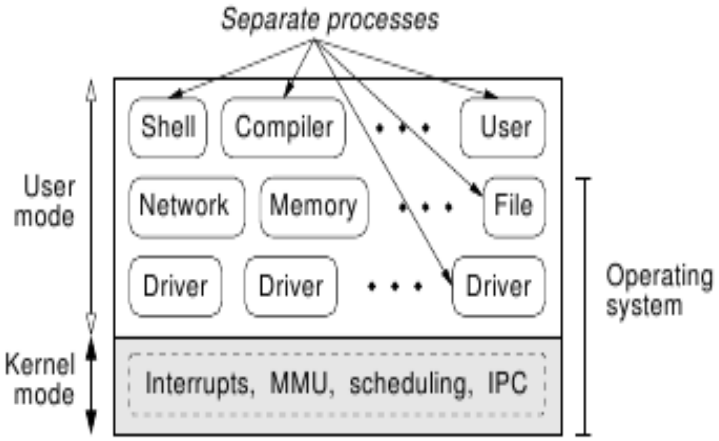

Figura 3. Estrutura de um Sistema Operacional baseado em microkernel[2]

Limitar os estragos que cada erro (bug) pode causar - Reduzir o tamanho do kernel não reduz a quantidade de código presente. Apenas desloca uma grande parte dele para o modo usuário. Entretanto, esta mudança tem um efeito profundo na confiabilidade. Código de kernel tem acesso completo para tudo o que a máquina pode fazer. Erros (bugs) que acontecem no kernel podem acidentalmente disparar operações de E/S, executar operações de E/S de forma errada, interferir com o mapa de memória ou muitas outras coisas que erros em programas em modo usuário não podem causar. Desta forma, é possível argumentar que quando um erro (bug) ocorre, os efeitos serão menos devastadores se os mesmos forem convertidos de bugs em modo kernel para bugs em modo usuário.

Recuperar de falhas comuns - Se tivermos capacidade de detectar o término ou queda de um driver ou serviço do sistema será possível automaticamente recuperá-lo através do seu re-ínicio. O monitoramento é feito periodicamente e quando um driver não responde corretamente dentro de um tempo especificado o procedimento é retirar definitivamente o driver do sistema e reinicia-lo. Sistemas monolíticos não tem esta capacidade de detectar drivers com falhas durante a execução dos mesmos.

4. Abordagens propostas

Existem várias propostas para melhorar a confiabilidade ou segurança a partir do isolamento de drivers. Vamos apresentar quatro abordagens diferentes em um contexto que inclui abordagens legadas e técnicas de isolamento novas.

4.1 Empacotamento de Drivers em Software

Um projeto de pesquisa importante que busca a construção de um sistema confiável na presença de drivers de dispositivos não confiáveis é o Nooks [3]. O Nooks tem como objetivo melhorar a confiabilidade dos sistemas operacionais atuais. Nas palavras dos autores: “o nosso alvo são extensões para sistemas operacionais existentes e não propor uma nova arquitetura. Nós queremos que estas extensões executem nas atuais plataformas se possível sem mudanças.” Ou seja, a ideia é ser compatível para com sistemas existentes, legados, mas pequenas mudanças são permitidas.
A abordagem Nooks, mostrada na Figura 4 [Tan, 2006], é manter drivers de dispositivos no kernel mas cercar os mesmos em um tipo de envólucro de proteção leve de forma que os bugs dos drivers não possam se propagar para outras partes do sistema operacional. O Nooks faz isto interpondo de

forma transparente um nível de confiabilidade (“Nooks isolation manager”) entre o driver de dispositivo que esta sendo envolvido e o resto do sistema operacional. Todo o tráfego de controle e dados entre o driver e o resto do kernel é inspecionado pelo nível de confiabilidade. Quando o driver é iniciado, o nível de confiabilidade modifica o mapa de páginas do kernel para desligar acessos de escrita para páginas que não são parte do driver, desta forma prevenindo o mesmo de modificar diretamente estas páginas. Para suportar acessos legítimos de escrita para as estruturas de dados do kernel, o Nooks copia os dados necessários no driver e depois os copia de volta depois da modificação, funcionando como o método “write-back” usado em memória cache.

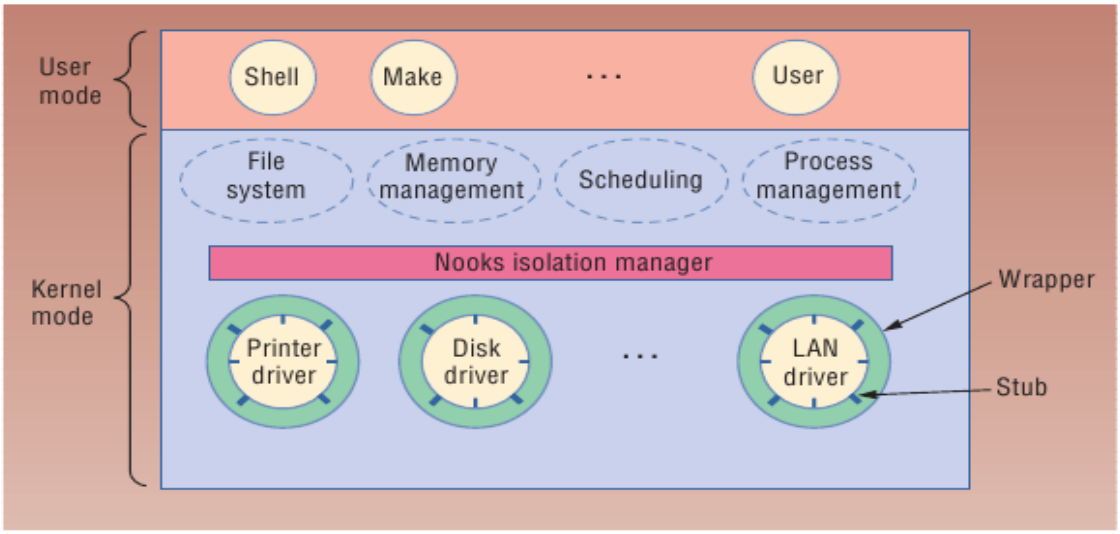

Figura 4. Modelo Nooks. Os drivers tem uma camada de proteção.
(fonte: [Tan et. All, 2006])

Embora a questão de tornar sistemas operacionais legados mais confiáveis seja importante, no que diz respeito a pesquisa também é importante atentar para a seguinte questão: Como os futuros sistemas operacionais deverão ser projetados para prevenir o problema da confiabilidade?
Com relação ao impacto do ponto de vista de código inserido a mais, o modelo Nooks tem 22.000 linhas de código, o que pode certamente significar um aumento no numero de erros de software (bugs).

4.2 Empacotamento de Drivers com Máquinas Virtuais

Outro projeto que encapsula drivers, utiliza o conceito de máquina virtual para isolar os drivers do resto do sistema [5,6]. Quando o driver é acionado, ele executa em uma máquina virtual diferente daquela que executa o sistema principal, como mostra a Figura 5. Assim, uma quebra ou outra falta não corrompe o sistema principal. Da mesma forma que o Nooks, esta abordagem é inteiramente dirigida para drivers legados em sistemas operacionais legados.
Mesmo considerando que esta abordagem realiza o que se propõe, alguns problemas podem ser apontados. Primeiro, existem questões que dizem respeio a confiança entre o sistema principal e a máquina virtual dos drivers, ou seja, até onde esta confiança vai ou deve ir. Segundo, executar um driver em uma máquina virtual levanta questões de tempo (timing) e de bloqueio (locking), porque todas as máquinas virtuais são compartilhadas no tempo e um driver do kernel que foi projetado para executar até o término sem interrupção pode inesperadamente ter seu tempo particionado, de forma involuntária e com consequencias imprevisíveis. Terceiro, alguns recursos, tal como espaço de configuração do barramento PCI, podem ser compartilhados entre várias máquinas virtuais.

Quarto, o mecanismo da máquina virtual consome recursos extras, embora o custo seja baixo.
O principal problema ainda é o fato de que esta abordagem é desajeitada e principalmente adequada para proteger drivers legados em sistemas operacionais legados ao contrário de ser usado em novos projetos.

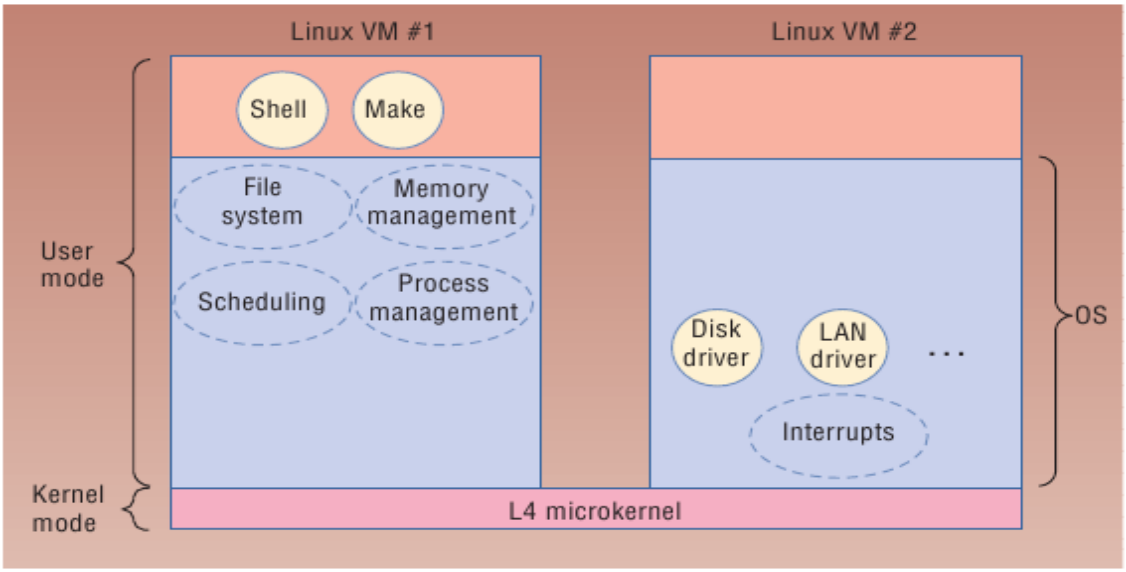

Figura 5. Encapsulamento com máquina virtual.
(fonte: [Tan et. All, 2006])

4.3 Proteção baseada em Linguagem

Outra abordagem é utilizar proteção baseada em linguagem e verificação formal para isolar drivers. Por exemplo, OKE [7] Open Kernel Environment fornece um ambiente seguro e controlado com relação a recursos que permite a carga de código nativo totalmente otimizado no kernel do sistema operacional Linux. O código é compilado com um compilador de propósito específico Cyclone que adiciona instrumentação ao código objeto de acordo com a política que corresponde aos privilégios do usuário. Cyclone, assim como Java, é uma linguagem tipada (type-safe language) na qual a maioria dos erros com apontadores (pointers) são evitados pelas definições da linguagem. O gerenciamento da confiança e o controle da autorização explícita garante que os administradores são capazes de exercer rigoroso controle sobre quais grupos são dados quais privilégios e este controle é automaticamente imposto nos seus códigos.
O compilador aplica regras extras (restrições) a compilação do código do usuário, baseado nas credenciais que o usuário pode apresentar. Como resultado, o código gerado, pode incluir verificações dinamicas para propriedades de segurança que não podem ser verificadas estaticamente. Depois da compilação, o módulo esta 'seguro' no que diz respeito as restrições de segurança impostas pelas regras extras. Por exemplo, ele pode garantir que o código: não irá usar muito tempo de CPU ou memória, não irá quebrar o kernel, pode apenas acessar certas funções do kernel e determinados bits da memória, etc.
A Figura 6 mostra no primeiro plano um usuário que deseja incorporar no kernel um módulo construído por ele. O usuário fornece ao compilador o código fonte juntamente com suas credenciais. O compilador seleciona, com base nas credenciais, o conjunto de customizações adequadas para incluir na compilação e compila o código. Se não ocorrer erros nesta etapa é gerado um código objeto pronto para a carga. Num segundo momento o usuário fornece ao carregador o código objeto e suas credenciais que são verificadas contra o código e se elas combinam o carregador carrega o código para o kernel.

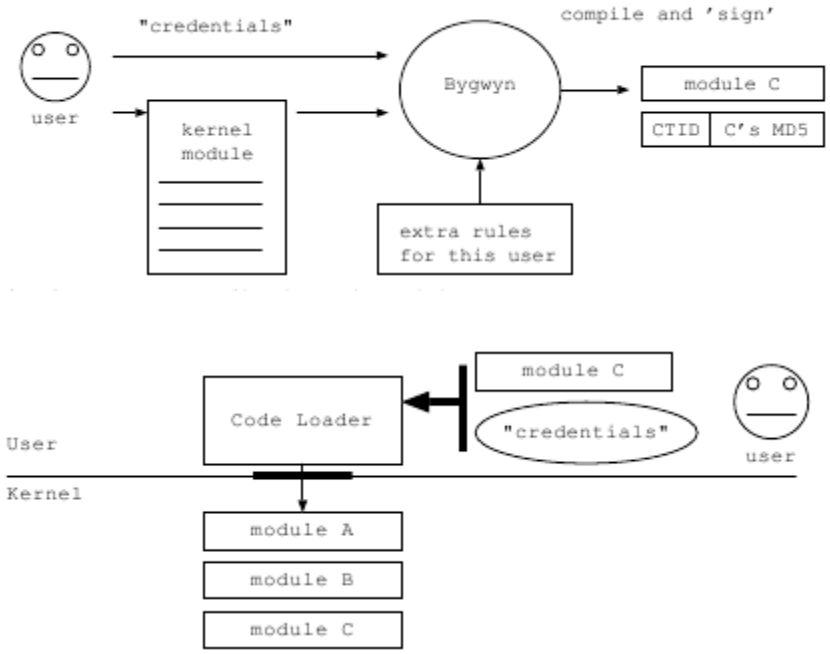

Figura 6. Fases de Compilação e Carga na Proteção baseada em Linguagem.
(fonte: [Tan et. All, 2006])

Proteção baseada em linguagem é a abordagem utilizada em outros projetos como: Singularity [8], que combina linguagens "type-safe" com verificação de protocolo e isola os processos depois da carga. O projeto seL4 [9], que visa um microkernel formalmente verificado através do mapeamento para uma implementação correta demonstrável. Devil [10], que é uma IDL (Interface Description Language) para dispositivos que permite verificação de consistência e geração de código de baixo nível. Dingo [11], que simplifica a interação entre drivers e o SO através da redução da concorrência e formalização de protocolos.

4.4 Sistemas Operacionais Multiserver

Outra proposta é a utilização de sistemas multiserver como o MINIX3, para encapsular drivers não confiáveis como processos que executam em modo-usuário com espaço de endereçamento privado. A arquitetura do sistema MINIX3 é mostrada na Figura 7 [12]. Neste caso, todos os servidores e os tratadores de dispositivos (drivers) executam como processos, modo usuário, independentes – cada um encapsulado em um espaço de endereçamento privado, protegido pelo hardware da MMU (*Memory Management Unit*) — no topo de um pequeno micro-kernel de cerca de 4000 linhas de código executável. A base do micro-kernel é responsável pela programação da CPU e MMU, tratamento de interrupção e IPC (Inter-Process Communication). As tarefas do kernel, *clock task* e *system task* fornecem uma interface para serviços do kernel, tais como E/S e alarmes, para partes do sistema operacional que estão no modo usuário. Os servidores mais comuns oferecem serviços de sistema de arquivos e gerência de processos. Um servidor especial, chamado de servidor de reincarnação (*Reincarnation Server*), gerencia todos os servidores e drivers e constantemente monitora o bom funcionamento do sistema.

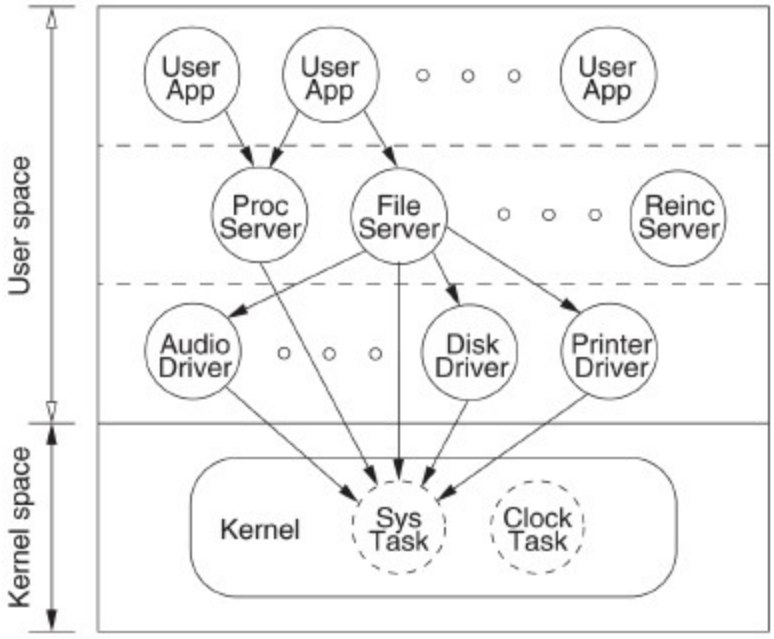

Figura 7. Arquitetura do MINIX3 – Sistema Operacional Multiservidor [12].

O MINIX3 [12] segue estritamente o princípio mínima autoridade (*least authority*) no sentido de não permitir ou garantir excessivo poder à componentes potencialmente com erros. A regra é que cada driver é executado em um processo tradicional UNIX separado com um espaço de enderecamento protegido por MMU e privado. Desta forma, os drivers executam sem privilégio algum e inofensivos. Além disso, devido a esta proteção ser de granularidade grossa são oferecidos vários mecanismos de granularidade fina para garantir acesso seletivo aos recursos necessários para o driver realizar seu trabalho. Diferentes políticas por driver podem ser definidas pelo administrador. O kernel e servidores confiáveis do sistema operacional agem como monitores de referência e mediam todos os acessos aos recursos privilegiados tais como CPU, dispositivo de I/O, memória e serviços do sistema. Os principais pontos utilizados para tornar o MINIX3 tolerante a falhas são: Isolamento das Faltas, Detecção de Defeitos e Recuperação em tempo de execução.
Isolamento de faltas: é necessário para prevenir que problemas sejam propagados e limitar o dano que os erros (bugs) podem causar. Quando um bug é capturado torna-se fácil localizar precisamente o defeito e a recuperação é possível. Mesmo que o sistema operacional fique em espaço de usuário, o *isolamento de faltas* não pode ser conseguido apenas pela separação de espaço de endereçamento. Isto porque os servidores e os drivers precisam mecanismos, que podem ser potencialmente perigosos, para comunicar e compartilhar dados com o objetivo de fazer o sistema funcionar. O MINIX3 não garante poderes aos processos e sim reduz os privilégios. Considerando os drivers, por exemplo, cada um deles é carregado com um arquivo de proteção que precisamente lista seus recursos, incluíndo dispositivo de memória, portas de E/S e linhas de interrupção, e capacidades de comunicação (IPC). O servidor de reincarnação assegura que a política de restrição esta ativa antes de o driver iniciar.
Detecção de Defeitos: o servidor de reincarnação (RS) é o componente central que guarda todos os servidores e drivers no sistema. Durante sua inicialização o RS adota todos os processos (do boot) como seus filhos, servidores e drivers que iniciam depois do boot também se tornam filhos do RS. Assim, conforme o modelo POSIX, o RS será notificado pelo gerente de processos quando um processo do sistema termina. Com base no tipo de término é possível identificar *classes de defeitos*. Outra forma que o RS tem de monitorar o sistema buscando anomalias é periodicamente enviar uma mensagem para o driver e esperar uma resposta. Se não receber uma resposta ele considera comoum defeito e inicia o procedimento de recuperação.

Recuperação em tempo de execução: quando um servidor ou driver é iniciado, ele pode ser

associado com um script shell que dirige o *procedimento de recuperação*. Quando um defeito é detectado, o RS procura o script associado ao processo que esta com malfuncionamento e executa o mesmo. Parametros como: componente que falhou, classe do defeito e contador de falta, são passados para que o script possa decidir o que fazer. A política mais simples pode registrar o erro e derrubar o componente em mal funcionamento, mas em muitos casos é possível trocar o mesmo por uma cópia nova. Uma vez que o componente é reiniciado ele precisa ser reintegrado ao sistema. Primeiro, o RS atualiza a entrada correspondente ao nome do servidor no banco de dados que usa um mecanismo do tipo publish/subscribe para informar componentes dependentes sobre a nova configuração do sistema. Por exemplo, o servidor de arquivos será notificado quando um driver de disco for reiniciado e o seu novo ponto de comunicação (IPC) é publicado no banco de dados. A partir dai, o servidor de arquivos pode reinicializar suas próprias tabelas e pode solicitar ao driver para reinicializar-se. Se o componente reiniciado perdeu estado durante sua queda, ele pode, em princípio, recuperar o backup feito pelo componente quebrado do banco de dados.

Conclusão

As abordagens apresentadas buscam principalmente a redução da acorrência dos erros (bugs) que podem tornar os sistema menos confiáveis e passíveis de quebra. Algumas tem uma bagagem significativa no que diz respeito a suportar sistemas legados, que de certa forma já tem um princípio de funcionamento propício a ocorrência de situações de quebra, como por exemplo: o modelo Nooks. Neste caso, é possível argumentar que: devido a quantidade de código necessária para a implementação da confiabilidade ser grande, o efeito pode ser contrário e gerar ainda mais erros, passíveis de quebra.

Algumas colocações sobre esta estratégia devem ser consideradas:
Embora a questão de tornar sistemas operacionais legados mais confiáveis seja importante, com relação a pesquisa é importante atentar para a seguinte pergunta: Como os futuros sistemas operacionais deverão ser projetados para prevenir o problema? O Nooks tem 22.000 linhas de código (mais código significa mais bugs).

Bibliografia

[1] J. Herder, H. Bos, B. Gras, P. Homburg, A. Tanenbaum. *Fault Isolation for Device Drivers.* In: Proc. 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09), pp. 33--42, Lisbon, Portugal, July 2009. (Conference track: Dependable Computing and Communication Symposium (DCCS).)

[2] J. Herder, H. Bos, A. Tanenbaum. *A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers.* Technical Report IR-CS-018, Vrije Universiteit, Amesterdam, The Netherlands, January 2006.

[3] A. Tanenbaum, et alli. *Minix 3: status report and current research.* USENIX ;login:, vol. 35, no.

3, pp. 7--13, Jun. 2010.

[4] M. Swift, B. Bershad, and H. Levy. *Improving the Reliability of Commodity Operating Systems*. 23(1):77–110, 2005.

[5] J. LeVasseur and V. Uhlig. *A Sledgehammer Approach to Reuse of Legacy Device Drivers*. In Proc. 11th ACM SIGOPS European Workshop, pages 131–136, Sept. 2004.

[6] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. *Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines*. In Proc. 6th Symp. on Oper. Syst. Design and Impl., pages 17–30, Dec. 2004.

[7] H. Bos and B. Samwel. *Safe Kernel Programming in the OKE*. 2002.

[8] G. Hunt, C. Hawblitzel, O. Hodson, J. Larus, B. Steensgaard, and T. Wobber. *Sealing OS Processes to Improve Dependability and Safety*. In Proc. 2nd EuroSys, 2007.

[9] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. *Towards a Practical, Verified Kernel*. In Proc. 11th HotOS, 2007.

[10] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. *Devil: An IDL for Hardware Programming*. In Proc. 4th OSDI, 2000.

[11] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. *Dingo: Taming Device Drivers*. In Proc. 4th EuroSys Conf., 2009.

[12] J. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum. *Roadmap to a failure-resilient operating system*. USENIX ;login:, vol. 32, no. 1, pp. 14--20, Fev. 2007.