

PARTE 3 - SISTEMAS OPERACIONAIS DISTRIBUÍDOS

- 9. Introdução
- 10. O kernel
- 11. Processos e Threads
 - 11.1 Espaços de endereçamento
 - 11.2 Criação de processos
 - 11.3 Threads
- 12. Nomes e Proteção
 - 12.1 Proteção de recursos
- 13. Comunicação
- 14. Memória
- 15. Estudo de Casos
 - 15.1 Amoeba
 - 15.2 Mach

PARTE 3 - SISTEMAS OPERACIONAIS DISTRIBUÍDOS

9. INTRODUÇÃO

O conceito de um sistema operacional distribuído foi introduzido anteriormente e o mesmo pode ser caracterizado da seguinte forma:

- Sua tarefa é permitir que um sistema distribuído seja convenientemente programado, de forma que o mesmo possa ser utilizado por um amplo conjunto de aplicações.
- Isto é conseguido fornecendo para as aplicações abstrações gerais dos recursos do sistema, como por exemplo, canais de comunicação e processos, ao invés de redes e processadores.
- Em um sistema distribuído aberto, o sistema operacional distribuído é implementado por uma coleção de kernels e servidores. Não existe uma linha divisória clara entre o SOD (seus serviços abertos) e as aplicações que rodam em cima dele.

Muitos esforços tem sido feitos para projetar e implementar sistemas operacionais distribuídos. Sistemas como Mach e Chorus são exemplos de SOD que atingiram um status significativo de interesse técnico e comercial. Outros sistemas (Amoeba, Clouds) são mais aceitos no meio acadêmico sem muita aceitação para uso geral. Um ponto em comum destes projetos é a utilização de um kernel mínimo, microkernel, como base do projeto. Aqui vamos focalizar uma parte do sistema operacional distribuído que age como infraestrutura para um gerenciamento de recursos geral e transparente a rede. A infra-estrutura gerencia recursos de baixo nível (processador, memória, interface de rede e outros dispositivos periféricos) fornecendo uma plataforma para a construção de recursos de alto nível. Um sistema operacional distribuído fornece transparência no acesso aos recursos. Uma coleção de kernels UNIX não constituem um SOD porque as fronteiras entre os computadores são claramente visíveis.

Um SOD deve fornecer facilidades para o encapsulamento de recursos de maneira modular e protegida ao mesmo tempo que fornece aos clientes de toda rede acesso a estes mesmos recursos. Tanto os kernels como os servidores são gerentes de recursos e, portanto, são requisitos deles:

- *Encapsulamento*: Eles deveriam fornecer uma interface de serviços útil para os seus recursos, ou seja, um conjunto de operações que atendam as necessidades dos seus clientes. Os detalhes da gerência de memória e de dispositivos usada para implementar os recursos deveriam ser escondidas dos clientes, mesmo quando locais.
- *Processamento concorrente*: Clientes podem compartilhar recursos e acessar os mesmos de forma concorrente. Gerentes de recursos são responsáveis pela transparência de concorrência.
- *Proteção*: Recursos necessitam proteção em função de acessos ilegais. Por exemplo, arquivos são protegidos contra leitura de usuários que não tem permissão de leitura e registradores de dispositivos são protegidos de processos de aplicação.

Clientes acessam recursos identificando-os nos argumentos das operações, em chamadas remotas de procedimento para um servidor ou em chamadas de sistema para o kernel. Os acessos a um recurso encapsulado são chamados de **invocação**, independente de sua

implementação. Uma combinação de bibliotecas de cliente, kernels e servidores podem ser chamadas para a realização das seguintes tarefas relacionadas a invocação:

Resolução de nomes: O servidor (ou kernel) que gerência um recurso tem que ser localizado a partir do identificador do recurso.

Comunicação: Parâmetros de operação e resultados tem que ser passados para/de os gerentes de recursos, através da rede ou no computador.

Escalonamento: Quando uma operação é invocada, seu processamento deve ser escalonado no kernel ou servidor.

10. O KERNEL

O kernel é um programa cujo código é executado com total privilégios de acesso aos recursos físicos do computador hospedeiro. Ele tem capacidade de controlar a unidade de gerência de memória e manipular os registradores do processador de forma a restringir o acesso aos recursos da máquina por outro código. Algumas vezes ele pode permitir que servidores acessem recursos físicos. O kernel pode manipular **espaços de endereçamento** para proteger processos e fornecer para eles o necessário layout de memória virtual. Um processo kernel executa com o processador em modo *supervisor*, enquanto que faz com que os outros processos executem em modo *usuário*. O mecanismo de invocação de recursos gerenciados pelo kernel é conhecido como chamada de sistema, que é implementada por uma instrução de máquina, TRAP, que coloca o processador em modo supervisor e muda para o espaço de endereçamento do kernel.

Quanto as abordagens de projeto do kernel existem 2 que são consideradas mais importantes: *kernel monolítico* e *microkernel*. A diferença básica diz respeito a decisão de quais funcionalidades pertencem ao kernel e quais devem ser deixadas para processos servidores (Figura 10.1).

O kernel do sistema UNIX é um exemplo de kernel monolítico. O termo sugere que o sistema realiza todas as funções básicas de um SO e tem um tamanho significativo e que é escrito de forma não modular. O resultado é um sistema difícil de alterar qualquer componente individual de software em função da mudança de requisitos. Um kernel monolítico pode conter alguns processos servidores (arquivo, rede) cujo código é parte da configuração padrão do kernel.

No caso da abordagem microkernel, o kernel fornece apenas as abstrações mais básicas (processos, memória, comunicação entre processos) e todos os outros serviços são fornecidos por servidores que são dinamicamente carregados onde os serviços são fornecidos. Estes serviços são acessados por mecanismos de invocação baseados em mensagens, principalmente RPC. Um microkernel, normalmente, ocupa um nível entre o nível de hardware e o nível que contém os principais componentes do sistema.

As vantagens de um SOD baseado em microkernel são principalmente a sua propriedade de sistema aberto e sua habilidade forçar modularidade como consequência dos limites de proteção de memória. Além disto, um kernel relativamente pequeno tem mais chance de estar livre de erros do um grande e complexo. A falta de estrutura em projetos monolíticos pode ser evitada através do uso de técnicas de engenharia de software como por exemplo: níveis e orientação a objetos. Mesmo assim, pode ser difícil mante-lo e o suporte fornecido para sistemas distribuídos abertos é limitada. A re-implementação de um módulo em um kernel monolítico implica na reconstrução do kernel e o seu posterior reinício. A vantagem maior dos projetos monolíticos é a eficiência na invocação das operações.

Apesar de não existir uma definição fixa de quais funções devem ser realizadas pelo microkernel, quase todas as propostas de sistemas baseados em microkernel incluem: gerenciamento de processos, gerenciamento de memória e mecanismo de troca de mensagens, sendo que o tamanho dos mesmos pode variar desde 10Kbytes até centenas de Kbytes de código e dados. A Figura 10.2 mostra um exemplo de arquitetura de um microkernel, seus componentes e suas dependências.

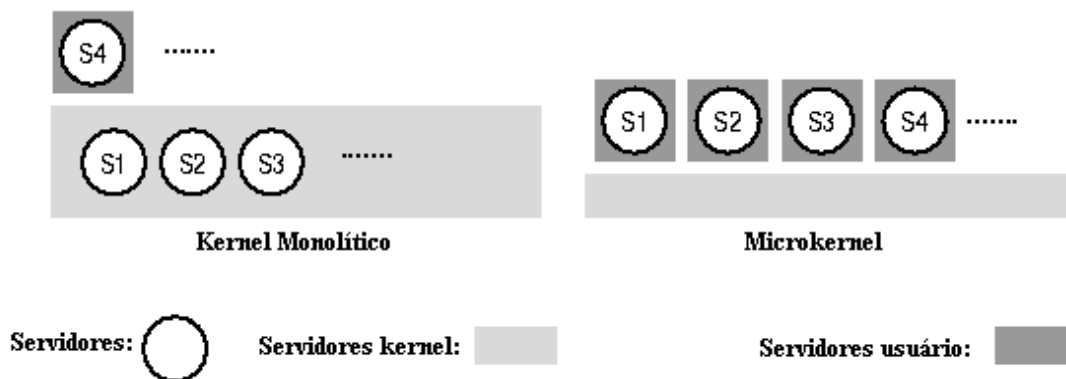


Figura 10.1 Kernel monolítico e microkernel

Gerente de Processos: Trata da criação e de operações de baixo nível em processos. As facilidades fornecidas por este módulo para as aplicações, poderiam ser incrementadas por um subsistema para suporte de linguagem ou emulação de sistemas operacionais.

Gerente de Thread: Trata da criação, sincronização e escalonamento de threads, que são atividades escalonáveis associadas aos processos.

Gerente de Comunicações: Trata da comunicação entre threads associadas a diferentes processos. Comunicação remota também pode ser incluída.

Gerente de Memória: Trata do gerenciamento dos recursos físicos de memória, MMU, caches.

Supervisor: Trata do atendimento de interrupções, chamadas de sistema via TRAP e outras exceções.

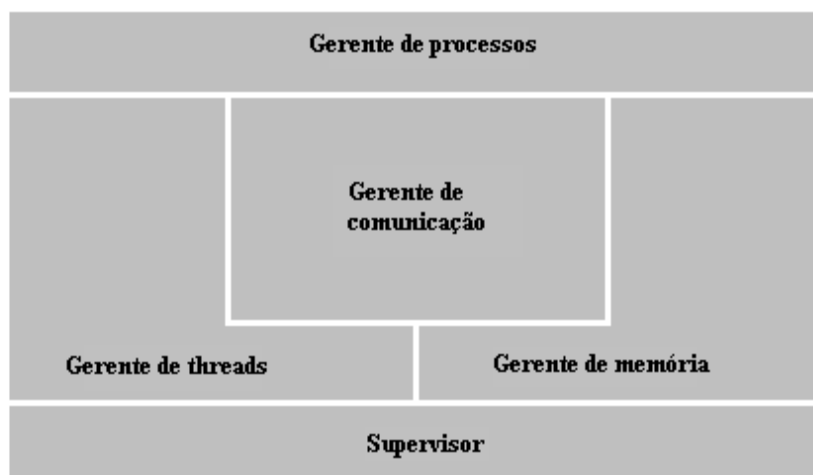


Figura 10.2 Arquitetura do microkernel

11. PROCESSOS E *THREADS*

A noção tradicional de processo, estilo UNIX, é considerada (1980) não apropriada para os requisitos de sistemas distribuídos e também para aquelas aplicações que requerem concorrência interna. O problema reside no fato de que processos UNIX associam apenas uma atividade de processamento com cada processo, mesmo sendo o processo um recurso de criação e gerenciamento caro. Isto torna o compartilhamento, entre atividades relacionadas, difícil e caro.

A solução encontrada foi a generalização da noção de processo de forma a associar um processo a várias atividades. Desta forma, um processo consiste de um ambiente de execução junto com uma ou mais *threads*. Uma **thread** é a abstração do sistema operacional que representa uma atividade. Um **ambiente de execução** é a unidade de gerenciamento de recurso: uma coleção de recursos locais gerenciados pelo kernel aos quais as *threads* tem acesso. Um ambiente de execução é composto por:

- Um espaço de endereços
- Recursos de sincronização e comunicação de *threads*

A criação e gerenciamento de ambientes de execução é normalmente cara, entretanto várias *threads* podem compartilhar (compartilhar todos os recursos pertencentes ao ambiente). A existência de múltiplos fluxos de execução (*threads* criadas/destruídas dinamicamente) permite a maximização do grau de concorrência entre operações relacionadas (Ex. servidores: o processamento concorrente das requisições pode reduzir a tendência dos servidores se tornarem gargalos). Um ambiente de execução fornece proteção, de forma que seus dados e outros recursos não sejam acessados por *threads* pertencentes a outros ambientes de execução. Entretanto, é possível permitir o compartilhamento controlado de recursos (memória) entre ambientes em um mesmo computador.

11.1 Espaços de endereçamento

É o componente mais caro de um ambiente de execução, podendo ser grande (2^{32} bytes) e composto por uma ou mais **regiões**, separadas por áreas não acessíveis de memória virtual. Uma região é uma área de memória virtual contígua que é acessível pelas *threads* do processo. As seguintes propriedades especificam regiões:

- Sua extensão (endereço inicial e tamanho)
- Permissões de acesso (read/write/execute)
- Quando pode crescer (crescente/decrescente)

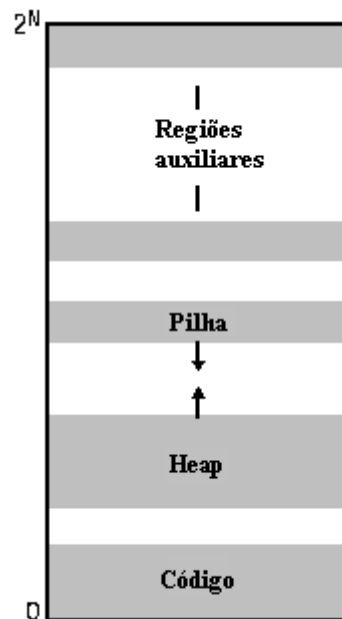


Figura 11.1 Espaço de endereçamento

Vários fatores motivam a existência de um numero indefinido de regiões. Um deles é permitir que arquivos em geral (não apenas seções de código e dados de arquivos executáveis) sejam mapeados em espaços de endereçamento.

Outro fator é o compartilhamento entre processos ou entre processos e kernel em uma mesma máquina. Uma *região de memória compartilhada* é vista por vários processos ao mesmo tempo podendo ser acessada por eles, alguns usos são:

Bibliotecas: O código de bibliotecas ao invés de ser carregado individualmente por cada processo pode ser compartilhado a partir do mapeamento deste como uma região no espaço de endereçamento dos processos que utilizam.

Kernel: Frequentemente código e dados do kernel são mapeados para todos os espaços de endereçamento na mesma localização. Quando um processo executa uma chamada de sistema ou uma exceção ocorre não existe necessidade de inicializar um novo espaço de endereçamento.

Compartilhamento de Dados e Comunicação: Dois processos ou um processo e o kernel podem necessitar compartilhar dados para cooperar em alguma tarefa. Pode ser mais eficiente os dados serem compartilhados através do mapeamento como regiões pertencentes aos dois espaços de endereços do que através de troca de mensagens.

Um espaço de endereçamento configurado como um conjunto de regiões permite alocar uma região de pilha para cada *thread* do processo e com isto é possível detectar tentativas de acesso fora dos limites da pilha assim como controlar o crescimento das mesmas. Outra

alternativa é alocar as pilhas no *heap*, e neste caso é difícil detectar o acesso fora dos limites.

11.2 Criação de processos

A criação de um processo é uma operação atômica fornecida pelo sistema operacional. Por exemplo, a chamada de sistema *fork* (UNIX) cria um processo cujo ambiente de execução é copiado do processo chamador. A chamada *exec* transforma o processo executante de um código constante de um arquivo executável. Para um sistema operacional distribuído, a criação de processos deve levar em conta dois novos requisitos. O primeiro é a utilização de vários computadores, o segundo é a divisão da infraestrutura de suporte do processo em serviços do sistema separados. A criação de um processo pode ser separada em 3 aspectos independentes:

- A escolha do computador,
- A criação de um ambiente de execução, e
- A criação de uma thread no mesmo.

Escolha do computador. A escolha do computador no qual o processo irá residir resume-se a uma questão de política. Por exemplo, o sistema V fornece um comando para os usuários executar um programa em um computador em particular ou em um ocioso escolhido pelo SO. No Amoeba, o servidor de execução escolhe um computador, do grupo de processadores, para cada processo e esta é transparente tanto para o programador quanto para o usuário.

Criação de um ambiente de execução. Um processo necessita um ambiente de execução composto de um espaço de endereçamento com conteúdo inicializado e um conjunto de portas para comunicar com os serviços. O kernel cria o novo ambiente de execução no computador local. Se necessário, um serviço do sistema pode requisitar a um processo servidor remoto a criação de um ambiente em um computador remoto. Existem duas alternativas para a definição e inicialização de um ambiente de execução para um processo. A primeira abordagem é usada quando o espaço de endereçamento é definido de forma estática, ou seja, o mesmo é composto somente por uma região de código, heap e pilha. Neste caso, as regiões do espaço de endereçamento são criadas a partir da definição do seu tamanho e são inicializadas de arquivos executáveis. Na segunda abordagem o espaço de endereçamento pode ser definido a partir de um ambiente de execução existente. No caso do UNIX os processos criados compartilham as regiões do pai (código, heap, pilha), que são copiadas para o processo filho. Alguns sistemas usam um sistema conhecido como **copy-on-write** para a cópia das regiões. Outros recursos como: arquivos, portas de comunicação, etc., também podem ser compartilhados pelo pai e filho.

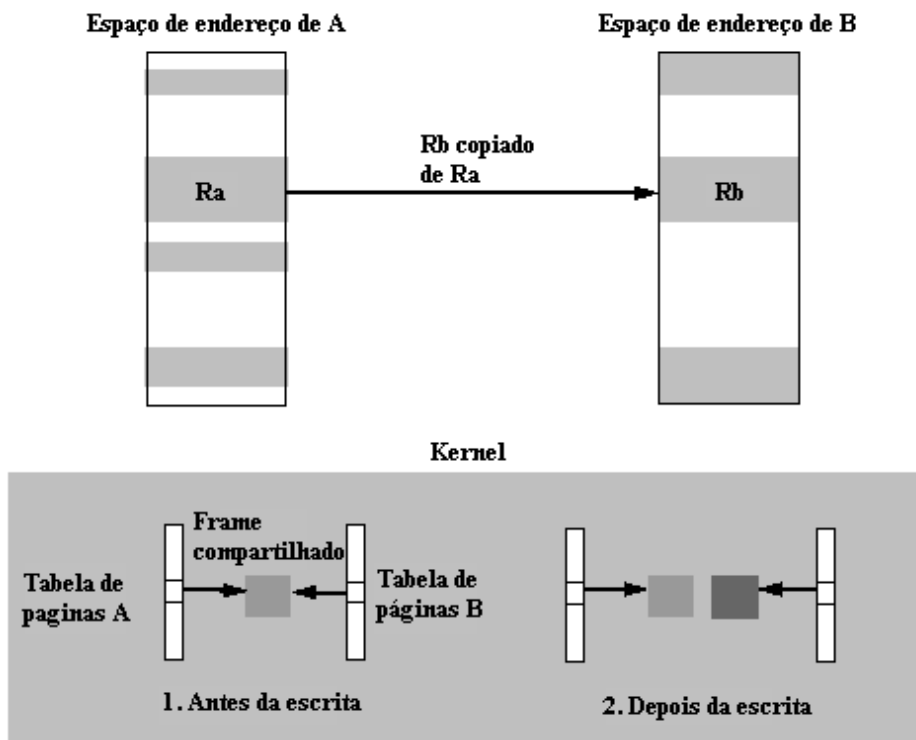


Figura 11.2 Técnica copy-on-write

11.3 Threads

Considere o exemplo de um servidor (Figura 11.3) onde existem uma ou mais threads sendo que cada uma recebe mensagens de requisição e as processa. Se cada requisição leva em média 2 ms para ser processada e 8ms de E/S e o servidor executa em um computador com um único processador sendo que uma única thread realiza todo o processamento, o desempenho do servidor será de 100 requisições por segundo. Se o servidor tem 2 threads e as mesmas são escalonadas de forma independente, ou seja, uma thread pode ser escalonada quando a outra fica bloqueada em E/S. Isto aumenta o desempenho do servidor. No exemplo, como as operações de E/S não podem ser realizadas em paralelo a taxa na qual as requisições podem ser atendidas está limitada a 125 requisições por segundo. Se utilizarmos o conceito de cache com taxa de acerto de 75%, o tempo de E/S é reduzido para 2ms, mas o tempo de processamento é aumentado para 2,5ms. Neste caso, poderíamos ter um desempenho de até 400 requisições por segundo.

Se considerarmos um computador com 2 processadores as threads poderiam executar em paralelo e neste caso o limite seria novamente o tempo de E/S.

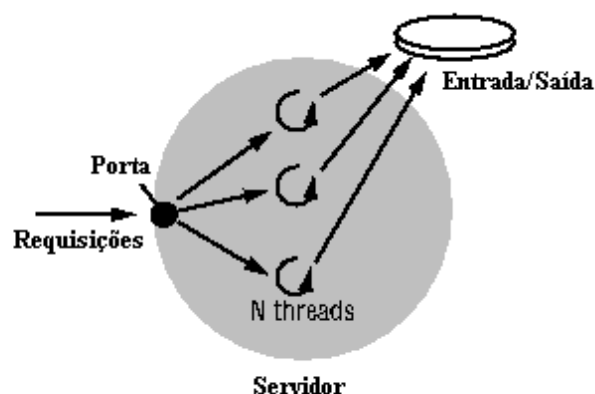


Figura 11.3 Servidor com treads

As threads podem ser úteis também para os processos clientes, como mostra o exemplo a seguir (Figura 11.4). O processo cliente tem 2 threads, sendo que a primeira gera resultados para serem passados para um servidor através de RPC, sem resposta, o que não é possível em RPC convencional. A segunda thread, realiza as chamadas remotas enquanto a primeira pode processar resultados, estes são colocados em buffers que são esvaziados pela segunda.

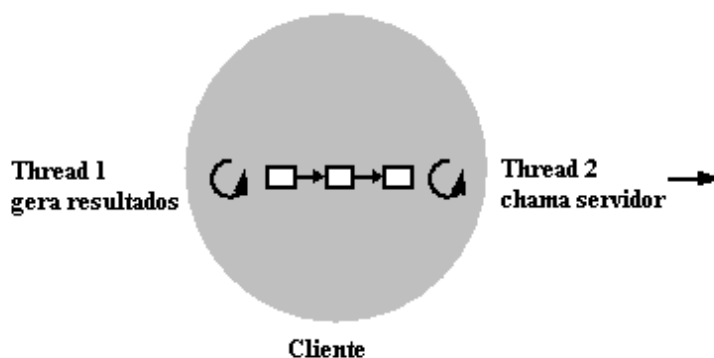


Figura 11.4 Cliente com threads

É possível a partir destes exemplos verificar a utilidade das threads, que permitem uma sobreposição das computações com operações de entrada e saída e no caso de multiprocessadores, com outras computações. Entretanto, isto pode ser conseguido com processos de uma única thread. Neste caso, porque o modelo multi-thread seria preferido? A resposta tem duas faces: threads são mais baratas de criar e gerenciar e o compartilhamento de recursos pode ser conseguido de forma mais eficiente entre threads do que entre processos porque threads compartilham um ambiente de execução. É possível comparar processos e threads como segue:

- Criar uma thread em um processo existente é mais barato do que criar um processo.
- O chaveamento de threads dentro do mesmo processo é mais barato do que o chaveamento entre threads de processos diferentes.
- Threads de um processo podem compartilhar dados e outros recursos de forma conveniente e eficiente se comparado com processos separados.
- Entretanto, threads de um processo não são protegidas umas das outras.

Quando criamos uma thread em um ambiente de execução existente as principais tarefas são : alocar uma região para a pilha, inicializar valores dos registradores do processador , estado de execução e prioridade. Uma vez que o ambiente existe, a thread só precisa conhecer seu identificador. Por outro lado, o overhead associado na criação de um processo é considerado bem maior. Ele envolve a criação de um ambiente, incluindo tabelas para o espaço de endereçamento e portas de comunicação. Uma comparação aponta um tempo de 11 ms para criar um processo UNIX e 1ms para criar uma thread, considerando a mesma arquitetura.

Do ponto de vista de chaveamento, ou seja, a troca de contexto de uma thread para outra, podemos dizer que o chaveamento entre threads que compartilham o mesmo ambiente é bem mais barato do que o chaveamento entre threads que pertencem a processos diferentes. O custo de chaveamento é o mais importante porque ele pode acontecer muitas vezes durante a vida de uma thread. O chaveamento envolve o escalonamento (escolha da próxima thread a executar) e a troca de contexto (transição entre threads ou chamada de sistema). No chaveamento entre threads de um mesmo processo em um kernel que suporta threads diretamente, a troca de contexto só ocorre do/para kernel. No caso do kernel ser mapeado no espaço de endereço da thread o custo é relativamente baixo. No chaveamento de threads de diferentes processos, é necessário também uma troca de contexto para um outro ambiente de execução.

No exemplo do processo cliente com duas threads, uma vez que as threads compartilham um espaço de endereçamento, não é necessário utilizar troca de mensagens para passar os dados o que representa uma vantagem. Entretanto, uma thread com erro pode causar falhas visto que não existe proteção entre as mesmas.

Programação com Threads

Programar com threads implica em programação concorrente e neste caso envolve conceitos como: condições de corrida, seção crítica, variáveis condição e semáforos. A maior dificuldade na programação com threads envolvem o escopo das variáveis e as técnicas usadas para a coordenação e cooperação entre threads. Cada thread tem uma pilha privativa e portanto variáveis locais de um procedimento são sempre privativas da thread que executa o procedimento. Entretanto, não existe área de heap privada ou cópia privada de variáveis estáticas.

Escalonamento de Threads

As abordagens de escalonamento de threads podem ser classificadas em 2 principais: *escalonamento preemptivo*, onde uma thread pode ser suspensa para dar lugar a uma outra

mesmo quando ela esta executando; *escalonamento não preemptivo*, onde uma thread executa até ela executar uma chamada que faz com que ela perca o processador para outra thread. Alguns pacotes permitem as duas alternativas. A vantagem do escalonamento não preemptivo é que qualquer seção de código que não tem uma chamada que pode causar um novo escalonamento é automaticamente uma seção crítica (evitando condições de corrida). Por outro lado, este tipo de thread não pode aproveitar as vantagens de um multiprocessador pelo fato de elas executarem de forma exclusiva. Estas threads também não são adequadas para sistemas de tempo real, nos quais os eventos estão associados com tempos absolutos nos quais eles devem ser processados. É possível associar prioridades as threads de forma que eventos urgentes possam ser atendidos antes que outros.

Implementação de Threads

Alguns kernels convencionais suportam apenas processos com uma única thread, enquanto que processos com múltiplas threads são implementados através de bibliotecas que são ligadas com os programas de aplicação. Um exemplo disto é o pacote de processos leves do SunOS 4.1. Neste caso, o kernel não tem conhecimento dos processos leves (lightweight processes) e não pode escalonar os mesmos separadamente, quem organiza o escalonamento é uma biblioteca *run-time*. Com isto, uma thread (a partir de uma chamada de sistema bloqueante) poderia bloquear o processo fazendo com que todas as outras threads associadas ao processo bloqueiem. Quando não existe suporte do kernel para multiplas threads, os pacotes apresentam os seguintes problemas:

- As threads de um processo não aproveitam a vantagem dos multiprocessadores;
- Threads de processos diferentes não podem ser escalonadas de acordo com um esquema único de prioridades.

Por outro lado, pacotes de threads a nível de usuário apresentam vantagens significativas em relação aqueles com suporte do kernel:

- Algumas operações são tem custo menor. Por exemplo, o chaveamento entre threads de um mesmo processo não implica necessariamente em chamada de sistema;
- Se o modulo de escalonamento de threads é implementado fora do kernel, ele pode ser personalizado para atender os requisitos de uma aplicação em particular;
- É possível suportar muito mais threads a nível de usuário do que pelo kernel.

Desta forma, é possível combinar as vantagens das threads a nível de usuário com as suportadas pelo kernel. Uma abordagem, aplicada no Mach, é permitir código a nível de usuário interferir no escalonamento das threads suportadas pelo kernel.

12. NOMEAÇÃO E PROTEÇÃO

Geralmente um serviço gerência vários recursos, sendo que cada um pode ser acessado independentemente por seus clientes. Para isto, o serviço fornece um identificador para cada um de seus recursos. Os clientes necessitam que a localização de um recurso seja transparente. O cliente acessa recursos através de requisições para o serviço gerenciador fornecendo os identificadores apropriados. Por exemplo, um cliente de um serviço de arquivo fornecerá o identificador do arquivo a ler ou escrever. As requisições são direcionadas para um identificador de comunicação que é obtido do serviço de ligação (cliente – servidor). Alguns sistemas utilizam como identificador de comunicação uma porta ou grupo de portas e neste caso a identificação do recurso requer conhecimento da porta (grupo de) o identificador do serviço específico.

O uso de identificadores de comunicação independentes de localização fornece transparência. O serviço de comunicação implementado pelo kernel (em conjunto com servidores de rede) é responsável pela procura do endereço físico correspondente ao servidor e pela determinação da rota da mensagem de requisição do cliente.

O formato dos identificadores usados por um serviço podem ser livremente escolhidos pelo implementador do serviço. Identificadores deveriam referenciar de forma não ambígua um recurso individual, seja no contexto do sistema distribuído ou pelo menos no do serviço gerenciador.

Como os identificadores deveriam ser gerados? Sempre que possível, kernels e servidores deveriam gerar identificadores para novos recursos sem precisar consultar seus pares. Uma forma conveniente de gerar identificadores que são garantidos únicos no sistema distribuído é pré-alocar um conjunto grande de identificadores para cada kernel ou servidor no sistema. Uma forma efetiva desta idéia é cada identificador começar com o identificador do site do kernel ou do servidor do serviço mas sem qualquer necessidade do recurso permanecer naquela localização física.

12.1 Proteção de recursos

O objetivo de um esquema de proteção é assegurar que cada processo pode acessar somente aqueles recursos para os quais ele tem permissão. A permissão especifica que operações o processo pode realizar no recurso. Assim como outros aspectos da gerência de recursos em sistemas distribuídos a proteção é muito específico do serviço. Tanto kernels como serviços de mais alto nível implementam seus próprios mecanismos de proteção. No caso do kernel é possível aplicar as facilidades do hardware como MMU para implementar proteção para si e os recursos gerenciados assim como proteção de memória para os processos mantidos. No caso dos servidores, eles podem receber mensagens de qualquer lugar do sistema e neste caso é preciso utilizar técnicas de software para proteger seus recursos contra acessos indevidos.

Domínios de Proteção. Um domínio de proteção é um ambiente de proteção compartilhado por uma coleção de processos: é um conjunto de pares <recurso,direitos>, listados os recursos que podem ser acessados por todos os processos executando dentro do domínio e especificadas as operações permitidas para cada recurso. Por exemplo, no UNIX o domínio de proteção de um processo é determinado pelos identificadores associados ao usuário e

grupo de usuários. Um domínio de proteção é uma abstração. Em sistemas operacionais as implementações usadas são baseadas em capacidades e listas de controle de acesso.

- *Capacidades*: um conjunto de capacidades é mantida para cada processo de acordo com o domínio ao qual ele pertence. Capacidades são identificadores que contém bits aleatórios tornando difícil de forjar. Serviços só fornecem capacidades para os clientes quando os mesmos são identificados como pertencentes ao domínio reclamado. Capacidades diferentes são utilizadas para diferentes combinações de direitos de acesso para um mesmo recurso.
- *Listas de controle de acesso*: uma lista é armazenada com cada recurso, indicando os domínios que tem acesso ao recurso e as operações permitidas em cada domínio.

Servidores podem ser projetados para usarem capacidades ou listas de controle de acesso:

- As requisições de clientes incluem uma capacidade para os recursos a serem acessados, servindo como prova de que o cliente está autorizado a acessar o recurso identificado pela capacidade com as operações especificadas na capacidade.
- As requisições de clientes incluem um identificador para o recurso a ser acessado. O servidor identifica o cliente e verifica a lista de controle de acesso em cada requisição. A identificação é conseguida através de comunicação entre o cliente, o servidor e um servidor de identificação.

13. COMUNICAÇÃO E INVOCAÇÃO

A comunicação não é um fim, mas normalmente parte da implementação do que denominamos uma invocação, uma construção tal como RPC, cujo propósito é permitir o processamento de dados em um escopo ou ambiente de execução diferente. As necessidades das aplicações em termos de comunicação incluem a abordagem produtor-consumidor, cliente-servidor e comunicação em grupo. Elas variam de acordo com a *qualidade de serviço* requerida, ou seja, garantia de entrega, tamanho de banda e demora, e segurança fornecida pelo serviço de comunicação. Por exemplo, dados de vídeo e voz devem ser transmitidos com baixas demoras enquanto que transferência de arquivos não. Algumas aplicações necessitam manutenção secreta dos dados, apesar dos mesmos passarem através de uma rede física insegura. As seguintes questões podem ser levantadas com respeito ao fornecimento de comunicação em um sistema operacional distribuído:

- Que primitivas básicas de comunicação são fornecidas?
- Que garantias de qualidade de serviço são feitas?
- Que protocolos são suportados?
- Quanto aberta é a implementação da comunicação?
- Que passos são tomados para fazer a comunicação tão eficiente quanto possível?

Protocolos. Podemos diferenciar alguns kernels em relação ao suporte as redes de comunicação. Por exemplo, Amoeba, V e Sprite, incorporam seus próprios protocolos sintonizados para interações do tipo RPC. Entretanto, estes protocolos não são muito utilizados a não ser nos ambientes de pesquisa onde estes SOD's foram projetados.

Protocolos tais como: TCP, UDP e IP, por outro lado, são muito utilizados tanto em LAN's como em WAN's mas não suportam diretamente interações do tipo RPC. No caso do Mach e Chorus, os projetistas, ao invés de projetar um protocolo próprio ou utilizar um protocolo em particular, fornecem suporte apenas para as comunicações locais e deixam o processamento de protocolos de rede para servidores que executam em um nível acima do kernel.

Primitivas de Comunicação. Sistemas operacionais distribuídos, através de seu kernel, fornece o mecanismo de passagem de mensagem em duas formas básicas: troca de mensagens e chamada remota de procedimentos. No primeiro caso, o mecanismo está associado a existência das primitivas básicas Send e Receive, enquanto que a abordagem RPC é similar ao mecanismo básico de chamada de rotina. É possível a partir da existência das primitivas básicas Send/Receive a implementação do modelo RPC. Além destes modelos, alguns sistemas fornecem a comunicação em grupo (Amoeba, V, Chorus) onde é possível um processo enviar mensagem para um grupo de processos.

Qualidade de Serviço. Mesmo que as primitivas básicas de comunicação do kernel não sejam confiáveis é possível a construção de uma versão confiável de Send ou de RPC. É possível implementar comunicação orientada a *stream* com bufferização, por exemplo. Segurança pode ser fornecida através de um servidor de rede ou a partir de criptografia. O problema maior talvez seja conseguir bandas passantes e demoras satisfatórias. Em dados multimídia, por exemplo, estes itens são necessários porque existem restrições de tempo,

sendo que os itens variam conforme o tipo de dado (vídeo ou voz). Uma aplicação profissional de vídeo, pode não funcionar de acordo com suas especificações a não ser que o sistema operacional possa garantir uma banda mínima e uma demora máxima. Jeffay, sugere que o sistema operacional deveria gerenciar seus recursos de processamento e comunicação de forma a garantir a qualidade de serviço especificada pelo cliente ou se recusar a fornecer e fazer com que o cliente tente novamente.

Chamar um procedimento local, uma chamada de sistema, o envio de uma mensagem, a chamada remota de procedimento e invocar um método em um objeto através do envio de mensagem, são exemplos de mecanismos de invocação. Cada mecanismo faz com que seja executado um código fora do escopo do procedimento chamador, e envolve a comunicação de argumentos para este código e o retorno de dados para o chamador. Estes mecanismos podem ser síncronos (chamada local ou remota) ou assíncronos (invocação de uma operação sem retorno). No que diz respeito a desempenho, distinções importantes entre os mecanismos de invocação, independente de ser síncrono ou assíncrono, são: quando eles envolvem uma transição de domínio, quando eles envolvem comunicação via rede e quando eles envolvem escalonamento e chaveamento de threads

14. MEMÓRIA

Memória virtual é a abstração de armazenamento de um nível que é implementado, de forma transparente, através da combinação de memória primária (RAM) e memória secundária (persistente, disco). A memória virtual é importante para o projeto de sistemas operacionais distribuídos. Primeiro, a implementação de memória virtual pode necessitar usar memória secundária em um computador separado daquele onde esta a memória principal. Segundo, é possível compartilhar dados que estão simultaneamente mapeados nos espaços de endereços de processos que residem em máquinas diferentes, na forma de memória compartilhada distribuída. Grande parte da implementação de memória virtual em um sistema distribuído é a mesma de um sistema operacional convencional. A principal diferença é que a interface com a memória secundária é feita através de um servidor ao invés de o disco local.

A principal vantagem de sistemas de memória virtual é possibilitar a execução de programas grandes (código e dados muito grandes para serem armazenados na memória principal ao mesmo tempo). Em sistemas com memória virtual, parte da memória principal é usada como uma cache dos conteúdos da memória secundária. A partir do armazenamento de apenas aquelas seções de código e dados atualmente sendo acessadas pelos processos, é possível : executar programas que excedem a capacidade da memória; aumentar o nível de multiprogramação a partir do aumento do número de processos que podem estar na memória; e eliminar as preocupações dos programadores quanto ao tamanho físico da memória.

A implementação mais comum de memória virtual é chamada de *paginação por demanda*. Cada página é buscada para a memória principal sob demanda, ou seja, quando um processo tenta executar uma leitura ou escrita em uma página que não esta na memória principal, a mesma é buscada da memória secundária. Um sistema de memória virtual necessita tomar decisões. Primeiro, é preciso quanto de memória principal deveria ser alocado para cada processo (política de alocação de frames). Segundo, é necessário uma política de substituição de páginas para quando uma página precisa ser buscada da memória secundária e não existe lugar na memória principal, sendo assim é preciso escolher uma página para ser substituída. Estas políticas são utilizadas em pontos de operação do sistema como: quando um processo faz referência a uma página não residente, o que causa uma interrupção de falta de página (page fault); periodicamente, em função da medição da taxa de faltas ou dos padrões de referência a páginas dos processos.

Em um sistema distribuído, o computador onde um processo que causa uma falta de página esta executando não é necessariamente o mesmo que gerência a página faltante (o primeiro pode não Ter disco). Mesmo quando disco local é usado para paginação, as páginas dos arquivos mapeados podem ser gerenciadas por um servidor de arquivos remoto. A implementação de memória virtual em sistemas distribuídos naturalmente leva a um desenvolvimento com base em servidores (páginas são armazenadas em um servidor e não tratadas diretamente pelo kernel). Estes servidores, nível de usuário, são conhecidos como *paginadores externos*. A utilização dos servidores externos para paginação, ao invés de manter esta funcionalidade no kernel, é possível a implementação de esquemas de

paginação personalizados. Os servidores também fornecem uma abordagem para a implementação de memória compartilhada distribuída.

O mapeamento de um objeto de memória (recurso contíguo endereçável tal como um arquivo ou um conjunto de páginas) em uma região é feito através do envio de uma mensagem de requisição por parte do processo para o servidor externo. Depois do mapeamento, as mensagens que tratam de paginação passam entre o kernel e o servidor externo. O kernel busca/envia páginas de/para o paginador externo da forma como o kernel de um sistema operacional convencional busca/envia páginas no disco.

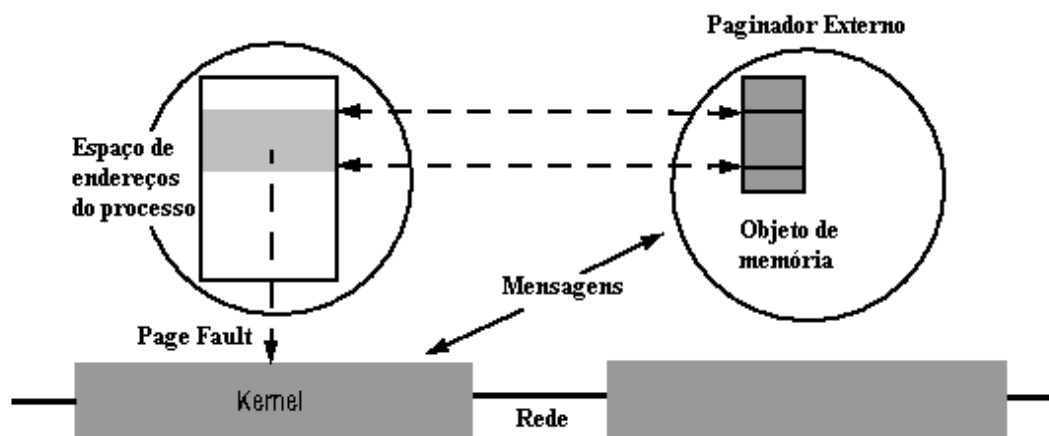


Figura 14.1 Gerenciamento de páginas com Paginador Externo

O kernel detém a responsabilidade de tratar os *page faults* que são gerados pelos processos locais. Ele também é responsável pelo gerenciamento da memória principal e portanto pela implementação da política de alocação de frames. O kernel normalmente implementa sua própria política de substituição de páginas. A informação necessária para aplicação do algoritmo de substituição, tal como os bits que são setados pela MMU quando as páginas são referenciadas ou modificadas, é local ao kernel. O papel do paginador externo inclui:

- Receber e tratar de forma apropriada os dados que são retirados pelo kernel da sua cache em função do algoritmo de substituição;
- Fornecer as páginas requisitadas pelo kernel para tratar um page fault;
- Impor restrições de consistência em relação aos objetos de memória devido a possibilidade de vários kernels tentar acessar as páginas pertencentes ao mesmo.

Os dois primeiros itens são meras extensões do sistema de memória virtual convencional que armazena cada página como um bloco de disco e grava seu endereço ou que encontra o endereço de páginas e busca as mesmas do disco. A diferença no caso distribuído é que o kernel necessita direito de acesso (capacidade) do objeto de memória e não seu endereço de disco. A capacidade é usada para transmitir ou requisitar páginas em mensagens para o paginador externo. O terceiro item é relevante para arquivos mapeados em memória e memória compartilhada distribuída.