Pontifícia Universidade Católica do Paraná
Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Informática Aplicada

Sobre as práticas de laboratório no ensino de Sistemas Operacionais

Documento submetido à Pontifícia Universidade Católica do Paraná
como requisito parcial no concurso de promoção na carreira docente
para o nível de Professor Titular por

Carlos Alberto Maziero

Curitiba, 14 de dezembro de 2001

Resumo

Este trabalho busca analisar as práticas de laboratório na disciplina de Sistemas Operacionais e apresenta a proposta de um novo sistema operacional didático como ferramenta de apoio ao ensino de sistemas operacionais. Esse sistema operacional, denominado SODA– Sistema Operacional Didático Aberto*, tem como objetivo prover a base mínima necessária para implantar uma abordagem construtivista no ensino de sistemas operacionais para cursos de graduação e pós-graduação em informática.*

Abstract

This work presents an analysis of undergraduate laboratory classes on operating systems. It also presents the proposal of a new toy operating system to support operating systems teaching activities. This system, called SODA *(from the portuguese translation for* Open Didactical Operating System*), has the goal of providing the minimal basis to build a constructivist approach to undergraduate operating systems teaching.*

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A disciplina de *Sistemas Operacionais* é uma disciplina fundamental nos currículos de graduação na área da Computação, ao lado de outras disciplinas ditas "de sistema", como *Microprocessadores*, *Arquitetura de Computadores*, *Redes de Computadores* e *Sistemas Distribuídos*. Nesse contexto, ela busca apresentar aos alunos os conceitos e mecanismos fundamentais usados na construção de sistemas operacionais e como esses conceitos podem ser usados – e influenciar – na construção de aplicações.

Os principais tópicos presentes em uma ementa típica dessa disciplina em cursos de graduação em Computação são os seguintes:

Arquitetura de sistemas de computação : componentes básicos do hardware (CPU, MMU, dispositivos de E/S); formas usuais de emprego de um sistema computacional (em lote, uso interativo, em tempo compartilhado, tempo-real, etc).

Arquitetura de sistemas operacionais : definição das principais funções do sistema operacional, dos componentes que as implementam e de como esses componentes podem ser organizados; apresentação das principais arquiteturas: monolítico, em camadas, micro-kernel, máquinas virtuais.

Processos e threads : conceitos básicos sobre contextos e fluxos de execução; multi-programação e tempo compartilhado; estruturas e mecanismos básicos para o controle da execução (PCB – *Process Control Block*, vetor de interrupções, trocas de contexto); ciclo de vida de processos e threads; escalonamento de CPU.

Comunicação e sincronização entre processos : condições de corrida e regiões críticas; mecanismos de sincronização; espera ocupada; deadlocks; dependência entre comunicação e sincronização; comunicações síncronas e assíncronas; troca de mensagens; memória compartilhada.

Gerência de memória : arquitetura de memória; resolução de endereços (em compilação, link-edição, carga, execução); endereços reais e virtuais; estratégias de alocação de memória; paginação e segmentação; memória virtual; *paging* e *swapping*; escalonamento de memória.

Gerência de Entrada/Saída : hardware de entrada/saída; polling e interrupções; DMA – *Direct Memory Access*.

Sistemas de arquivos : Dispositivos de armazenamento; conceito de arquivo e diretório; operações em arquivos; estratégias de alocação de arquivos; escalonamento de disco.

Redes : conceitos básicos de hardware de redes; comunicação por pacotes; protocolos; APIs de rede; serviços básicos; sistemas operacionais distribuídos.

Segurança : usuários, grupos e domínios de segurança; mecanismos de autenticação; propriedade e permissões de acesso a recursos; modelos de controle de acesso; problemas de segurança típicos.

Além de todos esses tópicos, poderiam ainda ser aprofundados na ementa assuntos específicos como escalonamento de tempo real, coordenação distribuída, sistemas de arquivos distribuídos, sistemas embarcados, máquinas virtuais, etc. Uma boa discussão sobre o conteúdo programático desta disciplina pode ser encontrada em [Ani00].

Como pode-se perceber, há uma profusão de tópicos inter-relacionados de forma nem sempre clara. Uma das principais características da disciplina de Sistemas Operacionais, observada por qualquer professor iniciando nessa área, é a relativa dificuldade em definir um seqüenciamento didático claro entre todos esses tópicos. Existem diversas inter-dependências, sobretudo no que se considera o "núcleo" da disciplina. Por exemplo, a noção de contexto de processo depende de conceitos associados à gestão de memória, como o espaço de endereçamento, enquanto esta depende de conceitos oriundos da gestão de processos, como o estado dos processos.

Situação similar ocorre com o conceito de memória compartilhada, necessário para a comunicação entre processos.

Como bem ressalta Tanenbaum em [TW99], muitos cursos não dão a ênfase adequada aos assuntos realmente importantes, como entrada/saída ou gerência de arquivos, e concentram-se demasiadamente em assuntos de importância secundária, como escalonamento de processos ou tratamento de deadlocks. Os programas da disciplina normalmente incluem até mesmo conceitos extremamente específicos ou complexos como a *anomalia de Belady* ou políticas de escalonamento para tempo real, mas normalmente falham em dar ao aluno uma visão global clara do sistema operacional, de suas partes constituintes e da integração entre elas. Com isso, observa-se que boa parte dos alunos concluem a disciplina conhecendo os principais conceitos e algoritmos, mas têm muita dificuldade em integrar todos aqueles conceitos em um conjunto coerente de software.

O sistema operacional constitui um conjunto de software extenso, complexo e de difícil compreensão global, por executar várias tarefas bastante diversas, mas complementares e inter-dependentes. De um lado, ele manipula detalhes do hardware, como registradores de CPU, MMU, DMA, TLB, IRQ, etc[1]. No outro extremo, ele deve definir políticas de alto nível para o uso dos recursos do sistema, e construir abstrações desses mesmos recursos para simplificar a construção e gerência das aplicações.

Como muito bem exposto em [PD95], o sistema operacional constitui muitas vezes uma "ponte" entre o mundo abstrato das teorias e algoritmos e o mundo prático e concreto do hardware. Essa característica de ponte entre dois mundos tão distintos pode tornar o ensino de sistemas operacionais uma tarefa trabalhosa e pouco efetiva. Notadamente, exige-se do alunos uma grande capacidade de construir abstrações mentais dos mecanismos envolvidos para poder compreender os aspectos mais densos da disciplina. Com isso, freqüentemente se observa que são raros os estudantes que completam a disciplina com uma visão coerente e abrangente sobre o tema.

Devido a essa característica de relativa complexidade, o ensino de sistemas operacionais deve ter uma componente prática significativa e muito bem elaborada, que complemente as aulas teóricas e, sobretudo, permita ao aluno ter uma compre-

[1] *Central Processor Unit, Memory Management Unit, Direct Memory Access, Translation Lookaside Buffer, Interrupt Request.*

ensão integrada das diversas partes de um sistema operacional. O autor acredita que a compreensão efetiva do tema só é alcançada quando o aluno torna-se capaz de questionar, argumentar e propor soluções próprias, ou seja, quando ele for capaz de reconstruir de forma espontânea (embora simplificada) o percurso histórico de desenvolvimento dessa área.

A experimentação é portanto uma atividade essencial no ensino de sistemas operacionais. Resta discutir a forma adequada de abordá-la, o que é um dos objetivos deste trabalho. O segundo objetivo é definir uma plataforma de software que permita aos alunos desenvolver atividades de experimentação em sistemas operacionais. Essa plataforma constitui um *sistema operacional didático*, que deve reproduzir com fidelidade as características mínimas de um sistema operacional e permitir a observação de seus mecanismos internos e de sua dinâmica de funcionamento. Além disso, ele deve permitir facilmente a inclusão de alterações, melhorias e novas funcionalidades.

Este documento está estruturado como segue: este capítulo introduz o contexto para o desenvolvimento do trabalho; o capítulo 2 apresenta os principais conceitos pedagógicos relacionados ao construtivismo e aplicáveis ao ensino de sistemas operacionais; o capítulo 3 analisa as principais abordagens para experimentação no ensino de sistemas operacionais; o capítulo 4 apresenta os principais sistemas operacionais didáticos em uso atualmente, analisando suas possibilidades e deficiências; o capítulo 5 descreve em detalhes uma proposta de sistema operacional didático, buscando cobrir as deficiências dos sistemas existentes e abrir novas possibilidades de uso; finalmente, o capítulo 6 apresenta as conclusões e perspectivas para a implementação e aprimoramento da proposta.

Capítulo 2

Abordagem Pedagógica

O tema central deste documento é a proposta de um ambiente computacional para desenvolver atividades de laboratório na disciplina de sistemas operacionais. Para que essa proposta tenha sucesso e seja bem recebida pelos alunos, ela deve ser baseada em uma metodologia pedagógica consistente e bem estruturada. Este capítulo busca introduzir alguns dos conceitos pedagógicos que norteiam a concepção da proposta, como a epistemologia genética e o construtivismo. Todavia, ele não deve ser entendido como uma contribuição original do autor ou mesmo uma introdução geral ao tema; material mais adequado pode ser encontrado em [Pia32, BS82, Bru90].

2.1 O desenvolvimento da inteligência humana

Conforme exposto em [BS82], a pedagogia moderna considera três teorias sobre o processo de desenvolvimento da inteligência humana, que são:

Empirismo : pressupõe que o desenvolvimento intelectual seja determinado pelo meio ambiente, ou seja, pela influência do meio. O indivíduo seria constantemente submetido a estímulos externos, desencadeando reações que podem ou não ser assimiladas como conhecimento. Desta forma, o desenvolvimento da inteligência partiria desses estímulos e não do ser humano, em um processo "de fora para dentro" que independe do mesmo. A inteligência de um indivíduo seria proporcional à qualidade do meio e à sua capacidade de receber e tratar os estímulos externos.

Racionalismo : esta teoria afirma que o indivíduo nasce inteligente e com o passar

do tempo reorganiza sua inteligência através das percepções do meio ambiente. Assim, o desenvolvimento intelectual seria determinado pelo sujeito e não pelo meio, em um processo "de dentro para fora". A capacidade do indivíduo determinaria como ele percebe a realidade ao seu redor.

Construtivismo : esta teoria afirma que o desenvolvimento intelectual é determinado pela relação do sujeito com o meio. Ela pressupõe que o ser humano não nasce inteligente, mas também não é totalmente dependente da influência do ambiente. Pelo contrário, o indivíduo interage com ele, respondendo aos estímulos externos, analisando, organizando e construindo seu conhecimento. Assim, através de um processo contínuo e interativo de fazer, analisar e refazer, o conhecimento vai sendo construído pelo indivíduo.

2.2 A epistemologia genética

O psicólogo suíço Jean Piaget lançou as bases do construtivismo, formulando uma teoria conhecida como *Epistemologia Genética* [Pia32, BS82], que analisa a evolução do raciocínio humano, do nascimento à maturidade do indivíduo. Através de suas pesquisas, Piaget concluiu que "o conhecimento se forma e evolui através de um processo de construção". A partir dessa conclusão, ele criou um modelo com base na interação *sujeito – objeto*, segundo o qual o conhecimento não está nem no sujeito, nem no objeto mas na interação entre ambos. A formação do conhecimento depende da ação simultânea do sujeito e do objeto um sobre o outro. A ação tem a função de estabelecer o equilíbrio rompido entre o sujeito e seu meio-ambiente, ou seja, é o elo entre o indivíduo e o mundo exterior.

Em seus estudos, Piaget considera duas formas de conhecimento:

Conhecimento físico : consiste no sujeito explorando os objetos.

Conhecimento lógico-matemático : consiste no sujeito estabelecendo novas relações com os objetos.

Para Piaget a *inteligência* é o processo dinâmico de interação entre o sujeito e o objeto, no qual o sujeito modifica os objetos e é modificado por eles. Piaget aborda a inteligência como algo dinâmico, decorrente da construção de estruturas de conhecimento que, à medida que vão sendo construídas, vão se alojando no cérebro.

A inteligência, portanto, não aumenta por acréscimo e sim por *reorganização*. Essa construção tem sua base biológica, mas vai se dando na medida em que ocorre interação com o objeto do conhecimento.

O conceito de *estrutura cognitiva* é essencial para sua teoria. Estruturas cognitivas são padrões de ação física e mental subjacentes a atos específicos de inteligência e correspondem a estágios do desenvolvimento infantil. Existem quatro estruturas cognitivas primárias (ou seja, estágios de desenvolvimento) de acordo com Piaget:

Estágio sensorial-motor , de 0 a 2 anos, no qual a inteligência assume a forma de ações motoras, através de reflexos neurológicos, através das quais a criança assimila as noções de tempo e de espaço.

Estágio de pré-operação , de 3 a 7 anos, de natureza intuitiva, no qual a criança se torna capaz de representar mentalmente o que ocorre no meio, priorizar certos aspectos em relação a outros e criar uma percepção global.

Estágio de operação concreta , de 8 a 11 anos, no qual a criança é capaz de executar ações concretas, consegue construir relações lógicas e abstrair dados da realidade, organizando a informação de forma sistemática.

Estágio de operação formal , de 12 a 15 anos, no qual o ato de pensar envolve abstrações profundas.

Sob a ótica de Piaget, pode-se afirmar que a criança aprende por si, construindo e reconstruindo suas próprias hipóteses sobre a realidade que a cerca. Nesse contexto, o erro, em vez de denunciar uma não-aptidão, torna-se uma etapa necessária do processo de construção do conhecimento [BS82]. Para Piaget os fatores mais influentes no desenvolvimento do conhecimento são:

- a maturação biológica;
- a interação com os objetos;
- a transmissão social (informações que o adulto passa à criança, ou transmitidas no seio do grupo social);
- a *equilibração*.

Este último ponto é o que contrabalança os outros três, ou seja, equilibra uma nova descoberta com todo o conhecimento até então construído pelo sujeito. Os mecanismos de equilíbrio são a *assimilação* e a *acomodação*.

Todas as idéias tendem a ser *assimiladas* às possibilidades de entendimento até então construídas pelo sujeito. Se ele já construiu as estruturas necessárias, a aprendizagem tem o significado real a que se propôs. Se, ao contrário, ele não possui essas estruturas construídas, a assimilação é deformante, resultando em um *erro construtivo*. Diante disso, havendo o desafio, o sujeito faz um esforço contrário ao da assimilação. Ele modifica suas hipóteses e concepções anteriores ajustando-as às experiências impostas pela novidade que não foi passível de assimilação. É o que Piaget chama de *acomodação*, onde o sujeito age no sentido de transformar-se em função das resistências impostas pelo objeto.

O desequilíbrio, portanto, é fundamental para que haja o erro construtivo, a fim de que o sujeito sinta a necessidade de buscar o reequilíbrio, o que se dará a partir da ação intelectual desencadeada diante do obstáculo: a *abstração reflexiva*. É nesse momento, na abstração reflexiva, que se dá a construção do conhecimento lógico-matemático (inteligência tradicional), resultando num equilíbrio superior e na conseqüente satisfação da necessidade. Portanto, o desenvolvimento da inteligência se processa para que o sujeito consiga manter o equilíbrio com o meio ambiente, redefinindo e recombinando suas estruturas de conhecimento de forma interativa e contínua.

Piaget afirma que a criança conhece como o cientista, ou seja: observando a realidade (epistemologia), interrogando-se sobre ela (levantando hipóteses) e investigando (elaborando uma teoria). A isso ele denominou *estudo crítico da ciência*. O aluno é o sujeito construtor do seu conhecimento; as deficiências ocorrem quando há falhas na interação entre o sujeito e o objeto do conhecimento.

2.3 O construtivismo

A partir da teoria de Piaget, J. Bruner e outros pesquisadores em educação elaboraram a *teoria construtivista* [Bru66, Bru90]. Um ponto relevante em sua teoria é que o aprendizado é considerado um processo ativo, no qual aprendizes constroem novas idéias, ou conceitos, baseados em seus conhecimentos passados e atuais. O aprendiz seleciona e transforma a informação, constrói hipóteses e toma decisões,

contando, para isto, com uma estrutura cognitiva. A estrutura cognitiva (esquemas, modelos mentais) fornece significado e organização para as experiências e permite ao indivíduo "ir além da informação dada" [BS82].

No contexto construtivista, o instrutor tem um papel especial. Ele deve incentivar os alunos a descobrir por si sós os princípios. O instrutor e o aluno devem se engajar em um diálogo ativo (aprendizado socrático). A tarefa do instrutor é traduzir a informação a ser aprendida em um formato apropriado ao estado verdadeiro de compreensão do aluno. O currículo deve ser organizado em espiral, estimulando o aluno a construir novos conhecimentos continuamente, sobre o conteúdo que já aprendeu, revisitando esse conteúdo de forma mais profunda e detalhada.

Bruner afirma em [Bru66] que a teoria de instrução deve se direcionar para quatro aspectos principais: 1) predisposição na direção do aprendizado; 2) modos nos quais um corpo de conhecimento pode ser estruturado, para que seja facilmente compreendido pelo aluno; 3) seqüências mais efetivas nas quais apresentar o material; 4) natureza e ritmo das recompensas e punições. Bons métodos para estruturar o aprendizado devem resultar em simplificação, geração de novas proposições e aumento da manipulação da informação. Isso permite enunciar alguns princípios para a organização do processo de aprendizagem:

- A instrução precisa estar preocupada com as experiências e os contextos que levam o aluno a estar pronto e apto para aprender.
- A instrução precisa ser estruturada para que possa ser facilmente compreendida pelo aluno (organização espiral do conteúdo).
- A instrução precisa ser criada para facilitar a extrapolação ou preencher as brechas no conhecimento (ir além da informação dada).

Tendo por base os princípios construtivistas, o professor tem de redimensionar seu trabalho. Ele passa a ser o mediador da relação entre o sujeito que aprende e o objeto do conhecimento. Essa mediação nada mais é que a intervenção planejada para favorecer a ação do aprendiz sobre o objeto. Para cumprir seu papel com êxito ele deve seguir alguns princípios que guiam seu relacionamento com os alunos:

- Respeito à produção individual do aluno, valorizando suas conquistas e estimulando-o na direção adequada;

- Prover um espaço adequado para o aluno interagir com o meio e testar suas hipóteses;
- Estimular o trabalho em grupo, favorecendo assim as interações sociais que facilitam o aprendizado.

Para o exercício da mediação, o professor precisa ter instrumentos para detectar com clareza o que seus alunos já sabem e o que eles não sabem. Para isso necessita de um conhecimento consistente do conteúdo, do objeto do conhecimento e de informações sobre o processo de construção, que lhe permita antecipar o caminho através do qual o aluno vai se apropriar desse conhecimento. Conforme [Wei99], um mediador é alguém que, em cada momento, em cada circunstância, toma decisões pedagógicas conscientes: nunca está limitado a corrigir ou deixar errado, pois além de informar e respeitar o erro quando construtivo, ele pode problematizar, questionar e ajudar a pensar.

Ainda segundo [Wei99], pode-se caracterizar uma atividade como uma boa situação de aprendizagem quando:

- os alunos precisam pôr em jogo tudo que sabem e pensam sobre o conteúdo em torno do qual o professor organizou a tarefa;
- os alunos têm problemas a resolver e decisões a tomar em função do que se propõem produzir;
- o conteúdo trabalhado mantém suas características de objeto sócio-cultural real sem transformar-se em objeto escolar vazio de significado social.

2.4 O construtivismo em sistemas operacionais

O ensino de informática envolve ao menos dois aspectos que devem ser abordados de forma distinta no que diz respeito à sua abordagem pedagógica. Por um lado, *informática é matemática*, o que direciona o aprendizado no sentido de favorecer a assimilação de conceitos lógico-formais, ou seja, a construção de abstrações mentais que permitam ao aluno compreender as bases matemáticas e algorítmicas subjacentes ao conteúdo apresentado. Essa é a perspectiva das disciplinas ditas

"fundamentais' como lógica de programação, algoritmos e estruturas de dados, ou mesmo de disciplinas mais avançadas como inteligência artificial e compiladores.

Por outro lado, *informática é engenharia*. Esse enfoque prioriza os aspectos arquiteturais do conteúdo apresentado, como as relações existentes entre os diversos elementos apresentados e as estruturas utilizadas para sua interação. Nesse contexto, os aspectos algorítmicos ou matemáticos do conteúdo apresentado devem ser colocados em segundo plano. Essa abordagem é certamente a mais adequada para as disciplinas ditas "de sistema", como é o caso de arquiteturas de computadores, sistemas operacionais, redes e sistemas distribuídos. Apesar disso, muitos professores dão uma clara ênfase aos aspectos algorítmicos no ensino de disciplinas de sistema, em detrimento dos seus aspectos arquiteturais. Citando Tanenbaum [TW99], "muitos cursos não dão a ênfase adequada aos assuntos realmente importantes, como entrada/saída ou gerência de arquivos, e concentram-se demasiadamente em assuntos de importância secundária, como escalonamento de processos ou tratamento de *deadlocks´´*.

O ensino de ciência da computação tem historicamente empregado uma abordagem *instrutivista*, que se baseia no desenvolvimento de um conjunto de seqüências instrucionais com objetivos pré-definidos [KA99]. O objetivo do ensino instrutivista é facilitar a transferência de conhecimento entre um mestre ativo e um aprendiz passivo. A abordagem clássica das aulas expositivas traz embutido o conceito do modelo instrutivista, buscando a exposição da maior quantidade possível de informação [Nul98].

Essa abordagem tem evidentemente suas falhas. Confrey [Con90] sugere que o modelo instrutivista imputa ao professor, ao invés do aluno, a responsabilidade pela simplificação do conteúdo didático. Assim, os alunos recebem um conhecimento simplificado de uma realidade complexa e são privados do conhecimento que o processo de simplificação em si pode trazer. A teoria construtivista coloca menos ênfase nas seqüências de aprendizagem e mais ênfase no projeto de um *ambiente de aprendizagem*, no qual o aprendiz desempenha um papel ativo e essencial. Sob sua ótica, aprender é um processo ativo de construção do conhecimento e instruir é sobretudo dar suporte ao processo de aprendizado, mais que simplesmente comunicar conhecimento [DS95].

Em [BA98], Ben-Ari expõe diversas possibilidades de uso de técnicas construtivistas no ensino de ciência da computação. Na prática do ensino de sistemas

operacionais, a aplicação de métodos construtivistas pode trazer diversos benefícios aos alunos. Sem dúvida, um de seus maiores benefícios é conduzir o aluno no processo de construir um modelo mental do computador e seu sistema operacional, que o permita conjeturar na direção correta e assim "ir além do conhecimento transmitido".

2.5 Conclusão

A teoria da epistemologia genética e as práticas construtivistas que dela decorrem têm mostrado ser ferramentas eficientes para o desenvolvimento da inteligência lógico-matemática. O emprego de técnicas construtivistas no ensino de ciência da computação já foi investigado por diversos autores [Den89, BA98]. No caso específico de sistemas operacionais, poucos estudos foram efetuados nesse sentido, como indica [PD95].

Através do exposto neste capítulo, é possível estabelecer algumas premissas básicas para o desenvolvimento de atividades de laboratório em sistemas operacionais sob uma ótica construtivista:

- O aluno é o *sujeito* que reflete sobre o *objeto* de seu conhecimento (o sistema operacional) e faz hipóteses sobre seu funcionamento.
- O *erro construtivo* é o resultado de uma hipótese lógica do aluno, necessário no seu processo de apropriação desse conhecimento. Para que o erro construtivo ocorra, é necessária a *interação* entre o aluno e o sistema operacional. Todavia é importante que o aluno possa descobrir e compreender as causas do erro, com o auxílio de ferramentas adequadas.
- O planejamento das *situações de aprendizagem* deve oportunizar a vivência de um contexto didático rico em interações com o objeto de estudo.
- A *troca entre iguais*, isto é, a interpretação entre os alunos, é fundamental para que eles avancem em suas concepções sobre o objeto de estudo. Por isso o desenvolvimento de atividades em grupo na sala de aula é fundamental. Como os alunos pensam de forma distinta, a heterogeneidade de uma turma enriquece e apóia o trabalho do professor.

- O *uso* do sistema operacional deve estar presente na sala de aula de forma ativa e concreta. Esse conhecimento deve ter significado para o aluno fora do contexto específico da disciplina, ou seja, no mundo real.

- Na *avaliação* é preciso considerar o processo que o aluno está vivendo e não apenas o produto da aprendizagem.

No próximo capítulo serão apresentadas as atividades usuais no ensino prático de sistemas operacionais, que serão analisadas sob a ótica do construtivismo e de sua possibilidade de aplicação efetiva no contexto desta proposta.

Capítulo 3

Laboratório de Sistemas Operacionais

Como exposto no capítulo 1, a componente experimental da disciplina de sistemas operacionais reveste-se de grande importância pedagógica, como complemento dos conceitos abordados nas aulas teóricas, mas sobretudo como ferramenta para que o aluno possa extrapolar seu conhecimento. Sob este aspecto, as aulas práticas de sistemas operacionais devem buscar os seguintes objetivos:

- Consolidar a compreensão dos mecanismos e estruturas básicas do funcionamento de um núcleo de sistema operacional típico.
- Compreender claramente como as diversas partes constituintes de um sistema operacional interagem e se integram.
- Observar o impacto das políticas internas do núcleo no funcionamento das aplicações.
- Compreender os compromissos envolvidos nas escolhas de implementação.
- Desenvolver a capacidade de propor e testar suas próprias soluções.

Para atingir esses objetivos, as aulas de laboratório devem ser planejadas de forma a incentivar o aluno a interagir com o sistema operacional e explorar seus diversos aspectos, tanto externos quanto internos. Essa certamente não é uma tarefa simples, haja vista a complexidade do tema e a ampla variedade de tópicos a

considerar. Outro fator dificultante é a dificuldade de observação da maior parte dos mecanismos envolvidos. Muitos dos mecanismos envolvem operações de baixo nível sobre os registradores da CPU (como as trocas de contexto), sobre os endereços de memória (como a memória virtual) ou em dispositivos de entrada/saída (como os *drivers* de disco).

As atividades de laboratório devem portanto explorar o uso dos serviços do sistema operacional para a construção de aplicações (aspectos externos) e a forma como esses serviços são construídos (aspectos internos). Na seqüência serão apresentadas e analisadas algumas das principais técnicas normalmente empregadas nas aulas práticas de sistemas operacionais, conforme constatado pelo autor, que são:

- Uso de abstrações do núcleo;
- Análise por inferência;
- Simulação de algoritmos;
- Exploração de código em sistemas reais;
- Uso de núcleos simulados.

3.1 Uso de abstrações do núcleo

Esta técnica consiste em propor a construção de programas aplicativos que façam uso dos mecanismos oferecidos pelo núcleo, como comunicação entre processos, controle de concorrência, escalonamento de tempo-real, etc. Seu objetivo básico é estimular a compreensão das abstrações oferecidas pelo núcleo do sistema operacional. O conteúdo pode ser organizado de forma a estimular o aluno a "sentir´´ a necessidade de novas abstrações, levando-o a reconstituir o desenvolvimento histórico.

Um exemplo típico de aplicação desta técnica é a construção de programas clientes e servidores para protocolos usuais de rede usando *sockets*, ou programas simulando problemas de sincronização usando semáforos, como o conhecido *Jantar dos Filósofos* e tantos outros.

Sob uma perspectiva construtivista, essa técnica deve ser abordada sob a forma de mini-projetos, nos quais o desafio é exposto aos alunos e o professor passa a ser uma referência de informação em caso de dúvidas (que não deve ser no entanto uma

referência passiva, mas instigante e desafiadora). De preferência, deve-se envolver o uso de programas ou ferramentas familiares ao aluno (como clientes de e-mail ou web comerciais, no caso dos servidores de rede). Esse fator traz diversos benefícios:

- contribui muito para contextualizar o conhecimento adquirido pelo aluno, deixando claras suas possibilidades e limitações;
- faz com que o aluno valorize seu trabalho e sua capacidade, na medida em que "o produto de suas próprias mãos" consegue interagir com softwares complexos do mundo real;
- desmistifica os produtos comerciais, demonstrando que, apesar de sua complexidade e sofisticação, estes seguem conceitos básicos e devem obedecer padrões para funcionar corretamente; freqüentemente o aluno "descobre" que consegue compreender o princípio de funcionamento do software comercial usado, e até arrisca desenvolver sua própria versão (obviamente simplificada) do mesmo.

Concluindo, esta técnica permite ao aluno alcançar uma boa compreensão das abstrações oferecidas pelo núcleo do sistema operacional, tornando-o capaz de usá-las para resolver seus problemas de programação. Todavia, ela não contribui para compreender como essas abstrações são construídas e gerenciadas pelo núcleo do sistema. Portanto, essa técnica é mais adequada para cursos onde o enfoque principal é o uso do sistema operacional, e não a construção e gerência de suas estruturas internas. Ela tem seu lugar na prática de laboratório de sistemas operacionais em ciência da computação, mas não pode ser a única abordagem utilizada. Além disso, as dificuldades de programação podem demandar um tempo considerável, sobretudo se os alunos não estiverem familiarizados com o ambiente computacional empregado.

3.2 Análise por inferência

Esta técnica tem como idéia motriz a utilização do sistema operacional de forma a suscitar conjeturas sobre o comportamento dos mecanismos do núcleo subjacente. Para tal, normalmente são desenvolvidos programas para testar a reação do núcleo a cargas computacionais diversas, como por exemplo demandas excessivas de uso de memória, de entrada/saída ou de criação de novos processos.

A atividade de laboratório a seguir representa, de forma bastante simplificada, um exemplo típico dessa abordagem:

Atividade: comparação de políticas de escalonamento

1. Escreva e compile o seguinte programa C:

```
void main ()
{
   while (1) ;
}
```

2. No LINUX, lance 5 processos a partir do executável gerado no passo anterior, cada um em uma janela de terminal distinta.
3. Analise a distribuição da CPU entre os processos lançados.
4. Repita os passos anteriores no ambiente WINDOWS.
5. Compare, discuta e estabeleça conclusões sobre os resultados obtidos.

Esta técnica busca estimular a capacidade do aluno em relacionar os conceitos abordados nas aulas teóricas ao comportamento de sistemas operacionais reais. Embora os mecanismos internos não sejam explicitamente "manipulados", seus efeitos e conseqüências diretas são analisados em situações concretas.

Qual o melhor contexto para a aplicação desta técnica ? Novamente, por explorar mais a influência das decisões do sistema operacional sobre as aplicações que seus aspectos internos, seu uso parece ser mais indicado a cursos cuja ênfase seja o uso do sistema operacional. Todavia, essa técnica também pode ser usada, de forma complementar, para auxiliar na compreensão das estruturas internas do núcleo, sobretudo no que diz respeito às estratégias de gestão dos recursos compartilhados (CPU, memória e dispositivos de E/S).

3.3 Simulação de algoritmos

O objetivo desta técnica é facilitar a compreensão dos algoritmos comumente encontrados na literatura para a definição das políticas internas do núcleo do sistema operacional. Para isso lança-se mão de programas que simulem algoritmos de

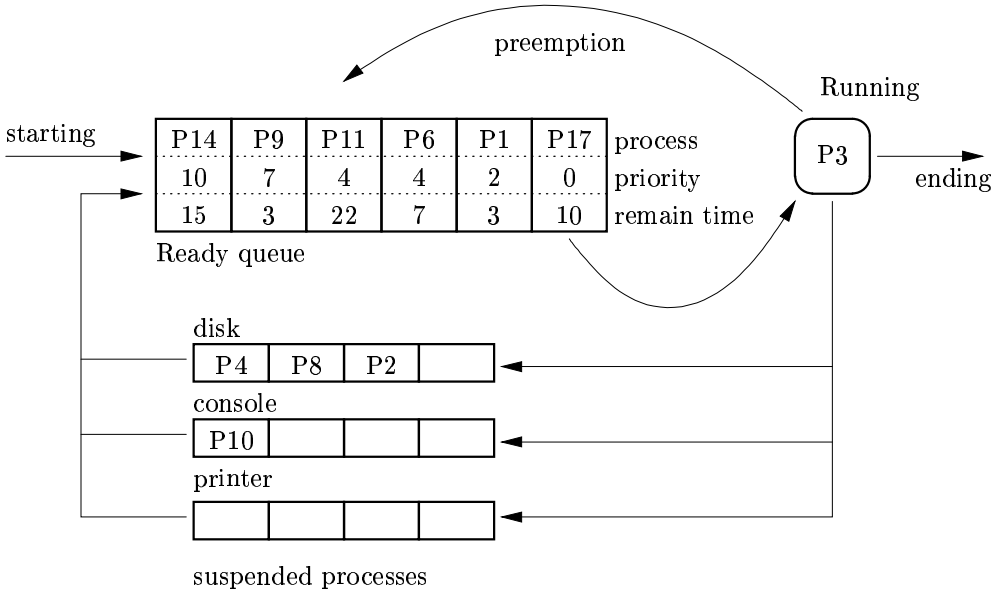

Figura 3.1: Simulador de escalonamento de processos

escalonamento de CPU, substituição de páginas de memória ou escalonamento de acesso a disco, gerando resultados estatísticos ou animações gráficas para facilitar a compreensão dos mesmos.

Um exemplo típico dessa técnica é a implementação de um simulador gráfico do escalonamento preemptivo de processos, mostrado na figura 3.1.

Esta técnica, também conhecida como *visualização de algoritmos*, proporciona oportunidades interessantes de aprendizado na observação e interação com a simulação. Este aspecto é bastante privilegiado pelo uso de tecnologias recentes, como a Internet. Podem ser encontradas muitas simulações de algoritmos típicos de sistemas operacionais implementados como *applets* Java e portanto disponíveis para interação via Internet [Win01, Tho01].

Alguns autores [PCJ96, Sta97] sugerem que os alunos devem atuar não somente no uso das simulações, mas na própria construção das ferramentas de simulação, para que estes adquiram a plena compreensão dos algoritmos simulados e suas implicações. A experiência pessoal deste autor confirma essa hipótese, indicando que a simples observação e interação com as simulações não é suficiente, pois muitos detalhes passam despercebidos. Somente através da implementação dos simuladores os alunos capturaram as sutilezas e deficiências dos algoritmos simulados.

Devido às possibilidades gráficas e interativas, esta técnica é bastante atrativa para os alunos e portanto mostra-se bastante efetiva sob o aspecto pedagógico. No entanto, geralmente as simulações tendem a ser excessivamente simplistas e concentrar-se em aspectos isolados do sistema, não contribuindo para a formação de uma visão global e integrada do mesmo. Além disso, a implementação dos simuladores pode se mostrar excessivamente trabalhosa, dependendo da sofisticação gráfica exigida.

3.4 Exploração de código em sistemas reais

Muitos professores consideram que a melhor forma de transmitir conhecimento aos alunos é fazendo-os interagir com o mundo real, através de problemas concretos. No caso de sistemas operacionais, isto poderia ser traduzido através da exploração de núcleos de sistemas operacionais reais e plenamente funcionais. Para tal, podem ser usados sistemas operacionais de produção com código-fonte disponível, como o LINUX [Lin01] ou o FREEBSD [Fre01]. Também podem ser utilizados sistemas operacionais simplificados, mas plenamente funcionais, desenvolvidos especificamente para esse fim, como o MINIX [TW99] e o XINU [Com84]:

MINIX (*mini*-UNIX): foi desenvolvido por Tanenbaum na *Vrije Universiteit Amsterdam* [TW99, Tan01]. Embora sua interface seja muito próxima do UNIX, inclusive implementando diversos de seus serviços e utilitários, como o shell `sh`, ele se baseia em uma arquitetura de micro-kernel com comunicação por mensagens. Será apresentado em detalhes na seção 4.1.

XINU (XINU *Is Not* UNIX): desenvolvido por Comer e outros na Purdue University [Com84]. Sua arquitetura é compacta e monolítica, mas estruturada logicamente em camadas. Seu código é sucinto e adequado ao ensino, sendo usado pelo autor sobretudo na demonstração de conceitos associados a protocolos e serviços de redes.

A análise e modificação do código-fonte de um núcleo estimula o aluno a "mergulhar" no sistema operacional, buscando com isso compreender simultaneamente estruturas, mecanismos, políticas e integração entre as partes. Entretanto, algumas barreiras podem tornar essa abordagem inviável no ensino de graduação:

- Essa abordagem exige uma carga horária de estudo significativa, tanto em classe quanto extra-classe, que pode não estar disponível aos alunos.

- O tamanho do código pode ser um fator complicador, sobretudo em núcleos reais. Por exemplo, o núcleo do LINUX na versão 2.4.10 conta com cerca de 3.3 milhões de linhas de código. Mesmo núcleos simplificados podem ser relativamente volumosos (o MINIX 2.0 tem cerca de 35000 linhas de código).

- Em núcleos reais, a profusão de detalhes de implementação pode tornar a compreensão do código uma tarefa árdua e pouco produtiva em termos didáticos, sobretudo se a documentação disponível for escassa, como é o caso do LINUX. Além disso, os núcleos reais são projetados buscando otimizar seu desempenho, o que pode tornar sua estrutura obscura e complexa. Por exemplo, o código do escalonador de processos do kernel LINUX é bastante difícil de ser interpretado, devido aos aspectos de escalonamento SMP (*Symmetric Multi-Processing*) que nele estão embutidos.

- A observação da dinâmica interna em núcleos reais pode ser uma tarefa complexa. Como se trata de um núcleo real, executando diretamente sobre o hardware, normalmente não é possível depurá-lo ou executá-lo passo-a-passo, como o permitiria uma aplicação convencional.

- O desenvolvimento em "modo núcleo" impõe uma boa dose de dificuldade técnica aos alunos. O ciclo de desenvolvimento intra-kernel consiste em editar, compilar (geralmente em uma outra máquina), instalar o núcleo na máquina-alvo, reinicializar o sistema e torcer para que funcione sem falhas. Esse é um processo longo e trabalhoso, que acaba prejudicando os objetivos efetivos da disciplina. Existe a possibilidade de executar o sistema operacional MINIX sobre um software emulador de hardware, como sugerido em [TW99], mas essa técnica resolve apenas parcialmente o problema. As etapas de instalação do kernel, reinicialização do computador e observação externa do funcionamento são bastante simplificadas, mas a depuração do núcleo através de ferramentas permanece um problema.

- Finalmente, o uso de núcleos reais impõe a necessidade de computadores dedicados a esse fim. Entretanto, a realidade habitual em cursos de graduação é o uso de laboratórios compartilhados entre várias disciplinas, ou mesmo entre

vários cursos. A instalação de núcleos reais nas máquinas, para experimentação pelos alunos, equivale a atribuir a eles privilégios de administração desses equipamentos. Isso é certamente indesejável do ponto de vista da segurança das instalações, pois os alunos poderiam (inadvertidamente ou intencionalmente) destruir ou corromper os softwares já instalados nas máquinas, prejudicando o andamento das demais disciplinas e gerando um custo administrativo significativo.

Do exposto, conclui-se que esta abordagem é adequada se os alunos tiverem um excelente nível técnico, que permita dominar rapidamente os problemas operacionais e a complexidade interna do núcleo, e se houver a disponibilidade de laboratórios específicos para esse fim. Finalmente, caso haja muita discrepância na capacidade técnica dos alunos da turma, esta abordagem pode provocar desistências por parte dos alunos mais fracos, ao invés de estimulá-los a sanar suas deficiências.

3.5 Uso de núcleos simulados

Este método envolve utilizar um sistema operacional simulado, total ou parcialmente, para demonstrar os conceitos teóricos e explorar sua implementação. Uma grande vantagem do sistema operacional simulado é o completo controle sobre sua execução por parte do aluno, que pode inclusive depurá-lo passo-a-passo em um ambiente controlado. Além disso, por se tratar de um software "convencional" sob o ponto de vista do sistema operacional das máquinas de laboratório, seu uso não impõe as restrições de exclusividade discutidas na seção anterior para o caso dos sistemas operacionais reais.

Os principais sistemas operacionais simulados de uso didático encontrados na literatura são os seguintes:

OSP : é um simulador de sistema operacional apresentado e usado em [KS92]. Não simula um sistema inteiro de forma integrada, mas possui diversos componentes isolados que modelam partes do sistema operacional.

PRMS (*Process and Resource Management Simulator*): um sistema operacional simulado com animação gráfica de algumas de suas estruturas e algoritmos internos, desenvolvido para o ambiente MS-DOS [HMOS90].

NACHOS (*Not Another Completely Heuristic Operating System*): desenvolvido na Berkeley University por Tom Anderson e outros [ACP93], é um simulador de sistema operacional bastante completo, escrito em C++ para diversos ambientes Unix. É um sistema muito popular, para o qual existem diversos manuais e projetos propostos. Será apresentado em detalhes na seção 4.2.

RCOS (*Ron Chernich's Operating System*): um simulador de sistema operacional destinado a demonstrar graficamente os principais conceitos e algoritmos internos do sistema. Ele também permite ao aluno modificar os algoritmos internos e acrescentar novas funcionalidades. Foi escrito em C++ sobre a plataforma MS-DOS [Che94, CJJ96].

RCOS.JAVA : não se trata de uma simples re-implementação em Java do sistema RCOS, mas de um novo ambiente, reformulado a partir da experiência do autor sobre o sistema anterior [JN01]. Será apresentado em detalhes na seção 4.3.

YALNIX : não é um sistema pronto, mas um projeto de graduação da *Carnegie Mellon University* para construir um sistema operacional completo *from scratch*, a partir de um hardware virtual simplificado [Kes00].

Um dos problemas enfrentados por esta abordagem é o custo de aprendizado (*learning curve*) associado à mesma, que pode ser significativo [Che94]. Além disso, o comportamento do sistema operacional simulado pode não representar exatamente o comportamento de um sistema real, ou pode ser excessivamente simplificado [WB92]. Devido a isso o aluno pode se ver confrontado a um "gap" significativo entre o universo simulado e um sistema operacional real.

3.6 Conclusão

Neste capítulo foram discutidas várias abordagens para as atividades práticas no ensino de sistemas operacionais para a graduação. No entanto, a perspectiva construtivista não considera uma separação clara entre teoria e prática, pois ambas são facetas do mesmo processo de construção do conhecimento. Portanto, para ser completa, esta discussão deveria envolver também as aulas teóricas, numa tentativa de reformular completamente os procedimentos de ensino dessa disciplina. No en-

abordagem	benefícios	deficiências
Uso de abstrações do núcleo	Compreensão do uso da API do núcleo para problemas de programação reais.	Os mecanismos internos do núcleo permanecem obscuros.
Análise por inferência	Compreensão do impacto das políticas do núcleo sobre as aplicações.	A forma como essas políticas são implementadas permanece obscura.
Simulação de algoritmos	Permite compreender os algoritmos internos do núcleo e como eles são implementados.	A observação pode ser simplista e omitir detalhes importantes no mundo real; geralmente a simulação se concentra sobre aspectos isolados.
Exploração de código em sistemas reais	Tem contato com o sistema real e completamente funcional.	Complexidade excessiva, dificuldades de utilização e desenvolvimento, dificuldades operacionais em laboratórios compartilhados.
Uso de núcleos simulados	Simulação da maioria das funcionalidades do núcleo em um ambiente controlado.	Risco de "gap" entre o ambiente e a realidade; curva de aprendizado muito longa.

Tabela 3.1: Quadro comparativo entre as abordagens apresentadas

tanto, esse objetivo exigiria um trabalho bem mais extenso e profundo, que o autor considera fora do escopo do presente trabalho.

As técnicas apresentadas têm características distintas em termos de necessidades (tempo disponível, conhecimento prévio do aluno, disponibilidade de equipamentos) e resultados pedagógicos, além de enfatizar certos aspectos dos sistemas operacionais em detrimento de outros. A tabela 3.1 apresenta um quadro comparativo que tenta sumarizar as principais benefícios e deficiências de cada uma das abordagens apresentadas.

Observa-se que nenhuma dessas abordagens é auto-suficiente para o ensino efetivo de sistemas operacionais. Segundo a análise do autor, a abordagem que se mostra mais promissora é a de uso de sistemas operacionais simulados. O uso de um sistema operacional simulado que forneça um ambiente rico em interações, com amplas possibilidades de modificação e suficientemente realista constitui uma abordagem privilegiada para o ensino dessa disciplina. Essa abordagem deve ser no

entanto complementada em segundo plano pelas demais abordagens.

No capítulo 4 serão apresentados e analisados os sistemas operacionais simulados Nachos e Rcos.Java, e também o sistema operacional didático Minix, amplamente utilizados no ensino de sistemas operacionais. Na seqüência, o capítulo 5 apresenta a proposta de um novo sistema operacional didático, que preenche lacunas deixadas por seus antecessores.

Capítulo 4

Sistemas Operacionais Didáticos

Neste capítulo serão apresentados com detalhes os sistemas operacionais didáticos mais populares atualmente. Serão abordados dois sistemas operacionais simulados (NACHOS e RCOS.JAVA) e um sistema operacional real reduzido (MINIX). Para cada um deles será detalhada sua arquitetura interna, enumeradas suas potencialidades pedagógicas e destacadas eventuais deficiências ou fontes de dificuldade para o aprendizado.

4.1 MINIX

O MINIX é um sistema operacional didático amplamente utilizado no ensino de graduação. Apesar de não ser um sistema simulado, mas um verdadeiro sistema operacional, operando nativamente sobre o hardware, sua presença no meio acadêmico e a qualidade de sua construção justificam plenamente sua apresentação neste trabalho. Alguns dos conceitos utilizados no MINIX serão considerados na proposta do capítulo 5.

Durante os anos 60 e 70 o estudo de sistemas operacionais tinha como referência prática o sistema UNIX, cujo código-fonte estava publicamente disponível sob licença da AT&T. Com o advento do UNIX versão 7, no início dos anos 80, a AT&T percebeu o potencial econômico do sistema e revogou a licença de acesso ao seu código-fonte. Não havia nesse momento outro sistema operacional com código aberto adequado às atividades de ensino, o que forçou muitos professores a reformular completamente suas atividades de laboratório em sistemas operacionais.

Nesse contexto, Andrew Tanenbaum, da *Vrije Universiteit Amsterdam*, resolveu

escrever um novo sistema operacional que pudesse ser usado no ensino, sem restrições de licença de uso. Esse sistema, a ser escrito a partir do zero, deveria manter compatibilidade com o UNIX do ponto de vista do usuário externo (ou seja, suas APIs[1] seriam as mesmas do UNIX), para aproveitar os programas já escritos e usufruir da simplicidade e eficiência dos conceitos desse sistema. Todavia, sua implementação teria de ser completamente independente de qualquer código proprietário, para não sofrer nenhuma restrição de uso devido a licenças.

O sistema, batizado de MINIX (de "mini-UNIX") foi escrito em C, com algumas pequenas partes em assembly, para a plataforma INTEL*X86*. Atualmente o sistema está em sua versão 2.0, que conta cerca de 35.000 linhas de código, e pode também ser executado sobre outras plataformas, como Macintosh, Amiga e Sparc. O MINIX implementa diversas funcionalidades presentes em sistemas operacionais modernos, como multi-programação por tempo compartilhado (processos), drivers de dispositivos, terminais, sistema de arquivos, memória compartilhada, etc.

Nas próximas seções será apresentada a arquitetura interna do MINIX e a implementação de suas principais funcionalidades. Boa parte do conteúdo aqui apresentado foi extraída e adaptada de [TW99].

4.1.1 Arquitetura

A arquitetura interna do MINIX é baseada em uma estrutura de camadas, mas incorpora também alguns conceitos de micro-kernel [RBF+89]. Essa arquitetura pode ser observada na figura 4.1.

A camada inferior (1) captura as interrupções e *traps*, gerencia a troca de contexto, o escalonamento de processos e as facilidades de IPC[2], oferecendo às camadas superiores um modelo abstrato de processos seqüenciais independentes que se comunicam via trocas de mensagens. As rotinas de troca de contexto e tratamento de interrupções são escritas em assembly, enquanto todo o restante do código é escrito em C.

A camada 2 contém os processos responsáveis pelas operações de entrada/saída, também chamados *tarefas de E/S* ou *drivers de E/S*. Cada tipo de dispositivo possui sua tarefa associada nessa camada. Uma das tarefas, a *tarefa de sistema*, não está

[1] *Application Programming Interfaces.*
[2] *Inter-Process Communication.*

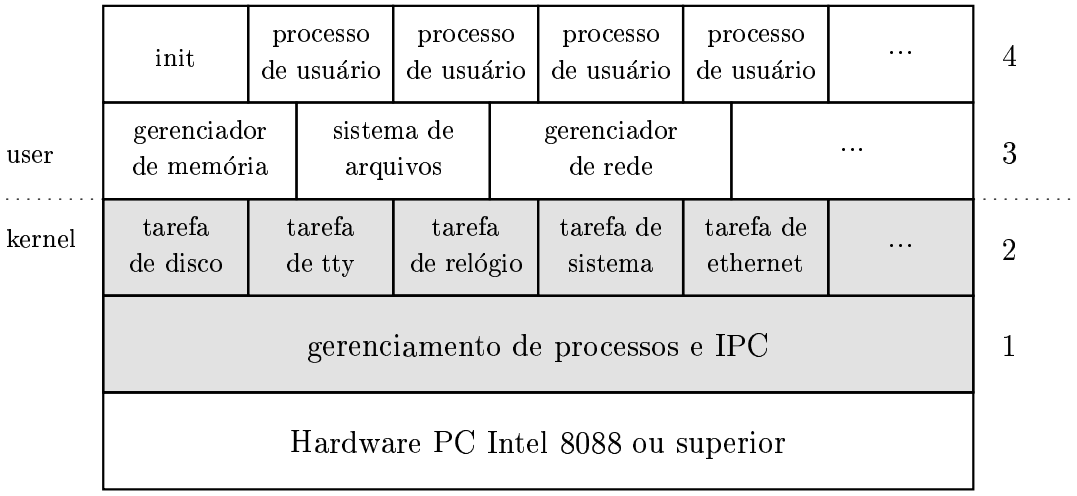

Figura 4.1: Estrutura em camadas do MINIX

associada a nenhum dispositivo; seu objetivo é fornecer serviços específicos a outras tarefas, como manipular regiões de memória, etc.

As duas primeiras camadas estão reunidas em um único arquivo binário que constitui o verdadeiro núcleo do sistema. Com essa estruturação, o MINIX pode ser caracterizado como tendo um micro-kernel, responsável pela noção de processo, troca de contexto, IPC, escalonamento e tarefas de entrada/saída. Estas últimas são independentes entre si, escalonadas como processos normais e têm prioridade abaixo das rotinas de tratamento de interrupção e troca de contexto, se o hardware permitir a definição de prioridades.

A camada 3 contém os chamados *processos servidores*, que fornecem serviços úteis aos processos de usuário, como a gestão de arquivos, de memória ou de rede. Esses processos executam em um nível menos privilegiado que o kernel e não podem acessar as portas de E/S diretamente, nem áreas de memória fora de seu próprio segmento. Enquanto as duas camadas inferiores se encarregam sobretudo da gestão dos recursos do sistema, esta camada é a grande responsável pela implementação das chamadas de sistema, oferecendo aos processos de usuário uma máquina abstrata.

A camada superior (4) contém todos os processos de usuário, como *shells*, editores, compiladores e programas do usuário. Estes processos são os menos prioritários do sistema, e não têm acesso direto aos seus recursos; todos os acessos a recursos têm de ser solicitados aos servidores da camada 3.

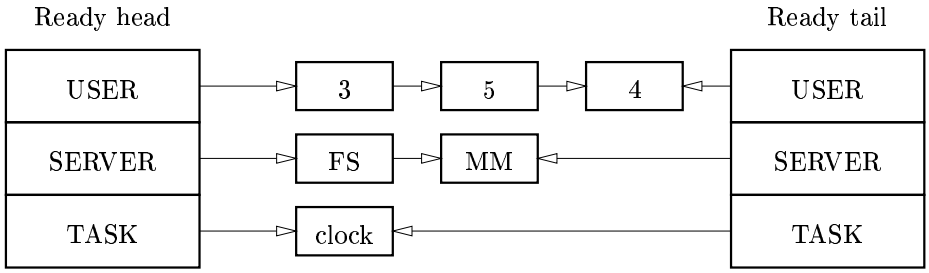

Figura 4.2: Escalonador do MINIX

4.1.2 Gerência de processos

O MINIX suporta apenas processos convencionais, no estilo do UNIX, onde cada processo possui um único fluxo de execução (*thread*). Os processos são lançados através das chamadas `fork()` e `exec()`, do padrão POSIX, e se organizam em uma árvore de processos cuja raiz é o processo `init`, lançado pelo kernel durante a inicialização do sistema.

O tratamento de interrupções obedece ao padrão para a arquitetura *X86*. São permitidas até 15 interrupções de hardware, associadas a dispositivos externos. O vetor de interrupções conta com 35 entradas que apontam para as respectivas rotinas de serviço. O tratamento de uma interrupção por sua rotina de serviço gera uma mensagem para a tarefa de E/S responsável pelo dispositivo em questão. Além disso, a rotina de serviço aciona o escalonador e pode agendar essa tarefa para execução logo em seguida. O processo de troca de contexto é relativamente complexo, pois envolve muito mais que a substituição do *Program Counter* atual; seu detalhamento foge ao escopo deste texto.

No que tange o escalonamento de CPU, o MINIX implementa uma política com três níveis de prioridade: o menos prioritário é o nível *user*, para processos de usuários (camada 4); a seguir vem o nível *server*, para processos servidores (camada 3) e finalmente o nível *task*, para as tarefas de kernel (camada 2). A figura 4.2 ilustra o escalonador. Os processos servidores e as tarefas de entrada/saída têm escalonamento FIFO não preemptivo, ou seja, podem executar até bloquear. Já os processos de usuário são escalonados em uma política *round-robin*, ou seja, FIFO com preempção por fim de *quantum*. A tarefa de relógio (*clock task*) é responsável por monitorar o tempo de uso da CPU pelos processos de usuário.

Os processos do MINIX se comunicam através de mensagens, usando o mecanismo conhecido como *rendez-vous*, ou comunicação síncrona. Essa escolha de projeto simplifica significativamente o sistema, pois não há necessidade de gerência de buffers de mensagens. Quando um processo executa um `send(dest,&msg)`, a camada inferior do núcleo verifica se o destino está esperando uma mensagem do remetente (ou de qualquer processo); se estiver, a mensagem é copiada do buffer do remetente para o buffer do destino e ambos são escalonados para execução. Se o destinatário não estiver esperando mensagens, o remetente é bloqueado e inserido em uma fila de espera associada ao destinatário. Reciprocamente, a recepção de uma mensagem por `recv(sender,&msg)` bloqueia o destinatário enquanto não houver processo tentando enviar uma mensagem para ele.

A comunicação via troca de mensagens é amplamente usada no sistema MINIX. Ela é empregada pelas rotinas de tratamento de interrupções para informar eventos às tarefas de entrada/saída; pelos processos de usuário, para solicitar chamadas de sistema aos processos servidores e por estes últimos, para solicitar operações específicas às tarefas de entrada/saída ou para informar eventos específicos (sinais) aos processos de usuário.

4.1.3 Gerência de dispositivos

O MINIX suporta vários tipos de dispositivos de blocos, como discos rígidos, discos de RAM e disquetes. Também suporta dispositivos de caracteres, como terminais e teclados. Para cada classe de dispositivos externos presente em um sistema MINIX, há uma tarefa distinta de E/S no núcleo (camada 2). Essa tarefa, também conhecida como *driver de dispositivo*, é um processo completo, com seu próprio estado, registradores e pilha, que encapsula todo o código dependente dos dispositivos físicos.

Os *drivers* de dispositivo são acionados pelos gerenciadores de sub-sistemas (camada 3), para atender pedidos dos processos dos usuários. A comunicação entre gerenciadores e *drivers* é feita através de trocas de mensagens, tornando o sistema menos eficiente que o UNIX (no qual as interações se dão por chamadas de procedimento), mas muito mais modular. A figura 4.3 ilustra essa estruturação, que torna o sistema suficientemente claro para ser facilmente compreendido pelos alunos.

A função de cada tarefa é aceitar solicitações de outros processos e executá-las.

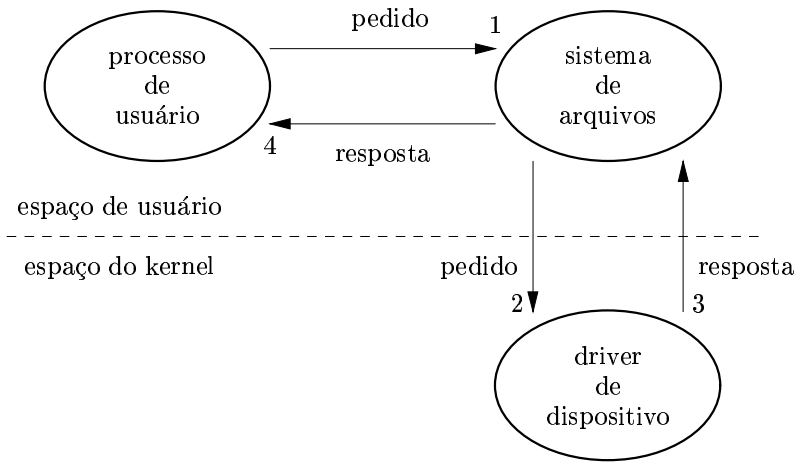

Figura 4.3: Interação entre processo de usuário, gerenciador e *driver* de dispositivo

Para dispositivos de bloco (discos), as tarefas são estritamente seqüenciais e não contém nenhuma multi-programação interna, para mantê-las simples. No caso de dispositivos de caracteres, há um certo grau de multi-programação, pois a mesma tarefa pode atender vários dispositivos (teclado, terminal, mouse, etc). Contudo, o tratamento de cada dispositivo em si é estritamente seqüencial: toda solicitação deve ser completada antes de aceitar uma nova solicitação. Como as tarefas tratam com dispositivos físicos concretos, sua implementação é relativamente extensa e complexa.

A *tarefa de sistema* (seção 4.1.1) desempenha um papel importante no sistema. Ao contrário das demais tarefas, ela não oferece acesso a um dispositivo de E/S, mas às estruturas internas do núcleo, de forma controlada. A tarefa de sistema oferece serviços para o gerenciadores de arquivos e de memória, permitindo que estes interajam com o núcleo para diversas atividades, como gerência de caches de arquivos, criação e destruição de processos, manipulação de mapas de memória, etc.

4.1.4 Gerência de arquivos

O sistema de E/S independente de dispositivo está tão intimamente relacionado com o sistema de arquivos que ambos foram fundidos em um só processo, denominado *gerenciador de arquivos*. Esse processo, localizado na camada 3 (e portanto fora do núcleo), gerencia tanto operações genéricas de E/S (interface com drivers, nomeação, proteção, alocação, bufferização e *caching*) quanto aspectos específicos dos sistemas

de arquivos (montagens, diretórios, etc.).

O gerenciador de arquivos do MINIX é portanto um processo autônomo que recebe mensagens com pedidos dos processos de usuários e interage com as tarefas do núcleo para atendê-los, também através de mensagens. No total, o gerenciador reconhece 39 mensagens distintas, das quais 31 são emitidas por processos de usuários e as demais pelas tarefas ou demais sub-sistemas. O sistema de arquivos do MINIX usa uma estrutura indexada similar à do UNIX, com arquivos referenciados por nós-i (*inodes*), e oferece diversas características importantes, como controle de acesso, diretórios, *pipes*, arquivos de dispositivos, *advisory locks* e *caching/buffering* de blocos, entre outras. Com todas essas características, o código do sistema de arquivos do MINIX torna-se bastante extenso, atingindo cerca de 100 páginas de código em *C*.

4.1.5 Gerência de memória

O gerenciamento de memória no MINIX é relativamente simples, pois não utiliza paginação nem troca (*swapping*). O gerenciador de memória mantém uma lista de lacunas classificada pela ordem de endereço inicial de memória. Quando uma área de memória é necessária (devido a uma operação `fork` ou `exec`), a lista é pesquisada através do algoritmo *first-fit*, até encontrar uma lacuna de tamanho adequado. Uma vez carregado na memória, um processo permanece na mesma posição até encerrar. O processo nunca muda de lugar, vai para disco ou muda sua quantidade de memória alocada. Essa estratégia simples deriva sobretudo do desejo de seu autor em poder usar o MINIX em qualquer computador compatível PC, mesmo os mais antigos, que não dispõem de hardware adequado para paginação ou segmentação.

O gerenciamento de memória é efetuado em um processo fora do núcleo, denominado *gerenciador de memória*, que se comunica com o núcleo através de mensagens. As decisões sobre a alocação da memória para os processos são feitas pelo gerenciador, enquanto a configuração real dos mapas de memória para os processos é feita pela tarefa de sistema, dentro do núcleo. Essa estrutura constitui um bom exemplo de separação entre *política* e *mecanismo*.

Os processos de usuário no MINIX podem ser compilados para usar o mesmo espaço de memória para código e dados, ou espaços separados. O primeiro caso torna a gerência mais simples, enquanto o segundo permite um uso mais eficiente da memória. O MINIX implementa o compartilhamento de memória entre processos,

permitindo que vários processos executando o mesmo programa compartilhem a mesma área de código (assim cada processo só precisa manter uma área privada de dados e pilha).

O gerenciador de memória possui duas estruturas de dados essenciais: a *tabela de lacunas* e a a *tabela de processos*. A tabela de lacunas é uma simples lista encadeada relacionando todas as áreas não utilizadas da memória. A tabela de processos contém as informações relativas à alocação de memória de cada processo do sistema. Para cada processo ela contém três entradas, indicando respectivamente os segmentos de texto, dados e pilha, que não necessitam ser consecutivos. Cada segmento é caracterizado por seu endereço virtual, endereço físico e tamanho. A conversão entre endereços virtuais e físicos é feita com o uso dos registradores de endereçamento indireto das CPUs *X86*.

4.2 NACHOS

O sistema operacional didático NACHOS (*Not Another Completely Heuristic Operating System*) foi desenvolvido na Universidade da Califórnia em Berkeley (UCB) no início dos anos 90, por Tom Anderson e outros [ACP93, SG97]. Ao contrário do MINIX, o sistema NACHOS não executa sobre um hardware nativo, mas sobre um hardware emulado, o que o classifica como um sistema operacional simulado. O sistema NACHOS executa inteiramente dentro de um processo UNIX, inclusive o hardware emulado, o que torna o desenvolvimento e depuração com esse sistema bem mais simples e eficiente que em um sistema operacional real. Ele pode ser executado em vários sistemas UNIX, entre os quais *Linux*, *Solaris* e *Digital Unix*.

A abordagem empregada na proposta do sistema NACHOS foi a de prover um sistema contemplando as principais áreas de sistemas operacionais – incluindo redes – da forma mais simples possível, mas plenamente operacional e utilizável. Para cada área foi elaborado um código mínimo, mas funcional, e um ou mais projetos para melhorar esse código, inserindo novas funcionalidades ou melhorando seu desempenho. O núcleo mínimo do NACHOS provê as seguintes funcionalidades: gerência de threads, sistema de arquivos, suporte básico para programas de usuário e um sistema de rede baseado em *mailboxes*. O código do sistema em sua versão mínima compreende cerca de 2500 linhas de código em *C++*, mas pode chegar a 15.000 linhas com sistema de arquivos nativo (o sistema mínimo é simplesmente um *stub*

para arquivos UNIX).

Na seqüência serão apresentadas as principais características do sistema NACHOS. Parte substancial do material aqui exposto foi extraído de [ACP93, Nar95, Kar01].

4.2.1 Arquitetura

Na concepção do sistema NACHOS, os autores buscaram aliar simplicidade e clareza com desempenho e facilidade de uso. A decisão de executar o núcleo sobre um hardware simulado permite simplificar a manipulação e depuração do código pelos alunos, pois o sistema pode ser depurado como um processo de usuário.

No entanto, executar *todo* o sistema operacional e as aplicações sobre esse hardware simulado também pode trazer problemas. Primeiro, todo o código do núcleo teria de ser escrito para executar sobre esse hardware simulado. Isso exigiria compiladores adequados para gerar código para o mesmo, ou então o núcleo teria de ser escrito no *assembly* dessa máquina. Além disso, a depuração do núcleo pelos alunos, ou mesmo sua observação seriam prejudicadas (pois teriam de ser feitas em *assembly*), a não ser que o depurador também possa ser executado sobre o hardware simulado. Finalmente, o desempenho de um sistema totalmente simulado poderia ser insatisfatório para executar aplicações realistas.

Essas restrições levaram a uma decisão de projeto bastante importante: a execução do núcleo é nativa, enquanto a dos processos de usuário é simulada. O núcleo do sistema NACHOS foi escrito em linguagem *C++* e está completamente envolvido pelo simulador de hardware, representado sob a forma de objetos. Dessa forma, as rotinas internas do núcleo podem ser depuradas normalmente, em uma linguagem mais facilmente compreensível pelos alunos. Sempre que o núcleo acessa um dispositivo externo (relógio, disco, interface de rede ou console), métodos nos objetos do simulador são invocados para efetuar a operação desejada.

Por outro lado, todo o código em modo usuário (ou seja, processos de usuário fora do núcleo) é executado via simulação da CPU e da memória: os processos de usuário são carregados em um grande vetor de bytes simulando uma memória paginada; cada acesso a esse vetor é analisado em relação a faltas de página ou outros erros; uma a uma, as instruções são lidas, decodificadas e executadas pelo objeto que representa a CPU. Eventos como chamadas ao sistema, faltas de página ou outras interrupções fazem com que o simulador de CPU pare seu trabalho e devolva o controle ao núcleo,

próximo do que faria um sistema operacional real.

Uma importante característica de software didático é a possibilidade de re-execução. Para permitir isso, o núcleo do NACHOS é completamente determinístico. Ao invés de usar sinais UNIX para representar o comportamento de dispositivos assíncronos (como discos ou o relógio), o núcleo mantém um relógio simulado, atualizado a cada instrução de programa de usuário e em certas chamadas dentro do núcleo. Esse relógio, independente do tempo físico, serve como base para ativar interrupções. Além disso, todos os eventos aleatórios do sistema usam o mesmo gerador de números pseudo-aleatórios. Com isso, as execuções são completamente reprodutíveis[3].

A arquitetura do sistema NACHOS pode ser vista na figura 4.4. Nessa arquitetura, existe uma pequena camada dependente do sistema operacional subjacente, devido à forma como as *threads* do núcleo são implementadas. Esse assunto será retomado mais adiante.

4.2.2 O hardware simulado

O hardware simulado sobre o qual o sistema NACHOS executa contém um processador, memória, relógio, discos, terminal serial (console e teclado) e interface de rede. Trata-se de um código relativamente compacto: o simulador do hardware ocupa apenas 2500 linhas *C++*, e simula os seguintes componentes de hardware:

Processador : o sistema simula um processador RISC MIPS R2/3000 (somente as operações com inteiros são suportadas). Esse processador possui 40 registradores, entre os quais alguns de uso específico, como o *PCReg* (contador de programa).

Memória : a memória está organizada em 31 páginas de 128 bytes acessíveis individualmente. Para a gerência de memória virtual, são suportadas tabelas de páginas e um TLB (*Translation Lookaside Buffer*).

Relógio : o sistema mantém um relógio simulado que serve como base para a geração de interrupções. Além disso, ele possui uma fila de eventos futuros, na

[3] Com exceção das ações envolvendo operações de rede, nas quais o assincronismo do mundo real se faz presente, pois o NACHOS usa *sockets* UNIX para simular interfaces de rede.

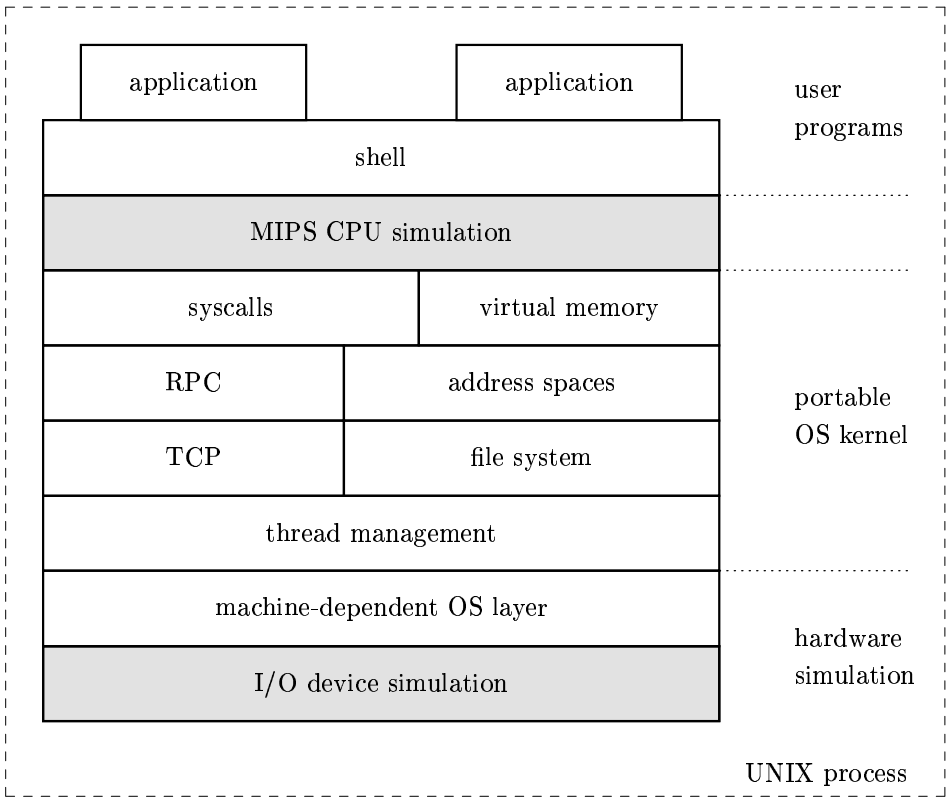

Figura 4.4: Visão geral da arquitetura do NACHOS

qual podem ser inseridas interrupções dos dispositivos de E/S (disco, terminal), permitindo simular o comportamento assíncrono dos mesmos.

Disco : um disco real é simulado através de operações para ler e escrever blocos em determinadas posições do mesmo. O NACHOS oferece um objeto `Disk` que oferece esses métodos e simula um disco rígido com múltiplas trilhas e setores. O tempo de resposta das operações varia de acordo com a distância "física" entre os blocos do disco. Como o funcionamento do disco é assíncrono e somente uma operação é executada por vez, cabe ao sistema operacional gerenciar a fila de pedidos de acesso pendentes. Para prover persistência de estado entre execuções, cada disco do NACHOS é mapeado em um arquivo real no UNIX.

Terminal : a classe `Console` provê mecanismos para criar terminais baseados em caracteres com interface serial, através da qual pode-se ler caracteres do teclado e enviar caracteres para a tela. A operação do terminal é assíncrona (*interrupt-driven*) e somente um caractere é transmitido por vez. Os descritores de arquivo que representam o terminal podem ser desviados para arquivos reais ou terminais do UNIX subjacente.

Rede : a interface de rede é similar à do terminal, exceto pelo fato de transmitir pacotes de tamanho fixo ao invés de caracteres. O núcleo provê suporte de rede para a transmissão ordenada, mas não confiável, de pacotes de bytes entre duas máquinas NACHOS. Pacotes podem ser perdidos pela rede, mas nunca corrompidos (o que torna desnecessário o uso de *checksums*). O percentual de perda de pacotes pode ser definido via linha de comando.

4.2.3 Sugestões de projetos

Um atrativo para o uso do sistema NACHOS é a sugestão de projetos feita por seus autores [ACP93]. O núcleo já implementa funcionalidades para a operação do sistema, mas essas implementações são minimalistas. Os autores sugerem então 5 projetos de pequeno/médio porte, para desenvolvimento durante um semestre de graduação, visando compreender, adicionar funcionalidades e melhorar o desempenho de cada um dos componentes do sistema:

Threads e concorrência : a gerência de threads do NACHOS é explícita, permitindo observar com detalhes o que ocorre em cada troca de contexto. O núcleo oferece threads cooperativas e semáforos. Este projeto envolve várias atividades: inicialmente, pede-se ao aluno observar o que pode ocorrer quando threads concorrentes acessam dados sem sincronização. Na seqüência, devem ser implementados *locks*, variáveis de condição e barreiras usando os recursos do núcleo. Finalmente, deve-se usar essas facilidades para resolver alguns problemas de concorrência típicos.

Sistemas de arquivos : o núcleo NACHOS oferece dois sistemas de arquivos: um sistema *stub*, que simplesmente traduz as chamadas do núcleo para chamadas ao UNIX, e um sistema nativo bastante primitivo. O sistema nativo provê arquivos de tamanho fixo (e bastante pequeno) e não possui sub-diretórios nem sincronização de acesso (apenas uma thread por vez pode acessar um disco). Além disso, o desempenho do sistema é sofrível, pois não há *caching/buffering* nem escalonamento de disco. Os objetivos deste projeto são a) corrigir algumas das limitações funcionais do sistema nativo e b) aumentar o desempenho no acesso a arquivos.

Multi-programação : o NACHOS traz código pronto para criar um espaço de endereçamento de usuário, carregar uma imagem executável nesse espaço e finalmente executar essa imagem (através do simulador de CPU MIPS). Somente um processo de usuário pode ser executado por vez. O objetivo deste projeto é alterar o código para suportar a multi-programação, implementar as chamadas de sistema necessárias (como `fork` e `exec`) e otimizar o desempenho do sistema, definindo políticas de escalonamento adequadas. Além disso, pede-se ao aluno que implemente um pequeno processo *shell* para lançar outros processos a partir de arquivos NACHOS. Uma vez concluído este projeto, o sistema pode executar aplicativos simples, via *shell*.

Memória Virtual : o objetivo deste projeto é substituir o gerenciamento de memória original do núcleo por um sistema de memória virtual paginado. A primeira parte do projeto é o tratamento das faltas de página: o núcleo deve interceptar a falta de página, localizar a página no disco (em um arquivo NACHOS), encontrar um quadro de memória livre onde carregar a página (possivelmente removendo outra página da memória para o disco), ler a página na

memória, atualizar as tabelas de páginas e retornar à execução do processo. Na seqüência, o aluno deve estabelecer políticas para a gerência de memória: como e quando descarregar páginas no disco (*swapping*), quando ler páginas antecipadamente (*read-ahead*), quantas páginas carregar inicialmente para um novo processo (*working set*), etc.

Operação em rede : cada processo UNIX executando o NACHOS é visto como um computador mono-processado. Uma rede de computadores pode ser simulada executando várias máquinas NACHOS e usando as interfaces de rede para transferir pacotes de rede de uma máquina para outra. O código do núcleo NACHOS provê, além da interface de rede, um protocolo básico de *mailbox*, que suporta troca de mensagens de tamanho fixo entre threads. A tarefa dos alunos consiste em a) implementar o suporte a mensagens de tamanho variável, b) tornar o protocolo robusto quanto à perda de pacotes e c) implementar um aplicação distribuída sobre esse protocolo (o autor sugere um sistema de arquivos distribuído, entre outros).

O sistema NACHOS vem sendo usado com sucesso em centenas de universidades ao redor do mundo. Além da documentação básica escrita pelos autores, a Internet oferece um vasto material sobre esse sistema. Alguns documentos relevantes para sua plena compreensão são o *A Road Map Through Nachos* [Nar95] e o *Nachos Project Guide* [Kar01]. Recentemente, um grupo de Berkeley iniciou o desenvolvimento de uma verão Java do sistema NACHOS, que ainda está em desenvolvimento [HC00].

4.3 RCOS.JAVA

O software RCOS.JAVA [JN01] é um simulador de sistema operacional escrito em Java. Entre suas características, destaca-se o amplo uso de animações gráficas para facilitar a observação dos mecanismos internos do núcleo e do funcionamento geral do sistema. Além disso, ele foi construído de forma a facilitar alterações pelos alunos, para implementar novas facilidades ou modificar seus mecanismos internos. O sistema RCOS.JAVA foi construído a partir da experiência anterior do autor na construção do sistema RCOS, construído em *C++* para o ambiente *MSDOS* [CJJ96].

Os sistemas RCOS e RCOS.JAVA foram desenvolvidos dentro de um contexto bastante específico: o ensino à distância. A *Queensland Central University*, na

Austrália, onde os mesmos foram desenvolvidos, possui vários *campi*, além de milhares de alunos seguindo seus cursos à distância. O módulo de Sistemas Operacionais da formação em Sistemas de Informação, por exemplo, tem apenas 20% de seus 200 alunos em aulas presenciais. Experiências com MINIX e NACHOS, entre outras, foram infrutíferas devido à dificuldade de instalação sem auxílio externo e/ou o baixo grau de interação entre o aluno e o sistema operacional em estudo [JN01].

O sistema RCOS.JAVA pode ser dividido em 5 grandes grupos de classes Java: hardware, núcleo, animação, sistema de mensagens e compilador *p-code*:

Hardware : é implementado um conjunto de dispositivos de hardware bastante simples, composto de processador, disco, memória e terminais. O processador modela uma máquina hipotética baseada em pilha, inspirada da proposta original de [KMH78] e também usada no sistema RCOS, denominada *P-machine*. Para o RCOS.JAVA essa máquina virtual foi levemente alterada para suportar o conceito de páginas. O disco representa um antigo modelo de disquete de 8"da IBM (IBM 3740). A memória RAM é representada por 20 páginas de 1024 bytes cada. Operações de E/S também podem ser feitas via 8 terminais de texto. A simplicidade do hardware é proposital, para amenizar a curva de aprendizado do aluno.

Núcleo : o sistema RCOS.JAVA possui um micro-núcleo (μ*-kernel*) inspirado do MINIX, que se ocupa de interrupções, trocas de contexto e comunicação entre processos. Além disso, ele gera mensagens para os demais componentes do sistema, informando eventos ou solicitando serviços. Os demais componentes do sistema são implementados como processos; atualmente estão implementados o escalonador de disco, o gerente de memória, o gerente de IPC (semáforos e memória compartilhada), o gerente de processos e o gerente de terminais.

Sistema de mensagens : os componentes do sistema comunicam através de um conjunto padronizado de mensagens que podem ser dirigidas aos animadores (explicados a seguir), ao sistema operacional ou a ambos. Todas as mensagens do sistema são encaminhadas através de um componente *Post Office*, que as encaminha aos seus destinatários. Como todas as interações no sistema ocorrem através de mensagens, foi adicionada uma facilidade de *registro*, que permite registrar uma determinada seqüência de operações no sistema e re-executá-la mais tarde. Essa funcionalidade permite passar aos alunos

execuções específicas ilustrando determinados conceitos de sistemas operacionais, como *starvation*, *deadlocks*, etc.

Animação : um grupo de classes denominados *animadores* é responsável pela representação gráfica do que ocorre dentro do sistema operacional. Alguns animadores oferecem a possibilidade de interagir com as estruturas internas ou o hardware (por exemplo, para mudar a política de escalonamento de processos). Os animadores atualmente implementados são: *escalonamento de processos*, *gerente de IPC* e *CPU*. A arquitetura do sistema foi projetada para prover um baixo acoplamento entre os animadores e o restante do sistema. Essa operação foi obtida através do sistema de mensagens: o componente *Post Office* automaticamente replica cada mensagem recebida, enviando a mensagem original ao destinatário e a cópia ao animador adequado. Com isso o código do sistema operacional propriamente dito permanece "limpo".

Compilador *p-code* : a CPU de RCOS.JAVA executa código de máquina *p-code* [KMH78]. O sistema contém um compilador *C/C++* simplificado, também escrito em Java, que gera código binário aceito pela CPU. Isto permite aos alunos desenvolver pequenos programas para o sistema. O compilador somente executa compilação simples, sem bibliotecas, *linking* ou características mais avançadas de *C/C++*. É possível portar o compilador para C e compilá-lo para executar na máquina RCOS.JAVA. Com isso, tornaria-se possível compilar aplicações *dentro* do próprio ambiente simulado.

A arquitetura do sistema foi desenvolvida com uso extensivo de técnicas modernas de programação orientada a objetos. São usados vários padrões de projeto (*design patterns*) para a estruturação do software, dentre os quais o padrões *model-view-controler*, *visitor*, *adapter*, *command* e *mediator*. Atualmente, o sistema RCOS.JAVA conta com mais de 250 classes e 100.000 linhas de código Java. Todo o código está disponível como *open source* em `http://rcosjava.sourceforge.net`.

Até a data deste documento, o sistema RCOS.JAVA não está pronto, e ainda não foi utilizado em uma experiência real de ensino. Como seu antecessor RCOS já foi amplamente utilizado, seus autores não esperam encontrar dificuldade em seu uso pelos alunos. A atividade atual de seus autores é o desenvolvimento de um contexto para as atividades didáticas, envolvendo exercícios e projetos com um claro enfoque construtivista, que possa ser utilizado com sucesso no ensino à distância. Os autores

planejam ainda a incorporação de funcionalidades de rede, CPUs mais complexas e a possibilidade de manipular dados multimídia, como áudio e imagens.

4.4 Conclusão

Neste capítulo examinamos em detalhe três sistemas operacionais didáticos comumente utilizados no ensino de sistemas operacionais. O objetivo didático foi o motor do desenvolvimento de todos, mas o enfoque e os resultados obtidos diferem significativamente entre eles.

O fato de executar sobre um hardware real torna o MINIX um sistema extremamente rico em oportunidades de aprendizado e interessante para os alunos, sobretudo por poder executar aplicações reais. A arquitetura do sistema é limpa e elegante, priorizando compreensão em vez de eficiência. Com exceção da comunicação em rede, praticamente todas as principais áreas do ensino de sistemas operacionais são cobertas nesse ambiente. Todavia, algumas delas são abordadas de forma superficial, por limitações do hardware (como a gerência de memória sem memória virtual) ou pela idade do projeto (ausência de threads nos processos de usuário).

No entanto, o uso do MINIX pode ser problemático em determinadas circunstâncias. Primeiro, ele exige um bom conhecimento prévio da arquitetura *X86* e da linguagem *C*. Segundo, o código do sistema é bastante extenso e envolve vários aspectos de baixo nível que podem tornar sua compreensão difícil, sobretudo se o aluno dispuser de pouco tempo para seu estudo. A observação interna do núcleo é difícil, pois não há possibilidade de executar o núcleo "passo-a-passo". Finalmente, como o MINIX executa diretamente sobre o hardware, seu uso exige instalação em um laboratório dedicado para esse fim. O desenvolvimento de projetos envolvendo a criação de novas funcionalidades pode exigir um espaço de trabalho para cada grupo durante toda a duração do projeto, o que nem sempre é viável.

Por sua vez, o sistema NACHOS tira proveito de sua simplicidade e do fato de ser um sistema simulado. Com as ferramentas de depuração adequadas, é possível observar e compreender a maioria dos mecanismos internos do núcleo. A abordagem dos autores, de prover um sistema funcional e pedir aos alunos a inclusão de novas funcionalidades ou a melhoria no desempenho tem se mostrado efetiva [Sla00].

Todavia, algumas deficiências podem ser observadas no NACHOS. A única forma de observar o comportamento do núcleo é através da depuração. O uso de animações

gráficas de certos mecanismos (como o escalonador de processos) traria benefícios para a compreensão dos mesmos, pois certos conceitos não são facilmente assimiláveis simplesmente observando o código. Além disso, a troca de contexto entre as threads do núcleo é confusa, pois é feita em *assembly* e manipula diretamente vários registradores da máquina real subjacente (e não da CPU simulada), o que pode confundir o software de depuração e o aluno. Além disso, as threads do núcleo são cooperativas, e todas as trocas de contexto entre threads do núcleo devem ser feitas explicitamente, o que diminui o realismo do sistema.

Outro ponto negativo é a falta de estruturas de dados globais (em virtude da abordagem orientada a objetos), o que torna a estrutura interna pouco convencional e isso pode dificultar a compreensão pelos alunos (por exemplo, não há uma tabela de processos contendo PCBs – *Process Control Blocks*). Além disso, os dispositivos de entrada/saída são modelados de forma excessivamente simplista.

A escolha de MIPS como CPU a simular não foi feita com critérios pedagógicos, mas pelo fato de ser a CPU do sistema onde o NACHOS foi desenvolvido. Apesar de ser uma CPU de tecnologia RISC [4], seu *assembly* certamente não está entre os de compreensão mais simples. Além disso, o desenvolvimento de programas de usuário necessita de ferramentas de desenvolvimento específicas, como um *cross-compiler* e um *linker* para essa arquitetura, caso o sistema seja executado sobre outra plataforma.

Quanto ao RCOS.JAVA, trata-se de um projeto recente e arrojado, construído a partir de experiências dos autores com sistemas anteriores, incluindo NACHOS e MINIX. As arquiteturas do sistema operacional simulado e do hardware subjacente são atraentes por sua simplicidade e conseqüente facilidade de aprendizado. Além disso, as animações podem oferecer boas possibilidades de observação e interação com o sistema. No entanto, não é possível afirmar sobre sua facilidade de uso ou efetividade no ensino, pois ainda não há experiência pedagógica concreta com ele. Além disso, a ausência de uma interface de rede priva o sistema de várias possibilidades didáticas interessantes. Outro ponto a ressaltar é que o volume de código (cerca de 100.000 linhas) e os conceitos empregados em sua construção podem exigir dos alunos um forte conhecimento em padrões de programação orientada a objetos. Por fim, o uso de Java pode distanciar muito o ambiente da realidade dos

[4] *Reduced Instruction Set Computer* – Computador com Conjunto Reduzido de Instruções.

sistemas operacionais, que são softwares de baixo nível geralmente escritos em *C*.

No próximo capítulo será apresentada uma proposta de sistema operacional didático que busca atender a vários requisitos, entre os quais a fidelidade aos conceitos, interatividade e facilidade de compreensão e uso. A proposta tem como base as características dos sistemas operacionais apresentados nesta seção, e tenta sanar algumas de suas deficiências.

Capítulo 5

SODA - um Sistema Operacional Didático Aberto

Conforme exposto nos capítulos precedentes, a disponibilidade de um ambiente propício à experimentação é importante para desenvolver atividades de ensino em sistemas operacionais. Também foram apresentados alguns sistemas operacionais didáticos em uso em universidades ao redor do mundo, sendo enumeradas as principais qualidades e deficiências dos mesmos. Este capítulo tem como objetivo descrever a proposta de um novo sistema operacional didático, que contemple adequadamente as necessidades de ensino de graduação no contexto da PUCPR.

Antes de propor um novo sistema, é necessário ter em mente o que se busca como resultado e, sobretudo, por que os sistemas existentes não contemplam as necessidades levantadas. É possível exprimir informalmente a idéia motriz deste trabalho pelo seguinte texto:

> *Propor uma solução mista entre sistema operacional real ou reduzido e simulador, que seja didática como um simulador e realista (empolgante) como um núcleo real. Idealmente, o sistema deve permitir desde a observação microscópica do núcleo, para estudar mecanismos de troca de contexto ou gerência de E/S, até a execução rápida de programas de usuário desenvolvidos pelos alunos, para estudar os efeitos macroscópicos das decisões e políticas internas do núcleo.*

Na seções subseqüentes essa "declaração de intenções" será detalhada, e a especificação (informal) de um sistema que a atenda será delineada.

5.1 Requisitos básicos

Idealmente o sistema a ser proposto deve atender aos seguintes requisitos:

Simplicidade : acima de tudo, o sistema deve ser construído em uma linguagem simples, com uma arquitetura simples, representando um sistema operacional simples, sobre um hardware simples. A estrutura do software resultante ou a linguagem de programação utilizada não devem ser empecilhos ao aprendizado.

Observabilidade : permitir observar e interagir com os mecanismos internos de baixo nível do sistema (visões microscópicas); possibilidade de associar a cada elemento físico (CPU, memória, dispositivos de I/O) ou lógico (escalonador de CPU, de memória, de disco, descritores diversos, sockets, ...) do sistema uma janela gráfica de observação e/ou interação.

Realismo : o sistema deve permitir a construção de aplicações sobre o mesmo, de forma similar a um sistema real. Os dispositivos de I/O deve ser capazes de permitir a implementação de aplicações simples, como jogos ou editores de texto. Além disso, o sistema deve prover meios para extrair informações estatísticas das execuções.

Arquitetura aberta : o software deve ser modular e incremental, permitindo facilmente a inclusão de novas características, como novos dispositivos de hardware e/ou novos serviços do núcleo. Obviamente, o acoplamento entre os elementos constituintes do sistema deve ser mínimo.

Além dos requisitos fundamentais acima apresentados, é possível enumerar diversas outras características desejáveis, embora não haja necessariamente um consenso sobre elas:

Uso de conceitos clássicos : os principais conceitos dos sistemas operacionais devem estar representados no sistema: processos, threads, arquivos, semáforos, etc. Por um lado, esta característica é desejável por retratar a realidade atual dos sistemas operacionais. Por outro lado, a pesquisa na área tem levado à revisão de vários desses conceitos e à criação de novos conceitos, como componentes, monitores e outros [Mil99].

Fidelidade : na medida do possível, o sistema deve apresentar as mesmas estruturas de dados internas usadas em sistemas reais, embora simplificadas. Em outras palavras, o sistema deve implementar descritores de arquivos, tabelas e listas de processos, tabelas de memórias, etc. Esta característica está obviamente ligada à anterior e, como ela, também é discutível. Por exemplo, o sistema NACHOS não apresenta uma tabela de processos no sentido clássico. Nesse sistema, cada processo é representado internamente por um objeto, e as referências desses objetos são manipuladas através de listas.

Re-execução : o sistema deve ter a capacidade de efetuar um registro histórico de sua execução e permitir a re-execução de seqüências previamente armazenadas. Esta característica é extremamente interessante, por tornar possível ao instrutor criar seqüências de execução representativas de situações especiais (*deadlocks*, *starvation*, *thrashing*, etc) e repassá-las aos alunos para análise. A capacidade de *undo/redo* também seria bem-vinda, mas sua implementação pode tornar o sistema excessivamente complexo e lento.

Portabilidade : o sistema deve ser pouco dependente do seu ambiente de execução. Esse é um requisito difícil de atender para um sistema operacional real, pois as estruturas de baixo nível sempre serão dependentes do hardware subjacente, mas pode ser alcançado no caso de um sistema simulado.

A lista de requisitos exposta é extensa, e atendê-la certamente não é uma tarefa simples. Na seqüência serão analisadas as principais decisões de projeto que elas implicam.

5.2 Decisões de projeto

Inicialmente devem ser estabelecidas as áreas a serem cobertas pelo sistema proposto. A partir dos sistemas operacionais didáticos expostos no capítulo 4, o autor acredita que o conjunto mínimo de áreas a serem abordadas deve incluir:

- Gestão de atividades
  - Suporte básico a threads (atividades)
  - Suporte básico a processos (contextos de execução protegidos)

  - Mecanismos de comunicação e sincronização (IPC)
  - Escalonamento de threads

- Gestão de memória
  - Endereçamento virtual
  - Tabelas de páginas e/ou segmentos
  - Tratamento de erros (faltas de página, endereços inválidos)
  - Paginação e swapping em disco

- E/S orientada a caracteres
  - Driver de porta serial
  - Terminais (console e teclado)
  - Controle de fluxo

- E/S orientada a blocos
  - Driver de disco
  - Sistema de arquivos
  - Escalonamento de acesso a disco

Posteriormente podem ser adicionadas outras características interessantes, como o suporte a dispositivos de E/S específicos (relógio, interface de rede, vídeo gráfico), protocolos de rede e controle de acesso (usuários, grupos e permissões), entre outros. Uma vez definidas as áreas a serem abordadas, as próximas decisões de projeto se referem a:

Modo de operação : Deve ser construído um sistema operacional simulado (como o NACHOS ou o RCOS.JAVA) ou um sistema real reduzido (como o MINIX) ?

Arquitetura do sistema operacional : o sistema operacional didático terá uma arquitetura monolítica (NACHOS), micro-kernel (RCOS.JAVA) ou em camadas (MINIX) ?

Arquitetura de hardware : que CPU, memória e periféricos utilizar ?

Implementação : qual será o ambiente, linguagem e metodologia de programação empregados na construção do sistema ?

Nas próximas seções serão definidas as principais características do sistema e analisadas as possibilidades de implementação e eventuais dificuldades esperadas.

5.3 Proposta do sistema operacional

De acordo com o exposto nos capítulos 3 e 4, o modo de operação do sistema mais adequado para o contexto de uso por alunos de graduação em laboratórios compartilhados ou em seus computadores pessoais é o núcleo simulado. Entretanto, o aspecto realista do sistema não pode ser esquecido, sob o risco do sistema ser considerado apenas como um "brinquedo" distante da realidade. Esta é, aliás, uma reclamação freqüente dos alunos confrontados a sistemas didáticos, conforme registrado em [Sla00].

No que toca a arquitetura do sistema, o núcleo monolítico é certamente o mais simples, compacto e rápido. No entanto sua compreensão não é necessariamente simples, pois as interações internas podem ser obscuras, prejudicando a compreensão pelos alunos. Além disso, erros podem se propagar por todo o sistema, dificultando o desenvolvimento de novas funcionalidades. Do exposto no capítulo 4, o autor presume que uma arquitetura em camadas, inspirada no modelo do MINIX, seja a melhor alternativa, por prover um modelo de fácil compreensão aos alunos.

Quanto ao ambiente de hardware, a escolha de um núcleo simulado faz com que o hardware também o seja, e portanto permite uma ampla escolha de dispositivos, conforme será discutido na seção 5.4. Caso fosse construído um núcleo real, a escolha se restringiria à arquitetura PC INTEL, por ser a mais popular em nosso contexto.

Idealmente, o núcleo simulado deve ser pouco dependente do sistema operacional e do hardware real subjacentes. Na medida do possível, as ferramentas de desenvolvimento empregadas (linguagens e ambientes) devem ser simples e familiares aos alunos. Caso a decisão fosse implementar um núcleo real reduzido, a escolha cairia necessariamente sobre a arquitetura PC INTEL, com desenvolvimento preferencialmente em C, por exigir programação em baixo nível.

O sistema proposto deve aproveitar os conceitos julgados mais significativos dos sistemas apresentados no capítulo 4, a saber:

Do NACHOS :

- Núcleo simulado, executando como um processo de um sistema operacional hospedeiro.
- Código do núcleo em uma linguagem compreensível aos alunos e depurável com ferramentas convencionais.
- Modelagem dos dispositivos de hardware como objetos.
- programas do usuário como binários executados via emulação da CPU e da memória.

Do MINIX :

- Estrutura μ-kernel, com as camadas inferiores implementando os mecanismos de IPC e o controle básico de threads.
- Servidores no espaço de usuário (embora implementados na mesma linguagem do núcleo).
- Comunicação entre threads através da troca síncrona de mensagens, para simplificar a implementação.

Do RCOS.JAVA :

- Implementação conjunta dos conceitos de threads (como unidades de execução) e processos (como unidades de contexto).
- Possibilidade de observação interna, interação e re-execução.
- Uso de uma CPU hipotética simplificada.

5.3.1 Arquitetura

O sistema proposto, denominado SODA (Sistema Operacional Didático Aberto) tem uma arquitetura de μ-kernel similar à do MINIX, mas executando sobre um hardware simulado, e pode ser vista na figura 5.1.

O núcleo propriamente dito (sombreado na figura) é constituído de três camadas. A *camada inferior* implementa threads, trocas de contexto, comunicação entre threads e recepção de interrupções, além de manter as estruturas de dados globais: descritores de processos e threads, filas de threads em espera de recursos, fila de

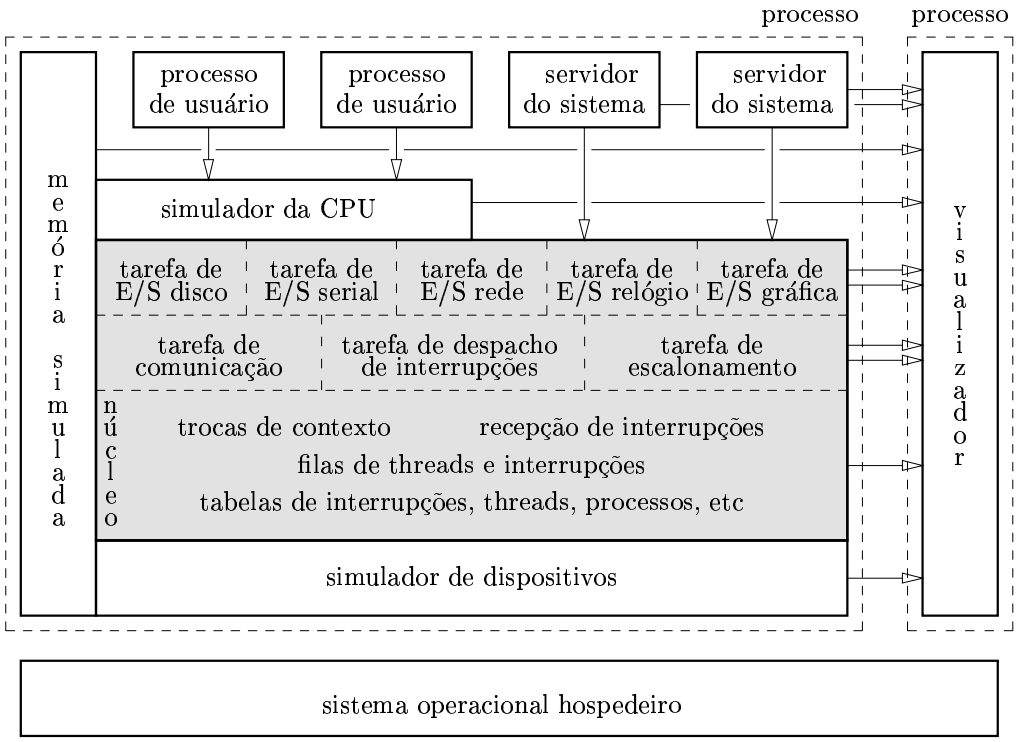

Figura 5.1: Arquitetura geral do SODA

interrupções pendentes, vetor de interrupções, etc. A *camada intermediária* é composta por tarefas (*tasks*) do núcleo, responsáveis pela gestão da comunicação entre threads, pelo escalonamento nas diversas filas de threads e pelo tratamento das interrupções recebidas. A *camada superior* é responsável pelas operações básicas de entrada/saída: ela contém as threads que gerenciam o acesso de baixo nível aos dispositivos físicos simulados (disco, terminal, rede, etc). Fora do núcleo, os processos servidores implementam os serviços de alto nível do sistema: sistemas de arquivos, gerência de memória, etc.

O hardware simulado está presente na forma de dispositivos de E/S, memória e processador. Para simplificar o uso do sistema, será empregada a mesma abordagem do NACHOS: somente os processos de usuário serão executados sobre o processador simulado. Todo o restante do código, inclusive os servidores do sistema, que executam em modo usuário, serão executados sobre o processador real da máquina hospedeira. No entanto, deve se tomar o cuidado de garantir que os processos servidores não tenham acesso ao espaço de endereçamento do núcleo.

O sistema operacional propriamente dito é implementado como um único processo do sistema operacional hospedeiro. Paralelo a ele, um segundo processo implementa o *visualizador*. Esse processo é responsável pelas janelas de observação e interação do usuário com o sistema operacional simulado. Sua execução é opcional; o núcleo deve funcionar normalmente caso o visualizador não esteja presente. A interação entre o sistema e o visualizador se dá também através de mensagens.

5.3.2 Processos e threads

Ao contrário de seus antecessores, o SODA deve implementar de forma coordenada as noções de thread, como uma unidade lógica de atividade (linha de execução), e de processo, como uma unidade lógica de contexto (espaço de endereçamento). Todas as threads do sistema estão associadas a algum processo[1] e um processo pode possuir mais de uma thread associada. Para simplificar, será adotado o modelo de multithreading *um-para-um*, no qual cada thread de processo em modo usuário corresponde a um thread em modo núcleo [SGG01]. Cada thread possui sua própria pilha de execução, na qual também pode armazenar suas variáveis locais.

[1] As threads internas do núcleo são consideradas como estando associadas a um processo "núcleo", com PID (*process identifier*) zero e espaço de endereçamento igual à totalidade de memória real do sistema, usando apenas endereçamento real.

Uma thread deve ser implementada como um registro contendo no mínimo os seguintes campos:

- *Thread Identifier (TID)*: identifica de forma única a thread no sistema. Pode ser o índice do registro da thread no vetor de threads do núcleo.
- *Host Process*: identifica o processo ao qual a thread está associada, que define seu espaço de endereçamento.
- *Program counter*: a posição corrente da thread em seu código, no espaço de endereçamento definido pelo *host process*.
- *Stack Base*: a posição de início da pilha da thread, idem.
- *Stack top*: a posição do topo da pilha da thread, idem.
- *Send buffer*: buffer de tamanho fixo contendo a mensagem a enviar no momento de uma chamada `send(msg,dest)`, onde `dest` é o número da thread a quem a mensagem deve ser entregue.
- *Recv buffer*: buffer de tamanho fixo no qual é depositada a mensagem recebida em uma chamada `recv()` ou `recv(from)`. Ambos os buffers estão localizados no espaço de endereçamento definido pelo *host process*.
- *State*: define o estado atual da thread, como sendo um dos seguintes:
  - `new`: a thread está sendo criada e ainda não é escalonável.
  - `ready`: a thread está somente aguardando pela CPU.
  - `waiting`: a thread está suspensa, esperando por um evento externo (interrupção de dispositivo físico), por um semáforo ou por um intervalo de tempo.
  - `finished`: a thread concluiu sua execução através da chamada `exit(code)` ou foi abortada devido a algum erro (instrução ilegal, endereçamento ilegal).
- *Queue*: indica a fila na qual a thread está suspensa (estado `waiting`) ou a fila de threads prontas (estado `ready`).

Um processo é minimamente implementado como um registro contendo:

- *Process Identifier (PID)*: identifica de forma única o processo no sistema. Pode ser o índice do registro do processo no vetor de processos do núcleo.
- *Thread Table*: lista ou vetor indicando as threads associadas ao processo. Se todas as threads associadas tiverem terminado o processo pode ser removido do sistema pelo núcleo. Esta tabela se encontra no núcleo.
- *Local Descriptor Table*: lista de descritores de arquivos abertos. Cada descritor contem informações mínimas sobre o respectivo arquivo. Caso se opte por usar a abstração de arquivo para acesso a outros dispositivos (portas seriais ou conexões de rede), os descritores terão de ser adequadamente ajustados. Esta tabela se encontra no espaço do processo.
- *Page table*: esta tabela indica o mapeamento entre as páginas do espaço de endereçamento virtual do processo e os quadros de memória real ao qual elas estão associadas. Esta tabela se encontra no espaço do núcleo e sua estrutura depende do modelo de MMU (*Memory Management Unit*) considerado (ver seção 5.4).

A camada inferior do núcleo SODA deve implementar:

- Os vetores contendo as threads e processos do sistema.
- As listas de threads para escalonamento e espera por eventos.
- As rotinas de criação e término de threads e processos.
- A rotina de troca de contexto entre threads, que pode ser acionada de forma explícita (cooperação) ou via interrupção de tempo (preempção).
- A rotina de troca de contexto entre processos, a ser acionada caso ocorra uma troca de contexto entre threads de processos distintos.
- O mecanismo básico de tratamento de interrupções.

5.3.3 Tratamento de interrupções

O tratamento de interrupções no SODA está divido em dois níveis: a recepção da interrupção e seu tratamento. A recepção de uma interrupção é feita pela camada inferior do núcleo onde, a cada interrupção, uma rotina simples é acionada para inserir um *registro de interrupção* em uma *lista de interrupções pendentes*. Após a recepção da interrupção, ocorre uma troca de contexto e a primeira thread da fila de prontos reassume o controle. Uma thread de sistema chamada *tarefa de despacho de interrupções* (*interrupt dispatch task*) é responsável por varrer a lista de interrupções pendentes e processá-las. Obviamente essa tarefa tem prioridade sobre as demais, para evitar o acúmulo de interrupções não tratadas.

Essa abordagem é bastante próxima do mecanismo de *bottom-half* do núcleo LINUX [Rus99]. Em [GK95] é proposta uma outra abordagem, na qual cada interrupção dá origem a uma nova thread no sistema, escalonada para tratar a interrupção que a gerou. Apesar de ser uma abordagem interessante, o autor acredita que isto poderia complicar em demasia o núcleo.

Um tipo particular de interrupção é a interrupção de tempo, essencial para implementar o escalonamento preemptivo de CPU. A definição correta do referencial de tempo do sistema é importante não somente para implementar a interrupção de tempo, mas para registrar a passagem do tempo e permitir medidas de desempenho do sistema. Então, como simular e registrar adequadamente a passagem do tempo ?

Vincular o relógio do sistema ao tempo físico do sistema operacional subjacente não é uma solução adequada, pois impediria a observação pelos alunos de fenômenos internos do núcleo, como as trocas de contexto, o tratamento das interrupções e a execução das chamadas de sistema. A abordagem usada no sistema NACHOS [ACP93] é manter um relógio simulado interno, completamente desconectado do tempo físico, com seu avanço determinado apenas por um escalonador de eventos. Esta técnica provém dos simuladores a eventos discretos e parece ser bastante adequada para modelar a passagem do tempo no sistema.

Assim, o sistema deve manter um *escalonador de tempo* junto ao modelo do hardware, para simular adequadamente a passagem do tempo e a geração de interrupções. Todo dispositivo de E/S deve gerar eventos futuros que serão inseridos nesse escalonador em resposta a solicitações do núcleo. Solicitações ao temporizador também serão inseridas no escalonador de tempo, para gerar futuras interrupções de tempo e preempções. A gerência do escalonador de tempo deve estar a cargo

de uma thread independente (não uma thread do núcleo simulado, mas uma thread do processo real). Deve-se observar que, do ponto de vista do sistema operacional hospedeiro, o processo do sistema possui apenas duas threads: uma que implementa o núcleo e demais estruturas simuladas e outra que implementa o escalonador de tempo). Extremo cuidado deve ser tomado para evitar que os alunos confundam as estruturas do escalonador de tempo com as estruturas do sistema operacional simulado.

5.3.4 Comunicação entre processos

Toda a comunicação dentro do sistema se dá através de mensagens síncronas (*rendez-vous*) de tamanho fixo e formato padronizado, usando o modelo do MINIX. Essa abordagem simplifica muito o sistema, pois torna desnecessária a gestão de buffers. Por comunicação síncrona, entende-se que tanto o envio quanto a recepção bloqueiam as respectivas threads até que a mensagem tenha sido transferida. Cada thread tem um buffer de mensagens com duas posições, uma para a próxima mensagem a enviar e outra para a última mensagem recebida.

As mensagens trocadas entre o sistema operacional propriamente dito e o processo visualizador podem ser síncronas ou assíncronas. Os eventos na interface de visualização que implicam em alterações internas no comportamento do núcleo ou dos servidores são comunicados via mensagens síncronas. Por sua vez, os eventos internos do sistema que implicam em atualização da interface de visualização são enviados via mensagens assíncronas (sem reconhecimento nem controle de fluxo). Com isso, a carga computacional representada pelas animações gráficas fica desacoplada da operação do sistema.

Em [VH96] é feita uma análise crítica da comunicação por mensagens e apresentada uma variante de RPC (*Remote Procedure Call*) para comunicação entre domínios de endereçamento distintos. A comunicação por mensagens é pouco eficiente, mas torna a implementação do sistema mais simples e modular, que são justamente alguns dos objetivos almejados na construção do SODA.

5.4 A máquina virtual

O sistema operacional SODA executa seu próprio código sobre o sistema operacional hospedeiro, mas seus processos de usuário devem executar sobre um hardware virtualizado, desconectado do hardware real da máquina. Com isso, todo o controle de baixo nível dos dispositivos de hardware do SODA pode ser feito de forma controlada, sem riscos para o sistema hospedeiro. Esta abordagem também é empregada no NACHOS e no RCOS.JAVA, embora cada um tenha suas peculiaridades.

O hardware virtual empregado no sistema NACHOS é composto por uma CPU MIPS (somente as operações inteiras são simuladas), uma unidade de memória com gestão de endereçamento virtual incorporada, unidades de disco e portas seriais para terminais [Nar95]. O NACHOS tem sido alvo de críticas porque, embora sua CPU simulada seja bastante fiel à realidade, o mesmo não ocorre com a memória e os dispositivos de E/S. Por exemplo, não há uma estrutura de MMU (*Memory Management Unit*) explícita; as conversões de endereçamento são feitas pela CPU. Além disso, a interface de acesso a disco e terminais é excessivamente abstrata. No NACHOS, cada disco é uma instância da classe `Disk`, que oferece as seguintes operações:

```
// create a new disk, bound to a real file on host OS;
// callWhenDone is a function to simulate disk IRQ
Disk (char *realFileName, VoidFuncPtr callWhenDone, int callArg) ;

// read a disk sector in a memory buffer
ReadRequest (int sectorNumber, char *data);

// write a disk sector from a memory buffer
WriteRequest (int sectorNumber, char *data);

// compute time to access a new sector from the current location
ComputeLatency (int newSector, bool writing);
```

Como pode-se observar, essa interface abstrai completamente o nível dos *device drivers*, o que priva o aluno da oportunidade de interagir com esses mecanismos de baixo nível que são parte fundamental dos sistemas operacionais modernos. O sistema SODA deve oferecer uma implementação mínima dos níveis inferiores de acesso

a dispositivos de E/S, deixando aos alunos a tarefa de implementar *drivers* para algum dispositivo específico (porta paralela ou placa gráfica, por exemplo). Como exemplo, um diagrama simplificado da estrutura de acesso a arquivos em disco no SODA pode ser vista na figura 5.2. Nessa estrutura, somente o nível *implementação do disco* deve ser opaco aos alunos.

No sistema SODA, os dispositivos de E/S devem ser simulados em seu nível mais baixo, para tornar o sistema mais próximo da realidade. Um exemplo hipotético de funções para acesso a disco em *C* seria:

```
// create a new disk with number diskNumber, associate to real file
// realFile, with IRQ number irqNumber and start IO port at ioPort.
// The other I/O ports follow at ioPort+1, ioPort+2, etc.
diskInit (int diskNumber, char *realFile, int irqNumber, int ioPort) ;

// read at IO port a byte value
ioGet (int port, short *value);

// write at I/O port a byte value
ioPut (int port, short value);
```

Essa abordagem exige a definição clara do *protocolo de acesso* a cada dispositivo. Idealmente os *drivers* para disco e terminal devem ser fornecidos aos alunos, mas é interessante que os mesmos desenvolvam *drivers* para outros dispositivos.

No tocante ao processador, há diversas possibilidades. A escolha do processador tem impacto na geração de código para os processos de usuário, no mecanismo de interrupções e E/S e no modelo de memória virtual a ser empregado. Pode-se optar por um processador atual de mercado, como a família *X86*, mas sua complexidade pode tornar a implementação do sistema muito complexa. Por exemplo, o 80386 tem um modelo de memória que combina segmentação e paginação, com tabelas de páginas com 4 níveis de indireção e uma série de mecanismos distintos para referência à memória.

Também é possível optar por um processador de mercado mais antigo e portanto mais simples. Por exemplo, o 8085 ainda é bastante usado em automação industrial e apresenta uma estrutura interna bastante simples. No entanto, ele não possui nenhuma forma de gerência de memória virtual. O sistema NACHOS usa uma CPU comercial MIPS simplificada, com suporte a memória paginada via tabelas de páginas

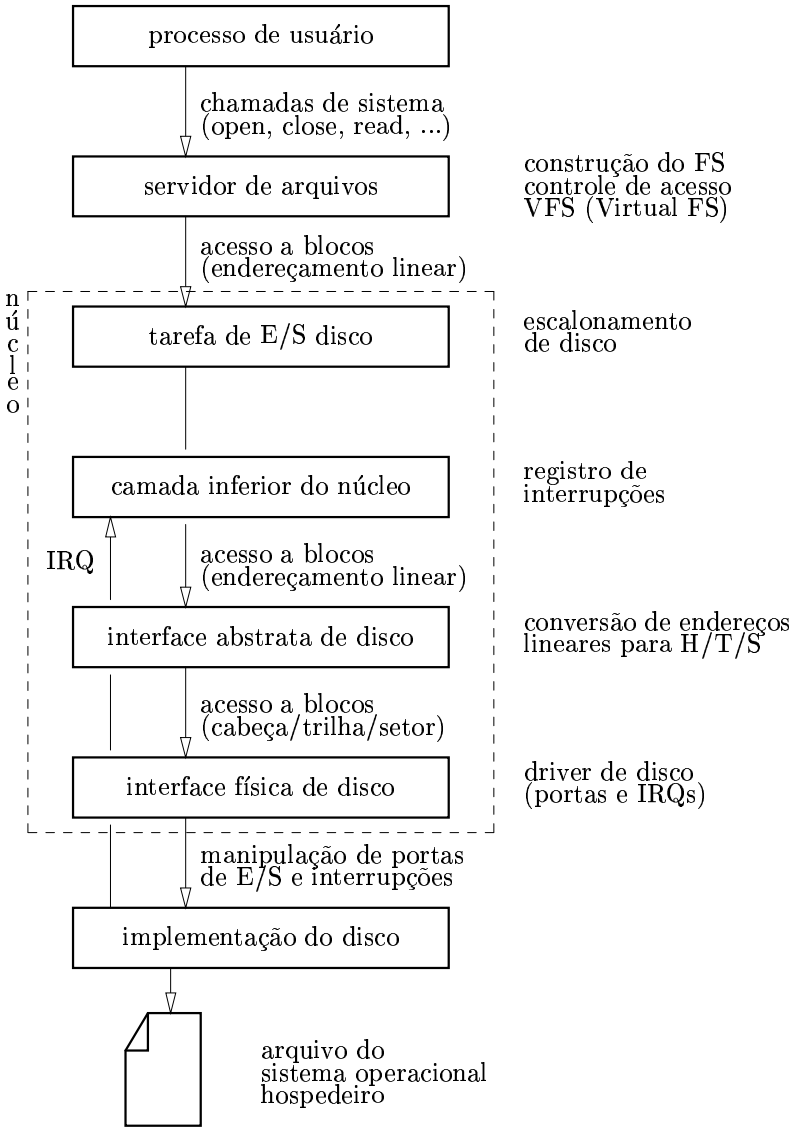

Figura 5.2: Arquitetura de acesso a arquivos em disco no SODA

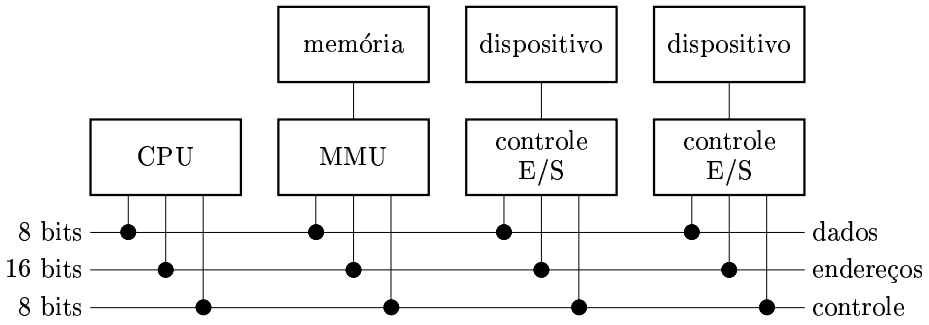

Figura 5.3: Modelo de hardware do SODA

de um nível. Por sua vez, o RCOS.JAVA usa uma versão modificada do processador hipotético *P-Machine*, ao qual foram adicionadas estruturas e operações para suportar uma memória virtual paginada. A *P-Machine* foi desenvolvida nos anos 70 para executar o código intermediário gerado pelos compiladores Pascal UCSD[2] sobre diversas plataformas, em uma abordagem similar à máquina virtual e bytecodes *Java* dos dias de hoje[KMH78].

O autor acredita que a melhor abordagem seja adotar um processador bastante simples, como a *P-Machine* ou algo similar, e empregar também uma MMU (*Memory Management Unit*) separada do processador para a implementação da memória virtual. Essa estrutura torna mais simples o uso do processador e permite maior flexibilidade no caso de alteração do modelo de memória. A figura 5.3 representa a arquitetura proposta.

Todo acesso à memória passa pela MMU, que efetua as conversões de endereço necessárias. A MMU propriamente dita deve ser inicializada e acessada da mesma forma que um dispositivo de E/S, ou seja, através de portas de E/S e interrupções. O acesso à MMU é necessário para informar a tabela de páginas corrente, entre outras operações. Os erros de acesso à memória são informados à CPU através de interrupções. Com essa estrutura, é possível prover os alunos com várias implementações distintas de MMU e solicitar que implementem rotinas de gerência de memória sobre elas. Também é possível solicitar que alterem a MMU para incorporar características avançadas, como acesso direto à memória (DMA) para operações de E/S.

[2] *University of California at San Diego.*

5.5 A arquitetura do software

Uma discussão bastante extensa pode ser tecida a respeito das técnicas de programação a empregar na construção do núcleo SODA. As abordagens usuais diferem bastante: o NACHOS foi desenvolvido em um subconjunto de *C++*, objetivando simplicidade do código. Há uma versão *Java* do NACHOS sendo desenvolvida, mas seus autores apontam uma série de dificuldades para torná-la plenamente operacional [HC00]. O MINIX foi desenvolvido em sua maior parte em *C*, tendo como justificativa o fato de operar diretamente sobre o hardware. Como seu próprio nome diz, o RCOS.JAVA foi inteiramente desenvolvido em Java, com uso extensivo de técnicas avançadas de programação orientada a objetos, como *design patterns*.

O uso de orientação a objetos em sistemas operacionais vem sendo debatido há algum tempo na academia [KTW92, KL93, TRC95], mas é fato concreto que a quase totalidade dos sistemas operacionais de mercado atuais são desenvolvidos em *C*, pois essa linguagem permite pleno acesso aos recursos de hardware. Assim, a definição concreta da linguagem de implementação não é uma tarefa simples, pois o quesito de realismo impõe *C*, enquanto os quesitos de boa qualidade de programação, clareza e portabilidade do código impõe uma linguagem orientada a objetos como *Java*. Qual a resposta ?

Independentemente da linguagem empregada, deve ser preservado o desacoplamento entre o sistema propriamente dito e o processo de visualização, para permitir um bom desempenho na execução de aplicações. As janelas de observação interna do sistema podem ser desativadas para permitir uma operação "full-speed" e com isso executar processos de usuário de forma adequada a poder extrair estatísticas ou poder usar o software desenvolvido sobre o núcleo de forma efetiva. Em [Sef95, JN01] são apresentadas algumas abordagens para a visualização das estruturas e mecanismos internos de forma desacoplada do funcionamento do núcleo propriamente dito.

5.6 Conclusão

Este capítulo delineia as linhas gerais de um novo sistema operacional didático denominado SODA (*Sistema Operacional Didático Aberto*), que busca solucionar as deficiências dos sistemas operacionais didáticos apresentados no capítulo 4.

A proposta do sistema está ainda bastante incompleta, e exigirá uma quantidade significativa de trabalho adicional para torná-la completa e coerente. Certamente será necessária a implementação de protótipos parciais, para subsidiar a tomada de algumas decisões importantes, como a linguagem de programação e o modelo da CPU e dos demais dispositivos de hardware, e validar a proposta inicial de arquitetura para o núcleo.

Capítulo 6

Conclusão

Is Nachos too dry or Minix too hard ? Get yourself some Soda !

Carlos Maziero, 2001

Este trabalho teve como objetivo lançar um olhar crítico sobre a prática de laboratório no ensino de sistemas operacionais. Muitas das reflexões aqui apresentadas são fruto da experiência do autor na área, e algumas outras provêm de trabalhos correlatos [WB92, Che94, Sla00, Ani00]. Esta discussão resulta do fato da área de sistemas operacionais normalmente representar um desafio ao aprendizado, devido a suas exigências tanto em programação quanto em compreensão dos mecanismos de baixo nível e das abstrações envolvidas [PD95].

Ao longo do texto, houve a preocupação de se definir um contexto pedagógico claro para a análise das práticas de laboratório e para a proposta de ambiente computacional que decorre dessa análise. A opção do autor pela abordagem construtivista justifica-se por ser esta uma abordagem consagrada no ensino das ciências, notadamente no campo da engenharia [Den89, DS95, BA98].

A definição clara do sistema operacional SODA está ainda incompleta, e irá necessitar muito esforço para sua finalização. O autor acredita que somente após um grande esforço de prototipagem a arquitetura do SODA está bem delineada, sobretudo seus pontos mais obscuros, como a implementação do escalonador de tempo e do mecanismo de *traps* para as chamadas de sistema. Além disso, é necessário definir claramente a abordagem construtivista empregada na construção do sistema SODA e, sobretudo, em seu uso. Para tanto, deverão ser definidos claramente os

projetos a serem desenvolvidos pelos alunos, a exemplo do que foi proposto para o sistema NACHOS [ACP93].

Na opinião do autor, as principais contribuições deste trabalho são:

- A análise crítica das práticas de laboratório em sistemas operacionais;
- A análise crítica de alguns sistemas operacionais didáticos em uso atualmente;
- A proposta de um novo sistema operacional didático, com base na análise dos sistemas anteriores, buscando aproveitar suas qualidades e sanar algumas de suas deficiências.

Como continuidade deste trabalho, espera-se o aprofundamento da especificação do sistema, incluindo a definição clara dos componentes e interfaces do hardware virtual. Na seqüência, deverão ser desenvolvidos os *drivers* de dispositivo até o nível de entidade abstrata como proposto no sistema NACHOS. A partir desse ponto será então possível definir claramente as demais estruturas internas do sistema operacional e partir para a implementação do núcleo propriamente dito.

Referências Bibliográficas

[ACP93] T. Anderson, W. Christopher, and S. Procter. The Nachos instructional operating system. In *Proceedings of the Winter Usenix Technical Conference*, pages 481–489, 1993.

[Ani00] Ricardo Anido. Uma proposta de plano pedagógico para a matéria Sistemas Operacionais. In *Anais do II Curso de Qualidade, Workshop sobre Educação em Computação, XX Congresso da Sociedade Brasileira de Computação*, pages 125–148, Curitiba PR, Julho 2000.

[BA98] M. Ben-Ari. Constructivism in computer science education. *ACM SIGCSE Bulletin*, 30(2):257–261, 1998.

[Bru66] J. Bruner. *Toward a Theory of Instruction.* Harvard University Press, 1966.

[Bru90] J. Bruner. *Acts of Meaning.* Harvard University Press, 1990.

[BS82] R.W. Bybee and R.B. Sund. *Piaget for Educators, 2^{nd} edition.* Columbus OH, 1982.

[Che94] R. Chernich. The design and construction of a simulated operating system. In *Proceedings of the Asia-Pacific Information Technology in Teaching and Eduaction Conference*, pages 1033–1038, Brisbane, Australia, 1994.

[CJJ96] Ron Chernich, Bruce Jamieson, and David Jones. RCOS: Yet another teaching operating system. In *Proceedings of the 1^{st} Australian Conference on Computer Science Education*, Sidney – Australia, July 1996.

[Com84] D. Comer. *Operating System Design - The XINU Approach.* Prentice-Hall, 1984.

[Con90] J. Confrey. What constructivism implies for teaching. In *Constructivist Views on the Teaching and Learning of Mathematics – National Council of Teachers of Mathematics*, pages 107–122, Reston Virginia, 1990.

[Den89] P. Denning. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[DS95] T. M. Duffy and J. R. Savery. Problem based learning: An instructional model and its constructivist framework. *Educational Technology*, 35(5):31–38, 1995.

[Fre01] FreeBSD Community. The FREEBSD project. `http://www.freebsd.org`, 2001.

[GK95] Sven Graupner and Winfried Kalfa. Adaptable infrastructures for operating systems. In *Proceedings of the 4th IEEE International Workshop on Object-Orientation in Operating Systems*, pages 162–165, Lund Sweden, 1995.

[HC00] Dan Hettena and Rick Cox. *A Guide to Nachos 5.0j.* Dept of Computer Sciences, University of California at Berkeley, 2000.

[HMOS90] J.H. Hayes, L.R. Miller, B.A. Othmer, and M. Saeed. Simulation of process and resource management in a multiprogramming operating system. In *Proceedings of the 21st ACM Technical Symposium on Computer Science Education*, page 125, 1990.

[JN01] David Jones and Andrew Newman. RCOS.Java: a simulated operating system with animations. In *Proceedings of the Computer-Based Learning in Science Conference*, Brno – Rep. Tcheca, July 2001.

[KA99] Heather Kanuka and Terry Anderson. Using constructivism in technology-mediated learning: Constructing order out of the chaos in the literature. *Radical Pedagogy*, 1(2), 1999.

[Kar01] Vijay Karamcheti. *Nachos Project Guide*. Dept of Computer Sciences, New York University, 2001.

[Kes00] Gregory Kesden. The Yalnix kernel project. `http://www-2.cs.cmu.edu/~412/projects/proj3/proj3.pdf`, 2000.

[KL93] Gregor Kickzales and John Lamping. Operating systems: why object-oriented. In *Proceedings of the 3rd IEEE Intl Workshop on Object Orientation in Operating Systems*, pages 25–30, Asheville - NC - USA, 1993.

[KMH78] Ghung Kin-Man and Yuen H. A tiny pascal compiler. *Byte Magazine*, 3(9-11), 1978.

[KS92] M. Kifer and S. Smolka. OSP – an environment for operating systems projects. *Operating Systems Review*, 26(4):98–99, 1992.

[KTW92] Gregor Kickzales, Marvin Theimer, and Brent Welch. A new model of abstraction for operating system design. In *Proceedings of the 2nd IEEE Intl Workshop on Object Orientation in Operating Systems*, pages 346–349, Dourdan - France, 1992.

[Lin01] Linux community. The Linux home page. `http://www.linux.org`, 2001.

[Mil99] Dejan Milojicic. Operating systems – now and in the future. *IEEE Concurrency*, pages 12–21, 1999.

[Nar95] Thomas Narten. *A Road Map Through Nachos*. Dept of Computer Sciences, Duke State University, January 1995.

[Nul98] U. Nulden. The ExCon project: Advocating continuous examination. *ACM SIGCSE Bulletin*, 30(1):126–130, 1998.

[PCJ96] J. Pane, A. Corbet, and B. John. Assessing dynamics in computer-based instruction. In *CHI'96 Conference Proceedings, ACM Press*, New York, 1996.

[PD95] A. Perez-Davilla. OS – bridge between academia and reality. *ACM SIGCSE Bulletin*, 27(1):146–148, 1995.

[Pia32] J. Piaget. *The Moral Judgement of the Child*. NY Harcourt, 1932.

[RBF+89] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi. Mach: a foundation for open systems. In *Proceedings of the 2nd IEEE Workshop on Workstation Operating Systems*, pages 109–113, Pacific Grove - CA - USA, 1989.

[Rus99] David A. Rusling. *The Linux Kernel.* LinuxDoc, 1999.

[Sef95] Mohlalefi Sefika. An open visual model for object-oriented operating systems. In *Proceedings of the 4th IEEE International Workshop on Object-Orientation in Operating Systems*, pages 103–112, Lund - Sweden, 1995.

[SG97] A. Silberschatz and P. Galvin. *Operating Systems Concepts, 4th edition.* Addison-Wesley, 1997.

[SGG01] A. Silberschatz, P. Galvin, and G. Gane. *Sistemas Operacionais - Conceitos e Aplicações.* Editora Campus, 2001.

[Sla00] SlashDot, Inc. Custom kernels used in computer science programs ? `http://slashdot.org/askslashdot/00/12/08/2339254.shtml`, december 2000.

[Sta97] J.T. Stasko. Using student built algorithm animations as learning aids. In *Proceedings of the 28th Technical Symposium on Computer Science Education*, pages 25–29, 1997.

[Tan01] Andrew Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 2001.

[Tho01] Thomson Learning, Inc. A simple CPU simulator applet. `http://www.brookscole.com/compsci/aeonline/course/7/5/index.html`, 2001.

[TRC95] See-Mong Tan, David Raila, and Roy Campbell. An object-oriented nano-kernel for operating system hardware support. In *Proceedings of the 4th IEEE Intl Workshop on Object Orientation in Operating Systems*, pages 220–223, Lund - Sweden, 1995.

[TW99] A. Tanenbaum and A. Woodhull. *Sistemas Operacionais: Projeto e Implementação, 2a edição.* Editora Bookman, 1999.

[VH96] Alistair Veitch and Norman Hutchinson. The Kea system: reconciling micro and monolithic kernel design. In *Proceedings of the 3rd Intl Conference on Configurable Distributed Systems*, pages 236–242, Annapolis - MD - USA, may 1996.

[WB92] J.M. Withers and M.B. Bilodeau. An examination of operating systems laboratory techniques. *ACM SIGCSE Bulletin*, 24(3):60–64, 1992.

[Wei99] Telma Weisz. *O Diálogo entre o Ensino e a Aprendizagem.* Editora Ática, 1999.

[Win01] Joakim Winkler. A deadlock simulator applet. `http://www.pt.hk-r.se/~pt95jwi/javastuffs/noholdandwait/deadlock.html`, 2001.