Design de Kernels: Microkernel, Exokernel e novos Sistemas Operacionais

Raoni Fassina Firmino
Glauber Módolo Cabral
Aleksey Victor Trevelin Covacevice

Universidade Estadual de Campinas - UNICAMP
Instituto de Computação - IC

MO806 - Tópicos em Sistemas Operacionais
Seminário

Novembro de 2007

Roteiro

Introdução

Conceitos de Kernel

- **Kernel = Núcleo**
  Kernel de um programa como parte central, fundamental de um programa/algoritmo.
- **Kernel = Modo Kernel(Supervisor)**
  Parte de um programa que executa em modo Kernel. Modo Kernel -> Espaço/Modo de Execução. Suporte do Processador a diferentes espaços, ou niveis, de execução(modo Kernel e modo usuario).

Kernel de um Sistema Oeracional

- Núcleo do Sistema Operacional -> Prove o Gerenciamento do Sistema e Funcionalidades Basicas.

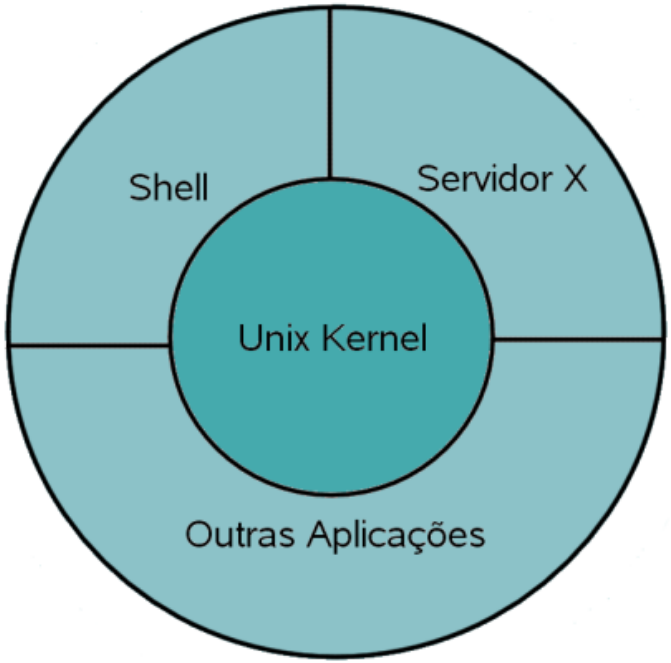


Figura: Diagrama do Sistema Operacional Unix

Ambientes de Execução

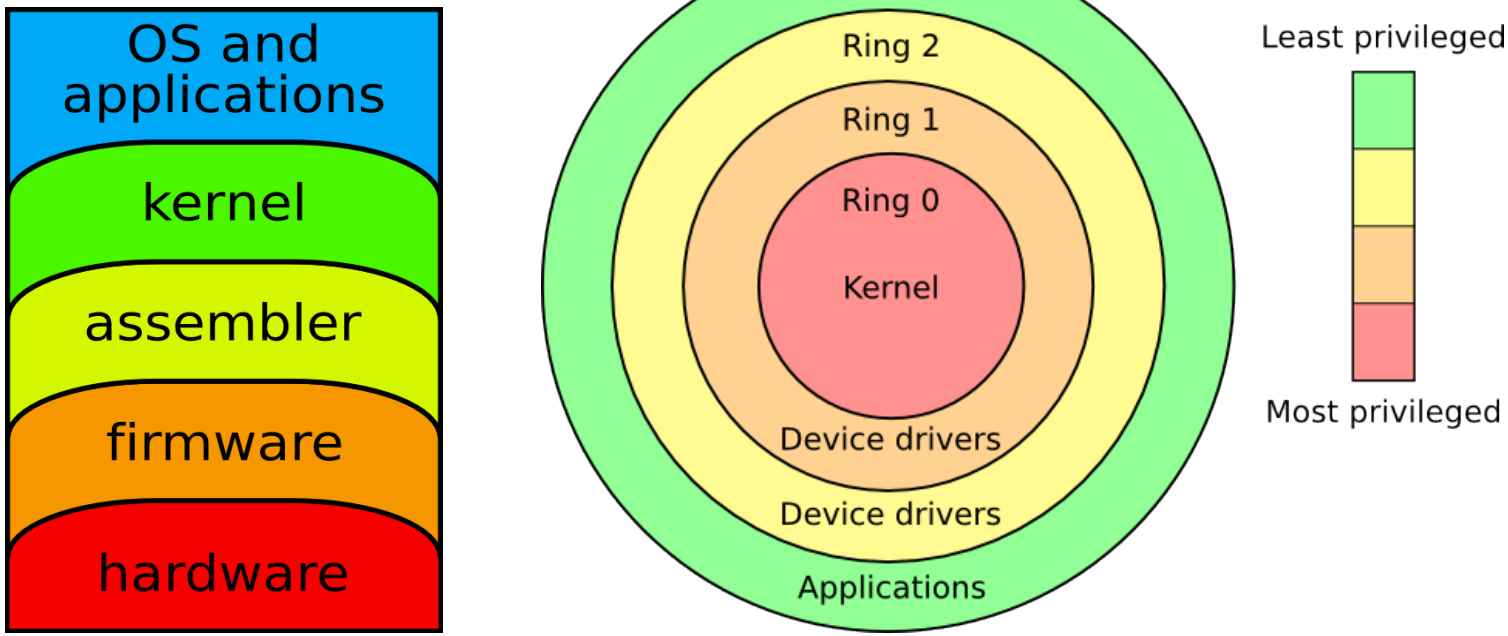


Figura: Diagramas de camadas de Abstração e Anéis de Privilégio de Execução

Kernel - Abordades de design

- MicroKernel
  - NanoKernel
- Kernel Monolitico
- Kernel Hibrido*
- ExoKernel

*Kernel Hibrido é tido como um termo controverso dentre a comunidade ciêntifica da área.

Kernel - Abordades de design

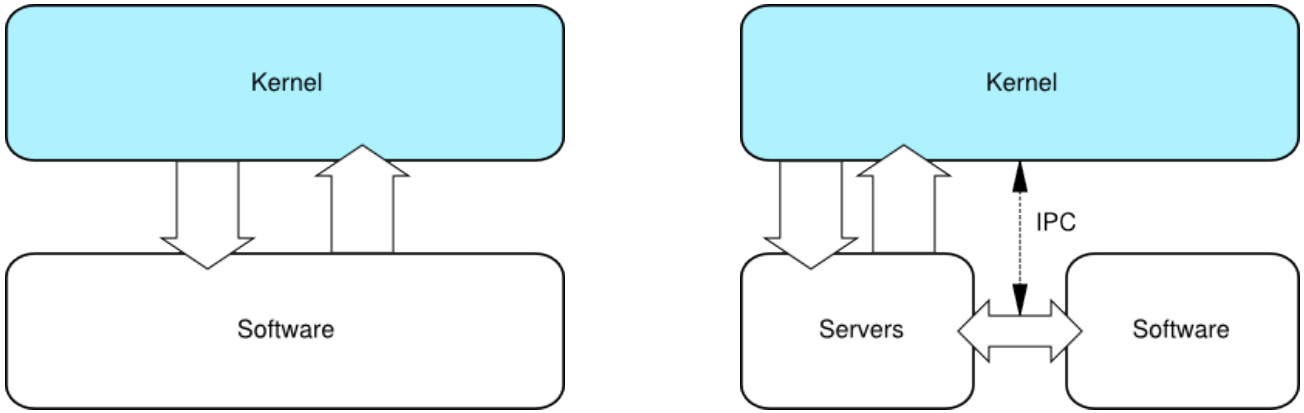


Figura: Diagramas Kernel Monolitico e MicroKernel

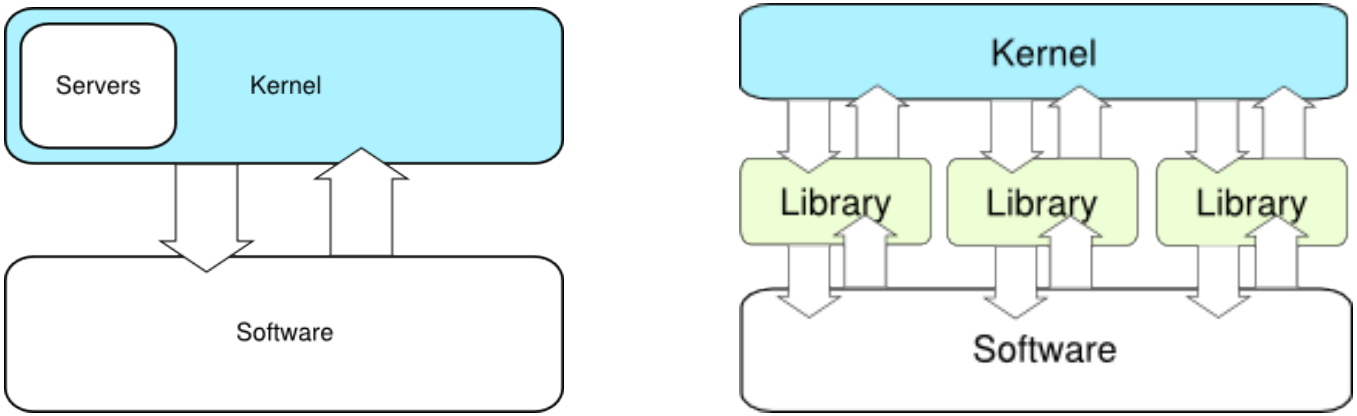


Figura: Diagramas Kernel Hibrido e ExoKernel

Exemplo de Kernel Hibrido

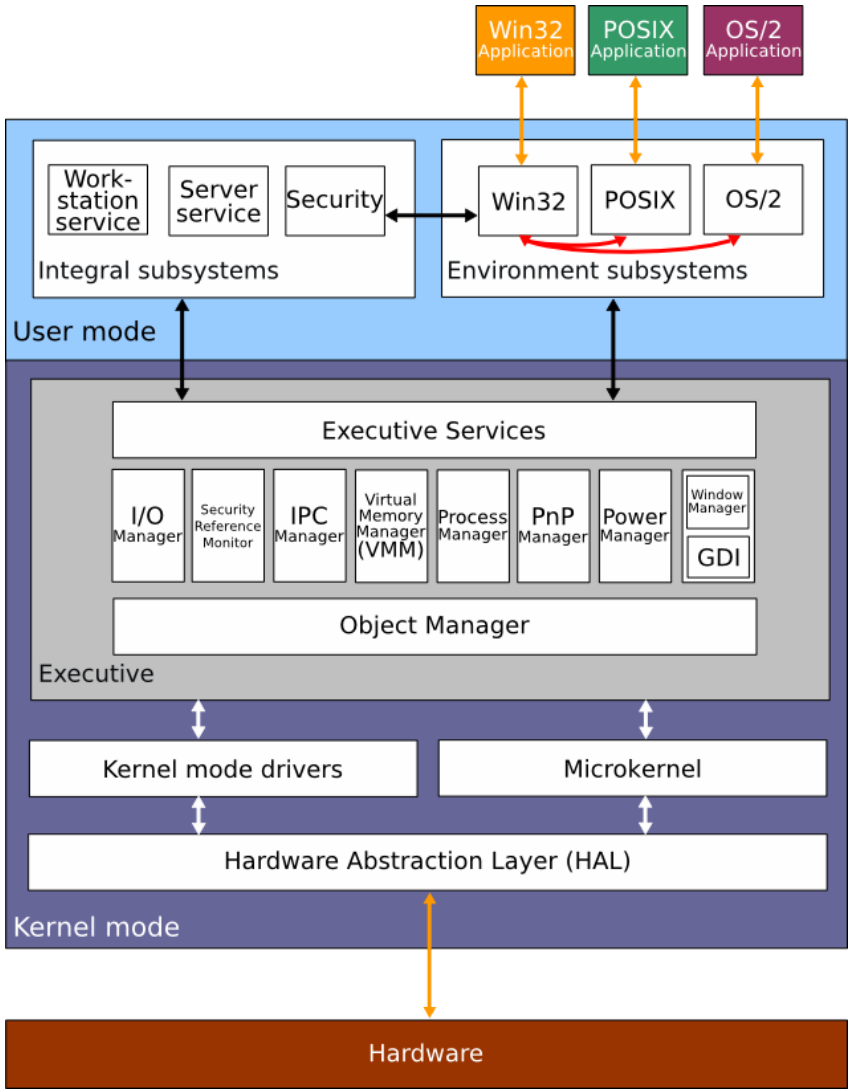


Figura: Diagrama do Kernel WindowsNT

Microkernel

Conceitos e objetivos

- Segurança
  Menor código possivel e menos tempo de execução do sistema possivel executando em modo Kernel.
- Separação de Serviços em Servidores.
- Basear comunicação do sistema em IPC(Comunicação Entre Processos)

interação de um Sistema com SO

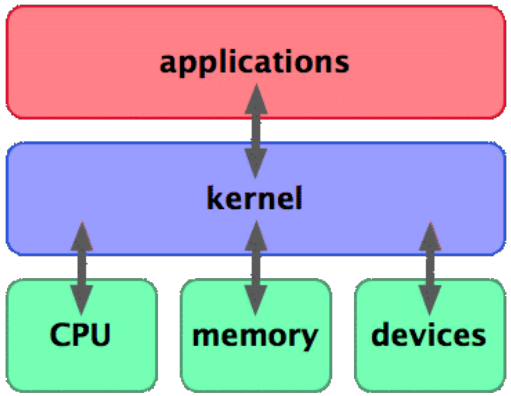


Figura: Diagrama da Visão da Aplicação da Comunicação do Sistema

Kernel Monolitico: System Calls e Sinais de Interrupções.
MicroKernel: IPC, Kernel Calls e Sinais de Interrupções.

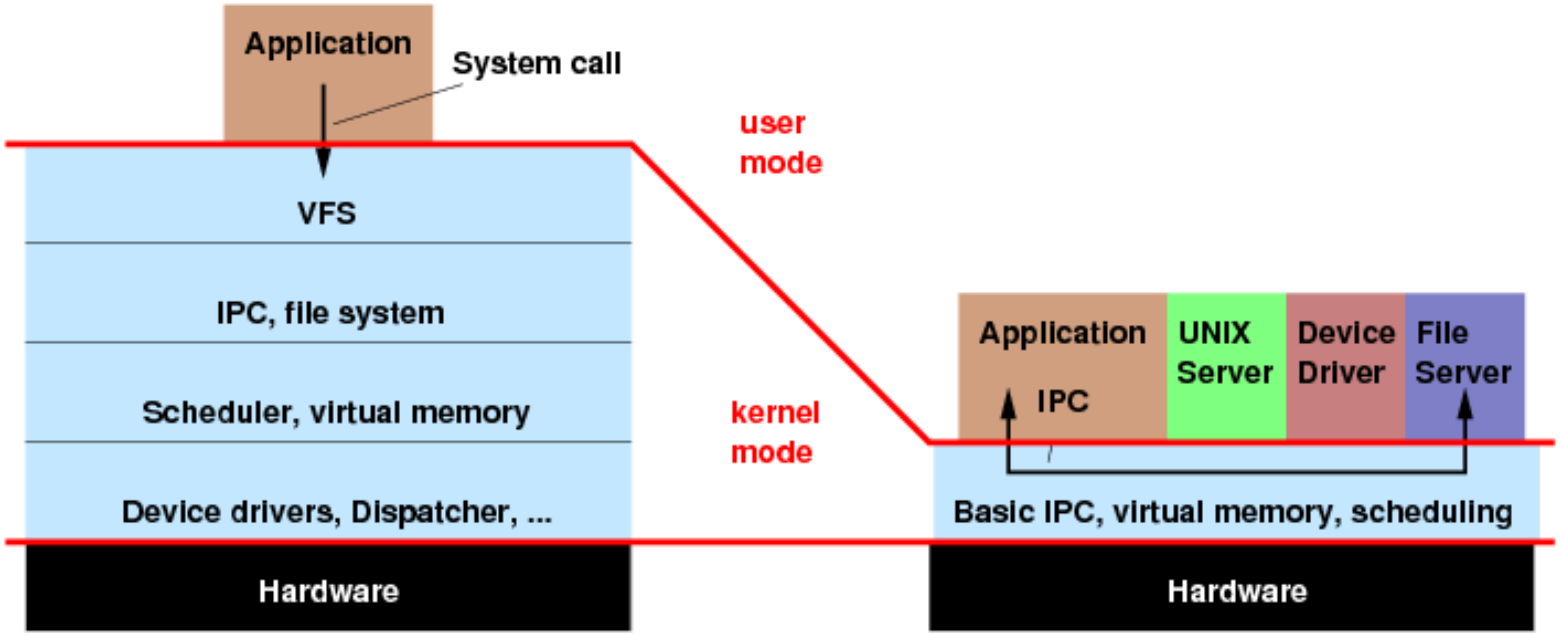


Figura: Diagra de comparação da estrutura de execução de Kernel monolitico e MicroKernel

Partes do Kernel em Modo supervisor:

- IPC;
- trocas de contexto;
- Parte do gerenciamento de memória;
- Gerenciador de interrupções;
- Outras rotinas criticas.
- Parte do escalonador de processos.

Partes do Kernel em Modo usuário:

- Device Drivers;
- Sistemas de Arquivos;
- Parte do gerenciamento de memória;
- Parte do escalonador de processos.

Caracteristicas

- Serviços Providos Através de Servidores em modo usuário;
- pedidos de serviços através de IPC;
- Segurança e Confiabilidade;
- desempenho:
  - Overhead de comunicação (IPC);
  - Overhead de troca de ambiente de execução.
- Porque usar ou não usar MicroKernel?

Exokernel

Exokernel: Introdução

- Exokernel é o nome de um kernel desenvolvido pelo MIT a partir de 1995;
- O termo exokernel passou a ser associado a um novo modelo de kernel, baseado em premissas diferentes dos modelos mais conhecidos;
- Ainda existente apenas como linha de pesquisa, estudos e discussões datam de, pelo menos, 1994.

Exokernel: Visão Geral (I)

- Sistemas Operacionais são sistemas que, em linhas gerais, gerenciam recursos computacionais:
  - Processamento;
  - Memória e armazenamento;
  - Dispositivos de entrada e saída (I/O);
  - Tais recursos são acessados por aplicações, através de interfaces fornecidas pelo SO;

Exokernel: Visão Geral (II)

- As interfaces de um SO comumente correspondem a um conjunto de protocolos de acesso:
  - APIs;
  - System calls;
  - Sinais, exceções e interrupções;
  - Protocolos de rede.
- Tais protocolos são criados sobre abstrações, ou modelos conceituais, pelos quais aplicações acessam recursos de forma transparente;

Exokernel: Visão Geral (III)

- Estes modelos abstraem informações desnecessárias do recurso subjacente e criam um padrão homogêneo de acesso:
  - Processamento (filas, prioridades);
  - Memórias (heaps, endereçamento virtual);
  - Armazenamento (sistemas de arquivo);
  - I/O (descritores, dispositivos virtuais, sockets).
- Eles têm como principal premissa a elegância, a simplicidade e a portabilidade.

Exokernel: O que são? (I)

- Um exokernel é um modelo alternativo de kernel baseado no princípio do ponta-a-ponta:
  - "Abstrações sobre um recurso devem ser feitas apenas pelas aplicações que o usam".
- Um exokernel não força abstrações sobre recursos: ele permite que as aplicações os utilizam da maneira mais próxima possível do *hardware* real;

Exokernel: O que são? (II)

- No modelo do exokernel, o kernel é apenas responsável pela alocação de um determinado recurso para a aplicação;
- A aplicação é responsável pelo uso eficiente do recurso, optando por criar mecanismos adicionais em cima deste;
- O exokernel permite, assim, um alto ou baixo nível de abstração sobre um recurso;

Exokernel: O que são? (III)

- A requisição por recursos se dá de uma maneira bastante concreta:
  - Fatias de tempo ao invés de filas e prioridades;
  - Blocos de disco ao invés de arquivos;
  - Páginas de memória ao invés de heaps;
  - Simples filtro de pacotes ao invés de sockets.
- Pode ser visto como um caminho intermediário entre a programação crua do hardware e um kernel "convencional";

Exokernel: O que são? (IV)

- O exokernel é, em si, um modelo estrutural próprio de kernel que contrasta com os modelos convencionais (monolíticos, microkernels, etc.);
- De forma direta, ele oferece às aplicações o hardware e todas as informações do sistema de maneira descoberta;
- De forma indireta, ele delega às mesmas aplicações a orientação final do sistema: desempenho, portabilidade, segurança, etc.

Exokernel: Princípios (I)

- Sistemas Operacionais modernos são desenvolvidos segundo alguns princípios de design:
  - Simplicidade;
  - Robustez;
  - Desempenho e eficiência;
  - Portabilidade;
  - Segurança;
  - Modularização.

Exokernel: Princípios (II)

- Tais princípios inferem modelos de design, que buscam definir e modelar características desejadas do sistema:
  - Kernels monolíticos;
  - Kernels real-time;
  - Microkernels;
  - Kernels híbridos;
  - Sistemas distribuídos;
  - SOs de uso geral.

Exokernel: Princípios (III)

- A proposta do modelo "exokernel" é seguir ao extremo o princípio do ponta-a-ponta;
- Aplicações selecionam bibliotecas segundo suas necessidades, auxiliando na simplicidade;
- Aplicações descartam ou omitem mecanismos desnecessários, aumentando seu desempenho;

Exokernel: Princípios (IV)

- Complexidades que seriam pré-existentes em sistemas convencionais não são obrigatórias em sistemas baseados em exokernels;
- O exokernel sustenta-se, portanto, sobre a simplicidade, eficiência e autonomia:
  - "Smart terminals for dumb networks."

Exokernel: Aplicações (I)

- Uma aplicação de segurança pode seguramente apagar rastros de dados no bloco de disco que lhe foi concedido;
- Uma aplicação de rede pode eficientemente mandar pacotes pré-formatados de dados sem a necessidade de uma pilha de protocolos;
- Uma aplicação multimídia pode escrever diretamente e de maneira segura no framebuffer, evitando overhead de outros mecanismos;

Exokernel: Aplicações (II)

- Regiões de performance crítica de execução podem requisitar fatias extras de tempo para serem processadas;
- Compartilhamento de dados pode ser feita simplesmente manuseando as permissões da página de memória desejada;
- Aplicações que compartilham mecanismos, funções ou abstrações podem utilizar das mesmas bibliotecas, como em sistemas convencionais;

Exokernel: Aplicações (III)

- Novas abstrações podem ser implementadas e experimentadas de maneira segura:
  - Novos sistemas de arquivo;
  - Novos escalonadores;
  - Novos protocolos de rede.
- Código "puro hardware" pode concorrer com código "puro software" sob o mesmo sistema operacional!

MIT Exokernel (I)

- Linha de pesquisa, referências constam desde 1995;
- Dois SOs para dois kernels:
  - Aegis kernel;
  - XOK.
- Princípio chave: “O SO deve oferecer o mínimo de serviços admissível em termos de concorrência segura”;

MIT Exokernel (II)

- Gerência de recursos seguindo a filosofia do exokernel;
- Processador:
  - Alocação de fatias de tempo;
  - Sinalização de eventos (interrupções, exceções, início e fim de um timeslice);
  - Aplicações gerenciam o próprio tempo;
  - O kernel penaliza aplicações "egoístas".

MIT Exokernel: Memória

- O kernel gerencia apenas a TLB e a alocação de páginas;
- Programas requisitam e liberam páginas;
- Páginas possuem capabilidades;
- Compartilhamento de dados é feito através das capabilidades.

MIT Exokernel: Disco

- Identificação por endereço físico de blocos no disco;
- Programas alocam ou liberam blocos no disco;
- Funções de callback são utilizadas como auxiliares na alocação e liberação de blocos;
- Possibilidade de alocar regiões contíguas ou repassar blocos para outros programas.

MIT Exokernel: Interface de rede

- Kernel providencia um filtro simples de pacotes;
- Filtro é programado utilizando uma linguagem auxiliar de simples avaliação pelo kernel;
- Programas enviam e recebem pacotes manipulando a "programação" do filtro:
  - Similar a iptables e ipchains!

Outros Exokernels

- Poucas abordagens: ainda como um conceito, não existem sistemas utilizados comercialmente;
- Pesquisa é desenvolvida em diversas localidades:
  - MIT;
  - University of Cambridge;
  - University of Glasgow;
  - Swedish Institute of Computer Science;
  - Citrix Systems.
- Alguns sistemas: Nemesis, ExOS (MIT).

Singularity Project

Motivação

Como seria uma plataforma de software projetada desde o início com o objetivo de prover confiabilidade e *dependability*?

Dependability

Termo utilizado para indicar um sistema que possui:

- *Availability*: estar pronto para fornecer um serviço;
- *Reliability*: continuidade no fornecimento do serviço;
- *Maintainability*: permitir modificações e reparos no sistema.

Estratégias usadas

- Linguagens de programação “seguras”: eliminar erros evitáveis, como *buffer overrun*;
- Ferramentas de verificação (*sound verification*): remover todos os erros possíveis no começo do ciclo de desenvolvimento;
- Arquitetura baseada em isolamento, por software, dos processos: impôr barreiras à propagação de erros de tempo de execução.

Produtos do projeto

- Linguagem de programação: Sing#
  - Extensão de C#;
  - Sintaxe possui suporte formalmente verificável a primitivas de comunicação
- Ferramentas de verificação
  - *Sound verification*: detecta os erros antes da execução.
- Sistema operacional: Singularity OS

Singularity OS: Arquitetura

- Processos isolados em software (SIP): provê um ambiente para execução de programas protegido de interferência externa;
- Canais com comunicação baseada em contrato: permite comunicação rápida, baseada em mensagens e com verificação das mensgens;
- Execução de programas baseada em manifesto: define o código de um programa e as suas propriedades comportamentais verificáveis.

Software-Isolated Process (SIP)

- SIP diz quais são os recursos em processamento e o contexto para a execução de programas;
- SIP usa a verificação de tipos e memória da linguagem de programação para diminuir o custo de isolar código seguro.
- Cada processo executa no contexto de um SIP, com uma ou mais threads;
- Possui um conjunto de páginas de memória associado, com códigos e dados;
- Executa com uma identidade de segurança e possui atributos de segurança do SO;
- Esconde informações entre os processos e isola falhas em processos.

Comunicação entre SIPs

- SIPs não compartilham dados;
- Comunicação feita por mensanges através de canais sob orientação de contratos estaticamente verificáveis;
- Contratos especificam as mensagens e o protocolo de comunicação para todos os canais de um mesmo tipo;
- SIPs acessam funções primitivas através de uma ABI (Application Binary Interface) com o kernel;
- ABI possui informação de versão: um programa declara de qual versão ele depende;
- ABI provê acesso local e seguro à memória, execução e comunicação sem contruções semanticamente ambíguas, como *ioctl*.

SIP - Espaços fechados de objetos

- O conteúdo de um SIP não pode ser alterado diretamente por outro;
- A trasnferência de dados ocorre por transferência de titularidade (*ownership*) dos dados nas mensagens;
- Dados a serem trocados são colocados em uma pilha de acesso comum (*exchange heap*);
- Permite execução autônoma: layout de dados, sistema de execução e *garbage collector* independentes em casa SIP;
- Torna cada SIP indepentende dos demais, mesmo com dependência de dados;
- Cada SIP pode ser desalocado a qualquer momento sem preocupação de interferência com os demais.

SIP - Espaços fechados de código I

- SIPs não podem carregar ou gerar código dinamicamente em si mesmos;
- Extensões rodam como um novo SIP independente;
- Aumentam a capacidade de verificação estática dos programas por analisadores de código automático;
- Permitem melhores mecanismos de segurança, como identificação de processos por seu código;
- Evitam que o controle de acesso precise ser duplicado nos ambientes de execução (entre SO e processo);

SIP - Espaços fechados de código II

- SIP são isolados através dos mecanismos de verificação de tipo e memória da linguagem de programação e não através do gerenciamento de memória pelo hardware;
- Verificação estática na compilação combinada com verificação em tempo de execução garantem que um SIP não acessa regiões de memória fora do SIP;
- Com isolamento por software, vários SIPs podem rodar no mesmo espaço de endereçamento físico ou virtual (no protótipo, kernel e SIPs compartilham enderaçamento em modo kernel).

SIP - Criação, gerenciamento e desempenho

- SIP são baratos de criar e destruir comparados com processos protegidos por hardware;
- SIPs podem ser criados sem a criação de tabelas de páginas ou liberação de memória na TLB;
- Troca de contexto entre SIPs são baratas porque não exigem liberação de memória na TLB e nem da cache de endereços virtuais;
- Recursos de um SIP podem ser reutilizados após o término de sua execução e suas páginas de memória podem ser recicladas sem envolver *garbage collector*;
- Precisa de apenas um modelo de recuperação de erros, um modelo de programação, um mecanismo de comunicação e uma arquitetura de segurança, em oposição aos vários níves de proteção e várias políticas de segurança dos sistemas atuais.

Canais baseados em contratos

- Canal: caminho bi-direcional de mensagens, FIFO e sem perdas, com exatamente duas extremidades;
- Cada extremidade tem uma fila de recepção e enviar uma mensagem significa colocar a mensagem na fila da outra extremidade;
- Cada extremidade pertence a exatamente uma thread a cada instante;
- Só esta thread pode tirar mensagens da fila ou enviar mensagens para a outra extremidade;
- Comunicação é descrita por um contrato.

Sintaxe do contrato

- Extremidades assimétricas:
  - extremidade importadora (Imp e arquivo.Imp)
  - extremidade exportadora (Exp e arquivo.Exp)
- Declaração de mensagens:
  - número e tipo dos parâmetros de cada mensagem;
  - sinal de direção da mensagem (opcional):
    - !: Exp -> Imp
    - ?: Imp -> Exp
  - Mensagens são declaradas com a direção (in, out) e os sinais de de direção só tornam o código dos estados legível;
- Nomeação de estados do protocolo:
  - Define as seqüências de mensagens que provocam transições de estados.
- Um SIP exporta um serviço e inicia no primeiro estado listado;

Contrato para acessar driver de rede (I)

```
contract NicDevice {
  out message DeviceInfo(...);
  in message RegisterForEvents(NicEvents.Exp:READY c);
  in message SetParameters(...);
  out message InvalidParameters(...);
  out message Success();
  in message StartIO();
  in message ConfigureIO();
  in message PacketForReceive(byte[] in ExHeap p);
  out message BadPacketSize(byte[] in ExHeap p, int m);
  in message GetReceivedPacket();
  out message ReceivedPacket(Packet * in ExHeap p);
  out message NoPacket();
```

Contrato para acessar driver de rede (II)

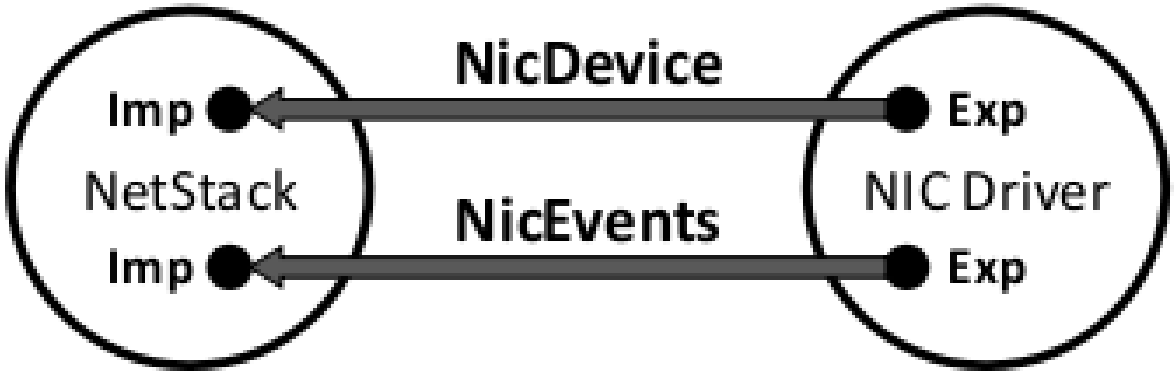
```
state START: one {
  DeviceInfo! -> IO_CONFIGURE_BEGIN;
}
state IO_CONFIGURE_BEGIN: one {
  RegisterForEvents? ->
  SetParameters? -> IO_CONFIGURE_ACK;
}
state IO_CONFIGURE_ACK: one {
  InvalidParameters! -> IO_CONFIGURE_BEGIN;
  Success! -> IO_CONFIGURED;
}
state IO_CONFIGURED: one {
  StartIO? -> IO_RUNNING;
  ConfigureIO? -> IO_CONFIGURE_BEGIN;
}
```

Contrato para acessar driver de rede (III)

```
  state IO_RUNNING: one {
    PacketForReceive?
      -> (Success! or BadPacketSize!)
        -> IO_RUNNING;
    GetReceivedPacket?
      -> (ReceivedPacket! or NoPacket!)
        -> IO_RUNNING;
    ...
  }
}
```

Contrato para eventos do driver de rede

```
contract NicEvents {
  enum NicEventType {
    NoEvent, ReceiveEvent, TransmitEvent, LinkEvent
  }
  out message NicEvent(NicEventType e);
  in message AckEvent();
  state READY: one {
    NicEvent! -> AckEvent? !READY;
  }
}
```



Verificação de contratos

- Compilador pode verificar estaticamente que as trocas entre canais não são aplicadas em estados inconsistentes;
- O verificador de contratos pode verificar estaticamente quais contratos o bytecode de um programa utiliza e que o código está de acordo com a máquina de estados descrita no protocolo do contrato
- Junto com o sistema de tipos, permite troca de grande volume de dados sem cópia (envia apenas a titularidade do dado);
- Permite eliminar o tratamento de erros no envio de mensagens, deixando o mesmo apenas para a recepção.

Manifesto: definições

- Manifesto descreve:
  - recursos de código
  - recursos de sistemas do qual o código depende;
  - funcionalidades esperadas do código;
  - dependência do código em relação a outros programas.
- Códigos não rodam sem possuir um manifesto;
- Para a execução, invoca-se o manifesto e não o binário;

Instalação, execução e verificação de manifestos

- Quando o manifesto é instalado, pode-se:
  - identificar propriedades necessárias para a execução e verificá-las;
  - garantir que as dependências de outros programas possam ser satisfeitas;
  - evitar que esta instalação possa interferir com programas previamente instalados.
- Antes da execução do manifesto, pode-se:
  - descobrir os parâmetros de configuração necessários ao programa;
  - verificar as restrições sobre estes parâmetros.
- Ao ser invocado, o manifesto:
  - guia a disposição do código em um SIP para execução;
  - determina as conexões de canais entre o seu SIP e os demais;
  - garante o acesso do SIP aos recursos de sistema.

Verificação através de manifestos

- Usado para verificação estática automática de propriedades;
- Garante:
  - que programas não interfiram uns nos recursos dos outros;
  - checagem de tipo e memória;
  - ausência de instruções de modo privilegiado;
  - conformidade com os contratos dos canais;
  - uso apenas de canais com contratos declarados;
  - uso da versão correta da ABI.

Conteúdo e extensões de manifestos

- Código entregue ao sistema em MSIL (Microsoft Intermediate Language) compilada;
- Características específicas a Sing # adicionadas a MSIL através de meta-dados de extensão;
- Código compilado para linguagem nativa durante a instalação do manifesto, ao invés de compilação JIT;
- Manifesto aceita extensões para definir propriedades avançadas e permitir a verificação das mesmas (verificação de tipos, subconjuntos de canais declarados);
- Extensões podem ser inline no código MSIL ou por meta-dados em arquivos separados.

Definições e linguagens

- Microkernel: serviços executam em SIPs fora do kernel;
- Mantido no kernel:
  - escalonamento
  - gerenciamento de acesso a recursos de hardware
  - gerenciamento de memória
  - gerenciamento de threads e pilhas das threads
  - criação e destruição de SIPs e canais
- Linguagens do código:
  - quase 90% escrito em Sing#, com verificação de tipos e *garbage collection*;
  - 48% do restante escrito sem verificação de tipos: código do *garbage collector*;
  - debbuger e código de início do sistema escritos em C++ (6% do total do código);
  - pequenos trechos em Assembly (vetor de interrupções, troca de contexto de threads)

Singularity ABI

- Fornece acesso a facilidades primitivas do kernel, como envio de mensagens;
- Por padrão, um SIP só pode manipular seu estado e iniciar e parar seus filhos;
- Demais permissões são passadas pelos canais, e não por funções da ABI;
- Extremidades dos canais são competências (*capabilities*): ou estão presentes desde o início da execução (definidas no manifesto) ou são recebidas pelos canais (pelas mensagens de titularidade);
- Competências só podem ser acessadas por quem possui a sua titularidade;
- Versionada: um código pode ser compilado para uma determinada versão de ABI e o sistema garante (pela análise do manifesto) que ela será usada;

Isolamento I

- ABI mantém estados isolados: um processo não altera o estado de outro e referências a objetos não podem ser enviadas de um processo para outro;
- Chamadas a ABI só afetam o processo atual: *garbage collector* pode atuar sem perigo de outro processo ter referência ao objeto;
- Embora mais cara que a chamada de uma função, a ABI com este isolamento permite troca de contexto rápida entre SIPs e kernel;

Isolamento II

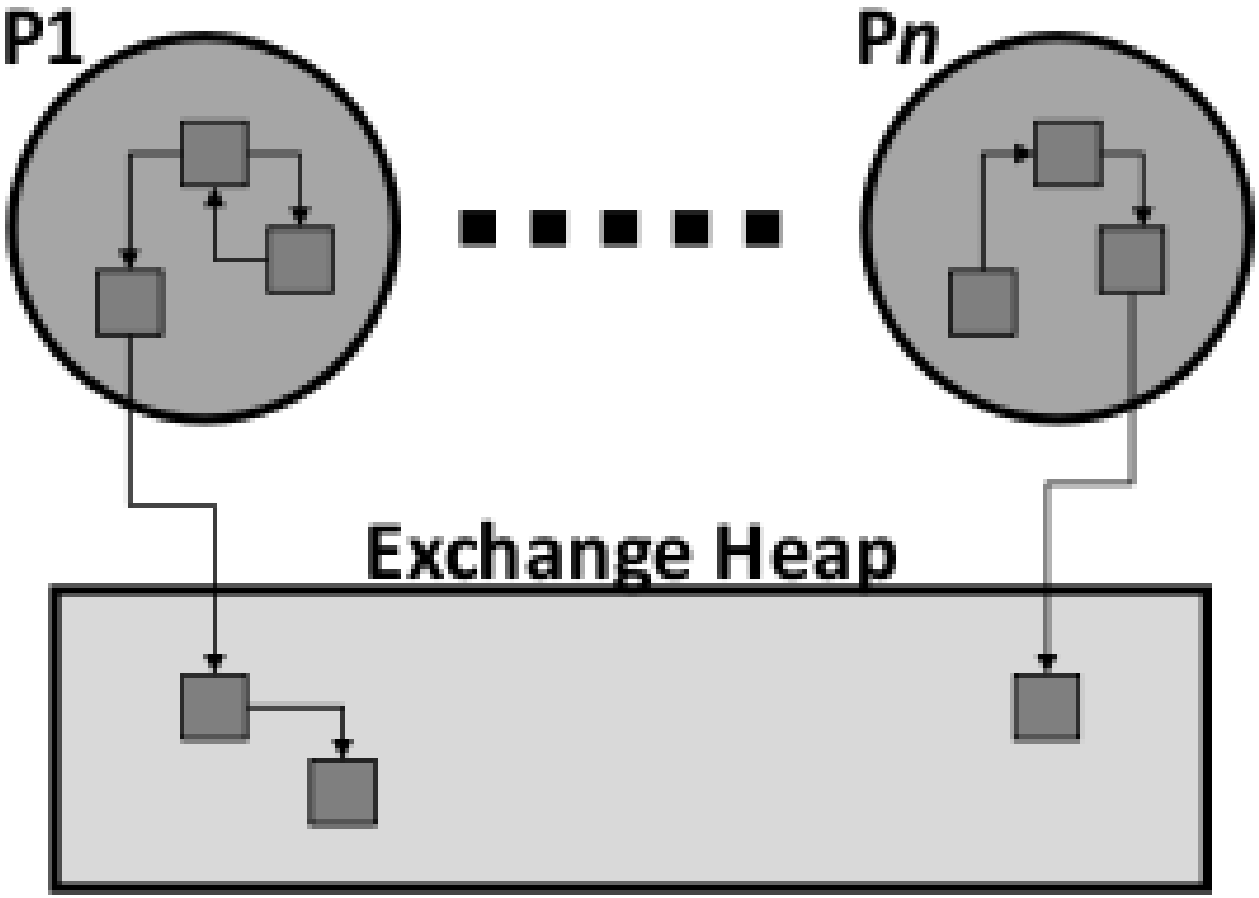
- Isolamento por software permite a inclusão de instruções privilegiadas dentro dos SIPs:
  - pode-se verificar quando e por quem serão executadas;
  - torna desnecessário o uso dos níveis de privilégrio de hardware;
  - titularidade garante a execução apenas por quem tem permissão.
- Acesso a objetos de kernel por mais de um SIP (mutex, etc) são feitos por (*handlers*) fortemente tipificados alocados pelo kernel e passados aos SIPs: como os ponteiros só são conhecidos pelo kernel, não interferem no isolamento dos SIPs.

Gerenciamento de Memória I

- Espaço único de endereçamento protegido com isolamento por software;
- Logicamente particionado em espaço de kernel, espaço para cada SIP e pilha de troca (*exchange heap*);
- Ponteiros em um SIP apontam, apenas, para memória interna ou para memória na pilha que pertença ao SIP; nunca para memória de outro SIP;
- Um SIP obtém memória por chamada a ABI: a memória não precisa ser contígua entre várias chamadas;
- Ponteiros na pilha só apontam para dados dentro da pilha, nunca dentro de processos ou do kernel;

Gerenciamento de Memória II

- Apenas um SIP possui acesso a um bloco de memória da pilha por vez: já garante exclusão mútua;
- SIPs podem ter ponteiros para memória inexistente: não é problema porque a memória apenas não está acessível, ao contrário de apontar para memória de outro processo;
- Um SIP não acessa a memória se tiver enviado sua titularidade para outro SIP;
- Em caso de falhas, os blocos da pilha são restaurados por contagem de referências ao bloco;
- A titularidade dos blocos é gravada para que sejam liberados quando um SIP finalizar sua execução.



Threads

- Sistema é multi-threads;
- Todas as threads são vísiveis ao escalonador do kernel;
- A troca de contexto entre threads tem o mesmo desempenho da troca de threads de usuários nos sistemas comuns porque não tem troca de modo de proteção de hardware;
- A memória das threads é alocada em pilhas ligadas, alocadas de forma não-contígua;
- SIP aloca memória por chamadas à ABI, sempre que não houver espaço para a chamada de uma nova função;
- Escalonador é otimizado para muitas threads que se comunicam freqüentemente (muitas dormem e muitas acordam).

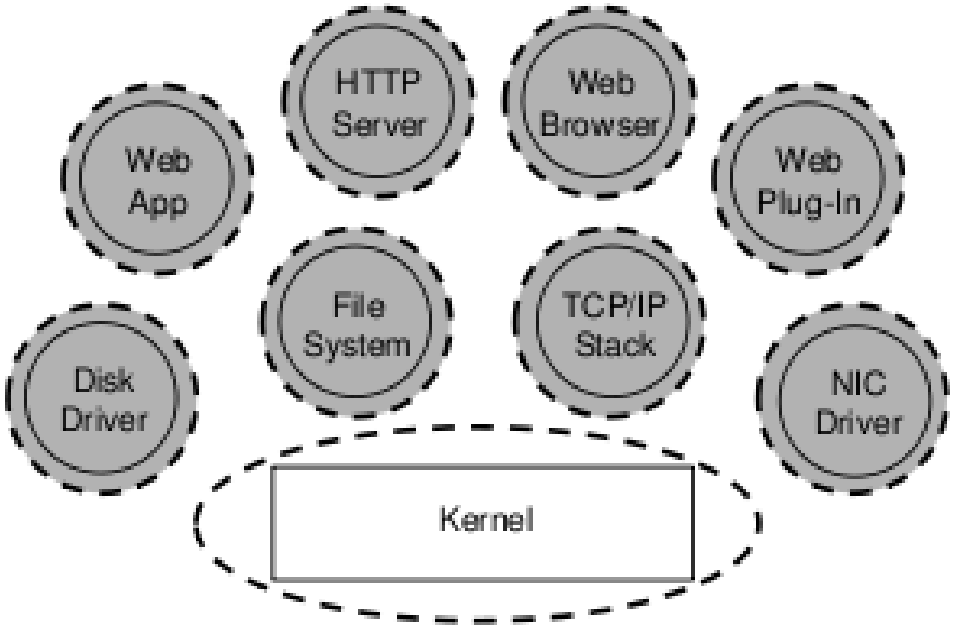
Garbage Collection

- Cada SIP pode escolher o algoritmo de garbage collection que vai executar;
- Possível porque:
  - cada SIP tem execução isolada;
  - ponteiros não ultrapassam os limites dos SIP ou do kernel: GC sabe que ponteiros não referenciam memória externa;
  - mensagens nos canais não são objetos: não precisa haver consenso no modelo de memória entre SIPs, apenas na *exchange heap*;
  - a alocação de memória é centralizada no kernel.

Pesquisas em andamento

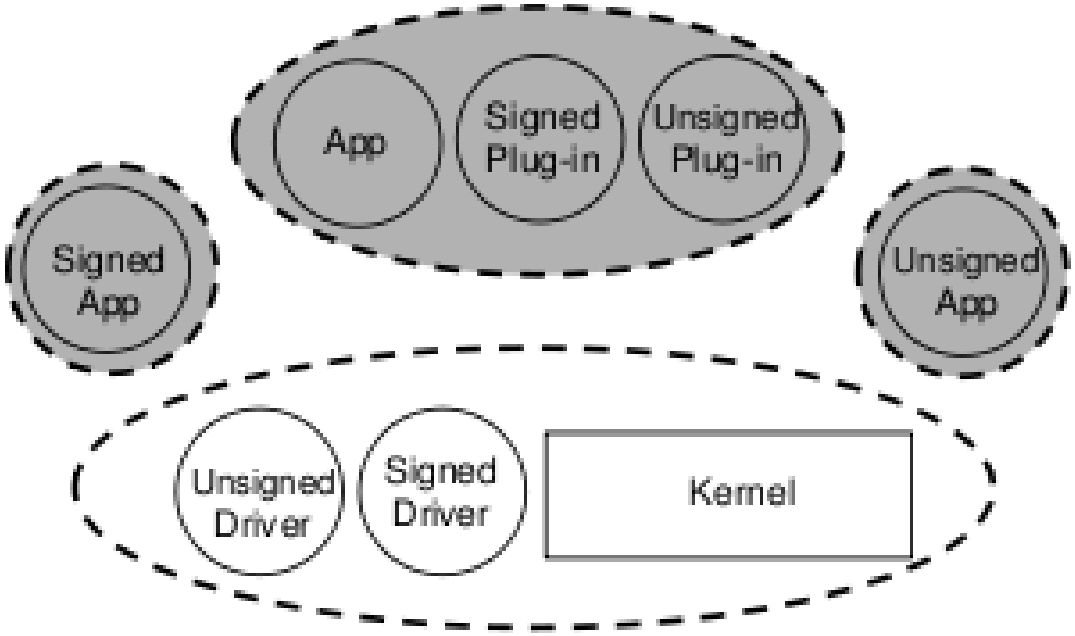
- Reflexão em tempo de compilação: evita o overhead da reflexão em tempo de execução;
- Domínios de proteção de hardware: permite configurar dinamicamente quais processos podem ou não ter privilégio (kernel-mode);
- Processamento heterogêneo: distribuir código para rodar nos processadores dos periféricos (*Programmable IO Processors*);
- Linguagem Assembly tipificada: permite verificar código nativo compilador em outros compiladores ou escritos à mão e rodá-lo com segurança no SO.

Domínios de proteção de hardware (I)



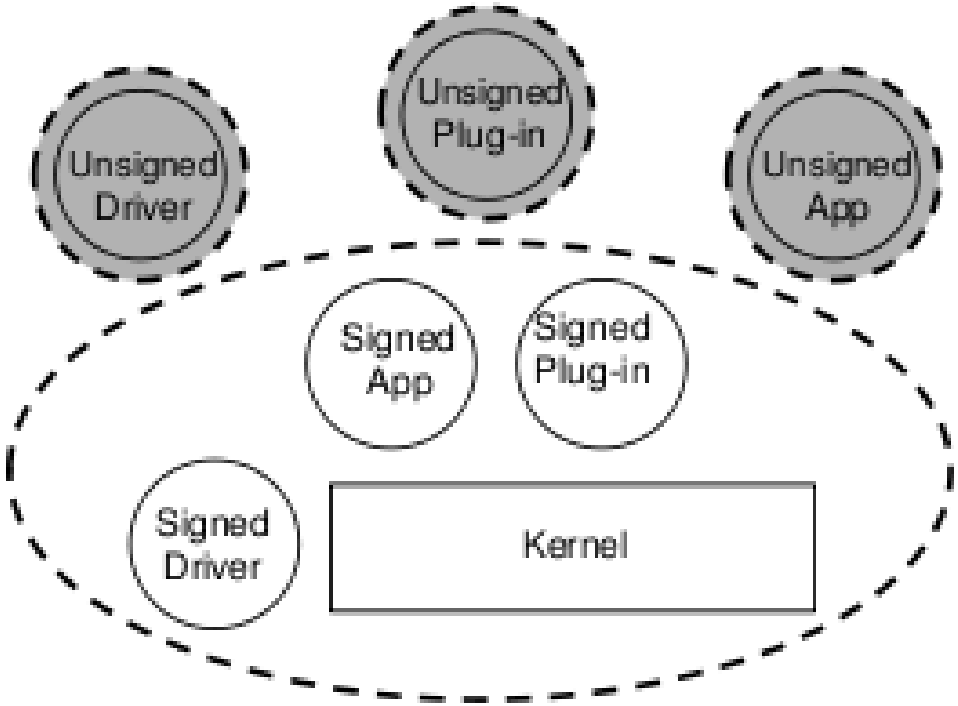
Micro-kernel configuration (like MINIX 3). Dotted lines mark protection domains; dark domains are user-level, light are kernel-level.

Domínios de proteção de hardware (II)



Monolithic kernel and monolithic application configuration.

Domínios de proteção de hardware (III)



Configuration with distinct policies for signed and unsigned code.

JNode

Java New Operating System Design Effort (JNode)

Retirado da Pagina Principal do Projeto JNode:

“The goal is to get an simple to use and install Java operating system for personal use. Any java application should run on it, fast and secure!”

- Portabilidade
  Estender as caracteristicas de execução de prgramas da JVM para todo o sistema.
- JVM executada diretamente sobre o Hardware.
- Desempenho.

História do JNode

- Ewout Prangsma (Fundador do JNode)
- Construir uma JVM em Java;
- Sistema como um ambiente de Execução Completo, não só Maquina Virtual;
  - Leve;
  - Sistema Flexivel.
- JBS (Java Bootable Syste)
  - Muito Código Nativo (C e Assembly).
- JBS2 -> JNode
  - Sem C e Pouco Código Assembly.
- JNode Primeira Versão: Maio de 2003;
- JNode Atualmente versão 0.25.

Design do Jnode

- Maquina Virtual java e Sistema Operacional;
- Escrito em Java e Assembly;
- Todo Módulo do JNode é um Plugin;
- PluginManager:
  Responsavel por permissões, suporte ao ciclo de vida, e carregar,descarregar e Recarregar Plugins;
- Class Library implementada usando recursos providos pelo Sistema Operacional do JNode (ex: Sistema de arquivo, GUI).

Design do Jnode

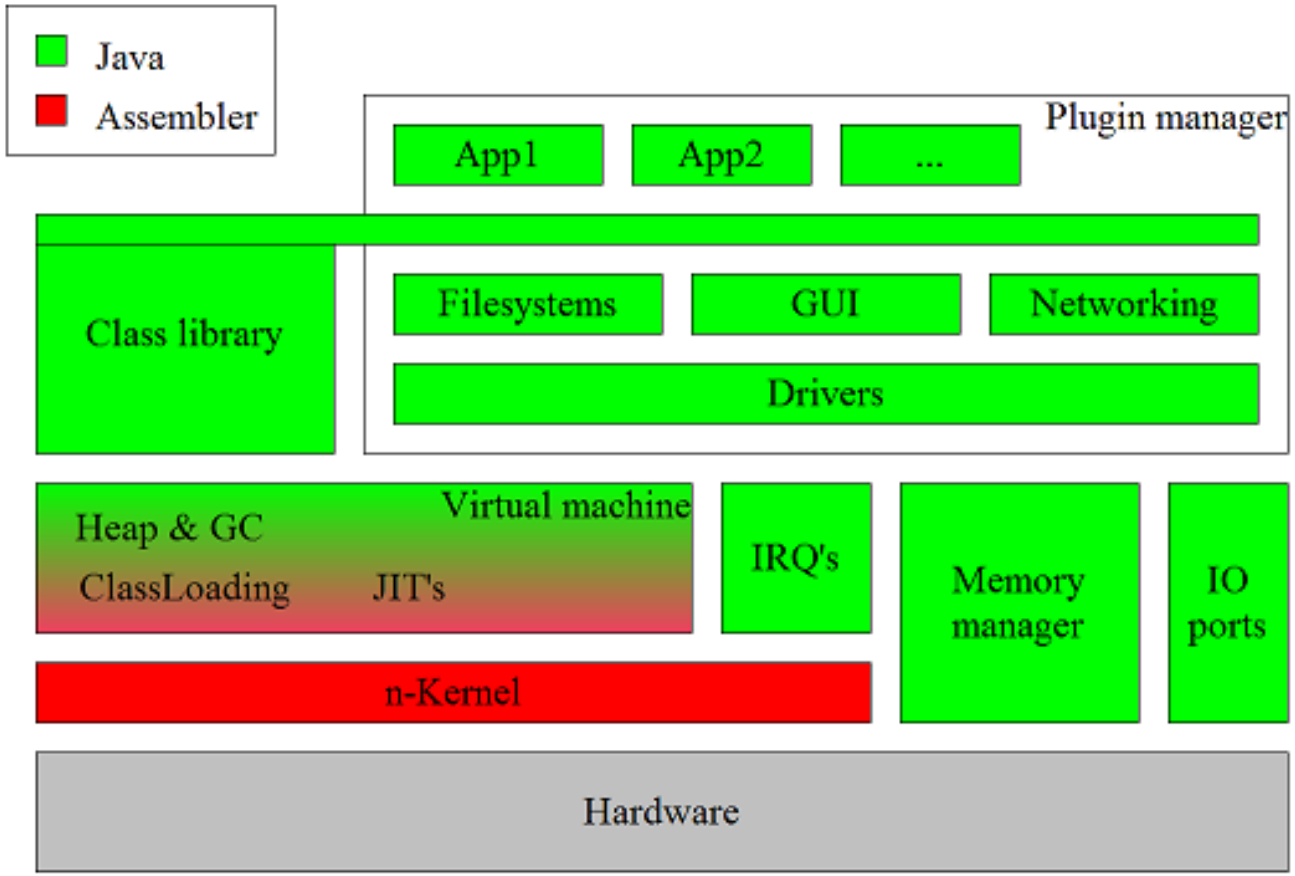


Figura: Diagrama do JNode

Nano-Kernel do JNode

- O Nano-Kernel executa no inicio do boot do JNode.
- Responsavem pela configuração da CPU em modo correto e inicialização das estruturas de gerenciamento da memória fisica(ex: Tabela de paginas, Segmentos.)
- Responsavel pelo gerenciamento de interrupções, fazer o Armazenamento(Cahce) e despacha-las para sua execução correta

Boot e Execução do JNode

Boot:

1. Carregamento da imagem do Kernel para a memória pelo Bootloader.
2. Bootstrapper Code.
3. Inicialização da JVM.
4. Apartir desse momento só se executa Código Java.

Execução de Aplicações:

- Aplicações são ByteCodes
- Executadas pela JVM
- Utiliza recursos providos pela Class Library
- Recursos da aquitetura providos diratamente da JVM e Nano-Kernel do sistem (Sem necessidade da JVM usar Syscalls como em um ambiente da JVM sobre um SO).

Portabilidade

Portar o JNode emvolve os seguintes Componentes:

- Nano-kernel (assembler)
- Native methods (assembler)
- Architecture specific classes
- Native code compilers
- Build process

JNode Shell e GUI

JNode Possui Shell Com recurso de “Command Line Completion”.

Apartir do Shell se tem acesso a interfase grafica (GUI).

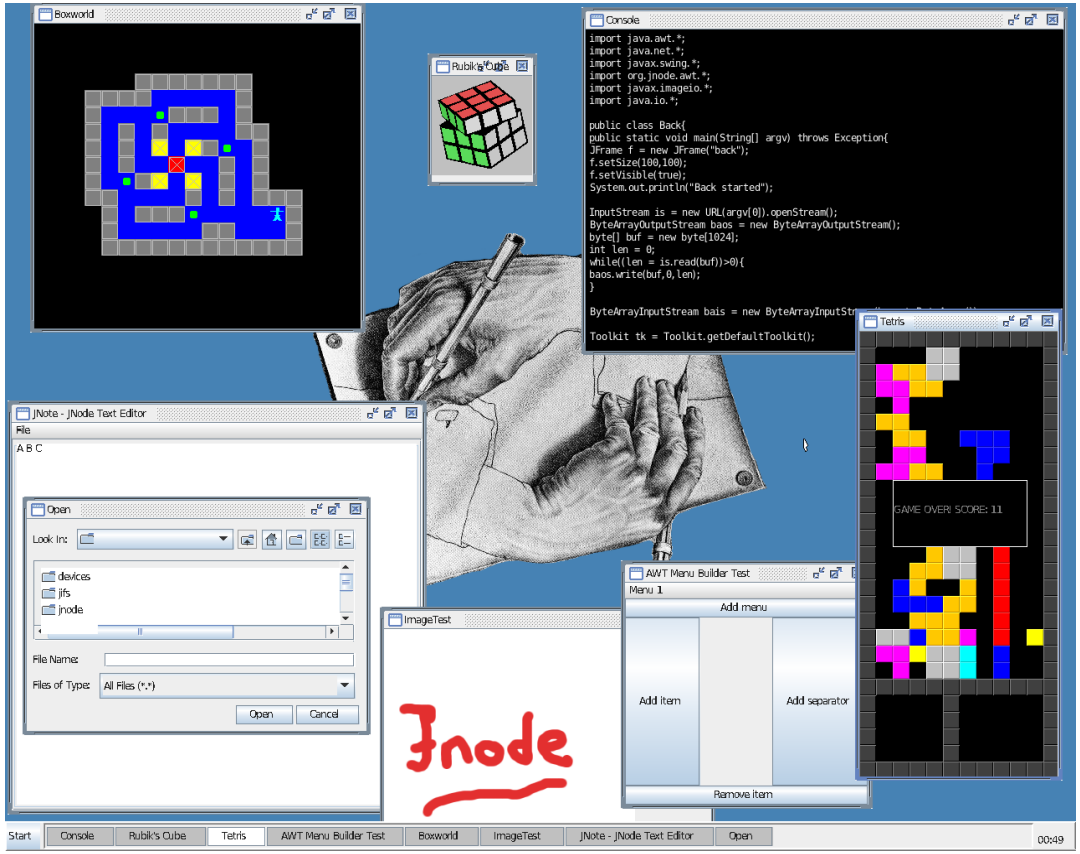


Figura: Screenshot da GUI do JNode

Referências I


D. R. Engler and M. F. Kaashoek.
Exterminate all operating system abstractions.
In *HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, page 78, Washington, DC, USA, 1995. IEEE Computer Society.


D. R. Engler, M. F. Kaashoek, and Jr. J. O'Toole.
Exokernel: an operating system architecture for application-level resource management.
In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM.


D. Engler.
The exokernel operating system architecture, 1999.

Referências II

Galen C. Hunt and James R. Larus.
Singularity: rethinking the software stack.
SIGOPS Oper. Syst. Rev., 41(2):37–49, 2007.

J. H. Saltzer, D. P. Reed, and D. D. Clark.
End-to-end arguments in system design.
ACM Trans. Comput. Syst., 2(4):277–288, 1984.

Agradecimentos

Obrigado!

Perguntas?

Contato:
glauber.cabral@students.ic.unicamp.br
raoni.firmino@students.ic.unicamp.br
aleskey.covacevice@stutents.ic.unicamp.br