Sistemas Operacionais

Programação Concorrente com Pthreads

Celso Maciel da Costa

Plano do Livro

Este livro é indicado para os cursos de graduação da área de computação e informática que possuem disciplinas de Sistemas Operacionais, especialmente os cursos de Ciência da Computação, Sistemas de Informação, Engenharia de Computação e Licenciatura em Computação. Também poderá ser usado como texto básico em disciplinas de Sistemas Operacionais e Programação Concorrente nos cursos de Pós-Graduação (Especialização, Mestrado e Doutorado).

Para o ensino de Programação Concorrente, devem ser estudados os capítulos 3, 5 e o anexo dedicado à biblioteca Pthreads.

O livro é formado por nove capítulos e um anexo. Cada capítulo se encerra com uma relação de exercícios. Nos capítulos que contém programas, todos estão escritos em C. Os programas concorrentes são escritos em C e com a biblioteca Pthreads.

Os sistemas operacionais Linux e Windows NT são estudados ao longo do livro. Para isso, sempre que pertinente, após a descrição dos aspectos conceituais referentes a um determinado tópico, são apresentadas, como estudo de caso, as soluções implementadas nos sistemas operacionais Linux e Windows NT .

O termo Windows NT utilizado ao longo do texto engloba uma família de sistemas, baseada no Windows NT, desde a primeira versão, NT 3.1 de 1993, até NT 6.0 de 2007, conhecido como Windows Vista.

O livro possui um anexo, dedicado à Programação Concorrente com Pthreads. Neste anexo são apresentados o protótipo e a semântica das principais primitivas de programação que permitem o gerenciamento (criação, destruição, ...) e a sincronização de threads. Exemplos de programas são apresentados, efetuando-se, desta forma, o estudo da interface de programação através de exemplos.

Sobre o autor:

Celso Maciel da Costa realizou Pós-Doutorado no Institut National de Télécommunications em Evry, France, no período julho/2002 a agosto de 2003 na área de sistemas distribuídos, é Doutor em Informática pela Université Joseph Fourier, Grenoble, França - 1993, nas áreas programação paralela e sistemas operacionais para máquinas paralelas, Mestre em Ciência da Computação pelo CPGCC-UFRGS em 1985 na área de Sistemas Operacionais e Analista de Sistemas pela PUCRS em 1978. Foi professor do Departamento de Ciências Estatísticas e da Computação da Universidade Federal de Santa Catarina de 1979 a 1985 e do Instituto de Informática da UFRGS de 1985 a 1994. Atualmente é professor Titular da Faculdade de Informática da PUCRS, onde leciona disciplinas relacionadas com programação concorrente e paralela, sistemas operacionais e programação de sistemas. Coordena projetos de pesquisa nas áreas de programação distribuída e paralela, e de sistemas operacionais distribuídos e paralelos. É também desde o ano de 1997, consultor do MEC para avaliação de processos de autorização, reconhecimento e recredenciamento de cursos de graduação da área de computação e informática.

Possui experiência em ensino e pesquisa de sistemas operacionais iniciada ainda durante a realização de seu curso de mestrado, na UFRGS. A sua dissertação foi à implementação de um sistema operacional do tipo Unix, na linguagem Pascal Concorrente. Este trabalho permitiu ao autor, além do aprofundamento nos aspectos funcionais dos sistemas operacionais, conhecer profundamente uma linguagem de programação concorrente, Pascal Concorrente, e a familiarização com o uso de processos, classes e monitores, construções suportadas pela linguagem.

No seu doutorado, o Prof. Celso Maciel da Costa se dedicou ao estudo de sistemas operacionais para máquinas paralelas. Trabalhou na definição e na implementação de um ambiente de execução para uma máquina a base de processadores Transputers. Este ambiente de execução consistiu de um Microkernel e de uma ferramenta de programação paralela que executava neste Microkernel.

Com relação ao ensino, desde o final de seu curso de mestrado o Prof. Celso leciona disciplinas relacionadas a sistemas operacionais. Inicialmente, na UFRGS, nos anos de 1985 a 1994, e, atualmente, na PUCRS, a partir de 1995.

Na UFRGS lecionou disciplinas de Programação de Sistemas, de Sistemas Operacionais e de Projeto de Sistemas Operacionais nos cursos de graduação e de Pós-Graduação (Especialização e Mestrado).

Na PUCRS leciona disciplinas de Sistemas Operacionais nos cursos de graduação (Sistemas de Informação, Ciência da Computação e Engenharia de Computação) e de Sistemas Distribuídos. Além disso, a partir de 1985, ministrou inúmeras palestras e seminários em sistemas operacionais e áreas relacionadas.

Sumário

1 Evolução e Funcionamento dos Sistemas Operacionais

Este capítulo apresenta a evolução dos sistemas operacionais, a partir de suas origens. Serão apresentados também os objetivos, a descrição do funcionamento dos Sistemas Operacionais, a estruturação dos SO e o funcionamento das chamadas de sistema.

1.1 Evolução dos Sistemas Operacionais

Sistemas operacionais são programas que controlam todos os recursos do computador e fornecem a base para o desenvolvimento dos programas de aplicação. É um gerenciador de recursos, responsável pela gerência do processador, pela gerência de memória, pela gerência de arquivos, pela gerência dos dispositivos de entrada e saída e pelos mecanismos de acesso aos dados.

Os sistemas operacionais virtualizam todos os recursos de hardware que gerenciam, criando uma máquina virtual. Por exemplo, os usuários tratam com arquivos, que são entidades lógicas gerenciadas pelo sistema operacional. Uma operação de escrita em um arquivo será traduzida pelo sistema operacional em uma operação de gravação de dados no periférico, completamente transparente ao usuário.

Monoprogramação e Multiprogramação

Nos sistemas operacionais monoprogramados existe um único programa de usuário em execução. Nos multiprogramados existem vários programas de usuário em execução simultânea. A figura a seguir mostra a organização de um sistema operacional monoprogramado. É formado por cinco componentes lógicos:

- Tratador de Interrupções: software do SO responsável pelo tratamento das interrupções;
- Drivers dos dispositivos: responsáveis pela execução das operações de entrada e saída. Existe um driver para cada classe de periférico;
- Gerenciador de Arquivos: responsável pela implementação do sistema de arquivos, permitindo ao usuário o armazenamento e a recuperação de informações.
- Seqüenciador de programas: módulo que, ao término da execução de um programa, faz com que o sistema passe a executar um novo programa;
- Programas de usuários: representa o programa em execução. Existe um único programa de usuário na memória. Ao término da execução, um outro programa será carregado pelo seqüenciador de programas e passará a ser executado.

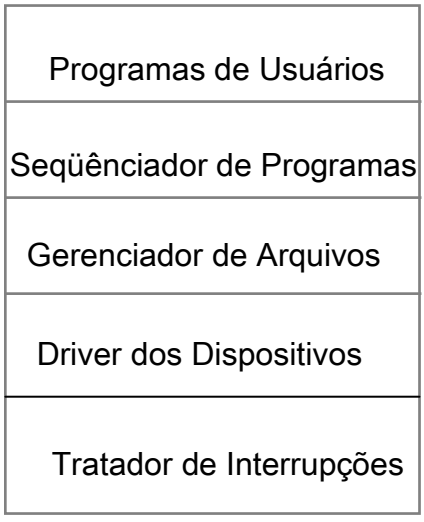

Fig. 1.1 - Sistema Operacional Monoprogramado

Um sistema operacional multiprogramado suporta a execução de múltiplos programas de usuários, em paralelo. Em sua organização possui Tratador de Interrupções, Drivers dos dispositivos, um gerenciador de memória, um gerenciador de processos que possui um escalonador, responsável pela seleção e disparo de programas, por um gerenciador de arquivos, por um seqüenciador de programas e pelos programas de usuários em execução. Num sistema multiprogramado o tempo do processador é distribuído entre os programas em execução. Cada programa executa por um certo tempo, ou até que solicite uma operação de entrada e saída, ou até que necessite esperar a ocorrência de algum outro evento. Nestes casos, o sistema operacional executa a rotina de tratamento referente ao pedido do processo e, após isso, um novo processo é selecionado e passa a ser executado. A figura a seguir mostra esquematicamente os componentes de um sistema operacional multiprogramado.

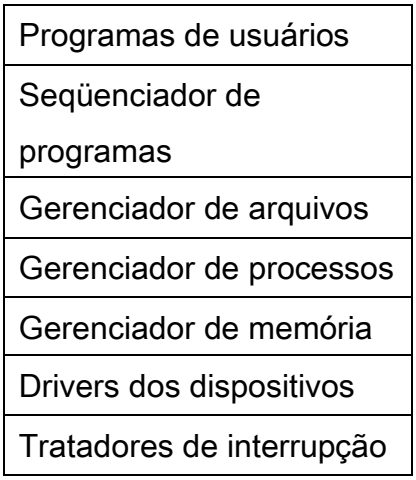

Fig. 1.2 - Sistema Operacional Multiprogramado

Evolução dos Sistemas Operacionais

Os primeiros sistemas eram totalmente manuais, com usuários altamente especializados e com um esquema de marcação de horas para utilização. Os usuários eram também os pesquisadores que trabalhavam no desenvolvimento dos equipamentos, e que possuíam um grande conhecimento do hardware, o que lhes permitia usar o computador. Estas

máquinas não possuíam sistema operacional e era necessário programar diretamente o hardware. A figura 1.3, apresentada a seguir ilustra esses primeiros ambientes computacionais.

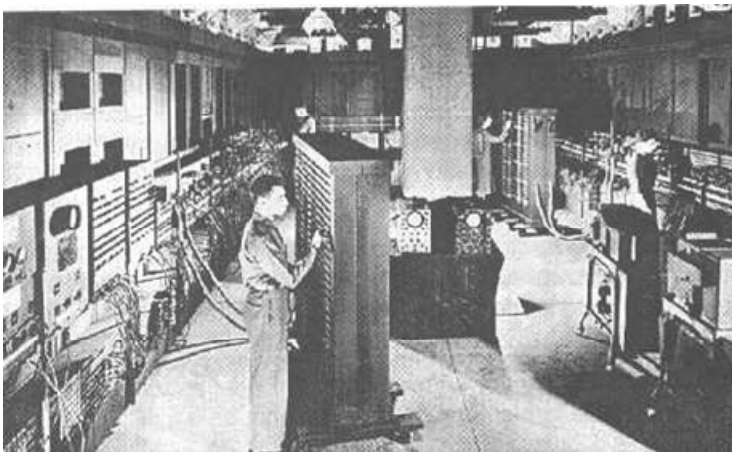

Fig. 1.3 – Primeiros sistemas computacionais

Nos primeiros sistemas computacionais, os programas eram escritos em assembler e tinham completo controle sobre a máquina. Cada usuário fazia a reserva de um determinado tempo de utilização do computador. Os problemas existentes eram significativos:

a) O usuário não conseguia terminar o seu trabalho no tempo reservado, o que poderia implicar em recomeçar a tarefa em uma nova data/hora;

b) O usuário gastava muito do tempo reservado depurando o programa, com a máquina ociosa;

c) O usuário terminava antes do tempo reservado o seu trabalho e a máquina permanecia ociosa até a chegada de um novo usuário;

d) Somente um usuário por vez podia utilizar a máquina.

Uma solução empregada foi à adoção de um operador humano, com a função de receber os diferentes programas dos usuários, executá-los e entregar os resultados. Naturalmente que nem sempre os programas eram executados na

ordem em que eram recebidos, podendo esta ordem ser determinada por fatores políticos.

O emprego de um operador humano eliminava os problemas *a, b*, e *c,* citados anteriormente. No entanto, a máquina continuava sendo monousuário e era executado um programa de cada vez. Além disso, a seqüência de execução era feita manualmente, pelo operador humano.

Outros problemas importantes existentes eram a necessidade de cada usuário desenvolver seus próprios drivers de acesso aos periféricos e as suas próprias rotinas, por exemplo, matemáticas.

Uma outra evolução, mais significativa, foi o desenvolvimento do conceito de monitor residente.

Um monitor residente é um pequeno núcleo concebido para fazer seqüenciamento automático da execução de programas. O monitor é formado, basicamente, pelos drivers de entrada e saída, por rotinas de biblioteca (reutilização de código), que possuem código de procedimentos e por um seqüenciador automático. O funcionamento do monitor é o seguinte:

Loop: carregar o código para execução;

executar;

go to Loop;

Com monitor residente, os programas e as diretivas (informações para o monitor de início de código, de início de dados, etc.) são perfurados em cartões (fig. 1.4 a seguir), e entregues pelo usuário para o operador humano para processamento, recebendo o resultado da execução posteriormente.

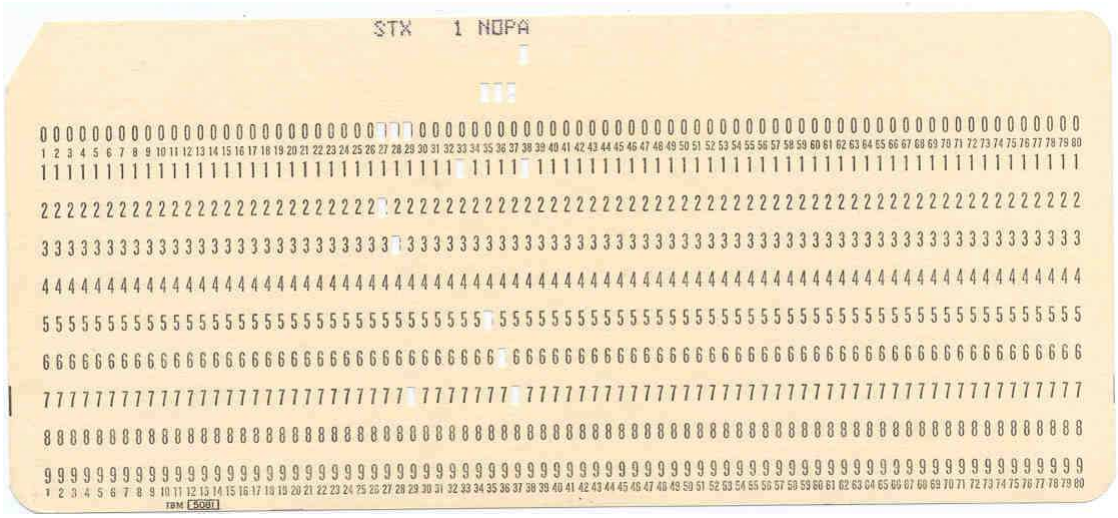

Fig. 1.4 – Cartão perfurado

Com monitor residente os passos para a execução de um programa são os seguintes

1. Elaborar a lógica da solução;
2. Programar em uma linguagem de programação (Ex. Fortran);
3. Perfurar os cartões correspondentes a cada comando da linguagem;
4. Entregar os cartões perfurados ao operador humano;
5. O operador humano transfere o programa dos cartões, com uma leitora de cartões perfurados, para um periférico mais rápido (por exemplo, um tambor magnético);
6. O computador é acionado e transfere o programa do tambor magnético para a memória;
7. O programa é executado;
8. Os resultados são gravados em fita magnética, impressos, ...

Com monitor residente a memória do computador é dividida em duas partes, uma para conter o monitor e outra para conter os programas dos usuários. Neste esquema, cada programa é executado completamente, sendo que somente após seu término um novo programa é carregado para execução. Trata-se, portanto, de um modelo de monoprogramação.

Os problemas existentes com monitor residente eram como saber a natureza do programa a executar, por exemplo, Cobol, Fortran ou Assembler, como distinguir um programa de outro e como distinguir código do programa dos dados. Uma solução para estes problemas foi à introdução do uso de cartões de controle, que fornecem ao monitor residente as informações necessárias à execução dos programas, como mostrado a seguir.

$JOB: indica o início de um programa;
$FTN: indica que a linguagem é Fortran;
$LOAD: indica que o programa deve ser carregado para a memória para execução;

“ comandos da linguagem de programação”

$RUN: indica que o programa deverá ser executado;
$DATA: indica início de dados;
$EOJ: indica fim de programa.

Com cartões de controle, o monitor residente possui as seguintes funcionalidades: Interpretar os cartões de controle, carregar os programas na memória, disparar a execução dos programas.

O monitor residente é um sistema operacional Batch monoprogramado, no qual o computador permanece ainda subutilizado, pois os programas são executados seqüencialmente.

Sistemas Operacionais Batch multiprogramados

Sistemas operacionais Batch multiprogramados representam uma evolução ao conceito de monitor residente apresentado anteriormente. Nestes sistemas, os programas de mesmas características são agrupados e executados em uma seqüência, sem a intervenção do operador, e vários programas são mantidos na memória ao mesmo tempo, sendo o tempo de CPU distribuído entre os programas residentes na memória.

Um exemplo de seqüência de programas a ser executado em um sistema operacional batch multiprogramado é o seguinte:

```
$JOB userName 1        ; identificar o usuário
$FTN                   ; carregar o compilador Fortran
cartão 1               ; cartões contendo o código do programa
cartão 2
- - -
cartão n

$LOAD                  ; carregar o programa compilado
$RUN                   ; executar o programa
cartão de dados 1      ; cartões de dados
cartão de dados 2
- - -
cartão de dados n
$EOJ                   ; final do programa
$JOB userName 2        ; identifica o usuário
$FTN                   ; carregar o compilador Fortran
cartão 1               ; cartões contendo o código do programa
cartão 2
- - -
cartão n
$LOAD                  ; carregar o programa compilado
$RUN                   ; executar o programa
cartão de dados 1      ; cartões de dados
cartão de dados 2
- - -
cartão de dados n
$EOJ                   ; final do programa
```

Duas funções fundamentais para os sistemas multiprogramados são gerência de memória e escalonamento. Deve existir uma política de alocação e liberação de memória para os programas em execução e devem existir políticas de seleção de programas para a entrega do processador.

Sistemas operacionais de tempo compartilhado (Time sharing)

Os sistemas operacionais de tempo compartilhado surgiram no início dos anos 1960 e se tornaram populares nos anos 1970. Nestes, existe uma comunicação " on line" entre o usuário e o sistema de computação. O usuário submete requisições ao sistema e recebe as respostas imediatamente. A interação entre o usuário e o equipamento é feita com o uso de terminais. O sistema possui um Interpretador de Comandos, que lê a linha de comando contendo o nome do programa a ser executado e o carrega para execução. Após a execução de um comando, o Interpretador de Comandos volta a ler o terminal. Assim, o usuário imediatamente compila e executa o seu programa. Um outro aspecto importante é que muitos usuários compartilham o uso do computador. É um sistema operacional multiusuário.

Nos sistemas operacionais de tempo compartilhado, o interpretador de comandos executa um loop eterno, no qual:

```
loop:   Ler a linha de comando contendo o nome do programa a executar;
        Carregar o programa para execução;
        Go to loop;
```

Nestes sistemas, o tempo de desenvolvimento de um programa é muito menor, comparando-se com os sistemas operacionais batch, pois o usuário interage com o equipamento. O usuário edita, compila e executa o seu programa em comunicação direta com o computador. Além disso, vários

usuários, simultaneamente, um em cada terminal, podem fazer uso da máquina. Os componentes de um sistema operacional de tempo compartilhado são os mesmos de um sistema operacional multiprogramado. Porém, o seqüenciamento de execução é feito por um interpretador de comandos, que possui a função de interagir com o usuário.

Sistemas Operacionais de Tempo real

Sistemas operacionais de tempo real são usados para controlar um dispositivo em uma aplicação dedicada, tais como controle de experimentações científicas, imagens médicas, controle de um processo industrial, na robótica, na aviação, etc. Em muitos destes sistemas, sensores coletam dados que são enviados ao computador, os dados são analisados e é executada uma ação, correspondente ao tratamento do sinal recebido e são enviados sinais de resposta.

A principal característica dos sistemas de tempo real é que existe uma restrição temporal, que implica na necessidade do sistema em garantir uma resposta atendendo os requisitos de tempo.

Os sistemas operacionais de tempo real podem ser classificados como *Hard real-time system* ou *Soft real-time system. Um Hard real-time system* é caracterizado pela necessidade responder a um evento, de uma certa forma, em um intervalo de tempo determinado, sob pena de que uma falha grave poderá ser provocada se isso não ocorrer. Este intervalo de tempo representa o deadline no qual uma resposta necessita ser gerada pelo sistema. Por exemplo, sensores colocados em um automóvel são capazes de detectar a faixa branca pintada nas laterais de uma estrada, gerar um sinal que será interpretado pelo sistema operacional que deverá sinalizar a ocorrência deste evento e será gerado, por exemplo, um sinal sonoro, de alerta ao motorista. Se o sinal sonoro não for acionado em um período determinado de tempo, um acidente poderá ocorrer. Existe, portanto, um intervalo de tempo determinado em que o sistema deverá responder a ocorrência do evento.

Soft real-time system são sistemas dedicados a aplicações nas quais se as restrições de tempo não forem obedecidas a falha que ocorrerá não será catastrófica, por exemplo, sistemas multimídia.

Um sistema operacional de tempo real possui tarefas que serão executadas quando da ocorrência dos eventos aos quais estão associadas. O componente fundamental de tal sistema é o escalonador de tarefas, que via de regra é acionado por uma interrupção disparada em um intervalo de tempo específico (ex. 20 ms). As tarefas possuem uma prioridade e o escalonador atribui o processador a tarefa de maior prioridade, garantindo o cumprimento dos deadlines de atendimento.

Um aspecto que deve ser ressaltado é que atualmente existem inúmeras aplicações de tempo real que necessitam de outros componentes, tais como drivers de rede, interface gráfica, etc. Estas necessidades são típicas de sistemas operacionais de uso geral, mas cada vez mais estão se tornando necessárias para aumentar as funcionalidades dos sistemas de tempo real. Pelas suas características, um sistema operacional de uso geral somente poderá ser utilizado para uma aplicação de tempo real que não possua deadlines críticos.

Sistemas Operacionais de Rede

Um sistema operacional de rede pode ser visto como sendo formada por um conjunto de máquinas interligadas por uma rede de comunicação, cada uma rodando o seu próprio sistema operacional e compartilhando recursos, por exemplo, um servidor de impressão. O acesso remoto a uma outra máquina se faz explicitamente, com comandos de login remoto. Além disso, o usuário necessita conhecer a localização dos arquivos e executar operações específicas para movê-los de uma máquina para outra.

Um sistema operacional de rede pode ser construído a partir de um sistema operacional tradicional, com a incorporação de um conjunto de funções que permitem a comunicação entre os diferentes processadores e o

acesso aos arquivos. Um exemplo significativo de sistema operacional de rede é Sun Solaris, que possui um sistema de arquivos distribuídos, o NFS (Network File System), que permite o acesso aos arquivos independentemente de sua localização física.

O NFS é um sistema que permite o compartilhamento de arquivos entre as estações pertencente ao pool de estações. Mais precisamente, gerencia os file servers e a comunicação entre os file servers. A idéia básica é a seguinte:

- Coleção de clientes e servidores compartilham um sistema de arquivos
    - Servidores exportam os diretórios;
    - Clientes, pela operação de montagem, ganham acesso aos arquivos.

Uma operação sobre arquivos ou diretórios é encaminhada a um componente do sistema denominado de Virtual File System (VFS). Tratando-se de um arquivo ou diretório local, a operação é executada e o resultado é enviado ao cliente. Se o arquivo ou diretório é remoto, é gerada uma mensagem sendo a mesma enviada ao sistema distante, sob a forma de um RPC (Chamada Remota de Procedimento). A operação será então realizada no sistema remoto e o resultado será enviado ao nodo solicitante, onde será tratada pelo VFS que a encaminhará ao cliente. A figura a seguir ilustra a organização do NFS.

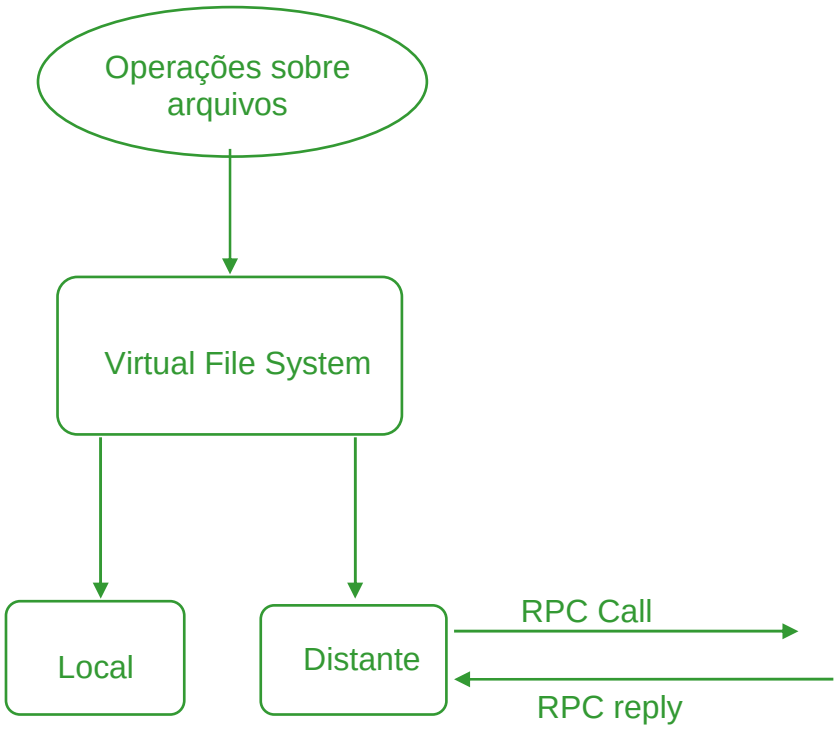

Fig. 1.5 - Visão Geral do funcionamento do NFS

O NFS suporta sistemas heterogêneos e possui um bem definido protocolo cliente/servidor. A operação “ mount” retorna um V-Node, que é um identificador de arquivo válido na rede. As operações read e write são executadas independente da localização física do arquivo e transparente ao usuário. Uma máquina é cliente NFS se ela monta ou “ importa” arquivos ou diretórios. Uma máquina é servidora se ela “ exporta” arquivos ou diretórios (figura abaixo).

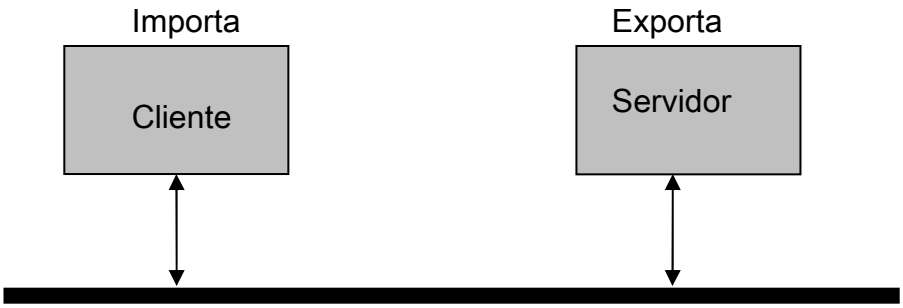

Fig. 1.6 - Visão Geral do NFS

Um sistema pode ser cliente e servidor, unicamente cliente, unicamente servidor, uma máquina servidora pode ter várias máquinas clientes e uma máquina pode ser cliente de vários servidores.

O funcionamento do NFS é centrado no VFS, que possui as seguintes características:

- É Implementado no núcleo do sistema;
- Permite acesso a diferentes tipos de sistema de arquivos (Unix, DOS,..);
- A cada sistema de arquivos montado corresponde uma estrutura VFS no núcleo;
- O sistema de arquivos virtual é mapeado no sistema de arquivos real se o arquivo é local .

A figura a seguir apresenta a arquitetura do NFS.

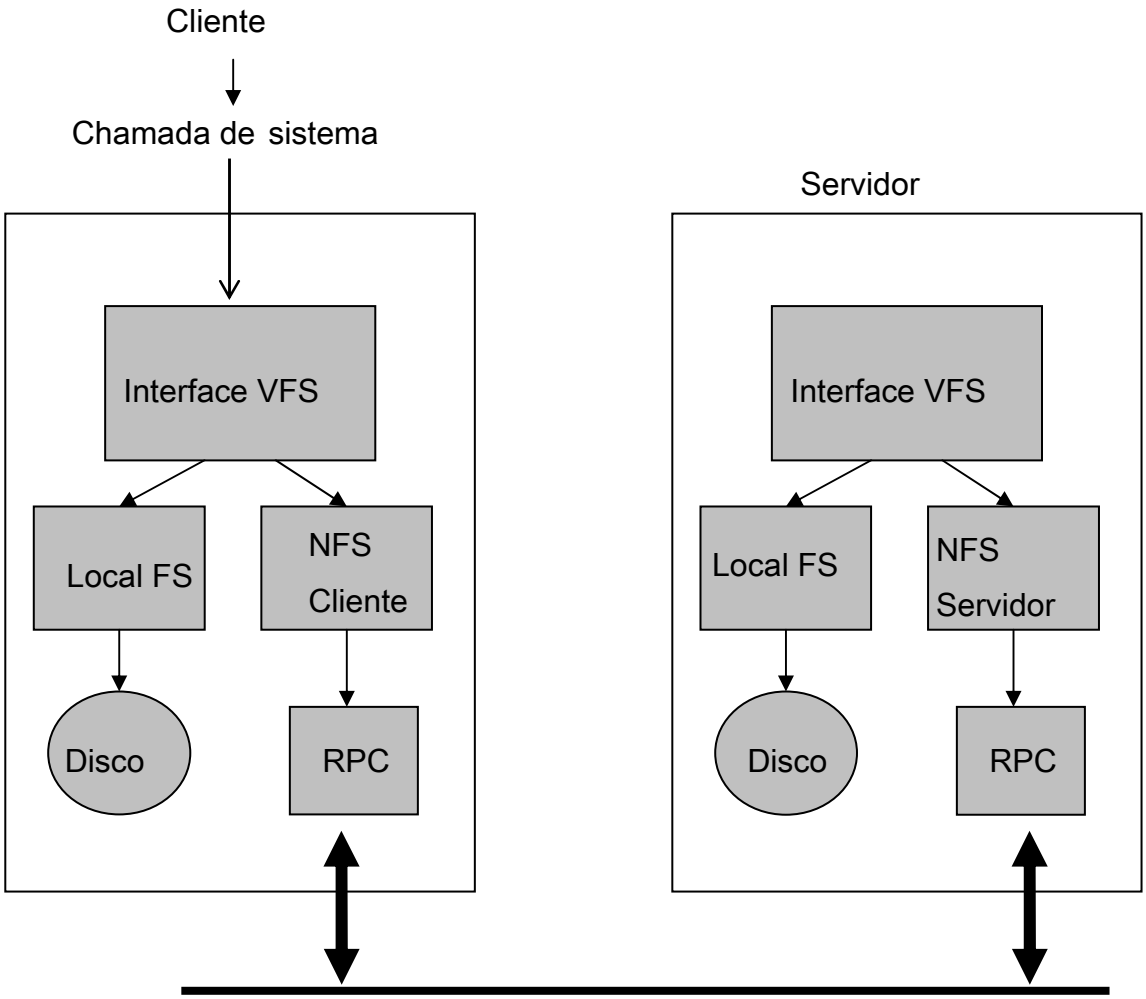

Fig. 1.7 - Arquitetura do NFS

Sistemas Operacionais Distribuídos

O objetivo de um sistema operacional distribuído é o de distribuir a computação entre vários processadores físicos e promover o compartilhamento de recursos. Cada processador, no conjunto de processadores, possui sua própria memória e a comunicação entre os processadores é feita através de linhas de comunicação.

Um sistema operacional distribuído oferece aos usuários a impressão que existe uma única máquina e um único sistema operacional, controlando todos os recursos da rede. Em um sistema operacional distribuído, em cada

processador executa um núcleo de sistema operacional e a comunicação entre os processadores é feita por troca de mensagens (ambientes sem compartilhamento de memória) ou por operações *read remoto e* write remoto (com memória virtual compartilhada). Alguns requisitos de um sistema operacional distribuído são:

- Transparência
  - Para o usuário é como existisse uma única máquina, com um único sistema operacional.
- Transparência de acesso
  - O usuário não deve necessitar saber se o recurso é local ou remoto.
- Transparência de localização
  - O nome do recurso não deve ser relacionado à localização;
  - O usuário deve poder acessar o recurso independentemente da máquina na qual está conectado.

Sistemas Operacionais para Máquinas Paralelas

Máquinas paralelas são compostas por mais de um processador operando de forma independente, podendo ter memória comum ou memória distribuída.

Nas máquinas paralelas com memória comum existe uma memória que pode ser acessada por qualquer processador da rede de processadores (figura a seguir).

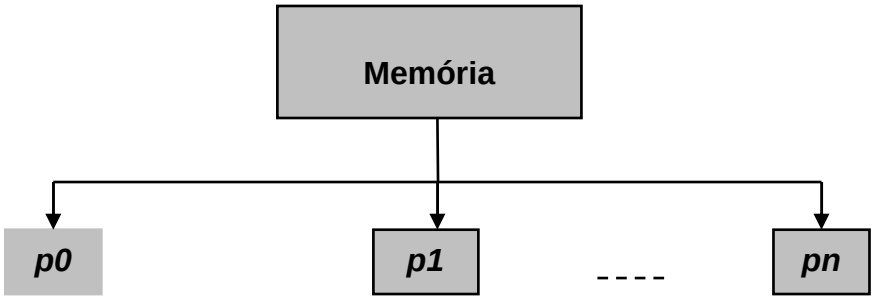

Fig. 1.8 - Máquina Paralela com Memória Comum

Nas máquinas paralelas com memória distribuída cada processador possui sua própria memória e um processador não possui acesso à memória dos outros processadores (figura 1.9 a seguir).

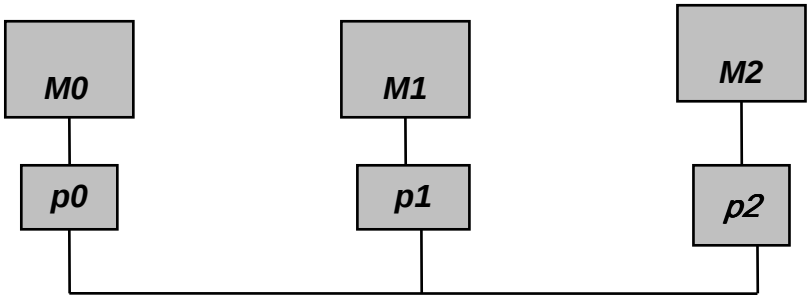

Fig. 1.9 - Máquina Paralela com Memória Distribuída

Os objetivos de um sistema operacional para máquinas multiprocessadoras são similares aos de um Sistema Uniprocessador Multiprogramado:

- Gerenciar recursos;
- Prover ao usuário a abstração de uma máquina de alto nível.

Sistemas operacionais para máquinas paralelas com memória comum

Os sistemas operacionais para máquinas multiprocessadoras com memória comum (MOS) possuem paralelismo físico, em oposição à concorrência nos sistemas uniprocessador, suportando verdadeira execução paralela de múltiplos processos.

Existem três configurações básicas para sistemas operacionais para máquinas paralelas com memória comum: Mestre/escravo, Mestre flutuante e Simétrico.

Mestre/Escravo: Um Processador (o mestre) monitora o status do sistema:

- Trata todas as chamadas de sistema;
- Atribui os processos a serem executados aos demais.

As vantagens desta configuração são facilidade de projetar e implementar e o fato de permitir a execução paralela de um único processo. As desvantagens desta abordagem são:

- Serialização das chamadas de sistema porque somente o mestre as executa;
- O mestre normalmente não fica com muita carga;
- O mestre pode se tornar um gargalo.

Mestre flutuante: O SO inteiro é tratado como uma seção crítica, e um processador por vez o executa. As vantagens desta abordagem são:

- Todos os processadores são completamente utilizados;
- Permite execução paralela de uma task . A principal desvantagem é que o mestre continua sendo o gargalo do sistema.

Simétrico: O SO pode ser acessado por múltiplos processadores, cada um acessando diferentes seções críticas. Este tipo de sistema possui uma configuração mais versátil e permite a execução paralela de um único

processo. Trata-se, no entanto, de uma organização mais difícil de projetar e implementar, que necessita de um kernel que possa ser executado em paralelo (reentrante). Um aspecto altamente importante é que muitas chamadas de sistema são executadas concorrentemente.

Sistemas Operacionais para máquinas paralelas com memória distribuída

Um sistema operacional para máquinas paralelas com memória distribuída possui um núcleo que executa em cada processador e oferece serviços de gerência de processos: criação e destruição locais ou distantes, de comunicação entre processos em um mesmo processador ou em processadores diferentes, de sincronização e também de gerência de memória. A isso se pode ainda acrescentar serviços como gerência de arquivos e de entrada e saída.

Um núcleo de sistema operacional paralelo pode ser organizado de maneira clássica. Cada processador possui um núcleo monolítico ao qual se incorporam a gerência de comunicações e o acesso aos serviços clássicos distantes. O conjunto de núcleos monolíticos cooperantes forma um núcleo monolítico paralelo.

Uma outra solução consiste na abordagem dita Microkernel. Um Microkernel é essencialmente um núcleo de comunicação entre processos, de gerência de processos e de gerência de memória. Não oferece nenhum dos serviços clássicos que são assegurados por processos servidores. Por exemplo, a abertura de um arquivo se traduz em uma mensagem do cliente ao servidor gerente de arquivos solicitando o serviço. O servidor executa a solicitação e envia os resultados. As vantagens de um Microkernel
são a extensibilidade e a modularidade. Os recursos do sistema são acessados da mesma maneira de acordo com um protocolo cliente/servidor. Para acrescentar um novo serviço basta acrescentar um novo servidor. Um Microkernel Paralelo é portanto composto por um conjunto de Microkernels locais cooperantes, um em cada nodo da máquina paralela. A função de

comunicação de cada Microkernel é estendida de maneira a permitir o acesso aos serviços oferecidos por servidores distantes.

O ponto crucial de um Microkernel Paralelo é o Microkernel de comunicação que permite a comunicação entre processos sobre um mesmo processador ou em processadores diferentes, de acordo com um protocolo cliente/servidor. O Microkernel de comunicação possui como funções básicas receber mensagens originárias de outros processadores, receber resultados de operações executadas remotamente e de enviar mensagens destinadas a outros processadores.

1.2 Estrutura e Funcionamento dos Sistemas Operacionais

Funcionamento dos Sistemas Operacionais

Os sistemas operacionais reagem a eventos, que representam solicitação de serviços. Por exemplo, uma operação de leitura no programa de usuário será realizada por uma chamada de sistema que corresponderá a uma função executada pelo sistema operacional que solicitará ao controlador do periférico a transferência dos dados para o endereço de memória informado no pedido do serviço. Quando a transferência se completar, o controlador do periférico pode gerar uma interrupção, que será sentida pelo processador e que fará com que o sistema operacional execute uma rotina de tratamento da interrupção. Neste exemplo, os dados solicitados pelo programa do usuário se encontram na memória principal, na área de dados do programa que solicitou ou na área de buffers gerenciada pelo sistema operacional. Se for na área de buffers, o sistema operacional os copiará para a área de dados do programa. Estando os dados na área do programa, o mesmo poderá novamente executar. A figura abaixo esquematiza o funcionamento de um sistema operacional.

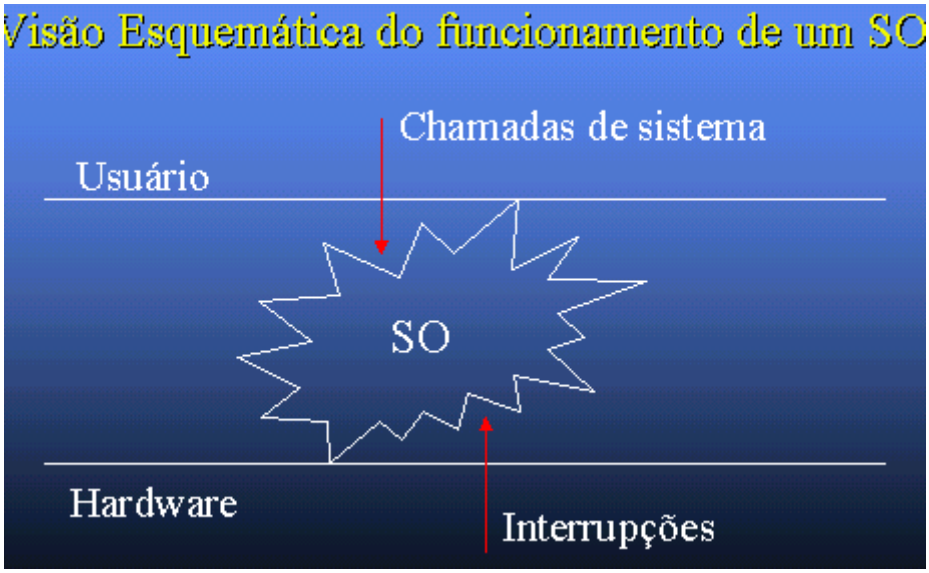

Fig. 1.10 - Visão esquemática do funcionamento de um Sistema Operacional

Estrutura dos Sistemas Operacionais

Um sistema operacional é um programa, formado por diversos processos concorrentes, situados entre os programas de aplicação e o hardware, que virtualiza o hardware tornando-o mais simples de ser utilizado. Desta forma, o desempenho do sistema operacional tem uma influência fundamental na performance das aplicações. A forma de estruturação dos sistemas operacionais têm evoluído, na tentativa de encontrar a estrutura mais apropriada. A seguir serão apresentadas as principais formas de estruturação dos sistemas operacionais.

Monolíticos

O SO é organizado como uma coleção de processos seqüenciais cooperantes, que recebem as solicitações dos usuários (chamadas de sistema), as executam e devolvem um resultado.

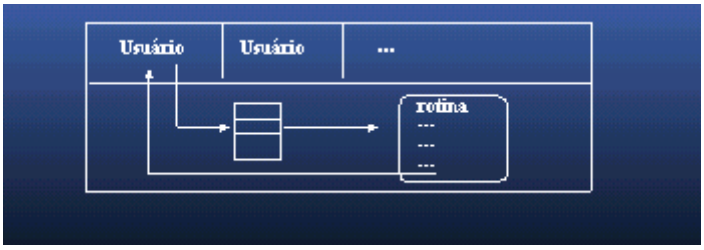

Fig. 1.11 - Sistema Operacional Monolítico

Hierárquico

Um sistema operacional pode ser organizado sob a forma de um conjunto de processos seqüenciais cooperantes, cada um executando uma função bem definida. Assim, cada programa de usuário, cada driver de periférico, a gerência de memória, o escalonamento, etc. é executado por um processo. O funcionamento do sistema é o resultado da cooperação entre os processos seqüenciais e a ação coordenada destes. Como as velocidades de execução dos diferentes processos que formam o sistema operacional é indeterminada, existe a necessidade de mecanismos explícitos de sincronização, para estabelecer o perfeito funcionamento do sistema.

No sistema operacional THE, construído por Dijkstra em 1968, todas as atividades do sistema são divididas em um certo número de processos seqüenciais. Estes processos implementam as abstrações dos diferentes componentes do sistema e são colocados nos vários níveis lógicos nos quais o sistema foi organizado.

O sistema possui seis níveis lógicos:

Nível 0: responsável pela alocação do processador para os processos em execução e pelo tratamento das interrupções.

Nível 1: responsável pela gerência de memória. No sistema foi utilizado o conceito de páginas e foi feita uma distinção entre páginas de memória e páginas de tambor magnético. Uma página de tambor magnético possui exatamente o tamanho de uma página de memória e um mecanismo de identificação de segmento permite a existência de um número maior de

páginas na memória secundária. Uma variável indica se a página está na memória principal ou não.

Nível 2: interpretador de mensagens: trata da alocação da console do sistema ao operador, para permitir a comunicação deste com os processos em execução. Quando o operador aperta uma tecla, um caractere é enviado ao interpretador de comandos, que pode então estabelecer uma conversação com um processo.

Nível 3: neste nível encontram-se os processos responsáveis pelos input streams e pelos output streams. Este nível implementa uma console virtual para cada processo, os quais compartilham uma mesma console física. O sistema permite uma única conversação por vez, usando para tal mecanismos de exclusão mútua. Este nível utiliza as funcionalidades do nível 2 (interpretador de comandos) para se comunicar como operador. Isto é necessário, por exemplo, no caso de problemas com um periférico.

Nível 4: formado pelos programas de usuário.

Nível 5: representa o operador do sistema.

A figura a seguir ilustra a estrutura do sistema.

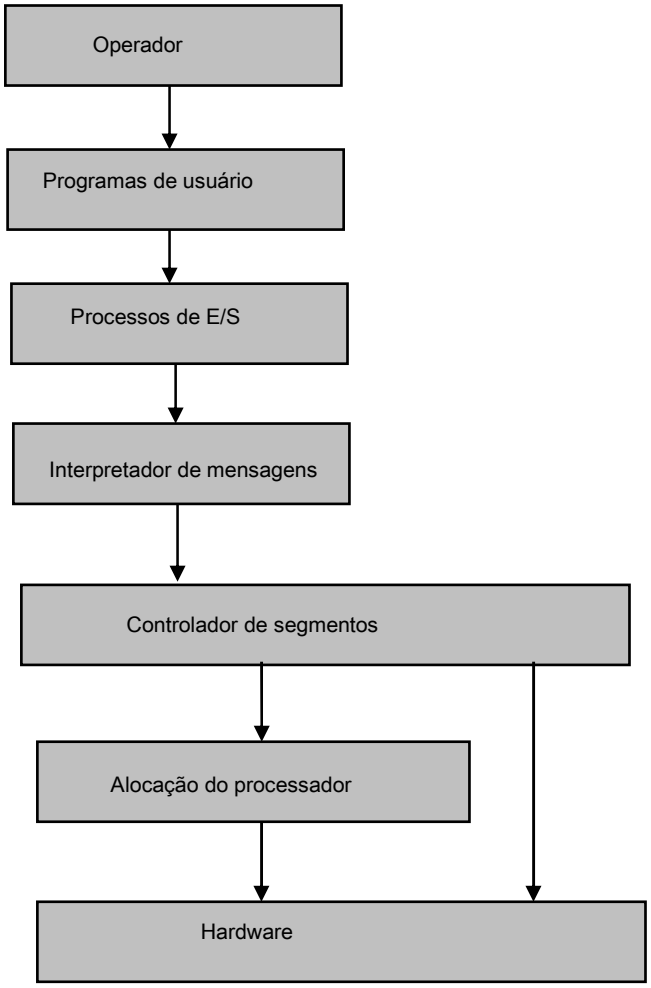

Fig. 1.12 - Organização do sistema operacional THE

A organização hierárquica, pelo fato de ser formada por processos independentes, é interessante de ser utilizada, pois é adequada a verificação da correção do funcionamento e permite o desenvolvimento modular, visto que cada processo seqüencial pode ser implementado de maneira completamente independente dos demais.

Máquinas virtuais

Máquinas virtuais são abstrações construídas por um monitor de máquinas virtuais, que distribui os recursos de hardware, permitindo a existência de vários ambientes de execução.

Uma máquina virtual é formada por recursos virtuais que são mapeados para recursos reais pelo monitor. Desta forma, uma máquina com grande capacidade de memória, disco, etc. pode ser particionada em várias máquinas virtuais, com menos recursos que a máquina real. O acesso a uma página de memória na máquina virtual é mapeado (traduzido) em um acesso a uma página de memória real. As instruções da máquina virtual são traduzidas para instruções da máquina real e executadas.

O VM (Virtual Machine) da IBM **[Creasy, 1981]** é um exemplo de sistema operacional que utiliza o conceito de máquinas virtuais. Cada sistema operacional VM é controlado por um programa chamado de Control Program, que gerencia o hardware físico, cria uma máquina virtual para cada usuário do sistema, sendo esta máquina virtual uma simulação exata de um sistema operacional (IBM/370 ou IBM/390). Cada usuário executa seus programas armazena seus dados, etc. sem interferência e sem interferir com os outros usuários.

O Control Program é o componente fundamental do sistema e roda diretamente no hardware do computador. Suas funções são gerenciar o hardware do computador, gerenciar máquinas virtuais (criação, deleção, ...), fazer o escalonamento, etc.

Microkernel

A organização Microkernel é baseada no modelo cliente/servidor, na qual os serviços do sistema são implementados por servidores especializados. Um Microkernel (mínimo) é responsável pelo tratamento das interrupções, pela gerência de memória nos seus aspectos mais básicos, e pelas funções de mais baixo nível do escalonamento. Todos os serviços do sistema são implementados por servidores, em um nível lógico acima do Microkernel. Os

clientes (programas de aplicação) solicitam os serviços ao SO (Microkernel) que os encaminha aos servidores. Os servidores executam um loop eterno, no qual recebem a solicitação de um serviço, o executam, enviam o resultado ao Microkernel e voltam a esperar a solicitação de um novo serviço. O Microkernel recebe do servidor o resultado e o transmite ao cliente, que de posse do resultado do serviço pedido, volta a executar o código da aplicação. A figura a seguir exemplifica uma organização Microkernel. As vantagens da organização Microkernel são a modularidade e a facilidade de se acrescentar novos serviços, que consiste na incorporação de um novo servidor.

Fig. 1.13 - Organização Microkernel

Exokernel

A estruturação de um sistema operacional em exokernel elimina a noção de que o mesmo deve fornecer uma abstração total do hardware sobre a qual são construídas todas as aplicações. A idéia é que, devido às necessidades distintas de diferentes aplicações, o sistema operacional deve, adequadamente, fornecer os recursos necessários a cada aplicação. Desta forma, o exokernel possui um conjunto básico de primitivas de baixo nível, com as quais gerencia completamente o hardware e, cada aplicação, utiliza somente o subconjunto adequado às suas necessidades. Assim, por exemplo, servidores podem implementar abstrações tradicionais, adequadas às aplicações às quais se destinam.

1.3 Organização do sistema operacional Linux

O Linux é um sistema operacional do tipo Unix, livremente distribuído. O nome Linux é uma combinação do nome de seu idealizador, Linus Torwalds e Unix. A primeira versão do Linux se tornou disponível em 1991. Em 1994 foi tornada pública a primeira versão estável do sistema. A evolução e a manutenção do sistema é realizada por voluntários, que trabalham no desenvolvimento, na depuração de erros e testando novas versões. O linux é hoje um moderno sistema operacional do tipo Unix, com suporte para:

- Multitarefa;
- Multiprocessamento;
- Diferentes arquiteturas;
- Diferentes file systems (MS/DOS, Ext2, Ext3, ..)
- Permite a carga de novos módulos;
- IPC (pipes, sockets,..)
- Um grande número de periféricos
- Gerência de memória com paginação por demanda
- Bibliotecas dinâmicas e compartilhadas
- TCP/IP;
- Etc.

O sistema operacional Linux é formado por seis grandes níveis lógicos, apresentados na figura 1.14 abaixo.

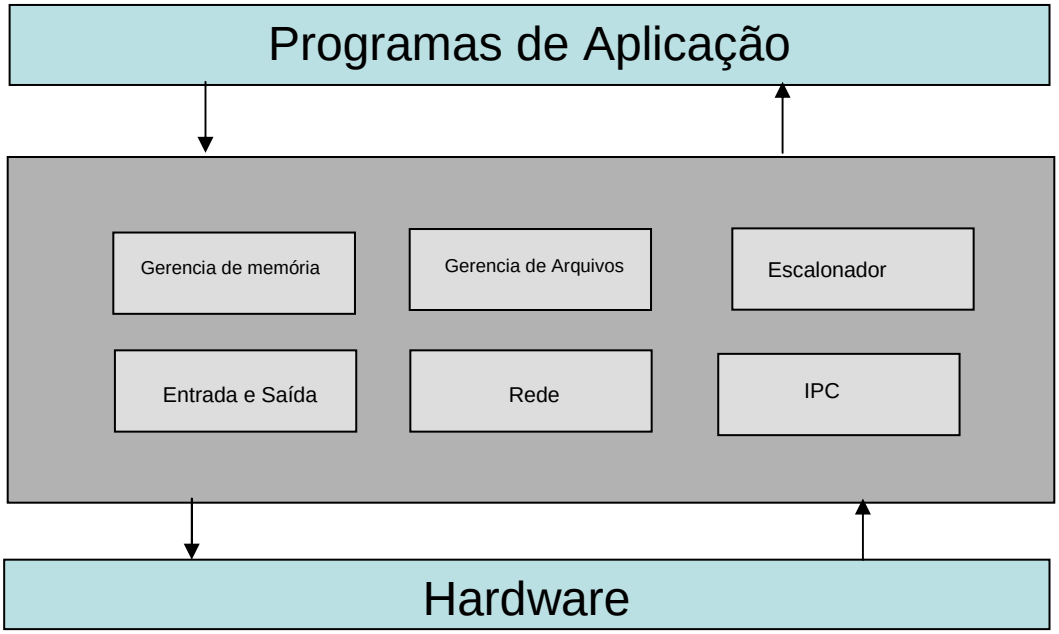

Fig. 1.14 – Arquitetura do Linux

As funções de cada subsistema são:

1. Escalonador (sched), responsável por selecionar processos para execução e entregar o processador ao processo selecionado.

2. Gerencia de memória, que permite a existência de vários processos na memória principal. O MM implementa memória virtual, que permite a execução de processos maiores que a memória real.

3. Gerencia do sistema de arquivos. Um conceito implementado é o de independência do dispositivo, que cria uma interface uniforme, comum a todos os diferentes tipos de dispositivos. Implementa também diferentes formatos de arquivos, compatíveis com outros sistemas operacionais.

4. Interface de rede, que permite o acesso as redes de comunicação.

5. Inter-Process Communication (IPC), subsistema que implementa os diversos mecanismos de comunicação entre processos.

6. Entrada e Saída, formado pelos drivers dos diferentes periféricos.

1.4 Organização do sistema operacional Windows NT

Windows NT designa uma família de sistemas operacionais da Microsoft, que evoluiu a partir de 1993, ano da sua primeira versão, conforme apresentado a seguir:

- NT 3.1 Windows NT 3.1 Workstation (denominado *Windows NT*), Advanced Server de 1993 ;
- NT 3.5 Windows NT 3.5 Workstation, Server de 1994;
- NT 3.51 Windows NT 3.51 Workstation, Server de 1995;
- NT 4.0 Windows N T 4.0 Workstation, Server, Server Enterprise Edition, Terminal Server, Embedded de 1996;
- NT 5.0 Windows 2000 Professional, Server, Advanced Server, Datacenter Server de 2000;
- NT 5.1 Windows XP Home, Professional, IA-64, Media Center (2002, 2003, 2004, 2005), Tablet PC, Starter, Embedded, de 2001;
- NT 5.2 Windows Server 2003 Standard, Enterprise, Datacenter, Web, Small Business Server de 2003;
- NT 5.2 Windows XP (x64) Professional x64 Edition de 2005;
- NT 6.0 Windows Vista Starter, Home Basic, Home Premium, Business, Enterprise, Ultimate Empresas: November 2006. Lançamento oficial: 30 de Janeiro de 2007.

A arquitetura do Windows NT é fortemente baseada na idéia de microkernel, na qual componentes do sistema implementam funcionalidades que são tornadas disponíveis a outros componente.

Utiliza também o conceito de níveis lógicos de maneira que cada nível pode utilizar funcionalidades do nível inferior e oferece funcionalidades ao nível superior. O conceito de objetos também é utilizado, sendo que arquivos, dispositivos de I/O, etc. são implementados por objetos e acessados por métodos a eles associados.

A estrutura do Windows NT possui duas partes: modo usuário, formada pelos programas de aplicação e pelos subsistemas protegidos (servidores) do sistema, e o modo kernel, chamado de executivo. Os subsistemas protegidos de ambiente fornecem API' s específicas dos sistemas operacionais suportados. Existem subsistemas (servidores) Win32, POSIX, Win16, OS/2 e MS-DOS. A figura 1.15 a seguir mostra um servidor Win32 se comunicando com um cliente Win32 através do Executivo do NT, e um servidor POSIX, a espera de solicitações de clientes.

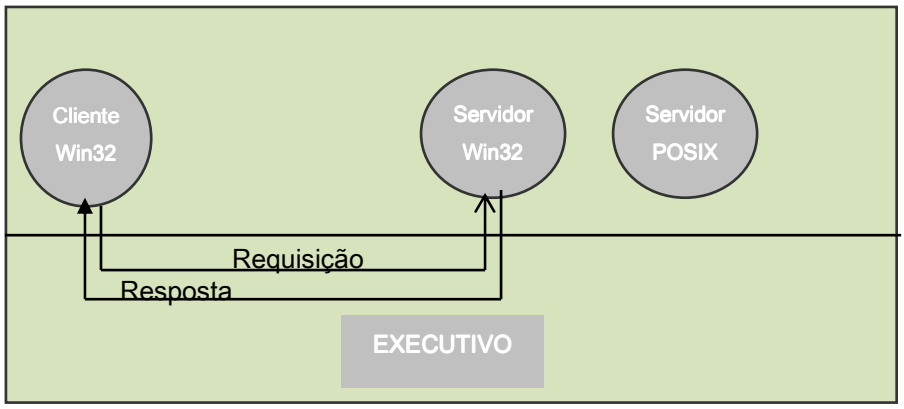

Fig. 1.15 – Comunicação clientes/servidores

O subsistema Win32 torna disponível aos demais subsistemas a interface gráfica do Windows NT e controla todas as entradas e saídas das aplicações, independentemente da API utilizada.

Cada subsistema protegido fornece uma API que um programa de usuário pode chamar. Quando um programa de usuário faz uma chamada de sistema, uma mensagem contendo a requisição é gerada e enviada para o

servidor apropriado através do mecanismo denominado LPC (Local Procedure Call).

O Executivo do NT é o kernel do sistema. Possui funções para Gerenciar objetos, Gerenciar processos, Implementar LPC, Gerenciar memória, Tratar interrupções, escalonar processos, Gerenciar I/O, gerenciar os drivers do sistema e gerenciar memória cache. O sistema possui a HAL, (Hardware Abstration Level), uma camada situada entre o executivo e o hardware no qual o sistema está sendo executado. Neste nível encontra-se o código dependente do hardware. Todo o restante do sistema é escrito em C. A figura 1.16 a seguir mostra a arquitetura do NT.

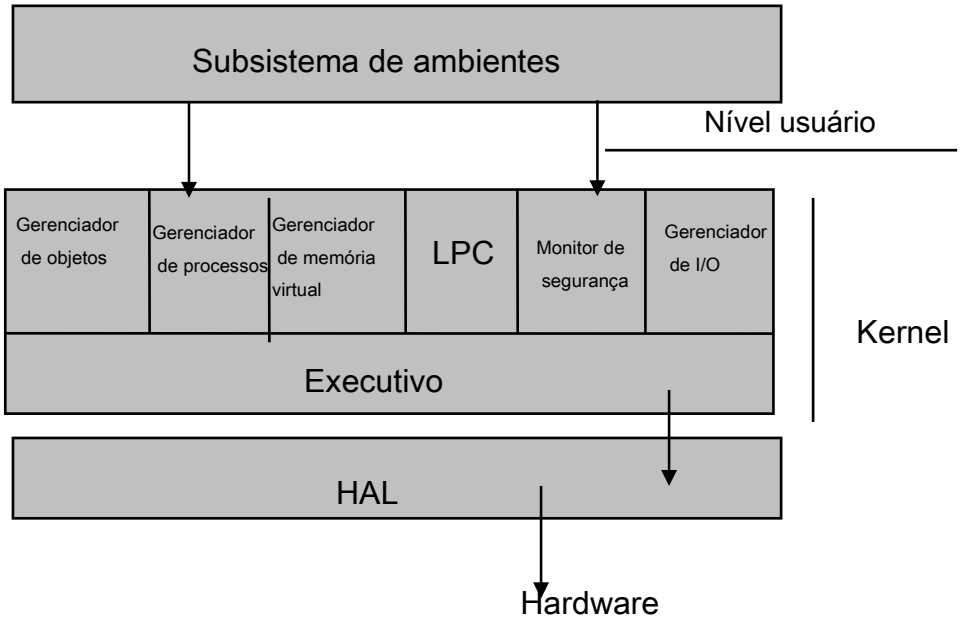

Fig. 1.16 – Arquitetura do NT

1.5 Chamadas de Sistema

Os serviços oferecidos pelo sistema operacional são acessíveis aos programas de aplicação sob a forma de chamadas de sistema. Portanto, as

chamadas de sistema são a interface entre os programas sendo executados e o sistema operacional e geralmente são implementadas com o uso de instruções de baixo nível. Um processo em execução, por exemplo, para abrir um arquivo utiliza uma chamada de sistema (open). Durante o tempo em que o sistema operacional trata a chamada de sistema, o processo que a solicita permanece bloqueado (a espera do recurso, no caso o arquivo) e um novo processo é selecionado para execução. Ao término da operação de abertura do arquivo, o sistema operacional torna o processo requisitante novamente apto a rodar. A figura a seguir mostra o grafo de transição de estados dos processos em uma chamada de sistema.

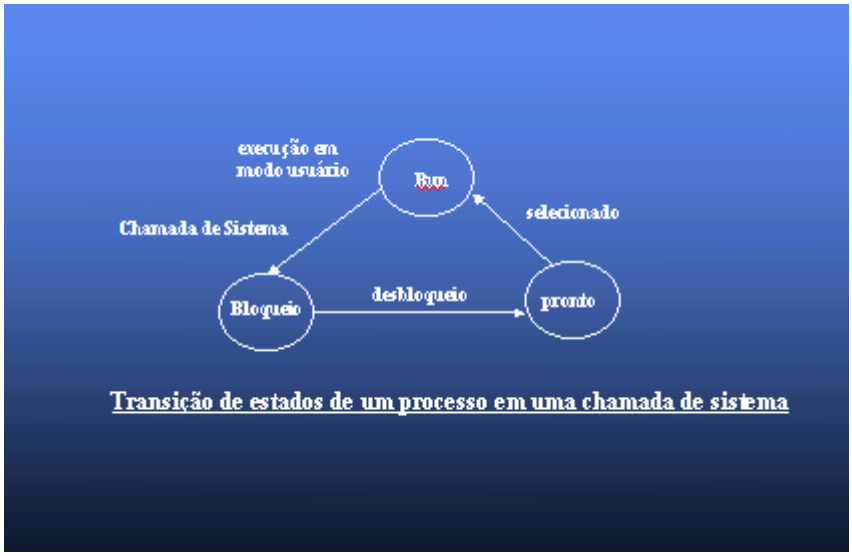

Fig. 1.17 – Grafo de transição de estados em uma chamada de sistema

Os processos em execução sofrem transição entre três estados: rodando, bloqueado e pronto para rodar. O processo quando é submetido para execução é inserido em uma fila de processos aptos para executar, está pronto para rodar. Quando o mesmo é selecionado passa para o estado rodando. Quando faz uma chamada de sistema perde o processador (por exemplo, solicitou uma operação de entrada e saída) e passa para o estado bloqueado. Ao término da chamada de sistema passa para o estado pronto para rodar e é inserido na fila de aptos. As chamadas de sistema podem ser relacionadas ao controle de processos, a manipulação de arquivos, a manipulação de dispositivos, à comunicação, etc.

Chamadas de sistema relacionadas ao controle de processos

- Criar;
- Terminar;
- Carregar, executar;
- Esperar por evento, sinalizar evento;
- Esperar passagem de tempo;
- Alocar e liberar memória;
- etc.

Chamadas de sistema para Manipulação de arquivos:

- Criar, deletar;
- Abrir, fechar;
- Ler, escrever;
- Posicionar;
- etc.

Chamadas de sistema para Manipulação de dispositivos:

- Alocar, liberar;
- Ler, escrever;
- Manutenção de informação do sistema. e.g.: ler, modificar a hora;
- Etc.

Chamadas de sistema referentes à comunicação entre processos:

- Criar, deletar canais de comunicação;
- Transferir informação;
- etc.

Implementação das Chamadas de Sistema

A implementação das chamadas de sistema depende do hardware existente. Por exemplo, os processadores Intel possuem uma instrução especial, INT, usada para troca de contexto e que transfere o controle da execução para o sistema operacional. Em processadores que não possuem este tipo de instrução, outras soluções necessitam ser utilizadas, por exemplo, chamada de uma rotina especial para executar o serviço. Outro aspecto importante é a passagem de parâmetros para o sistema operacional. Isto pode ser feito através de registradores, através de um bloco de memória, cujo endereço é colocado em um registrador, ou através da pilha de execução do processo em execução. Por questões de performance, é desejável que a passagem de parâmetros seja feita com o uso de registradores. Esta solução somente não pode ser utilizada para aquelas chamadas de sistema que possuem um número de parâmetros maior do que o número de registradores disponíveis, o que na prática são muito poucas. Para estas chamadas de sistema pode ser utilizada a outra solução, na qual o endereço da área de parâmetros é colocado em um registrador.

Exemplo de implementação de chamada de sistema

A seguir será apresentado um exemplo de implementação da chamada de sistema read, a partir de um programa escrito em C.

```
Programa do usuário

int main() {
   …
   …
   …
   read ( fd, &b, 20) ;
```

```
   …
   …
   }

Biblioteca

_read ( int fd, char *b, int l )
{
  move fd, regA;
  move b, regB
  move l, regC
  move READ, regD
  int 80H;
}
```

No exemplo acima, o usuário faz uma chamada de sistema read, passando como parâmetros o identificador do arquivo (fd), o endereço da área de memória na qual os dados lidos devem ser armazenados (&b) e o tamanho dos dados a serem lidos (20).

Esta chamada de sistema dispara a execução do procedimento de biblioteca read, que coloca os parâmetros recebidos em registradores e, com o uso da instrução especial int, passa o controle para o sistema operacional.

A instrução int coloca na pilha a palavra de status do sistema e o program counter e é atribuído ao program counter o endereço que se encontra na posição 80H do vetor de interrupções. A rotina chamada salva o restante do contexto e invoca o procedimento correspondente à chamada de sistema (read).

Após o acionamento do driver, que é feito pela rotina que executa o tratamento da chamada de sistema, um novo processo é selecionado para execução.

O processo que efetuou a chamada de sistema read será acordado (estará pronto para executar) ao término da transferência dos dados do periférico para o endereço de memória colocado na chamada de sistema, o que será feito pelo sistema operacional.

Chamadas de sistema no Linux

O sistema operacional Linux faz distinção entre o espaço de endereçamento do usuário e o espaço de endereçamento do sistema. A um processo de usuário não é permitido acessar diretamente os serviços do kernel. Nas operações que necessitam ser desenvolvidas pelo kernel, o processo do usuário solicita o serviço ao sistema operacional com uma chamada de sistema. Nos processadores Intel, a troca de contexto, do modo usuário para modo kernel, é feita com o uso da instrução **int 80h**.

Atualmente o Linux suporta mais de 300 chamadas de sistema, que podem ser acionadas pela instrução int 80h. Cada chamada de sistema possui um número que a distingue das demais, que é passado para o kernel no registrador EAX. O arquivo " /usr/include/asm/unistd.h " contém o nome e o número das chamadas de sistema. As linhas a seguir apresentam os números associados a algumas chamadas de sistema existentes no Linux.

```
#define __NR_exit                  1
#define __NR_fork                  2
#define __NR_read                  3
#define __NR_write                 4
#define __NR_open                  5
#define __NR_close                 6
#define __NR_waitpid               7
#define __NR_creat                 8
#define __NR_link                  9
#define __NR_unlink               10
#define __NR_execve               11
```

Assim, exit é a função 1, fork é a 2, read é a 3, etc. A seguir serão apresentadas as principais chamadas de sistema no sistema operacional Linux, referentes ao controle de processos e a manipulação de arquivos.

A seguir serão apresentadas as chamadas de sistema para controle de processos fork, exec, exit e wait.

- fork: cria um novo processo que é uma cópia do processo pai. O processo criador e o processo filho continuam em paralelo, e executam a instrução seguinte à chamada de sistema. Mais detalhadamente,

uma chamada de sistema fork dispara *do_fork()*, no núcleo do Linux, que possui as seguintes funções :

1. Criar um descritor (task_struct), com alguns valores iguais ao do processo pai;
2. Criar uma pilha para o novo processo;
3. Atribuir um PID ao novo processo;
4. Inicializar demais campos do descritor (ex. PID, PPID) ;
5. Atribuir o estado TASK_RUNNING ao processo filho;
6. Inserir o processo na fila TASK_RUNNING, para seleção pelo escalonador;
7. O PID do filho retorna para o processo pai ao final da chamada de sistema;
8. Como o processo filho é uma cópia do processo pai, o código é compartilhado. Retorna zero para o processo filho no final da chamada fork, o que permite a seleção de trechos diferentes de código para execução pelo pai e pelo filho.

- Exec: executa um programa, substituindo a imagem do processo corrente pela imagem de um novo processo, identificado pelo nome de um arquivo executável, passado como argumento. As ações executadas pela função do_exec são:

    1. Verificar a existência do arquivo passado, como argumento;
    2. Verificar permissões de acesso ao arquivo;
    3. Verificar se o arquivo é executável;
    4. Liberar a memória do processo que fez o exec;
    5. Atualizar o descritor do processo (task_struct);

6. Atualizar a tabela de páginas do processo (o código e os dados permanecem no disco, virão para a memória sob demanda);

7. Carregar o segmento de dados não inicializados (variáveis) na memória;

8. inicializar os registradores e o PC;

9. A execução continua no novo programa.

- Exit: termina o processo corrente. Esta primitiva libera os recursos do processo corrente, chamando, no núcleo, do_exit, que executa as seguintes ações:

    1. Liberar as estruturas de dados(semáforos, timers, arquivos,...);

    2. Liberar memória;

    3. Avisar o pai do término. Se o pai está esperando (primitiva wait), é acordado. Se o processo pai já terminou, o processo é herdado pelo processo 1. Se este também terminou, é então herdado pelo processo 0.

    4. O estado do processo para para TASK_ZOMBIE e permanece neste estado até que o pai execute a primitiva wait;

    5. Chamar o escalonador.

- Wait: suspende a execução do processo corrente até que um filho termine. Se um filho terminou antes desta chamada de sistema (estado zombie), os recursos do filho são liberados (descritor) e o processo não fica bloqueado, retornando imediatamente. A função do_wait, do núcleo do Linux executa as seguintes ações:

    1. Se o processo não possui filhos a primitiva retorna imediatamente;

2. Se existe um processo filho no estado TASK_ZOMBIE e trata-se do processo que o pai deseja esperar pelo término, o descritor do processo filho é liberado e o pai retorna imediatamente;

3. Se existe filho, o processo passa para o estado TASK_INTERRUPTIBLE, esperando pelo sinal do filho (exit) e chama o escalonador.

Outras chamadas de sistema importantes são Kill, usada para enviar um sinal para um processo ou grupo de processos, sendo que o sinal pode indicar a morte do processo e sleep, que suspende o processo pelo tempo especificado como argumento.

Chamadas de sistema para manipulação de arquivos

Um conjunto de primitivas permite aos usuários executar operações em arquivos e diretórios. Algumas primitivas para manipulação de arquivos são:

- Open: abre o arquivo especificado, convertendo o nome do arquivo no descritor do arquivo, um valor inteiro não negativo.
- Close: fecha um descritor de arquivo, liberando bloqueios e permitindo que o arquivo possa ser acessado por outros usuários.
- Read: recebe três argumentos (descritor do arquivo (fd), buffer (b) e quantidade de bytes que deverão ser lidos (l). Tenta ler do arquivo fd bytes, e os armazena no buffer b.
- Write: recebe três argumentos (descritor do arquivo (fd), buffer (b) e quantidade de bytes que deverão ser escritos (l). Tenta

escrever no arquivo fd l bytes, transferidos a partir do endereço b.

- Creat: cria um novo arquivo.
- Pipe: retorna em um vetor de dois elementos um par de descritores de arquivos criados para permitir a comunicação entre processos. O escritor retornado na entrada 0 do vetor é usado para leitura no pipe e o retornado na entrada 1 é usado para escrita no pipe.
- Link: cria um sinônimo para um arquivo já existente. Os dois nomes se referem ao mesmo arquivo.
- Unlink: elimina um nome de arquivo. Tratando-se do último nome associado ao arquivo, o mesmo é deletado.
- Lseek: coloca o byte corrente do arquivo na posição indicada pelo argumento recebido.

Para executar operações em diretórios as principais primitivas são:

- mkdir: cria um novo diretório com o nome especificado pelo argumento.
- rmdir: remove o diretório especificado.
- ls: lista o conteúdo do diretório especificado.
- chmod: usada para modificar os direitos de acesso aos arquivos especificados.

O programa a seguir, escrito em assembler, exemplifica o uso de chamadas de sistema no Linux.

```
# Exemplo de programa que executa chamadas de Sistema no Linux


# Para montá-lo digite    as syscall.s -o syscall.o
# Para gerar o executável digite    ld -s syscall.o  -o syscall



.data                   # Seção de dados inicializados
     O_RDONLY  = 0
     STDOUT    = 1
     SYS_EXIT  = 1
```

```
      SYS_READ  = 3
      SYS_WRITE = 4
      SYS_OPEN  = 5
      SYS_FORK  = 2
      len       = 13

      file:   .string   "Teste.txt\0" # Arquivo a ser lido

      msg:    .string   "Alo mundo!\0" # Mensagem a ser escrita

      .comm   buf, 512                # buffer com 512 Bytes

.text                    # Seção de código
      .global  _start

_start:
      # Alô mundo
      mov   $len, %edx        # Tamanho da mensagem
      mov   $msg, %ecx        # apontador para o buffer
      mov   $STDOUT, %ebx     # Arquivo de saída: STDOUT
      mov   $SYS_WRITE, %eax  # Número da chamada de sistema (write)
      int   $0x80             # Chamada do kernel


      # int open (const char *pathname, int flags, mode_t mode)
      mov $O_RDONLY, %edx       # Abertura do arquivo para leitura
      mov $0 , %ecx             # Flag para abertura (0)
      mov $file, %ebx           # Nome do arquivo a ser aberto
      mov $SYS_OPEN, %eax       # Número da chamada de sistema (open)
      int $0x80                 # Chamada do kernel

      # int read(int fd, void *buf, size_t count)
      mov $40,   %edx           # Quantidade de bytes a serem lidos
      mov $buf,   %ecx          # apontador para o buffer
      mov %eax,  %ebx           # Descritor do arquivo a ser lido
      mov $SYS_READ, %eax       # Número da chamada de sistema (read)
      int $0x80                 # Chamada do kernel

      # int write(int fd, const void *buf, size_t count)
      mov $40, %edx              # Tamanho da mensagem
      mov $buf, %ecx             # apontador para o buffer
      mov $STDOUT, %ebx          # Arquivo de saída: STDOUT
      mov $SYS_WRITE, %eax       # Chamada de sistema (write)
      int $0x80                  # Chamada do kernel

      # int _fork()
      mov $SYS_FORK, %eax      # No. da chamada de sistema (sys_fork)
      int $0x80                # Chamada do kernel

      # Alô mundo
      mov   $len, %edx        # Tamanho da mensagem
      mov   $msg, %ecx        # apontador para o buffer
      mov   $STDOUT, %ebx     # Arquivo de saída: STDOUT
      mov   $SYS_WRITE, %eax  # Número da chamada de sistema (write)
      int   $0x80             # Chamada do kernel

      # void _exit(int status)
      mov $0, %ebx             # Código do exit
      mov $SYS_EXIT, %eax      # Número da chamada de sistema (exit)
      int $0x80                # Chamada do kernel
```

O programa apresentado acima é formado por sete chamadas de sistema: write, que escreve o string “ Alô mundo” na saída padrão (console), open que abre o arquivo “ teste.txt” para leitura, read que lê um string de até 40 caracteres do arquivo “ teste.txt” e o armazena em “ buf” , write que escreve na saída padrão (console) o conteúdo lido pela primitiva read, fork que cria um novo processo, write, executada pelos dois processos, que escrevem na saída padrão o string “ Alô mundo” e exit, executada pelos dois processos para finalizar a execução.

Nas chamadas de sistema no Linux, os argumentos são passados nos registradores, obedecendo a seguinte ordem:

- %eax - número da função;
- %ebx - primeiro argumento;
- %ecx - segundo argumento;
- %edx - terceiro argumento;
- %esx- quarto argumento ;
- %edi - quinto argumento.

No linux quase que a totalidade das chamadas de sistemas possuem até cinco argumentos. Para chamadas de sistema que possuem seis argumentos, o endereço de uma tabela contendo o sexto argumento é colocado no %ebx.

O resultado da execução da chamada de sistema é colocado pelo sistema operacional no registrador %eax. No programa acima, na chamada de sistema open, o descritor do arquivo aberto, ao término da execução da chamada de sistema pelo Linux, está no registrador %eax.

Exercícios

1. Compare as organizações Monolítica e Microkernel.
2. Descreva o funcionamento de um sistema operacional baseado no modelo Microkernel.

3. Descreva as funcionalidades necessárias em um sistema operacional monoprogramado de uso geral. Que funcionalidades devem ser acrescentadas em um sistema deste tipo para transformá-lo em um sistema operacional multiprogramado de uso geral? Justifique.
4. Escreva a rotina de biblioteca **sleep(t),** que faz com que o processo que a executa fique bloqueado por **t** segundos, sabendo que os parâmetros devem passados para o SO através dos registradores (R0, R1, R2, ...) e que existe uma instrução especial (INT) que serve para troca de contexto.
5. Descreva todos os passos executados pelo sistema operacional na criação de um processo.
6. Apresente os estados que um processo assume durante a sua execução e faça o grafo de transição, indicando o evento causador da transição.
7. Sabendo que a chamada de sistema ***wait(int flag)*** bloqueia o processo que a executa até que um processo filho termine (se o argumento passado como parâmetro for igual a – 1 o processo fica bloqueado até que todos os filhos terminem) e que a chamada de sistema ***exit()*** termina o processo que a executa e acorda o processo pai se o mesmo estiver bloqueado (pela primitiva wait):
    a) Escreva os procedimento de biblioteca que implementam as chamadas de sistema ***wait(int flag)*** e ***exit()***, sabendo que as informações (parâmetros) necessárias para a realização do serviço devem ser colocadas em registradores (R1, R2, ...), que a identificação do serviço a ser executado pelo SO deve ser colocada no registrador R0 e que existe uma instrução **int 80H** utilizada para troca de contexto.
    b) Descreva os procedimentos executados pelo sistema operacional na realização destes serviços.

2 Princípios de Entrada e Saída

Neste capítulo serão apresentados conceitos básicos dos dispositivos de entrada e saída, .os mecanismos de comunicação entre a CPU e os periféricos, a organização dos subsistemas de entrada e saída e os princípios de entrada e saída no sistema operacional Linux.

2.1 Dispositivos de entrada e saída

Um sistema de computação é formado por Unidade Central de Processamento (UCP), memória e periféricos. Periféricos são dispositivos que permitem aos usuários a comunicação com o sistema. Os periféricos podem ser de entrada, de saída ou de entrada e saída. Exemplos de periféricos de entrada são mouse, teclado, scanner, leitora ótica, etc. Periféricos de saída de dados são impressora, monitor de vídeo, etc. e periféricos de entrada e saída são discos magnéticos, fitas magnéticas, pendriver, CD´s disquetes, etc.

Os periféricos podem ser orientados a bloco e orientados a caractere. Os periféricos orientados a blocos armazenam as informações em blocos de tamanho fixo e cada bloco possui o seu próprio endereço. Os blocos podem ser de 512, 1024, 2048, ... bytes. A principal característica destes dispositivos é a capacidade de se endereçar blocos aleatoriamente, o que permite que se acesse um dado pertencente a um arquivo sem ter que acessar todos os anteriores. Fazem parte deste tipo de periférico os discos magnéticos.

Os periféricos orientados a caractere executam operações de transferência de cadeia de caracteres. Não possuem estrutura de blocos,

portanto não possuem a capacidade de endereçamento aleatório. Exemplos desses periféricos são os teclados e as impressoras.

2.2 Comunicação entre a CPU e os Periféricos

Em um sistema de computação a Unidade Central de Processamento (UCP) acessa a memória para fazer busca de instruções e dados, executa as instruções e interage com os periféricos nas operações de entrada e saída. Em relação a UCP, os periféricos são extremamente lentos, e operam de forma assíncrona. A comunicação da UCP com o periférico é feita através de registradores. Diferentes periféricos possuem número distinto de registradores. Três registradores são importantes:

- Registrador de dados: Neste registrador o processador coloca os dados a serem gravados no periférico e o periférico coloca os dados resultantes de uma operação de leitura.
- Registrador de status: Contém informações sobre o estado do periférico (livre/ocupado).
- Registrador de controle: Neste registrador o processador escreve os comandos a serem executados pelo periférico.

Os dispositivos de entrada e saída podem ser mapeados em um espaço de endereçamento separado, distinto do espaço de endereçamento de memória e a comunicação e a troca de informações entre o processador e o controlador do periférico é feita através deste espaço, ou o espaço de endereçamento pode ser mapeado diretamente na memória, ocupando endereços no espaço de memória.

Com o espaço de endereçamento distinto do espaço de endereçamento de memória são necessárias instruções específicas de leitura e escrita de dados (Ex. IN e OUT). IN é usada para leitura de dados dos registradores usados na comunicação com o controlador do periférico. OUT é usada para escrita de dados nestes registradores.

Com o I/O mapeado na memória, para a comunicação entre o processador e o controlador do periférico, podem ser usadas todas as instruções que fazem movimentação de dados na memória.

Tipos de Entrada e Saída

I/O Programado (pooling).

As operações de entrada e saída são realizadas sob o controle do Sistema Operacional, que fica testando, na interface com o periférico, o estado do I/O (pooling). O controlador do periférico não interrompe a CPU. Ex. de funcionamento:

1. SO envia ao periférico um comando
2. Ler *status* do periférico
3. Se *Status NOT* ready goto 2
4. Se comando executado é de leitura então
   Ler informação do periférico
   Escrever a informação lida na memória
6. Se existe novo pedido de I/O então go to 1

O grande inconveniente do I/O programado é que a maior parte do tempo do processador é desperdiçado testando o status do periférico (pooling: linhas 2 e 3).

I/O com interrupção

O periférico, ao término de uma operação de transferência de dados (entrada ou saída), sinaliza o processador gerando um pedido de interrupção. O processador passa então a executar o procedimento que implementa a função correspondente a interrupção ocorrida. Existe um tratamento específico à cada fonte de interrupção. Como a ação do periférico é independente da ação do processador, durante a operação de transferência de dados o processador fica livre para executar outros processos.

Acesso Direto à Memória (DMA)

A informação é transferida diretamente do periférico para a memória sem intervenção do processador. Ao término da operação de transferência de dados é gerada uma interrupção, que será tratada pelo Sistema Operacional. Com DMA o processador, durante as operações de I/O, o processador fica executando outros processos.

Origem das Interrupções

As interrupções podem ser:

a) Geradas pelo programa: implementadas com o uso de instruções especiais (ex. INT, TRAP).
b) Geradas por um erro: divisão por zero, referência a memória fora do espaço permitido, etc.
c) Geradas pelo relógio;
d) Geradas pelos periféricos: sinalização de final de operação de E/S ou condição de erro.

Tratamento de Interrupções

O sistema operacional reage à ocorrência de interrupções, que provocam a execução da rotina de tratamento correspondente à fonte de interrupção.

Tratamento de interrupções de software

Estas interrupções são geradas por uma instrução especial (int, trap, ...). Quando ocorre uma interrupção de software:

- O programa para de executar;
- O sistema operacional executa o serviço solicitado pelo processo do usuário ou o encaminha ao periférico (no caso de E/S);
- Ao final da execução do serviço, o controle retorna ao processo solicitante ou um novo processo é selecionado (no caso do processo solicitante necessitar esperar pela realização do serviço).

Tratamento de interrupções de hardware

Quando ocorre uma interrupção de hardware, as ações executadas pelo processador são as seguintes:

- O processador acaba execução da instrução atual;
- O processador testa existência de interrupção;
- O processador salva estado atual;
- O processador carrega contador de programa com o endereço da rotina de tratamento da interrupção;
- A execução começa na rotina de tratamento da interrupção;
- A rotina de tratamento da interrupção executa;
- O contexto de execução anterior é restaurado;
- A execução retorna para a rotina interrompida.

2.3 Organização do subsistema de Entrada e Saída

O objetivo de um subsistema de entrada e saída é permitir a um programa de aplicação acessar os periféricos para realização de operações de transferência de dados. Para tal, deve possuir uma camada de mais alto nível lógico para servir de interface com o programa de aplicação e uma camada para interagir diretamente com o hardware dos periféricos, formada pelos drivers de periféricos. Também são relacionados diretamente também com as operações de entrada e saída os tratadores de interrupção. Possuem, portanto, três níveis lógicos: tratadores de interrupção, drivers dos periféricos e Interface com os programas de aplicação,.

Tratadores de Interrupção

Quando uma operação de entrada e saída é solicitada por um processo, o sistema operacional recebe o pedido e aciona o driver que o encaminhará ao periférico específico. O processo solicitante fica bloqueado, sem disputar o processador, até que o pedido se complete. Os tratadores de interrupção são componentes do núcleo do sistema operacional, acionados pelas interrupções. Quando a interrupção ocorre, a rotina de tratamento executará as ações correspondentes à interrupção. O resultado de uma interrupção é

que o processo bloqueado na operação será desbloqueado pelo tratador da interrupção.

Além dos periféricos, outra fonte de interrupção é timer. Quando uma interrupção do timer ocorre, em um sistema de tempo compartilhado, muitas ações relacionadas ao tempo devem ser executadas.

No sistema operacional Linux, a interrupção do timer está associada ao IRQ0 e ocorre a cada 10 ms. A entrada IRQ0 contém o endereço da rotina *timer_interrupt(),* que executa as seguintes ações:

- Calcula o tempo de retardo entre a ocorrência da interrupção e o disparo da rotina de tratamento, que será utilizado para corrigir o tempo atribuído ao processo do usuário;
- Chama a rotina *do_timer_interrupt ().*

A rotina *do_timer_interrupt ()* roda com interrupções desabilitadas. Os passos que executa são:

- Atualiza a variável jiffies, que armazena o número de ticks ocorridos desde que o sistema foi disparado (um jiffie = 1 tick e um tick ocorre a cada 10 ms);
- Atualiza o tempo do processo do usuário:
    - Atualiza o campo do descritor do processo corrente que mantém o tempo de execução em modo usuário;
    - Atualiza o campo do descritor do processo corrente que mantém o tempo de execução em modo kernel;
    - Conta o número de processos prontos para rodar e atualiza estatísticas de uso da CPU;
    - Verifica se o tempo do processo corrente se esgotou.
- Dispara a rotina *timer_bh ()* .

A rotina *timer_bh ()* dispara os procedimentos responsáveis pela execução das seguintes funções:

- Atualizar tempo corrente do sistema e a data do computador e calcula a carga corrente do sistema:
- Tratamento dos relógios lógicos: a implementação de relógios lógicos é feita com com cada relógio possuindo um campo que indica quando o tempo expira. Este campo é obtido somando o número de ticks ao valor corrente da variável jiffies. Cada vez que jiffies é incrementado, é comparado com este campo. Sendo maior ou igual, o tempo expirou e a ação correspondente deve ser executada. Relógios lógicos são usados:
    - Por programadores para disparar uma função em um tempo futuro ou para esperar por passagem de tempo;
    - Por drivers para detectar situações de erros (ex. periféricos que não respondem em um certo período de tempo)
    - Etc.

Driver dos dispositivos

Drivers de dispositivos são componentes do sistema operacional que interagem com um periférico, ou com uma classe de periféricos. Para cada tipo de periférico deve existir um driver específico, com pequenas variações para periféricos diferentes dentro da mesma classe. A função de um driver é receber um pedido do sistema operacional, que corresponde a uma solicitação de um usuário (ex. ler dados de um arquivo em disco), preparar a requisição, e encaminhar para a controladora do periférico. Se o periférico está ocioso, a operação deverá ser realizada imediatamente. Caso contrário, o pedido será colocado na fila de requisições do periférico.

No caso de uma operação de leitura de dados de um disco, o sistema operacional recebe do usuário os parâmetros (identificador do arquivo, tipo da operação (leitura ou escrita) , endereço de memória onde devem ser armazenados os dados, quantidade de bytes a serem lidos), calcula o número do bloco e encaminha para o driver o número do bloco a ser lido, juntamente com os demais parâmetros. O driver constrói a requisição (estrutura de dados na qual coloca os parâmetros recebidos e na qual acrescenta o identificador do processo solicitante, o posicionamento do braço no cilindro próprio e a operação requerida (leitura ou escrita) e coloca na fila de requisições do disco, se o periférico estiver ocupado.

A interrupção da controladora sinaliza final de operação de entrada e saída. A rotina de tratamento da interrupção salva o contexto e acorda o driver relacionado à fonte específica de interrupção. Tratando-se de uma interrupção do disco, sendo final de operação de leitura, o driver identifica o processo para o qual a transferência de dados se completou e o incorpora na fila de processos aptos a executar. A seguir, pega a primeira requisição da fila de requisições e a entrega para a controladora, escrevendo os parâmetros nos registradores da controladora. Após essas ações, a execução retorna ao processo que estava sendo executado quando ocorreu a interrupção.

Interface com os programas de aplicação

Os componentes do subsistema de entrada e saída são quase que completamente implementados no núcleo do sistema operacional. O software que não faz parte do sistema operacional está implementado a nível usuário, pelas bibliotecas de funções. Assim, quando um usuário necessita ler de um arquivo no Linux, utiliza a função

```
n = read (f, &b, nbytes);
```

que lê do arquivo ***f*** e armazena no **&b**, quantidade de **nbytes.**

Esta função faz parte de uma biblioteca de funções e é incorporada ao código do programa do usuário. Assim, a comunicação dos programas de aplicação com o subsistema de entrada e saída é implementada por um conjunto de

procedimentos de biblioteca, que possui a interface de cada primitiva. Fazem parte desta biblioteca, além da função read, as funções write, open, close, create, etc.

2.4 Entrada e saída no Linux

A nível usuário, a comunicação do programa de aplicação e o subsistema de entrada e saída é feita pelas chamadas de sistema relacionadas às operações de entrada e saída. Cada chamada de sistema possui a identificação da operação e os parâmetros necessários à sua execução. Exemplos de chamadas de sistema são as primitivas open, close, read, etc. As chamadas de sistema referentes as operações de entrada e saída são quase que exclusivamente tratadas pelo VFS (Virtual File System).

O Linux implementa o conceito de independência do dispositivo que, permite que, por exemplo, o comando

write(f, &b, nbytes);

que escreve no arquivo ***f*** , ***nbytes,*** armazenados no endereço ***&b***, seja executado para o arquivo ***f*** armazenado em um disco ou em um disquete. Todos os procedimentos referentes a denominação do arquivo, proteção de acesso, alocação e liberação de buffers, tratamento de erros são realizados pelo software independente de dispositivo. O driver é responsável pelas operações de mais baixo nível, que dependem do tipo de periférico específico.

O sistema operacional Linux suporta um grande número de periféricos. Para cada periférico deve existir um driver específico. Para o LInux existem dois tipos de dispositivos: orientados a caractere e orientados a bloco. Cada dispositivo é identificado por um maior número e um menor número. O maior número identifica o tipo de periférico e o menor número é usado para acessar o periférico específico, no caso de um disco, um inteiro entre 0 e 255.

Para um driver ser utilizado, necessita ser registrado e inicializado. Registrar um driver significa fazer a ligação com os arquivos necessários a sua execução. Na inicialização, são alocados os recursos necessários a execução do driver. Um driver pode ser compilado no kernel, neste caso é registrado quando o kernel executa as rotinas de inicialização, ou pode ser compilado como um módulo do kernel. Neste caso, sua inicialização é realizada quando o módulo é carregado.

O Linux identifica o final de uma operação de entrada e saída através de uma interrupção ou fazendo pooling. O funcionamento geral de um driver de impressora, que opera com pooling é apresentado a seguir.

```
void  imp (char c, int lp) {
   while (!READY_STATUS(status) && count > 0) {
        count - - ;
        }
    if (count == 0) return (TIMEOUT) ;
    out (char, lp) ;
   }
```

A variável count é utilizada para implementar um tempo máximo em que o driver testa o estado da impressora para enviar um novo caractere.

Exercícios

1. O que significa independência do dispositivo?
2. Escreva o pseudocódigo do procedimento executado pelo sistema operacional no tratamento de uma interrupção de relógio.
3. Escreva o pseudocódigo do procedimento executado pelo sistema operacional no tratamento de uma interrupção do disco.
4. Escreva os procedimentos *save_Context() e restore_context()* , executados pelo sistema operacional para, respectivamente, fazer salvamento e restauração de contexto. Considere que o sistema

possui dois registradores de uso geral (R0 e R1), dois registradores utilizados para gerência de memória (Mbase e Msize), respectivamente base e deslocamento, um registrador Stack Pointer (SP), um acumulador (AC) e um apontador de instruções (PC). Defina a estrutura de dados na qual são mantidas as informações sobre os processos (descritor de processos) e observe que o contexto do processo é salvo em seu registro descritor. Leve em conta a existência de uma operação *mov op1, op2* onde *op1* e *op2* podem ser variáveis ou registradores.

5. Descreva o funcionamento de um driver de disco.
6. Defina a estrutura de dados necessária para solicitação de uma operação para um driver de disco.
7. Quais são as ações executadas por um driver de disco no tratamento de uma interrupção?
8. Justifique a existência de buffers de entrada e saída, utilizados para conter dados (blocos) de disco nas operações de leitura e escrita.

3 Processos Concorrentes

Processos são programas carregados na memória do computador para execução, e podem ser classificados como leves e pesados. Este capítulo é dedicado ao estudo dos processos concorrentes. Apresenta o conceito de processos concorrentes, os estados dos processos, as transições de estado, as operações sobre processos concorrentes e mecanismos de especificação de concorrência. São também apresentados exemplos de programas concorrentes com fork no sistema LInux e o conceito de threads é ilustrado com a apresentação de programas concorrentes escritos com a biblioteca Pthreads.

3.1 Conceitos básicos

Os processadores executam instruções que representam algoritmos. Um processo é uma entidade abstrata, formada pelos recursos de hardware e pela execução das instruções referentes a um algoritmo sendo executado pelo processador. Um algoritmo é programado em uma linguagem de programação, por exemplo C, e o programa compilado (código objeto) é carregado na memória para execução. Pode-se classificar os processos em dois tipos: Processos pesados e Processos leves.

Os processos pesados são os processos tradicionais. Possuem uma thread inicial que começa a sua execução (ex. *main* de um programa C), executam um código seqüencial e normalmente são carregados do disco para execução. Processos leves (threads) são criados para permitir paralelismo na

execução de um processo pesado. As principais características dos processos leves são:

- Cada um roda um código seqüencial;
- Possuem sua própria pilha de execução e o seu próprio *program counter;*
- Compartilham o uso do processador;
- Podem criar processos (threads) filhos;
- Compartilham o mesmo espaço de endereçamento (dados globais).

Um processo (Pesado ou Leve) está no estado running se suas instruções estão sendo executadas pelo processador. Está em um estado ready se possui condições para ser executado e está esperando pelo processador, e está em um estado blocked se está a espera de alguma condição para rodar, por exemplo, está a espera de uma operação de entrada e saída. A figura 3.1 a seguir ilustra a transição de estados dos processos.

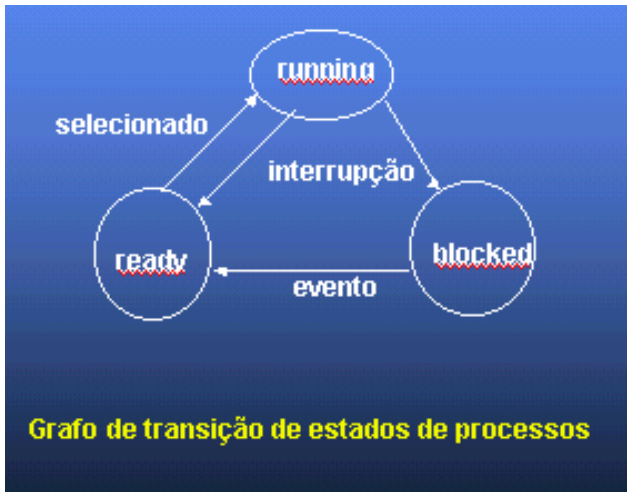

Fig. 3.1 – Grafo de transição de estados

Um processo passa do estado ready para o estado running quando é selecionado, ganha o processador e suas instruções começam a ser executadas. Passa de um estado blocked para ready quando ocorre o evento

pelo qual estava esperando, e passa de um estado running para blocked quando necessita esperar pela ocorrência de um evento (ex. Operação de entrada e saída). A transição de running para ready ocorre em sistemas que atribuem fatias de tempo do processador aos processos em execução. Quando este tempo termina, o processo perde o processador e passa para o estado ready, a espera novamente do processador.

3.2 Condições de Concorrência

Supondo que três processos compartilhem uma variável x, sobre a qual dois executam operações de modificação e de consulta, a ordem em que estas operações são feitas é importante. Se a variável x for modificada simultaneamente pelos dois processos, o valor impresso irá depender da ordem em que as operações de escrita forem realizadas. Por exemplo, se x = 5, na seqüência de operações

```
x = x + 10 ; (P0)
x = x + 15 ; (P1)
print  x ;  (P2)
```

o processo P2 irá imprimir o valor 30. Por outro lado, se o comando print for executado após a primeira atribuição, o valor impresso será 15. Assim, o valor da variável x que será impresso irá depender da ordem de execução dos comandos, sendo portanto indeterminado.

As condições de Bernstein[Bernstein 1966] são usadas para verificar se um programa é determinístico. O conjunto *read* e o conjunto *write* de uma aplicação A são respectivamente o conjunto das variáveis lidas (consultadas) por A e escritas por A. O conjunto *read* de uma aplicação B, R(B), é a união dos conjuntos *read* de todas as operações de B. Similarmente, o conjunto *write* de B, W(B), é a união dos conjuntos *write* de todas as operações de B. Por exemplo, se B for:

```
x = u + v;
```

y = x * w;

então R(B) = {u, v, x, w}, W(B) = {x, y}. Observe que x faz parte do conjunto *read* de B, R(B) e do conjunto *write* de B, W(P). As condições de Bernstein são:

Considerando dois processos P e Q, se

a) A intersecção do conjunto write de P, W(P) com o conjunto write de Q, W(Q) é vazia, e

b) A intersecção do conjunto write de P, W(P) com o conjunto read de Q, R(Q) é vazia, e

c) A intersecção do conjunto read de P, R(P) com o conjunto write de Q, W(Q) é vazia, então

a execução de P e Q é determinística.

Grafos de precedência

Um grafo de precedência é um grafo dirigido, acíclico onde os nós representam atividades seqüenciais e onde um arco de um nó i para um nó j indica que a atividade i deve terminar antes que a atividade j possa começar.

Um nó em um grafo de precedência pode representar um processo para o sistema operacional, ou pode representar uma única instrução, a ser executada por uma máquina paralela (granularidades diferentes). Considerando o seguinte programa seqüencial:

a = x + y; (s0)
b = z + 3; (s1)
c = a + b; (s2)
d = f + g; (s3)
e = d + 5 ; (s4)

h = e + c ; (s5)

O grafo de precedência deste programa seqüencial é o seguinte:

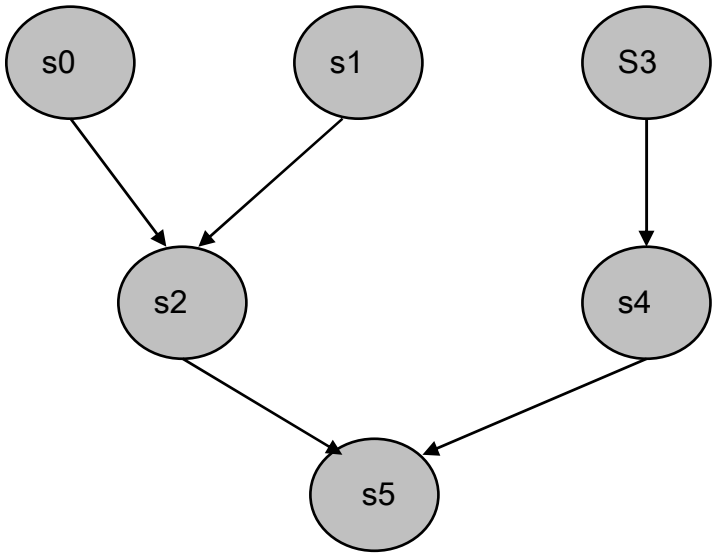

Fig. 3.2 - Grafo de processos para o programa seqüencial acima

Neste grafo de precedência, os comandos s0, s1 e s3 podem ser executados em paralelo. A execução de s2 somente pode começar depois do término de s0 e s1, a execução de s4 somente depois do término de s3. s5 somente pode ser executado após o final de s2 e s4.

Dois aspectos fundamentais ressaltam pela análise dos grafos de precedência: como expressar o paralelismo de um grafo e como sincronizar as execuções de maneira a obter-se determinismo na execução do programa. A seguir serão apresentados mecanismos de expressão do paralelismo e o capítulo seguinte será dedicado aos mecanismos de sincronização de processos.

3.3 Especificação de Concorrência

As primitivas *fork/join* join [Dennis and Van Horn, 1966], [Conway 1963] foram as primeiras notações de linguagem para especificar concorrência

A primitiva

fork v

inicia um novo processo que começa a execução na instrução com o label *v*. O novo processo compartilha o espaço de endereçamento do processo criador. O processo que efetuou a chamada *fork* e o processo criado continuam a execução em paralelo. A figura a seguir ilustra a operação *fork.*

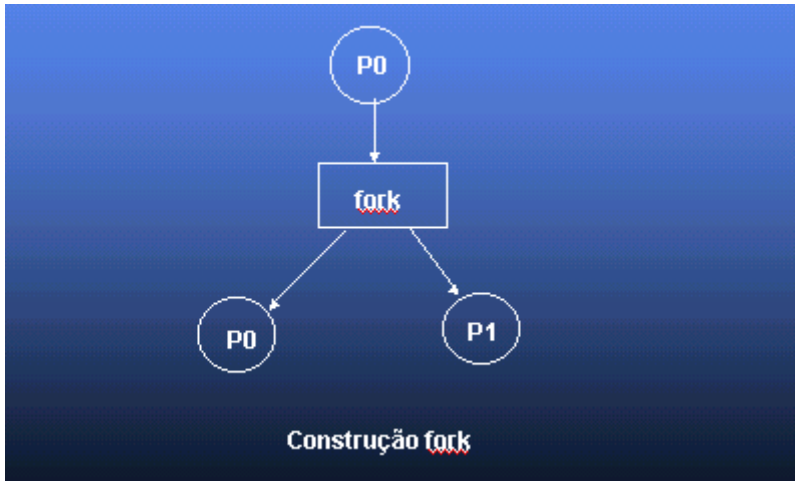

Fig. 3.3 – Visão esquemática de um fork

Na figura acima, o processo P0 executa a operação fork, criando o processo P1, dividindo a computação em duas. Uma que será executada pelo processo P1 e a do processo P0, que continua em paralelo com o processo criado. Considerando os comandos:

S0: a = 5 ; b = a + 7 ;

S1: c = 3 ; d = c + 2 ;

S2: e = 4 ; f = e / 2 ;

A execução paralela com o comando *fork* pode ser obtida com

```
        fork L0 ;
        fork L1 ;
        S0 ;
        go to Fim ;
L0:     S1 ;
        go to Fim ;
L1:     S2 ;
Fim:    exit ;
```

A execução começa com o processo que executará o comando S0, que inicialmente executa a operação *fork L0*; que cria um novo processo que começará sua execução no comando com o label *L0* e que executará o comando S1. Após a execução desta primitiva, o programa possui dois processos que executam em paralelo. O processo inicial, que executa o comando S0, continua a execução no comando seguinte e executa uma nova operação *fork* para criar um novo processo, que inicia no label L1 e executará o comando S2. Existirá, portanto, dependendo da velocidade de execução, três processos em execução paralela, S1, S2 e S3. A figura a seguir ilustra o fluxo de execução do programa acima.

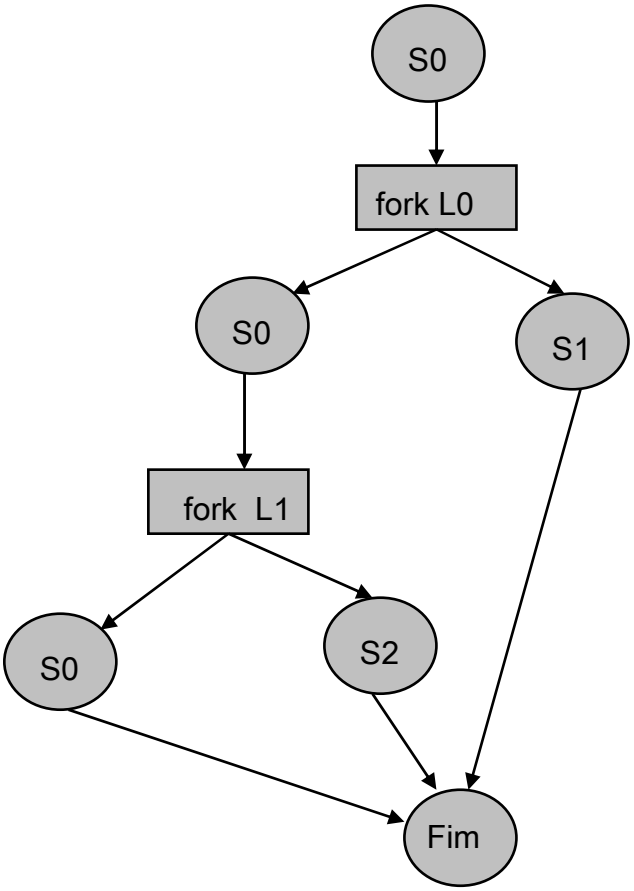

Fig. 3.4 - Programa paralelo com uso de fork

Existem situações nas quais um processo deve continuar somente após um conjunto de processos terem terminado. Por exemplo, no programa apresentado acima, se for necessária a execução de um comando S3, que soma todas as variáveis calculadas por S0, S1 e S2, o mesmo somente poderá ser executado ao final da execução destes comandos. A instrução

join counter, addr

definida por Conway, decrementa *counter* e o compara com zero. Se o valor for igual a zero, o processo executa a partir do endereço *addr,* caso contrário termina. Desta forma, a operação *join* combina várias execuções concorrentes em uma. Cada processo participante do grupo de processos

deve requisitar um join. O último processo a executar o *join* continua, os demais, como o valor de *counter* é maior que zero, terminam (figura abaixo).

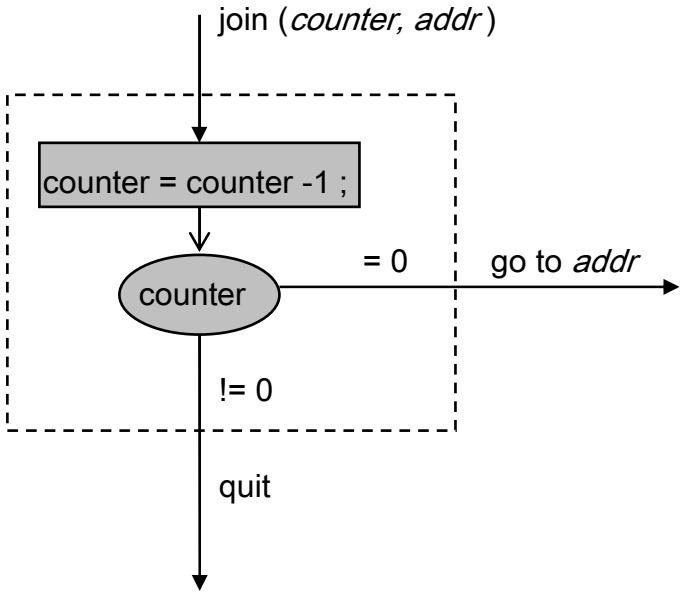

Fig. 3.5 - Construção join

Como vários processos podem chamar concorrentemente a operação *join*, a mesma deverá ser atômica. Isto significa que as operações que decrementam a variável e a que faz o teste devem ser executadas de maneira indivisível.

Na construção *join*, um processo termina sua execução quando o valor da variável *count* é diferente de 0. Isto é feito com uma operação *quit,* que destrói o processo que a executa. O exemplo a seguir ilustra o uso das construções *fork/join*. O programa calcula a soma dos elementos de dois vetores, A e B, e está programado em uma pseudolinguagem C. Um processo calcula a soma dos elementos do vetor A, outro dos elementos do vetor B e o último processo a terminar a soma, calcula a soma total.

```
main() {
```

```
      int FIM, L0, L1, count = 2 ;
      int soma0=0, soma1=0, somatotal=0 ;
      int N = 100 ;
      int i, j ;
      int A[1, N], B[1, N] ;
      fork L0 ;
      for (i=0; i<N; i++){
          soma0 = soma0 + A[i] ;
          }
          go to L1 ;
  L0: for (j=0; j<N; j++)
          Soma1 = soma1 + B[j] ;
  L1: join count, L2 ;
  L2: somatotal = soma0 + soma1 ;
}
```

No programa acima é criado um processo, que executa a partir do label L0, a soma do vetor B. O processo que efetuou a chamada *fork* soma os elementos do vetor A. Para o primeiro processo que executar a operação *join* o valor de count será maior que zero e o mesmo não executará a partir do rótulo FIM, executando, portanto, a operação *quit*, que o termina. O outro processo, conseqüentemente, calculará a soma total dos elementos dos dois vetores.

As operações *fork/join* podem aparecer em qualquer trecho de código, podendo fazer parte de loops e de comandos condicionais. Programas escritos desta forma se tornam difíceis de ser entendidos, pois é necessário saber quais procedimentos serão executados para saber os processos que irão ser criados no programa. Porém, se usados de forma disciplinada representam um excelente mecanismo de criação dinâmica de processos. O sistema operacional Linux implementa uma variação da primitiva *fork* apresentada anteriormente. A construção fork no sistema Linux possui a seguinte forma:

id = fork() ;

As características do *fork* no Linux são as seguintes:

- Duplica o processo que executa a chamada e os dois executam o mesmo código;
- Cada processo tem seu próprio espaço de endereçamento, com cópia de todas as variáveis, que são independentes em relação às variáveis do outro processo;
- Ambos executam a instrução seguinte à chamada de sistema;
- *id*, no retorno da chamada, contém, no processo pai, o identificador do processo filho criado;
- Para o processo filho o valor da variável *id* será zero;
- Pode-se selecionar o trecho de código que será executado pelos processos com o comando *if;*
- Não existe o comando join (a sincronização é feita com wait()), que bloqueia o processo que a executa até que um processo filho (criado pelo que executou a operação wait) termine.

Exemplo de programa com fork no Unix

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>


int main() {
int id ;
id = fork () ;
if (id != 0) {
    printf(“Eu sou o pai\n”) ;
```

```
    wait(0) ;
    }
else
  printf("Eu sou o filho\n") ;
}
```

No programa acima, escrito em C, é criado um processo com a chamada da primitiva *id = fork ()*. Para o processo pai, o que executou a chamada, o valor retornado na variável *id* é o número de criação do processo filho, que o identifica no sistema, e para o processo filho o valor da variável *id* é zero. Desta forma, para o processo pai o teste *if (id != 0)* é verdadeiro e são executados os comandos *printf* e *wait (0),* que o bloqueia até que o processo filho termine. Como para o processo filho o teste do *if* é falso, ele executa *printf* e a seguir termina. O seu término desbloqueará o processo pai, que estava bloqueado no wait, que também terminará.

O grafo a seguir

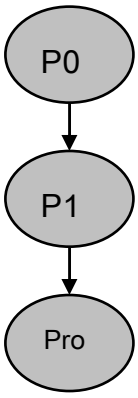

Fig. 3.6 - Grafo com três processos

representando três processos diferentes, indicando que P1 executa depois que P0 terminar e que P2 executará somente após o término de P1, pode ser programado com *fork* no Linux da seguinte forma:

```
#include <stdio.h>
```

```
#include <sys/types.h>

#include <signal.h>


int main () {

   int id0, id1 ;
   // código de P0
   id0 = fork () ;
   if (id0 != 0)  exit (0) ; // P0 termina
   else{
        //código de P1
        id1 = fork () ;
        if (id1 != 0) exit (0) ; //P1 termina
        else {
             // código de P2
             }
       }
   }
```

No programa acima é utilizada a primitiva *exit (),* que termina o processo que a executa, liberando os recursos alocados ao processo (memória, etc.).

O grafo de processos a seguir

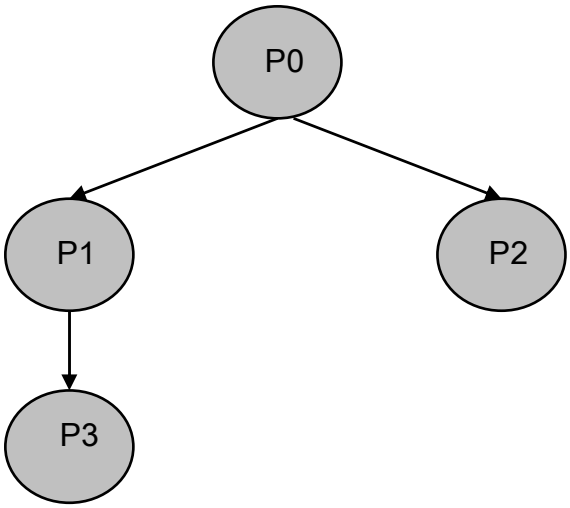

Fig. 3.7 - Grafo de processos com quatro processos

representando quatro processos, com *fork* no Linux pode ser programado como apresentado a seguir:

```
#include <stdio.h>
```

```
#include <sys/types.h>

#include <signal.h>


int main () {

   int id0, id1, id2 ;
   // código de P0
   id0 = fork () ;
   if (id0 != 0) {
      id1 = fork () ;
      if (id1 != 0) exit (0) ; // P0 termina
      else  {
             // código de P2
            }
      }
   else{
        //código de P1
        id2 = fork () ;
        if (id2 != 0) exit (0) ; //P1 termina
        else {
              // código de P3
             }
       }
   }
```

Construções CoBegin CoEnd

Esta construção, também chamada ParBegin ParEnd, foi criada por E. W. Dijkstra [Dijkstra1965]. É uma construção de mais alto nível, se comparada a construção fork, e mais fácil de ser utilizada. A forma geral é a seguinte:

cobegin

S1;

S2;

..

Sn;

coend;

Quando o cobegin é executado é criado um processo para executar cada um dos comandos entre o cobegin e o coend. Baseado na linguagem Algol, os comandos entre o cobegin/coend podem ser simples ou compostos. Comandos compostos são delimitados por begin/end. Quando todos estes

processos terminarem, o coend é executado, e o processo pai, o que executou o cobegin, recomeça a execução. Assim, todos os processos entre o cobegin e o coend são executados concorrentemente. A figura a seguir ilustra a construção cobegin/coend.

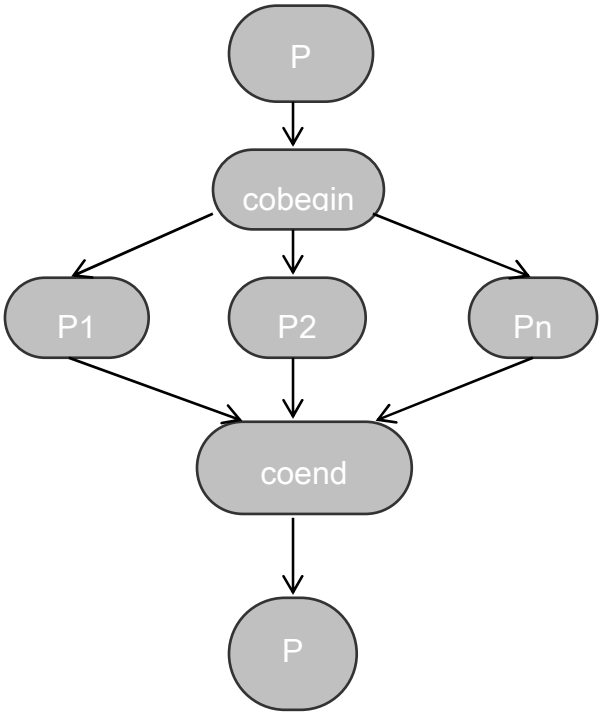

Fig. 3.8 - Construção cobegin/coend

O exemplo a seguir

```
Cobegin
        a (x, y, z) ;
        b (i, j) ;
        begin
```

c (r, s) ;

d (g, h) ;

end

Coend

é formado por dois comandos simples, a chamada do procedimento a (x, y, z) e do procedimento b (i, j) e por um comando composto, delimitado por begin/end, contendo as chamadas de procedimento c (r, s) e d (g, h). Os comandos simples serão executados em paralelo com o comando composto. No comando composto, serão executadas seqüencialmente as chamadas dos procedimento c e d. A figura a seguir ilustra a execução deste comando concorrente.

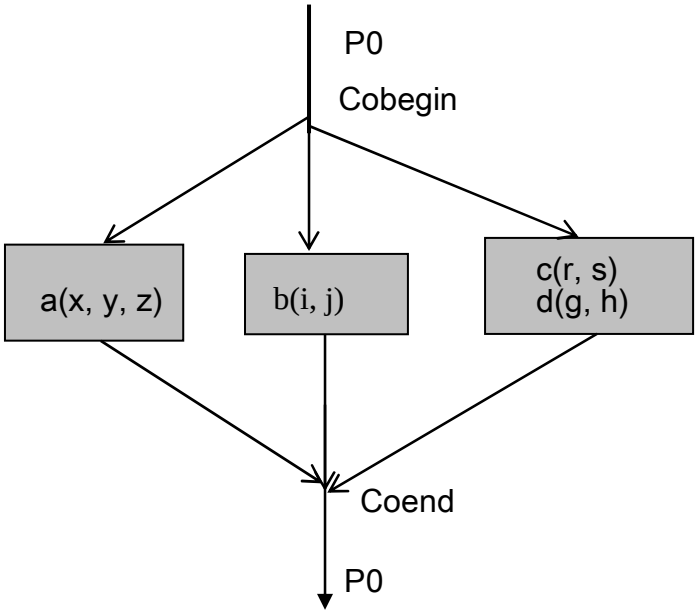

Fig. 3.9 – Funcionamento de Cobegin/Coend

O programa a seguir utiliza comando concorrente para fazer a cópia de um arquivo f para um arquivo g.

```
FILE f, g ;
char buffer0[4096], buffer1[4096] ;
```

```
int i, j, k ;
int main() {
      i = read (f, &buffer0, 4096) ;
      while (i > 0) {
            for (j=0; j<i; j++) buffer1[j] = buffer0[j] ;
            cobegin
                  k = write (g, &buffer1, i) ;
                  i = read (f, &buffer0, 4096) ;
            coend ;
                  }
}
```

No programa acima, enquanto um processo grava os ***i*** bytes no arquivo g, um outro processo faz a leitura dos próximos ***i*** bytes do arquivo f. Estas operações se repetem até que a cópia se complete.

Primitivas para manipulação de processos

O sistema operacional deve oferecer primitivas que permitam criação, destruição, bloqueio e sinalização de processos. Para a criação de processos as primitivas são do tipo *cobegin/coend* e *fork*, apresentadas anteriormente. Na primitiva *fork,* o processo pai executa concorrentemente com o processo filho. Com *cobegin/coend* o processo pai fica bloqueado no *coend* até que todos os processos concorrentes criados na construção *cobegin/coend* terminem. Com relação aos dados, na implementação Linux do comando *fork* o processo filho é uma cópia do processo pai e, a partir do *fork*, compartilham um subconjunto das variáveis (arquivos abertos, ...).

Término de processos

Um processo termina em três situações:

- Quando acaba a execução de seu último comando;

- Em circunstâncias especiais um processo pode acabar antes de executar o seu último comando (ex. endereço de um vetor fora dos limites);
- Um processo pode causar a morte de outros processos com primitivas especiais (ex. um processo está em um loop eterno e necessita ser terminado).

As primitivas que implementam o término de processos são:

- exit ()
    - O processo que a executa termina normalmente. Seus recursos são liberados pelo S.O.
- abort (id), kill (id)
    - id é a identificação do processo à terminar;
    - abort e kill são chamadas pelo pai do processo id;
    - para poder matar, o pai necessita conhecer a identidade do filho, que retorna na operação de criação.

O exemplo a seguir ilustra o uso da primitiva *kill* no sistema operacional Linux.

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
int main() {
int id ;
id = fork () ;
if (id != 0) {
    printf("Eu sou o pai\n") ;
    sleep(5) ;
    kill (id, SIGKILL) ;
    }
else{
```

```
        printf("Eu sou o filho\n") ;
       while(1) ;
      }
}
```

No programa acima o processo pai, após o *fork*, executa a primitiva *sleep(5)*, que o bloqueia durante cinco segundos. Transcorrido este tempo, o processo filho, que estava em um loop eterno (*while(1)*) é eliminado com a primitiva *kill (id, SIGKILL)*, executada pelo processo pai.

3.4 Processos leves (Threads)

Threads são processos leves, definidos para permitir paralelismo em um programa de aplicação. Um programa com múltiplos pontos de execução possui tantas threads de execução quantos forem estes pontos. As threads possuem sua própria pilha de execução e o seu próprio *program counter,* cada uma roda um código seqüencial, definido no programa de aplicação (processo pesado) no qual foram definidas, compartilham o uso do processador, podem criar processos (threads) filhos e compartilham o mesmo espaço de endereçamento (dados globais) do processo pesado ao qual pertencem. Portanto, quando uma thread altera um valor compartilhado, todas percebem esta mudança e os arquivos abertos por uma thread são disponível as outras threads do mesmo processo pesado. Para manter a coerência dos dados, mecanismos de sincronização devem ser utilizados pelo programador.

As threads sofrem transição entre os estados ready, running, blocked. Não existe estado " swapped" , isto é, uma thread não pode, durante a sua execução, ser retirada da memória principal e gravada em disco para liberar memória e permitir que um novo programa possa ser carregado para execução. A operação de swap não existe para uma thread, somente para todo o processo pesado. Outra característica é que a terminação do processo pesado termina todas as threads no processo.

Implementação de Threads

Threads podem ser implementadas por bibliotecas. Existem três modelos básicos para a implementação das bibliotecas de threads , que identificam a forma de representação das threads no sistema operacional. Os modelos são: N:1 (many-to-one), 1:1 (one-to-one), e M:N (many-to-many).

Modelo N:1

Neste modelo, a implementação é feita em nível de usuário, por uma biblioteca que implementa os procedimentos de gerenciamento de threads. Estes procedimentos incluem primitivas para criar, destruir, bloquear, etc., além de implementarem os algoritmos de escalonamento. Desta forma, a gerenciamento de threads é realizado pela aplicação (biblioteca de threads). O núcleo do sistema não conhece threads, executa o escalonamento do processo pesado no qual as threads foram definidas. O tempo de processador do processo pesado é distribuído, pelo escalonador de threads da biblioteca de threads, entre as diferentes threads que fazem parte da aplicação. Assim, o chaveamento de threads não requer privilégios do modo kernel, sendo específico da aplicação. As principais características das bibliotecas de threads a nível usuário são:

- Quando uma thread faz uma chamada de sistema, todo processo pesado será bloqueado. No entanto, para a biblioteca de threads, a thread rodando estará ainda no estado running;
- O estado das threads é independente do estado do processo pesado.

Vantagens e desvantagens de threads no nível de usuário

Vantagens

- Chaveamento de threads não envolve o kernel, não troca de modo de execução (usuário/sistema);
- Escalonamento pode ser dependente da aplicação: escolha de algoritmo adequado a cada aplicação;
- Pode rodar em qualquer SO, bastando para tal a instalação da biblioteca.

Desvantagens

- Maioria das chamadas de sistema são bloqueantes e o kernel bloqueia todo processo pesado, apesar de outra thread do processo poder continuar;
- O núcleo só associa processos pesados a processadores. Duas threads do mesmo processo pesado não podem rodar em processadores diferentes.

Modelo 1:1

Neste modelo, a gerência de threads é feita no núcleo do sistema. Assim, a cada primitiva de manipulação de threads corresponde uma rotina de biblioteca que empacota os parâmetros da chamada e executa uma troca de contexto, passando a execução para o núcleo do sistema operacional. O núcleo mantém informações de contexto sobre threads e processos pesados. Desta forma, as operações de criação, destruição, etc. de threads são realizadas pelo sistema operacional. A cada thread no programa de aplicação corresponderá uma thread no núcleo. Portanto, o chaveamento de threads envolve o núcleo.

Vantagens e desvantagens da implementação de threads no núcleo do sistema

Vantagens

- O núcleo pode escalonar simultaneamente várias threads do mesmo processo pesado em diferentes processadores;
- O bloqueio é em nível de thread e não de processo pesado;
- As rotinas do SO podem ser também multithreaded.

Desvantagens

- Chaveamento de threads dentro do mesmo processo pesado envolve o núcleo o que representa um impacto significativo no desempenho do sistema.

Modelo M:N

Neste modelo, cada processo de usuário pode conter M threads, que são mapeadas em N threads sistema. O gerenciamento de threads é realizado a nível usuário, assim como o escalonamento e a sincronização, sendo que o programador ajusta o número de threads sistema que serão utilizadas para mapear as threads usuário. Existe portanto uma distinção entre threads do usuário e threads do sistema. As threads do usuário são invisíveis ao SO e oferecem a interface de programação. As threads do sistema identificam uma unidade que pode ser escalonada em um processador, e suas estruturas são mantidas pelo núcleo.

pthreads (POSIX Threads)

A biblioteca pthreads (POSIX Threads) define uma interface de programação que permite o desenvolvimento de programas C, C++ utilizando threads.
Com pthreads, a criação de uma thread é feita com a primitiva

*int pthread_create(pthread_t *thid, const pthread_attr_t *atrib,*
*void *(*funcao), void *args);*

que pode aparecer em um bloco qualquer de comandos. O parâmetro *thid,* do tipo `pthread_t,` contém, no retorno da chamada, o identificador da thread

criada, *atrib* permite ao programador definir alguns atributos especiais (política de escalonamento da *thread*., escopo de execução da thread, isto é, modelo 1:1 ou modelo N:1, etc.). Se for passado o argumento `NULL`, são utilizados os atributos *default*. *args* `contém` o endereço de memória dos dados passados como parâmetros para a *thread* criada. Se a operação falha, a thread não é criada e o valor retornado indica a natureza do erro:

- EAGAIN: O sistema não possui os recursos necessários para criar a nova thread.
- EFAULT: o nome da função a ser executada (thread) ou attr não é um ponteiro válido.
- EINVAL: attr não é um atributo inicializado.

No caso de sucesso da operação é retornado o valor 0.

Em C/C++, o conjunto de instruções a ser executado por uma *thread* é definido no corpo de uma função.

A primitiva

*pthread_join(pthread_t thid, void **args);*

permite a uma *thread* se bloquear até que a *thread* identificada por `thid` termine. A thread pode retornar valores em *args*: 0 em caso de operação bem sucedida, um valor negativo em caso de falha, e um valor de retorno calculado pela thread. Se o programador não deseja valor de retorno, pode passar NULL como segundo argumento.

O programa a seguir, escrito em C, demonstra o uso destas primitivas para a implementação de um programa concorrente.

```
#include <pthread.h>
#include <stdio.h>

void * Thread0 ()
        {
          int i;
          for(i=0;i<100;i++)
                printf(" Thread0 - %d\n",i);
```

```
        }
void * Thread1 ()
        {
          int i;
          for(i=100;i<200;i++)
                printf(" Thread1 - %d\n",i);

        }

int main(){
   pthread_t t0, t1;

   pthread_create(&t0, NULL, Thread0,NULL) ;
   pthread_create(&t1, NULL, Thread1,NULL) ;
   pthread_join(t0,NULL);
   pthread_join(t1,NULL);
   printf("Main ....\n");
}
```

O programa acima é formado por três threads, uma thread to, que executa o código da função Thread0 que imprime os valores entre 0 e 99, uma thread t1, que executa o código da função Thread1 que imprime os valores entre 100 e 199, e a thread main, que executa o código da função main do programa. A execução começa na thread main, que cria a thread to, com a chamada da primitiva pthread_create. A partir da execução desta primitiva, a thread main e a thread criada passam a compartilhar o uso do processador. A seguir, a thread main cria a thread t1 (passando então a existir três threads em execução) e se bloqueia, com a primitiva pthread_join(t0,NULL), a espera que a thread t0 termine. Após o término da thread t0, a thread main se bloqueia, com a operação pthread_join(t1,NULL), e será acordada com o término da thread t1. O uso da primitiva pthread_join é indispensável, pois impede que a thread main continue a execução e termine, pois as threads definidas em um processo pesado são eliminadas se o processo pesado termina, o que ocorre com o término da thread main.

Exercícios

1. Compare o uso de threads e de processos pesados no desenvolvimento de programas concorrentes.
2. Faça o grafo de precedência do programa concorrente apresentado a seguir.

```
    count1 = 2 ;
    count2 = 2 ;
    s1 ;
    fork L1 ;
    s2 ; s4 ; s7 ; go to L4 ;
L1: s3 ; fork L2 ; s5 ; go to L3 ;
L2: s6 ;
L3: join count2 ;
    s8 ;
L4: join count1 ;
    s9 ;
```

3. Considerando os comandos apresentados a seguir, faça o grafo de precedência e escreva o programa concorrente com o uso de fork/join.

```
a = b + c ; (s0)
d = e + f ; (s1)
g = d + k ; (s2)
h = g + a ; (s3)
m = n + h ; (s4)
x = m + h ; (s5)
y = x + g ; (s6)
z = y + 3;  (s7)
```

4. Defina um conjunto de no mínimo 8 comandos (ex. S0: a = b+2 ;), construa o grafo de precedência e escreva o programa correspondente, com o uso de fork/join.
5. Fale sobre as possibilidades de implementação de threads, relacionando vantagens/desvantagens de cada modelo.
6. Descreva as ações executadas pelo sistema operacional Linux na execução da primitiva fork.
7. Defina um grafo com no mínimo 5 processos e o implemente com fork no sistema operacional Linux.
8. Relacione vantagens da programação com threads em relação ao uso de fork na programação concorrente.
9. Discuta uma possível forma de implementação da primitiva fork.

10. Na primitiva fork, tal como implementada no Linux, não existe compartilhamento de dados entre o processo pai e o processo filho, após o fork. Apresente um exempo de programa em que este problema é tratado com o uso de pipe.

4 Escalonamento

Conceitos Básicos de escalonamento. Algoritmos de escalonamento. Escalonamento de processos em máquinas paralelas. Escalonamento no Linux e no Windows NT .

4.1 Conceitos Básicos

Em sistemas com multiprogramação o uso da CPU é compartilhado entre os diferentes programas em execução, criando uma ilusão de paralelismo (pseudoparalelismo). Isso ocorre porque o tempo do processador é distribuído entre os processos que estão em execução. Para o usuário é como se cada processo tivesse a sua própria CPU, porém com uma velocidade menor do que a CPU real. Assim, pode-se imaginar que cada processo tem sua própria CPU virtual. Nos sistemas uniprocessador, existe um processo running (o que detém o processador) e os demais processos aptos a executar esperam pela CPU em uma fila (a ready list) que contém os processos aptos a rodar. O sistema possui ainda outras filas:

- filas de dispositivos: contém os processos esperando I/O;
- filas de eventos: contém os processos bloqueados a espera de eventos (ex. passagem de tempo).

Process Control Block (PCB)

Os processos são representados no sistema por um registro descritor, que mantém as informações referentes ao processo. Informações típicas são:

- estado do processo: ready, running, blocked, etc.;
- program counter;
- registradores da CPU;

- informação de escalonamento (dependem do algoritmo - prioridades, ponteiros para filas, etc.);
- Informação para gerência de memória: valor de registradores base e limite, registradores que apontam para a tabela de páginas, etc.;
- Informação de contabilizarão: tempo de CPU utilizada, limites, etc.;
- informação de estado de I/O: lista de dispositivos alocados ao processo, arquivos abertos, etc.

O sistema operacional gerencia uma tabela de descritores, cada um descrevendo um processo em execução. Quando um processo é submetido, um registro descritor é criado e incorporado a tabela de descritores. Quando o processo termina, seu registro descritor é eliminado.

As filas existentes no sistema (ready list, filas de periféricos, etc.) são igualmente gerenciadas pelo sistema operacional. As figuras abaixo exemplificam a fila de processos aptos à rodar, na qual estão os processos 0,5 e 9, a fila do disco0, com os processos 7 e 13 e a fila do disco1, com o processo 12.

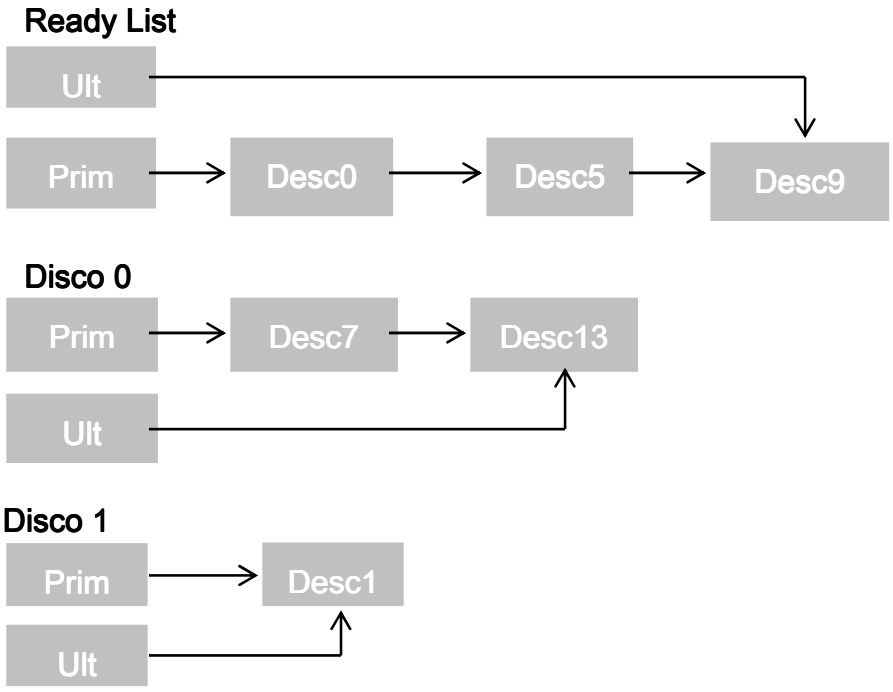

Fig. 4.1 Filas gerenciadas pelo Sistema Operacional

Escalonamento da CPU

Com multiprogramação vários processos residem na memória e a CPU é compartilhada pelos mesmos. A execução de um programa é uma seqüência de execução e espera por I/O. Assim, para otimizar o uso da CPU, quando o programa faz I/O a CPU é entregue a outro processo. Escalonamento é a troca da CPU entre os vários programas na memória prontos para executar.

Os escalonadores são processos do sistema operacional que implementam a Política de Escalonamento. Podem ser de longo termo e de pequeno termo.

Escalonadores de Longo Termo

Considerando-se, por exemplo, um sistema operacional do tipo Batch, no qual os processos submetidos pelos usuários são colocados em uma lista de execução, no disco, os escalonadores de longo termo selecionam processos desta lista para serem carregados na memória para execução. Uma vez na memória, o processo é inserido na ready list, de onde será selecionado pelo escalonador de pequeno termo para ganhar o processador. Escalonadores de longo termo normalmente não existem em sistemas interativos (Unix, Dos, Windows NT, ...).

Escalonadores de Pequeno Termo

Os escalonadores de pequeno termo (Escalonadores da CPU) selecionam um processo, entre os aptos (que estão na ready list), e o coloca para rodar. A diferença entre estes dois tipos de escalonadores é a freqüência de execução. O escalonador de pequeno termo executa muito mais freqüentemente que o de longo termo. Por exemplo, a cada 20 mseg o escalonador de pequeno termo seleciona um processo para execução, enquanto que o de longo termo executa ao término de um processo, para colocar um novo na memória.

Dispatcher

O dispatcher é o componente do escalonador responsável pela entrega da CPU para o processo selecionado pelo Escalonador de Pequeno Termo (Escalonador da CPU). Para tal executa as seguintes ações:

- carrega os registradores de uso geral do processo selecionado (armazenados na pilha ou no descritor do processo) nos registradores de máquina (restauração de contexto);
- carrega o Program Counter (dispara a execução).

4.2 Algoritmos de escalonamento

Os algoritmos implementam as políticas de escalonamento. Os objetivos dos algoritmos são:

- Minimizar o tempo de resposta;
- Maximizar o número de processos executados por unidade de tempo;
- Distribuir uniformemente o tempo de CPU.

Algoritmos de Scheduling

A seguir serão apresentados os principais algoritmos de escalonamento.

Algoritmo 1: Fila de processos (First-Come-First-Served)

Neste algoritmo, o primeiro processo que requisita a CPU é o primeiro que a recebe.

1. Processo pronto para executar entra no final da lista ready;
2. Quando a CPU é liberada é alocada para o primeiro da lista.

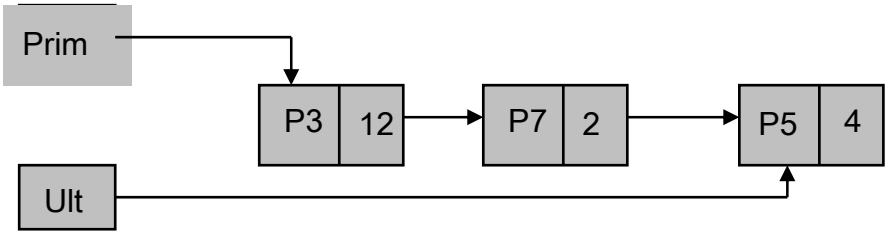

Fig. 4.2 - Ready List com FCFS

A figura acima mostra três processos (P3, P7 e P5), com seus respectivos tempos de execução, inseridos na ready list em ordem de chegada. Assim, a execução de P5 começara após o término de P3 e P7, ou seja, ficará na fila 14 unidades de tempo. O tempo médio de execução é de 6 unidades de tempo.

Este algoritmo é inadequado para sistemas interativos, pois processos CPU bound, isto é, que possuem muita necessidade de processamento e pouca de operações de entrada e saída, ao ganharem o processador o monopolizam por muito tempo. Os demais processos na fila são retardados e o tempo de resposta se torna inviável a um sistema em que a resposta deve ser imediata. No exemplo acima, os tempos de espera na fila são:

P3: 0

P7: 12

P5: (12 + 2) = 14

Algoritmo 2: Menores processos primeiro (SJF – Shortest Job First)

O SJF associa a cada processo o seu próximo tempo de ocupação de CPU. Quando a CPU esta livre é atribuída ao processo que irá utilizar a CPU por menos tempo. Este algoritmo é baseado em prioridades, pois cada processo

tem uma prioridade associada (próximo tempo de CPU) e a cpu é alocada ao processo de maior prioridade (o que necessita de menor tempo).

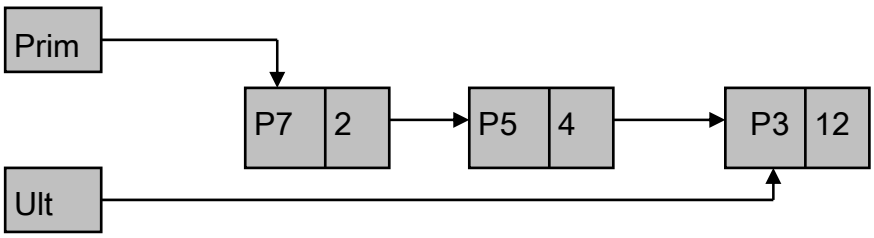

Fig. 4.3 Ready List com SJF

Para este algoritmo, os tempos de espera na fila são:

P3: (2+4) = 6

P7: 0

P5: 2

Em relação ao algoritmo anterior, o tempo de espera na fila é menor. O processo que fica mais tempo na fila espera 6 unidades de tempo, enquanto que no FCFS o maior tempo de espera é de 14 unidades de tempo.

O problema do SJF é determinar próximo tempo de CPU para cada processo. Uma solução adotada é fazer uma análise histórica da execução de cada processo. Com base nesta análise, determinar o próximo tempo de uso de CPU para cada processo, e colocá-lo na ready list na posição correspondente a este valor (esta prioridade).

Nos algoritmos baseado em prioridades existe um campo de variação (0..N) que indica a prioridade do processo. Em alguns sistemas 0 é a maior, outros 0 é a menor. Nos algoritmo baseado em prioridades um problema que pode surgir é o de postergação indefinida de processos com baixa prioridade. Para solucioná-lo, uma possibilidade é de tempos em tempos incrementar a prioridade dos processos de baixa prioridade, garantindo, desta forma, suas execuções em um tempo finito.

Outro aspecto importante é a chegada na ready list de um processo com prioridade mais alta do que a do processo que está rodando. As alternativas são:

a. Um processo chegando na fila ready tem sua prioridade comparada com a do processo que está rodando. Se for maior, ganha a cpu, caso contrário fica na fila. Neste caso, diz-se que o algoritmo é preemptivo.

b. Um processo chegando na fila ready sempre é colocado na fila, de acordo com a sua prioridade. Algoritmo não preemptivo.

Em sistemas de tempo real, nos quais os tempos de resposta são muitas vezes fundamentais para alguns processos, os algoritmos devem ser preemptivos.

Algoritmo 3: Round-Robin

Neste algoritmo, a lista de processos aptos a executar é uma fila circular e existe um tempo máximo que o processo pode ficar com o processador. O funcionamento do algoritmo é o seguinte:

- Pegar o primeiro processo da ready list;
- O processo executa por uma fatia de tempo, é preemptado e reinserido no final da ready list;
- O primeiro processo da ready list é selecionado e ganha o processador.

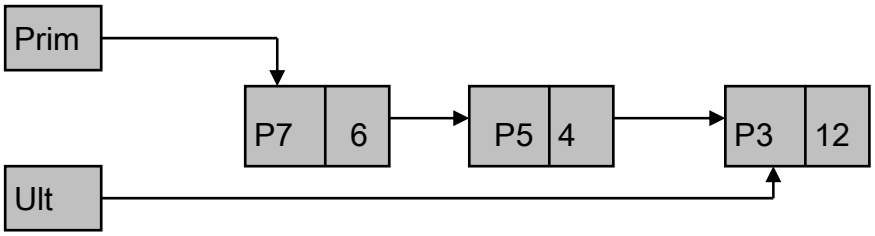

Fig. 4.4 - Ready List com Round Robin

Supondo que a fatia de tempo seja de 2 unidades de tempo, o processo P7 ganhará o processador, executará por uma fatia de tempo e será colocado no final da ready list. O processo P5 ganhará o processador, executará por uma fatia de tempo e será colocado no final da ready list, após o processo P7. O processo P5 sai da ready list após duas execuções e o processo P7 após 3 execuções. Mesmo restando somente o processo P3 na ready list e faltando ainda três execuções, o mesmo executará uma fatia de tempo e será inserido como último na ready list. Como será também o primeiro, ganhará novamente o processador. Isto significa que a ação do escalonador será sempre a mesma, independentemente do número de processos na ready list.

Este algoritmo é adequado aos sistemas interativos, pois evita que um processo monopolize o uso do processador. Um aspecto a considerar é o overhead determinado pelo tempo salvamento de contexto e de restauração de contexto. Este tempo depende:

- Da velocidade de memória;
- Do número de registradores;
- Da existência de instruções especiais. Por exemplo, uma instrução que permita mover o valor de todos os registradores para o descritor do processo.

Tempos típicos de salvamento/restauração de contexto variam de 10-100 microsegundos. Um outro problema na implementação deste algoritmo é a definição do quantum. Um quantum muito grande degenera o algoritmo, aproximando-o do algoritmo de fila (FCFS). Quantum muito pequeno aumenta o tempo gasto com troca de contexto. Valores comuns de time slice variam de 10-100 miliseg.

Algoritmos que utilizam Múltiplas filas

Uma outra solução a ser empregada é a utilização de múltiplas filas de escalonamento. Estes algoritmos possuem as seguintes características:

- Os processos são classificados em grupos;
- Existe uma fila para cada grupo;
- Cada fila pode ter seu próprio algoritmo;
- Uma fila pode ter prioridade absoluta sobre as demais filas com prioridades menores.

Para evitar o problema de postergação indefinida de uma fila, pode existir uma fatia de tempo atribuída a cada fila. Desta forma, assegura-se que processos em filas de baixa prioridade ganhem o processador.

Uma alternativa a este algoritmo é a existência de múltiplas filas com reavaliação. Neste caso, os processos se movem entre as filas, variando portanto as prioridades. Nestes algoritmos, deve ser definido o número de filas, critérios para movimentar processos para filas de mais alta prioridade, critérios para movimentar processos para filas de mais baixa prioridade e critérios para determinar em qual fila o processo deve iniciar.

4.3 Escalonamento de Processos em Máquinas Paralelas

As arquiteturas paralelas do tipo MIMD [Flyn 1972] são compostas por processadores que operam de forma independente. Dois tipos de máquinas, na classificação de Flyn são: máquinas paralelas com memória comum e máquinas paralelas com memória distribuída. Com memória comum, existe uma memória global que pode ser acessada por qualquer processador, figura 4.5, a seguir.

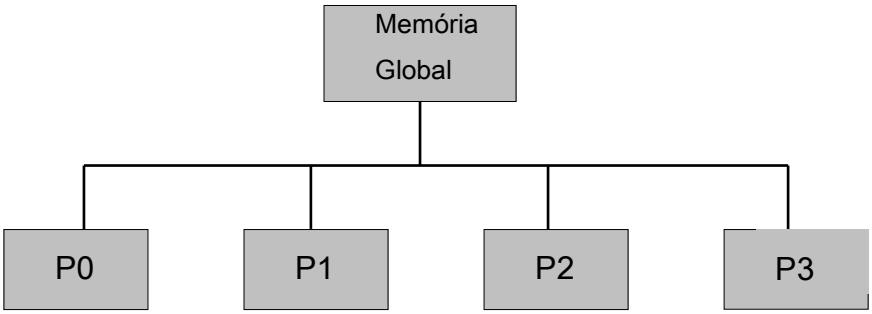

Fig. 4.5 - Máquina paralela com memória comum

A figura acima exemplifica uma máquina paralela com memória comum, formada por quatro processadores, que compartilham uma memória global. Uma máquina sem memória comum é mostrada na figura 4.6 a seguir.

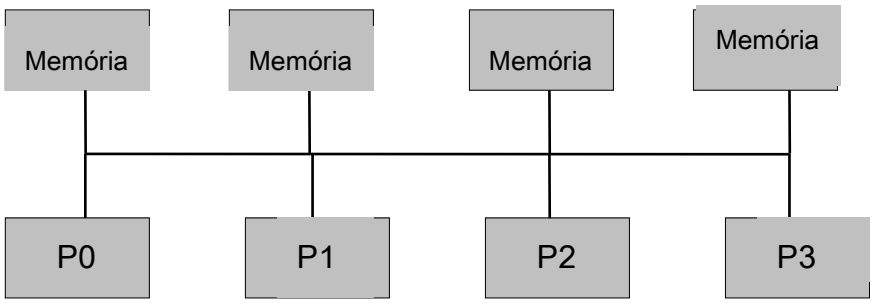

Fig. 4.6 - Máquina Paralela Sem Memória Comum

Na figura acima, cada processador possui a sua memória e não pode acessar a memória de outro processador.

Escalonamento em sistemas com memória compartilhada

Os sistemas com memória compartilhada, podem possuir na memória comum uma única fila de processos aptos à rodar, compartilhada pelos processadores. O conjunto de processadores que compartilham memória

ficam sob o sob controle de um único sistema operacional. Os processadores são tratados como recursos e o sistema operacional associa processos a processadores, sob demanda, isto é, sempre que um processador fica livre o sistema operacional lhe atribui um processo. Esta associação pode ser estática ou dinâmica. É estática quando o processo roda no mesmo processador e é dinâmica quando em tempos diferentes o processo pode ocupar diferentes processadores. As abordagens podem ser mestre/escravo, mestre flutuante e simétrico.

Mestre-escravo: As funções do kernel rodam sempre em um processador, os demais processadores rodam somente processos de usuário. O mestre faz escalonamento e se o escravo precisa serviço (chamada de sistema), faz pedido ao mestre

- Vantagens
    - Simples de projetar e implementar;
    - Permite execução paralela de um único processo.
- Desvantagens
    - serializa as chamadas de sistema porque somente o mestre as executa;
    - O mestre normalmente não fica com muita carga;
    - O mestre pode se tornar um gargalo
    - a falha do mestre para todo o sistema.

Mestre flutuante: Somente um processador por vez executa o código do escalonador, que pertence ao código do sistema operacional.

- Vantagens:
    - Todos os processadores são completamente utilizados;
    - Permite execução paralela de um processo .

- Desvantagens
    - O gargalo do mestre continua.

Configuração simétrica: o sistema operacional pode ser acessado por múltiplos processadores simultaneamente, sendo que os dados compartilhados somente podem ser acessados por um processador por vez.

- Vantagens:
    - Permite execução paralela de uma única task;
    - Configuração mais versátil.
- Desvantagens:
    - Mais difícil de projetar e implementar;
    - Requer kernel reentrante (permite que mais de um processo execute o código do sistema operacional simultaneamente, em posições distintas);
    - Muitas chamadas de sistema são executadas concorrentemente.

Escalonamento em sistemas sem memória compartilhada

Uma aplicação paralela, é classificada quanto ao número de processos, como estática ou dinâmica. Em uma aplicação estática o número de processos é conhecido no início de sua execução e não se modifica até o término do processamento. O grafo de processos não se altera. Aplicações dinâmicas são aquelas que o número de processos podem variar durante a execução. No primeiro caso, os processos são alocados aos nós processadores da rede durante a inicialização e permanecem no mesmo nó

até o final de sua execução. Não ocorre migração de processos e não são criados novos processos.

No segundo caso, a alocação dos processos criados dinamicamente deve ser feita de maneira a garantir um bom desempenho global do sistema (*throughput*). A carga de trabalho deve ser distribuída de uma forma balanceada para evitar-se que coexistam no sistema nós ociosos e nós sobrecarregados. Pode-se utilizar a migração de processos para obter este balanceamento de carga.

Em [Casavant, T. L. 1988] é apresentada uma classificação dos algoritmos de escalonamento. De acordo com esta classificação, um algoritmo global pode ser estático ou dinâmico. Ele é estático quando o processador onde o processo será executado é definido antes da execução. Depois de realizada a alocação o processo permanece neste processador até o final de sua execução. Nos algoritmos dinâmicos a decisão do processador é feita à medida que os processos são criados. Para melhorar o balanceamento de carga, pode ocorrer a migração de processos.

Outro importante aspecto da classificação é relacionado à escolha do processador onde será executado o processo. Em algoritmos centralizados a informação sobre a carga dos processadores é mantida em um processador do sistema, e a decisão de alocação também é centralizada. Em algoritmos distribuídos a informação sobre a carga dos processadores é distribuída ao longo dos processadores do sistema, a decisão de alocação ocorre também de forma distribuída. Quanto às decisões, elas podem ser cooperativas quando envolvem os outros processadores do sistema ou não cooperativas quando não envolvem os outros nós.

Os algoritmos são constituídos de quatro elementos básicos, que implementam a política de informação, a política de transferência, a política de localização e a política de negociação. A política de informação é responsável pelo armazenamento dos dados relativos a carga dos processadores. A política de transferência é responsável pelas tarefas de transferência entre processadores. A política de localização determina em que processador do sistema o processo será criado ou para que

processador o processo será migrado. A política de negociação representa a interação entre os processadores durante o processo de decisão.

4.4 Escalonamento no Linux e no Windows NT

A seguir serão apresentados os algoritmos de escalonamento utilizados nos sistemas operacionais Linux e Windows NT .

Escalonamento no sistema operacional Linux

Em seu desenvolvimento, os requisitos definidos para o escalonador do Linux foram:

- Apresentar boa performance em programas interativos, mesmo com carga elevada;
- Distribuir de maneira justa o tempo da CPU;
- Ser eficiente em máquinas SMP;
- Possuir suporte para tempo real.

Em relação ao primeiro requisito, tradicionalmente se classifica os processos em I/O bound e CPU bound. Por exemplo, um editor de textos é tipicamente um processo I/O bound, pois se caracteriza pelo altíssimo número de operações de entrada e saída e pela interação com o usuário, podendo também ser classificado como interativo. Processos CPU bound, ao contrário, usam muita CPU e fazem poucas operações de I/O. De maneira a oferecer um bom tempo de resposta, o Linux favorece os processos interativos, em relação aos processos CPU bound. Cada processo no Linux possui uma prioridade, que é recalculada dinamicamente. O escalonador entrega a CPU para o processo que possui a maior prioridade. O escalonador do Linux é preemptivo. O kernel verifica a prioridade do processo que passa para o estado running. Se for maior do que a do processo que estava rodando, o kernel chama o escalonador, que irá selecionar um novo processo para execução (o que se tornou running).

O Linux considera dois tipos de processos, cujas características definem o algoritmo de scheduling usado: processos interativos(time-sharing) e tempo real. Para os processos interativos o algoritmo é o Round Robin. Para processos de tempo real os algoritmos são FCFS e Round Robin. Processos de tempo real possuem prioridade sobre todos os demais processos. Se existe um processo de tempo real para rodar, ele sempre ganha o processador primeiro. Processos de tempo real Round Robin possuem uma prioridade, e o escalonador escolhe o processo de maior prioridade. Processos FCFS executam até terminarem ou até se bloquearem.

Nas versões que antecederam a 2.6, o escalonamento do Linux sofria alguma justificadas críticas:

a) Aumento do tempo de execução do escalonador proporcionalmente ao aumento do número de processos no sistema;

b) Uma ready list global em máquinas SMP.

O escalonador deve selecionar o processo de maior prioridade para entregar o processador. Para fazer isso, mantém uma fila de processos, cada um possuindo uma prioridade, baseada em uma prioridade básica (Nice) e em seu comportamento em tempo de execução. Uma vez selecionado, o processo ganha a CPU por uma fatia de tempo (timeslice). Quando o timeslice chega a zero, o processo é marcado como expirado. O escalonador, quando chamado, seleciona o processo não expirado com a maior prioridade. Quando todos os processos estiverem com seu timeslice expirado, o escalonador recalcula a prioridade de todos e então seleciona o próximo a rodar. Este recálculo de prioridades representa tempo de execução para o escalonador. Aumentando o número de processos no sistema, aumenta o tempo necessário para o escalonador recalcular as prioridades. Para aplicações com um número pequeno de processos este tempo não é significativo. Para aplicações com um número muito grande de processos, este tempo poderia penalizar enormemente o desempenho do sistema.

Referente ao segundo aspecto das críticas ao escalonador do Linux, nas versões anteriores a 2.6, a existência de uma ready list global nas

máquinas SMP, os problemas são de duas naturezas: primeiro a necessidade de acesso a mutuamente exclusivo a ready list por parte dos processadores. Se um processador está acessando a ready list os demais necessitam esperar até que a mesma seja liberada. Para um número elevado de processadores este bloqueio dos processadores pode acarretar em uma queda considerável no desempenho do sistema. Segundo, um processo interrompido, por exemplo, porque solicitou uma operação de entrada e saída, quando esta se completa é recolocado na ready list, que é global, e poderá ser executado por um processador diferente daquele no qual estava rodando previamente. Isto acontecendo, dados do processo porventura existentes em caches não poderão mais ser usados, provocando uma sobrecarga de processamento, influenciando negativamente na performance.

Escalonamento no Kernel 2.6

As principais características do escalonador do kernel 2.6, tornado público no final de 2003, chamado de $O(1)$ scheduler são:

- Tempo constante de execução para selecionar um processo para rodar, independentemente do número de processos;
- Boa performance com programas interativos;
- Eficiente em máquinas SMP;
- Afinidade de processador, com uma fila de aptos para cada processador;
- Suporte a tempo real.

O escalonador do kernel 2.6 possui uma fila de todos os processos prontos para executar. Esta fila é implementada por dois arrays, um com os processos ativos, elegíveis para rodar e outro expirado, com processos temporariamente inelegíveis. Quando um processo termina sua fatia de tempo, é colocado na fila dos expirados, e aguarda até que isso aconteça com todos os processos do array ativos. Então, o array expirado se torna ativo, com uma simples troca de apontadores. A figura a seguir esquematiza uma fila de processos aptos. Em máquinas SMP, existe uma fila de aptos para cada CPU.

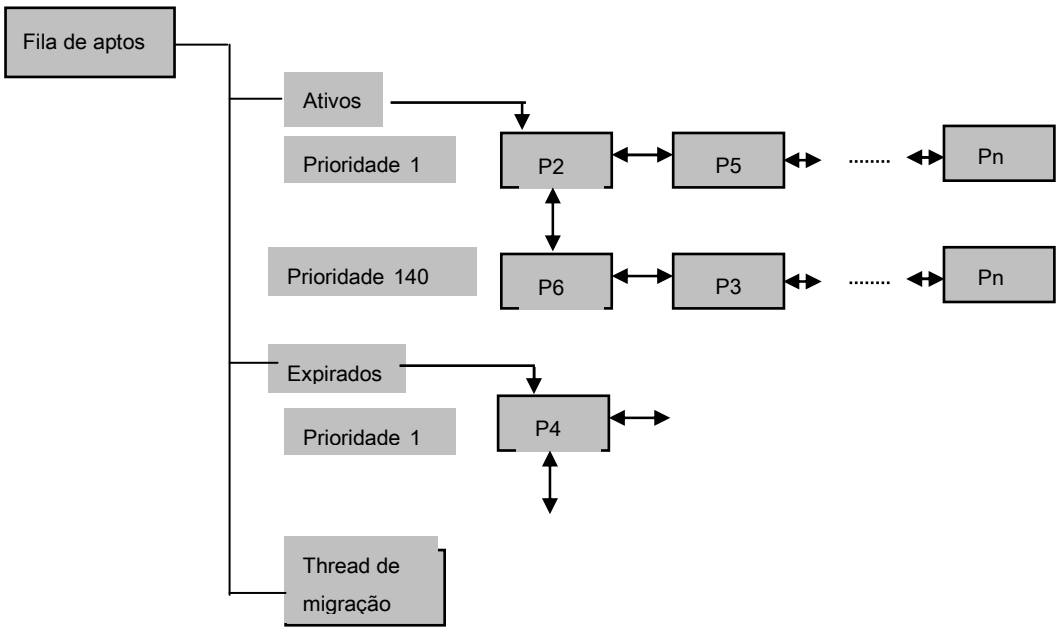

Fig. 4.7 - Fila de aptos no Linux

O escalonador possui 140 níveis de prioridade, sendo que quanto menor o nível, maior é a prioridade. Prioridades de 1 a 100 são para processos de tempo real. De 101 a 140 para demais processos de usuário (interativos ou não interativos).

Os processos de tempo real podem ser FIFO ou Round Robin, e possuem uma prioridade estática. Processos FIFO executam até voluntariamente liberarem a CPU. O nível de prioridade é mantido e não são preemptados. Processos Round Robin recebem uma fatia de tempo e executam até terminar esta fatia de tempo. Quando todos os processos de um dado nível de prioridade tenham terminado uma outra fatia de tempo é atribuída e eles continuam rodando, no mesmo nível de prioridade.

A prioridade dos processos de usuário é a soma de sua prioridade, que é o valor de seu *nice* e seu bônus dinâmico de prioridade, que varia de +5 a – 5. Nos níveis de prioridade 101 a 140, os processos de usuário possuem uma fatia de tempo de 20 ms. Esgotado este tempo, o próximo processo da fila de mesma prioridade ganha o processador (Round Robin).
Para o cálculo da prioridade dos processos de usuário, é avaliado o seu nível de interatividade. Os processos são classificados, pelo escalonador, em interativos e não interativos pela observação de seu comportamento.

Processos interativos são aqueles que ficam bloqueados por longos períodos esperando por I/O, executam rapidamente e ficam novamente esperando por I/O. O escalonador calcula a média de tempo bloqueado de um processo e a utiliza para calcular o bônus de prioridade do processo.

Assim, os processos são classificados em interativos e não interativos e a idéia é aumentar a prioridade de processos interativos (I/O bound) e diminuir a prioridade de processos não interativos (CPU bound).

Finalmente, em cada processador roda uma *thread de migração*, que tem a função de movimentar processos de um processador para outro, para balancear o uso dos processadores. Como existe uma *ready list* por processador, a idéia é evitar que existam processadores ociosos enquanto outros estejam sobrecarregados. A *thread de migração* é chamada explicitamente, quando o sistema esta desbalanceado, e periodicamente, a cada tick do relógio.

Linux I/O scheduler

No linux, as operações de leitura (READ) são síncronas e bloqueantes, porque existe a suposição de que o processo que executou a operação de leitura necessita dos dados para continuar, e as operações de escrita (WRITE) são assíncronas e não-bloqueantes.

Por este motivo, as operações de leitura possuem prioridade em relação as operações de escrita. Além disso, as requisições são classificadas, de forma a minimizar a distância de deslocamento do cabeçote do disco. Para evitar postergação indefinida, a cada requisição é atribuído um deadline de atendimento. O escalonador do kernel 2.6 implementa duas políticas: deadline scheduler e antecipatório scheduler(AS)

Deadline Scheduler

O escalonador atribui a cada processo um tempo limite de atendimento e implementa duas filas: uma fila de READ, com um deadline de 500 ms e uma fila de WRITE, com um deadline de 5 seg. Quando a requisição é submetida, é inserida na fila apropriada (READ ou WRITE), na posição correspondente a

sua requisição. O escalonador dispara as requisições, seguindo a ordem das filas. Se um deadline expira, então o escalonador dispara as requisições seguindo a ordem da fila, garantindo a execução das requisições cujo deadline expirou.

Esta política assegura que as operações de posicionamento da cabeçote (seek) são minimizadas (pela classificação das requisições) e o deadline garante que não haverá postergação indefinida de requisições. Outro aspecto importante é o aumento da interatividade, com atribuição um deadline menor e com a priorização das requisições das operações READ.

Antecipatório Scheduler

O Antecipatório Scheduler (AS) tenta antecipar futuras operações de leitura de um processo. Para isso, mantém algumas estatísticas de cada processo e tenta ver se, após um read, um outro não ocorrerá, e espera um certo tempo antes de executar a requisição seguinte na fila.

Quando um READ se completa, o escalonador não executa a próxima requisição na fila. Espera 6ms por uma nova requisição do mesmo processo. Se ocorrer, a mesma será atendida. Para muitas aplicações, esta espera evita a ocorrência de inúmeras operações de posicionamento dos cabeçotes (seek). Se o processo não executa outra operação READ, este tempo é perdido.

Escalonamento no sistema operacional Windows NT

Um processo rodando representa uma instância de uma aplicação, e possui os recursos necessários à sua execução. Cada processo é formado por um conjunto de threads, que é a unidade de execução no sistema operacional Windows NT. Cada processo possui ao menos uma thread, a thread primária que é criada quando o processo é carregado para execução. Para alguns processos, outras threads são criadas, de maneira a explorar o paralelismo da aplicação. A memória alocada para cada processo é proporcional a

quantidade de threads que o processo possui. Cada thread possui a sua própria pilha de execução, seu contexto de execução (representado pelos registradores da CPU) e a sua prioridade.

Escalonamento de threads

No Windows NT o escalonador utiliza múltiplas filas de processos aptos a rodar, e os processos interativos (I/O bound) possuem prioridade sobre os CPU bound.

O escalonamento de threads no Windows NT é baseado em prioridades. A cada thread nos sistema é atribuído um nível de prioridade, que varia de 0 a 31, sendo que 0 representa o menor e 31 o maior. A prioridade 0 é atribuída a uma thread especial do sistema, chamada zero thread, que é responsável por zerar as páginas livres no sistema. Somente esta thread pode receber a prioridade 0. As prioridades são divididas em duas classes:

- Real time: prioridades de 16 a 31;
- Normal: prioridades de 0 a 15.

Existe ainda uma classe especial chamada idle, a de mais baixa prioridade. Threads nesta classe somente executam quando não existirem outras no sistema. Por exemplo, uma thread que faz monitoramento de carga no sistema poderia executar somente quando não existirem outras, de maneira a não interferir na performance. Neste caso, a classe idle deveria ser especificada para esta thread.

Para atribuir a CPU a uma thread, o escalonador escolhe a de maior prioridade. Por exemplo, existindo threads de prioridade 31, a primeira da fila ganha o processador. Como se trata de uma thread da classe real time, executará até terminar ou até esperar por um evento. Então, existindo outra thread de prioridade 31 esta será escolhida para ganhar o processador. Não existindo, a primeira thread com prioridade 30 será selecionada. Desta forma, as threads são escolhidas de acordo com a prioridade. Threads com prioridade normal (0 a 15) recebem uma fatia te tempo. Esgotado o tempo, ou

quando a thread necessita esperar por um evento, uma outra thread será selecionada, sempre terão preferência as threads de maior prioridade.

Cada thread ao ser criada recebe uma prioridade base. Para processos com prioridade entre 16 e 31 esta prioridade não se altera. Processos com prioridade entre 0 e 15 possuem sua prioridade ajustada em tempo de execução. Processos que retornam operações de I/0 recebem um bônus de aumento de prioridade, que depende do periférico (ex. 1para disco e 6 para teclado). Após operações de sincronização, recebem também um aumento de prioridade, dependendo da natureza do processo (ex. 2 para líder de sessão e 1 para os demais processos).

Para processos na classe normal, no sistema Windows 2000 Professional (NT 5.0), a fatia de tempo é de 20 ms (para favorecer a interatividade). Para o Windows 2000 Server (NT 5.0), a fatia de tempo é de 120 ms. A figura a seguir exemplifica a fila de escalonamento do sistema operacional Windows NT .

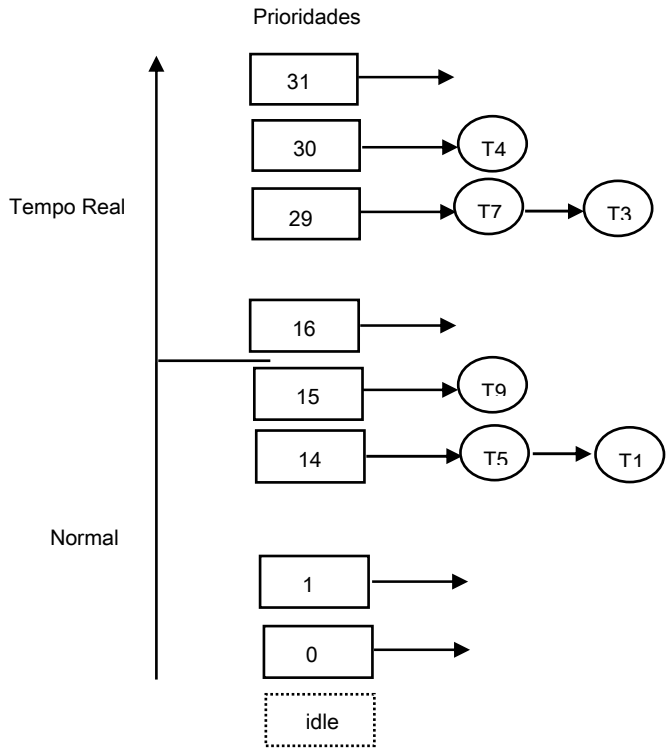

Fig. 4.8 - Representação da fila de aptos no Windows

Escalonamento em máquinas SMP

Durante a instalação do kernel do Windows 2000 (NT 5.0), a presença de mais de um processador é detectada e o suporte ao multiprocessamento é carregado. Existe somente uma fila de processos prontos para rodar, e sua estrutura é a mesma, com as mesmas classes e níveis de prioridade para uma máquina com um único processador ou com vários processadores. No entanto, a existência de múltiplos processadores permite paralelismo na execução das threads. O escalonador seleciona a thread de mais alta prioridade para rodar em cada um dos processadores. Além disso, o Windows 2000 implementa o conceito de afinidade, que define o processador específico no qual a thread deve executar. A afinidade pode ser *hard* , que indica a existência de uma dependência que obriga a que a thread rode sempre no processador especificado, ou *soft*, na qual o sistema tenta executar a thread no processador no qual havia sido executada previamente, para aproveitamento de dados da execução anterior que podem se encontrar em cache, o que implica em um ganho de performance.

Exercícios

1. algoritmo de escalonamento FCFS é adequado a um sistema de tempo compartilhado? Por quê?
2. Considere um sistema operacional multiprogramado no qual existem três filas scheduling:

   0: de mais alta prioridade na qual rodam os processos do sistema operacional.

   1: de prioridade intermediária na qual rodam servidores especializados.

   2: de mais baixa prioridade, na qual rodam os programas de usuários.

   Sabendo que cada uma das filas possui prioridade absoluta sobre as inferiores (isso é, somente roda um processo de prioridade 2 se não houver nenhum na fila de prioridade 0 nem na fila de prioridade 1), que

na fila de prioridade 0 algoritmo utilizado é o FCFS e nas demais é o round-robin:

i. Escreva o pseudocódigo do procedimento de tratamento da interrupção do relógio.

ii. Escreva o pseudocódigo do procedimento que seleciona um processo para execução.

iii. Cite duas situações em que cada procedimento é chamado.

3. Alguns sistemas operacionais possuem um conjunto de buffers, cada um com tamanho igual ao de um bloco de disco, e que serve de área de armazenamento. Supondo que o bloco que contém " b" não esteja na memória, uma operação de leitura **READ (f, &b, nbytes)** é tratada pelo sistema operacional. da seguinte forma:

- existe uma lista encadeada de blocos livres, de onde o SO seleciona um bloco para conter o bloco a ser lido do disco;
- o endereço desse bloco é colocado na requisição de IO fabricada pelo SO e encadeada na fila de requisições do periférico;
- o SO seleciona um novo processo para execução;
- o controlador do periférico executa a transferência do bloco que contém o dado indicado na operação de leitura para o bloco alocado previamente pelo SO, no conjunto de buffers;
- o controlador gera uma interrupção, indicando que terminou a transferência.

Sabendo que:

- o endereço onde o dado deve ser armazenado e o no. de bytes definido na operação de leitura do usuário também fazem parte da requisição fabricada pelo SO;
- que o sistema possui duas filas de scheduling (0: alta prioridade e 1: baixa prioridade);
- que o algoritmo de scheduling utilizado nas duas filas é o round robin;

- que os processos que têm seu IO completado possuem sua prioridade aumentada.

    a) Escreva o pseudocódigo do procedimento que trata a interrupção do periférico.

    b) Supondo a existência de uma única fila de scheduling, em que os processos não possuem prioridade, reescreva o pseudocódigo do algoritmo de tratamento da interrupção do periférico.

4. Considerando as características de um sistema operacional de tempo real *hard,* composto por três processos, de mesmo nível de prioridade, ativados por três sensores diferentes (cada um associado a um processo), qual(is) o(s) algoritmo(s) de escalonamento mais adequado(s)? Justifique.
5. Considere a seguinte afirmação: " em um sistema operacional multiprogramado interativo, diminuindo o tempo de espera nas filas dos processos CPU bound (que necessitam de mais CPU) se diminui o tempo médio de execução dos processos" . Esta afirmação está correta? Justifique sua resposta apresentando, para quatro processos, a ordem de chegada na ready list, o tempo de execução, o tempo de espera e o tempo médio de execução de cada processo.
6. Justifique a necessidade da existência da tabela de descritores de processos em sistemas multiprogramados. Esta tabela é necessária em sistemas monoprogramados? Por quê?
7. Compare os escalonadores do Linux e do Windows NT.
8. Compare os algoritmos de escalonamento dos sistemas operacionais Linux e Windows NT para máquinas SMP.

5 Sincronização de Processos

Neste capítulo são estudados princípios de Concorrência, algoritmos de exclusão mútua, semáforos, spin-lock, região crítica condicional, região crítica e monitores. Os problemas clássicos de sincronização, buffer limitado, jantar dos filósofos, barbeiro dorminhoco, leitores e escritores, são programados com o uso de semáforos da biblioteca Pthreads e a linguagem C.

5.1 Princípios de Concorrência

O programa apresentado a seguir, programado em C e com a biblioteca Pthreads, implementa o problema clássico de produtores e consumidores, com buffer circular. No exemplo, um processo produtor gera valores inteiros e os coloca no buffer, de onde o processo consumidor os retira.

```
#include <pthread.h>
#include <stdio.h>

#define N 10

pthread_t th0,th1;


int buffer[N], addrprod = 0, addrcons = 0 ;


void * produtor(){

  int i, p=50  ;

  for(i=0; i<20; i++)
     {
      while(((addrprod + 1) % N) == addrcons) {}
       p++;
       buffer[addrprod] = p ;
       addrprod= ( addrprod + 1 ) % N ;
     }
```

```
  }

void * consumidor(){

  int i, c ;

  for (i=0; i<20; i++)
    {
      while(addrprod == addrcons) {}
      c = buffer[addrcons] ;
      printf(" Consumi o valor %d da posição %d \n",c, addrcons) ;
      addrcons= ( addrcons + 1 ) % N ;
    }
}

int main(){


   pthread_create(&th0, NULL, produtor, NULL) ;
   pthread_create(&th1, NULL, consumidor, NULL) ;
   pthread_join(th0,NULL);
   pthread_join(th1,NULL);
}
```

No programa acima, a thread main, após criar a thread produtor e a thread consumidor se bloqueia, com as primitivas pthread_join, a espera que as threads criadas terminem. Após a morte das threads produtor e consumidor a thread main termina.

Supondo que a primeira thread a ganhar o processador seja a consumidor, ela executa o comando while(addrprod == addrcons) {}. Como a igualdade é verificada, o que significa que a posição do buffer na qual a thread produtora deve colocar um dado é a mesma que a thread consumidora deve consumir, portanto, o buffer está vazio, esta thread fica em loop no while durante o tempo em que estiver com o processador.

Quando a thread produtora ganha o processador, ela executa o comando while(((addrprod + 1) % N) == addrcons) {}. Como o teste é falso, a thread deposita na posição addrprod o valor da variável p e incrementa addrprod.

Existindo elementos no buffer, o teste feito pela thread consumidora, no comando while, é falso. Portanto, são executadas as instruções seguintes ao while, e após o loop do comando for.

O buffer circular é implementado com o uso da operação que utiliza o resto da divisão inteira. Por exemplo, se addrcons possui o valor 3, e N possui o valor 10, a próxima posição a ser acessada pelo consumidor é

Addrcons = (3 + 1) % 10) , ou seja, 4.

Se addrcons possui o valor 9, a próxima posição será ((9 + 1) % 10)), ou seja, a posição 0. Observe que em um buffer de 10 elementos, os índices variam de 0 a 9.

O programa a seguir é formado por duas threads , que compartilham uma variável inteira. O processo to adiciona o valor 5 à variável, e o processo t1 adiciona o valor 2.

```
#include <pthread.h>
#include <stdio.h>


pthread_t tid0,tid1;

long a=0, b, c ;

void * t0(){

     long i ;

     for (i=0; i<1000000; i++){
           a = a + 5 ;
           }
     printf("Encerrei a t0 %d\n",sizeof(int));
     }

void * t1(){
     long i ;

     for (i=0; i<1000000; i++)
         {
          a = a + 2;
          }
     printf("Encerrei a t1\n");
     }


int main(){
   pthread_create(&tid0, NULL, t0, NULL) ;
   pthread_create(&tid1, NULL, t1, NULL) ;
   pthread_join(tid0,NULL);
   pthread_join(tid1,NULL);
   printf("O valor de a e: %d\n",a);
```

```
}
```

No programa acima, as threads t0 e t1 compartilham a variável a. Considerando que após a compilação deste programa, o código assembler gerado para a thread t0, na operação de atribuição de um novo valor a variável *a* seja

```
load a ; // carregar o valor de a no acumulador
add 5 ; // adicionar o valor 5 ao acumulador
store a ; // armazenar o valor do acumulador na variável a
```

e para a thread t1 seja

```
load a ; // carregar o valor de a no acumulador
add 2 ; // adicionar o valor 2 ao acumulador
store a ; // armazenar o valor do acumulador na variável a
```

Se o valor de a é 60, com a seguinte seqüência de operações, o valor de ***a*** será inconsistente.

T0: t0 executa *load a* e perde o processador: o valor do acumulador (60) é salvo.
T1: t1 ganha o processador e executa *load a*: o valor do acumulador é 60
T2: t1 executa *add 2*: o valor do acumulador é 62.
T3 t1 executa **store a**: o valor do acumulador é armazenado na variável a (62).
T4: t0 ganha o processador, o valor do acumulador é restaurado (60) e executa a operação *add 5*: o valor do acumulador passa a ser 65.
T5: t0 executa *store a*: o valor do acumulador é armazenado na variável a (60).

Desta forma, uma atualização foi perdida, o valor da variável *a,* que deveria ser 67, é 65.

Seção Crítica de Código

É um trecho de código no qual os processos executam operações de modificação em dados compartilhados, e que, portanto, não pode ser executado em paralelo. No exemplo acima, para o processo t0 a seção crítica de código é

a = a + 5 ;

e para o processo t1 é

a = a + 2 ;

Causa do problema:

O acesso simultâneo à variável por t0 e t1 ocasiona uma inconsistência na variável a. Isto significa que o acesso a uma seção crítica de código deve ser feito de forma mutuamente exclusiva.

Definição geral do problema da seção crítica

- sistema com N processos, N > 1;
- cada processo executa seu código próprio;
- os processos compartilham dados variáveis, de qualquer tipo;
- cada processo possui SC' s, onde atualizam os dados compartilhados;
- a execução de uma SC deve ser de forma mutuamente exclusiva no tempo.

O modelo de solução para o problema é a utilização de um protocolo de entrada e saída na seção crítica. Assim, para acessar dados compartilhados o processo executa um código de entrada na seção crítica. Ao final, executa um

código de liberação da seção crítica, que deve permitir aos outros processos entrarem na sua seção crítica, como exemplificado abaixo.

entry-section
código-seção-crítica

exit-section

A entry-section deve garantir que somente 1 processo por vez execute a sua seção crítica e a exit-section permite que outro processo entre na seção crítica.

Requisitos para solução do problema da seção crítica

Para resolver o problema da seção crítica, as seguintes condições necessitam ser obedecidas:

a) exclusão mútua: somente um processo por vez é permitido entrar na seção crítica;

b) progresso: deve ser permitido a um processo entrar na sua seção crítica se nenhum outro processo está usando a seção crítica.

c) espera limitada: um processo não pode ficar indefinidamente esperando para entrar na seção crítica, enquanto outros processos, repetidamente, entram e saem de suas respectivas seções críticas.

Além disso, deve-se considerar que

- os processos possuem velocidades indeterminadas;
- não se deve fazer suposições sobre a velocidade relativa dos processos.

5.2 Algoritmos de Exclusão Mútua

A seguir serão apresentados os algoritmos de exclusão mútua utilizados para resolver o problema da seção crítica.

Soluções por SW para 2 processos

algoritmo 1

Os processos compartilham uma variável inteira, TURNO, que indica qual processo pode entrar na seção crítica. Se turno tiver o valor 0, é a vez do processo 0, se o valor for 1, deve entrar o processo 1.

```
var
   TURNO: integer;          % variável compartilhada; 0 ou 1
    EU, OUTRO: integer;     % constantes locais, com valores opostos
(0,1 e 1,0)
repeat
   while (TURNO !=EU)  do {}; % mutexbegin
      código-da-seção-crítica;
   TURNO := OUTRO       % mutexend
            resto-do-código;
until false;
```

Este algoritmo não resolve o problema da seção crítica de código, pois requer uma alternância na execução. Por exemplo, se o valor inicial de TURNO é 0, primeiro deve executar o processo 0, depois o 1, depois o 0, e assim por diante. Isso significa que a condição de progresso não é observada. Assim, se o processo 1 executa, passa a vez para o processo 0. Se o processo 0 não deseja ainda entrar na seção crítica, e o processo 1 deseja entrar novamente, o 1 não poderá entrar pois a vez é do 0. A seguir será apresentado o algoritmo 1, programado com pthreads

```
#include <pthread.h>

pthread_t tid0,tid1;


int turn = 0 ;
int shared ;

void * p0 (){

  int i ;

  for(i=0; i<10; i++)
     {
       while(turn != 0 ) {} //entra se turn = 0
       shared =shared + 50 ;
       printf("Thread1: INCREMENTEI \n");
       turn = 1 ;
     }
}
```

```
void * p1 (){

int i ;

  for (i=0; i<10; i++)
    {
       while(turn != 1 ) {} //entra se turn = 1
       printf("Thread2: SHARED: %d  \n",shared);
       turn = 0 ;
    }
}

int main(){
  pthread_create(&tid0, NULL, p0, NULL) ;
  pthread_create(&tid1, NULL, p1, NULL) ;
  pthread_join(tid0,NULL);
  pthread_join(tid1,NULL);
}
```

Soluções por SW para 2 processos

Algoritmo 2

O problema do primeiro algoritmo é a alternância de execução, o que não satisfaz a condição de progresso. O processo que está saindo da seção crítica passa a vez para o outro processo, sem saber se o mesmo deseja entrar. No algoritmo a seguir, a variável TURNO é substituída por um vetor do tipo boolean de duas posições, flag. Se flag[i] é verdadeiro significa que o processo i está na seção crítica. Um processo somente pode executar a sua seção crítica se o valor de flag correspondente ao outro processo for falso.

```
// algoritmo 2
var
   var flag[0..1] of boolean // variável compartilhada, inicializada
com false
repeat
      while flag[j] do {};          % mutexbegin
      flag[i] := true ;
      código-da-seção-crítica
      flag[i] := false              % mutexend
      resto-do-código;
until false;
```

O algoritmo acima não resolve o problema da seção crítica, pois os dois processo podem entrar ao mesmo tempo na seção crítica. Considerando a seguinte seqüência de execução:

To: p0 ganha o processador e encontra o valor de flag[1] false e perde o processador antes de executar a operação flag[i] = true;

T1: p1 ganha o processador e encontra o valor de flag[0] false; executar a operação flag[1] = true; e entra na seção crítica. Se p1 perde o processador dentro da seção crítica, p0 ganha o processador e executa a instrução flag[0] = true e vai para a seção crítica. Logo, os dois processos estão na seção crítica, e portanto o algoritmo não implementa exclusão mutua.

Algoritmo 2 programado com pthreads:

```
#include <pthread.h>

pthread_t tid0,tid1;
int shared = 0 ;
int flag [2];

void * p0(){

  int i ;
  for(i=0; i<1000000; i++)
       {
       while(flag [1] == 1 ) {} /* entra se flag [1] = 0 */
       flag[0] = 1 ;
       shared =shared + 5 ;
       flag [0] = 0 ;
       }
}
void * p1(){

  int i ;
  for (i=0; i<1000000; i++)
       {
       while(flag [0] == 1 ) {} /* entra se flag [0] = 0 */
       flag [1] = 1 ;
       shared = shared + 2 ;
       flag [1] = 0 ;
       }
}
main(){
  flag [0] = 0 ;
  flag [1] = 0 ;
  pthread_create(&tid0, NULL, p0, NULL) ;
  pthread_create(&tid1, NULL, p1, NULL) ;
  pthread_join(tid0,NULL);
  pthread_join(tid1,NULL);
  printf("O valor final de shared e:%d\n", shared);
}
```

Soluções por SW para 2 processos

Algoritmo 3

O problema do algoritmo 2 é que o processo pode perder o processador antes de mudar o valor da variável flag, que indica se o mesmo está na seção crítica ou não. No algoritmo a seguir, o processo primeiro modifica o valor de flag para indicar que o mesmo deseja entrar na seção crítica e posteriormente testa a posição que indica que o outro processo está na seção crítica. Se o outro processo não estiver, ele sai do loop e entra na seção crítica

```
// algoritmo 3
var
   var flag[0..1]  // variável compartilhada, inicializada com false
repeat
      flag[i] := true ;
      while flag[j] do {};    // mutexbegin
      código-da-seção-crítica;
      flag[i] := false        // mutexend
      resto-do-código;
     until false;
```

O algoritmo acima possui as seguintes propriedades:

a) satisfaz a exclusão mútua, pois somente 1 processo por vez pode entrar na seção crítica;

b) não satisfaz a condição de progresso pois se

T0: P0 faz flag[0] = true; e perde o processador

T1: P1 faz flag[1] = true;

Agora, tanto flag[0] quanto flag[1] possuem o valor verdadeiro. Portanto, P0 e P1 ficam em um loop eterno.

Algoritmo 3 programado com pthreads

```
#include <pthread.h>

pthread_t tid0,tid1;
```

```
int shared = 0;
int flag [2];

void * p0(){

  int i ;
  for(i=0; i<1000000; i++)
     {
       flag[0] = 1 ;
       printf("Thread0%d\n", shared) ;
       while(flag [1] == 1 ) {} /* entra se flag [1] = 0 */
       shared =shared + 5 ;
       flag [0] = 0 ;

     }
}
void * p1(){

  int i ;
  for (i=0; i<1000000; i++)
    {
       flag [1] = 1 ;
       printf("Thread1%d\n", shared) ;
       while(flag [0] == 1 ) {} /* entra se flag [0] [0] = 0 */
       shared = shared + 2 ;
       flag [1] = 0 ;
    }
   }

int main(){
  flag [0] = 0 ;
  flag [1] = 0 ;
  pthread_create(&tid0, NULL, p0, NULL) ;
  pthread_create(&tid1, NULL, p1, NULL) ;
  pthread_join(tid0,NULL);
  pthread_join(tid1,NULL);
  printf("O valor final de shared e:%d\n", shared);
}
```

Soluções por SW para 2 processos

O algoritmo a seguir, desenvolvido pelo matemático T. Dekker em 1962, foi a primeira solução correta apresentada para o problema da exclusão mútua entre dois processos

```
var flag[0..1] of boolean;  // variável compartilhada, inicializada
com false
    var  turn of integer;
    repeat
      flag[i] := true ;
      turn := j ;
      loop
         exit when not(flag[j]) or turn = i);   // mutexbegin
```

```
        código-da-seção-crítica
    flag[i] := false;                    // mutexend
        resto-do-código;
until false;
```

O algoritmo satisfaz as propriedades de exclusão mútua, progresso e de espera limitada.

A exclusão mútua seria violada se os dois processos entrassem na seção crítica, que necessitaria que flag[0] e flag[1] tivessem o valor verdadeiro (1). No entanto, como a variável turn somente pode ter um valor (0 ou 1), favorecendo a um único processo e garantindo que somente um poderia entrar na seção crítica.

A condição de progresso é satisfeita porque o processo que deseja entrar encontrará a posição referente ao outro processo, no vetor flag, com o valor 0, indicando que o mesmo não está na seção crítica. Com isso o processo requisitante irá executar a sua seção crítica.

A condição de espera limitada somente seria violada se o processo requisitante ficasse bloqueado eternamente no loop, testando a condição de entrada. Isso somente seria possível se o (a) outro processo também estivesse no loop, (b) se o outro processo repetidamente entrasse e saísse da seção crítica e, finalmente, (c) o outro processo não desejasse entrar. O caso (a) é impossível porque a variável turn sempre favorece um processo. O caso (b) também é impossível de ocorrer porque sempre que o processo sai da seção crítica favorece ao outro, setando a variável turn. No caso (c) a posição no vetor flag correspondente ao outro processo contém o valor 0, portanto o processo requisitante entra na seção crítica.

Soluções por SW para 2 processos

O algoritmo de Peterson, apresentado em 1981, implementa uma solução correta para o problema da exclusão mútua entre dois processos. Utiliza uma

variável turn, que indica qual processo deverá entrar na seção crítica, e um vetor booleano de duas posições, flag, que indica se o processo está na seção crítica. O processo i não entra na seção crítica se flag[j] = true e turn = j. Neste caso, não passa no comando while.

```
var flag[0..1] of boolean;  //compartilhada, inicializada com false
   var turn of integer;
   repeat
        flag[i] := true ;
        turn := j ;
        while flag[j] and turn = j do {}; // mutexbegin
            código-da-seção-crítica;
        flag[i] := false;                  // mutexend
        resto-do-código;
   until false;
```

O algoritmo de Peterson satisfaz as propriedades de exclusão mútua, de progresso e de espera limitada.

Exclusão mútua: um processo somente entra na seção crítica se o outro não quiser entrar ou se for a sua vez, o que é indicado pela variável turn, o que garante exclusão mútua.

Progresso: o processo entra na seção crítica se o outro não quiser entrar.

Espera limitada: o processo saindo da seção crítica (ex. p0) dá a vez para o outro processo (ex. p1), setando a variável turn, o que garante que p1 somente esperará uma vez que p0 execute a seção crítica, garantindo, portanto, a condição de espera limitada.

Algoritmo de Peterson programado com pthreads.

```
#include <pthread.h>

pthread_t tid0,tid1;

int turn ;
int shared ;
int flag [2];

void * p0(){

  int i ;
```

```
  for(i=0; i<10000; i++)
    {
     flag[0] = 1 ;
     turn = 1 ;
      while(flag [1]==1 && turn==1 ){}/*entra se flag[1]=0 */
      shared =shared + 5 ;
      flag [0] = 0 ;
     }
}
void * p1(){

  int i ;
  for (i=0;i<10000; i++)
    {
     flag[1] = 1 ;
     turn = 0 ;
     while(flag [0]==1 && turn==0 ){}/*entra se flag[0]=0 */
     shared =shared + 2 ;
     flag [1] = 0 ;
    }
}

int main()
{
  flag [0] = 0 ;
  flag [1] = 0 ;
  pthread_create(&tid0, NULL, p0, NULL) ;
  pthread_create(&tid1, NULL, p1, NULL) ;
  pthread_join(tid0,NULL);
  pthread_join(tid1,NULL);
  printf("O valor final de shared e:%d\n", shared);
}
```

Soluções por software para N processos

Os algoritmos de Dekker e Peterson, apresentados anteriormente, implementam uma solução correta para o problema da exclusão mútua entre dois processos. Lamport, em 1974, criou o algoritmo da padaria, a seguir apresentado, que soluciona o problema da seção crítica para n processos.

No algoritmo de Lamport, para entrar na seção crítica, cada processo precisa obter um “ ticket” e o processo com o ticket de menor valor tem preferência para entrar na seção crítica. Como a obtenção do “ ticket” é feita concorrentemente, mais de um processo pode obter o mesmo valor. Neste caso, o número do processos é usado para escolha. O processo com o menor número terá preferência.

// algoritmo da padaria

- antes de entrar na padaria cliente recebe um número;
- cliente com menor número é servido;
- clientes podem receber mesmo número, neste caso o identificador dos processos é usado para a escolha;
- número menor tem preferência.

O pseudocódigo do algoritmo de Lamport é apresentado a seguir.

// algoritmo da padaria

```
 // n = número de processos

// variáveis compartilhadas
    var status: array [0..n-1] of boolean;
    var ticket: array [0..n-1] of integer;

// estrutura do processo Pi
//inicialmente

status[i] = false; para i = 0 ...  i = n-1 ;
ticket[i]= 0 ; para i = 0 ...  i = n-1 ;

Loop {
   status[i] = true;
   ticket[i] = 1 + (max(ticket[0], ticket[1], .. , ticket[n-1]);
   status[i] = false ;
   for (j=0; j<n-1; j++) {
      while (status[j]) {} // comando vazio
      while ( (ticket[j] !=0 && (ticket[j],j < ticket[i], i)
                                  {}//Comando      vazio
      }

     Seção crítica de código

   ticket [i] = 0 ; // libera seção crítica

código não crítico
End_loop
```

O algoritmo possui dois vetores, *status,* no qual o processo sinaliza que deseja tirar um ticket, inicializado com false para todos os processos, e *ticket,* que contém, para o processo *i, 0* quando o mesmo não deseja entrar na seção crítica, ou o valor do ticket, quando o mesmo deseja entrar na seção crítica.

No pseudocódigo acima, nos primeiros três comandos após o *Loop*, o processo p_i sinaliza, no vetor *status,* que deseja entrar na seção crítica

(*status[i] = true ;*) e retira um ticket, que será " *o maior valor já obtido pelo seus pares que querem entrar na seção crítica*" + 1.
A seguir, no primeiro comando *while*, o processo p_i fica em loop enquanto o processo *pj* está pegando um ticket. O segundo comando *while*, garante que o processo p_i somente entrará na seção crítica se, para todos os processos j, j não quer entrar na seção crítica ou o ticket do processo p_i é o de menor valor.

A liberação da seção crítica é feita por `ticket [i] = 0,` que permitirá a entrada de um novo processo. Este algoritmo satisfaz as condições de exclusão mútua, de progresso e de espera limitada.

Exclusão mútua: somente o processo com o menor ticket, ou no caso de dois ou mais processos possuírem o mesmo número de ticket, o de menor número de identificador entra na seção crítica.

Progresso: se nenhum outro processo está usando a seção crítica (*status [j]=false para todo o j*), o processo *i* entra na seção crítica.

Espera limitada: Se todos os demais processos desejarem usar a seção crítica, o processo *i* pegará o maior valor de ticket, (pior caso), e o processo *i* espera que n – 1 processos executem a seção crítica.

Algoritmo de Lamport programado com pthreads, com quatro threads.

```
#include <pthread.h>
#define N 4
#define TRUE 1
#define FALSE 0

pthread_t tid0,tid1, tid2, tid3;

int turn ;
int shared ;
int status [N] , ticket [N], proc_id [N] ;

int max_number () {
  int i, maior ;
  maior = 0 ;
  for (i=0; i<N; i++){
     if (ticket[i] > maior) maior = ticket [i] ;
     }
   return (maior + 1) ;
  }
void mutex_begin (int i ) {
   int j ;
   status[i] = TRUE;
```

```
   ticket[i] = max_number () ;
   status[i] = FALSE ;
   for (j=0; j<n-1; j++) {
      while (status[j]) {} // comando vazio
      while ( (ticket[j] !=0 && (ticket[j] < ticket[i]) ||
              (ticket[j] !=0 && (ticket[j] == ticket[i]) &&
              (proc_id[j] < proc_id[i] )) {}//Comando      vazio
      }
   }

void mutex_end (int i ) {

   ticket [i] = 0 ;
   }

void * p0(){
  int my_number = 0 ;
  int i ;
  proc_id[my_number] = tid0 ;
  for(i=0; i<1000000; i++)
    {
      mutex_begin (my_number) ;
      shared =shared + 1 ;
      mutex_end (my_number) ;

     }
}
void * p1(){
  int my_number = 1 ;
  int i ;
  proc_id[my_number] = tid1 ;

  for(i=0; i<1000000; i++)
    {
      mutex_begin (my_number) ;
      shared =shared + 1 ;
      mutex_end (my_number) ;

     }
}
void * p2(){
  int my_number = 2 ;
  int i ;
  proc_id[my_number] = tid2 ;

  for(i=0; i<1000000; i++)
    {
      mutex_begin (my_number) ;
      shared =shared + 5 ;
      mutex_end (my_number) ;

     }
}
void * p3(){
  int my_number = 3 ;
  int i ;
  proc_id[my_number] = tid3 ;
  for(i=0; i<1000000; i++)
    {
      mutex_begin (my_number) ;
      shared =shared + 1 ;
```

```
      mutex_end (my_number) ;

     }
}

int main()
{
  int i ;

  for (i=0; i<N; i++) {
    status[i] = FALSE ;
    ticket [i] = 0 ;
    }

  pthread_create(&tid0, NULL, p0, NULL) ;
  pthread_create(&tid1, NULL, p1, NULL) ;
  ptrhead_create(&tid2, NULL, p2, NULL) ;
  pthread_create(&tid3, NULL, p3, NULL) ;

  pthread_join(tid0,NULL);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  pthread_join(tid3,NULL);
  printf("O valor final de shared e:%d\n", shared);
}
```

5.3 Semáforos, Spin-lock, Região Crítica e Monitores

As soluções apresentadas contém um grave problema, qual seja o de espera ocupada. Os processos que desejam executar a sua seção crítica necessitam testar a condição de entrada. Se a mesma é falsa, toda a sua fatia de tempo do processador será desperdiçada. Em seções críticas grandes, esta demora pode comprometer seriamente o desempenho do sistema.

Semáforos

Uma solução mais genérica, e que elimina o problema da espera ocupada, são os semáforos, desenvolvidos por Dijkstra em 1965. Características dos semáforos:

- Um semáforo *s* é uma estrutura de dados, formada por um contador e um apontador para uma fila de processos bloqueados no semáforo;
- Somente pode ser acessado por duas operações atômicas (P e V);

- A operação P bloqueia o processo que a executa se o valor do semáforo é 0;
- A operação V incrementa o valor do semáforo. Existindo processos bloqueados, o primeiro da fila do semáforo é acordado;
- As modificações no valor do semáforo são executadas atomicamente;
- Se dois processos tentam, simultaneamente, executar P(s) ou V(s), essas operações irão ser executadas seqüencialmente, em uma ordem arbitrária;
- Semáforos podem ser usados para exclusão mútua com *n* processos, quando inicializados com o valor 1.

A estrutura de utilização dos semáforos, para cada processo Pi, é a seguinte:

```
repeat
        P(mutex) ;
        “ seção crítica”
        V(mutex)
        “ seção não crítica”
until false ;
```

A implementação de semáforos é feita com o uso de uma estrutura de dados que contém o identificador do semáforo, o valor do semáforo, um apontador para o primeiro processo bloqueado no semáforo e um apontador para o último processo bloqueado no semáforo. A operação P no semáforo s é a seguinte:

```
P(s): s.value = s.value - 1 ;
    if (s.value < 0 ) {
        “ adicionar o processo na lista de processos bloqueados em s”
```

```
        block(p) ;  /*bloqueia o processo p*/
        }
```

Esta operação é executada pelo processo que deseja entrar na seção crítica. Se o valor do semáforo é zero ou menor que zero, o processo fica bloqueado. Caso contrário, o processo continua a execução dentro da seção crítica.

A operação V(s) é executada pelo processo para liberar a seção crítica de código. Se existirem processos bloqueados no semáforo, o primeiro da fila é liberado.

```
 V(s): s.value = s.value + 1 ;
      if (s.value < =0 ) {
        "  remover o processo "  p"   da lista s.l"
         wakeup(p) ; /*acorda o processo p*/
         }
```

Semáforos Binários e não binários (contadores)

Um semáforo é binário quando somente assume os valores 0 e 1. Se o valor do semáforo é 0, o processo que executa a operação P fica bloqueado. A operação V verifica a fila do semáforo. Se existe algum processo bloqueado, o primeiro é acordado, caso contrário é atribuído 1 ao valor do semáforo. Um semáforo não binário (semáforo contador) pode assumir valores diferentes de 0 e 1. A operação P verifica o valor do semáforo. Se o valor do mesmo é 0, o valor permanece 0 e o processo fica bloqueado. Sendo maior que 0, é decrementado e o processo não fica bloqueado (continua a execução). A operação V verifica a fila do semáforo. Se existe um processo bloqueado, o mesmo é acordado. Caso contrário, o valor do semáforo é incrementado.

Semáforos na biblioteca pthreads

A biblioteca pthreads inclui a definição de semáforos não binários e contém primitivas que permitem a inicialização e a utilização de semáforos (operações para inicialização, P e V). A biblioteca <*sys/semaphore.h*> contém a definição do tipo semáforo *sem_t*. A declaração de um semáforo *s* realizada da seguinte forma:

```
#include <semaphore.h>
 sem_t s ;
```

Após ter sido declarado, a atribuição do valor inicial do semáforo é feita com a primitiva

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

onde,

sem: é o endereço da variável semáforo;
pshared: 0 indica que o semáforo não é compartilhado com threads em outro processo e diferente de 0 caso contrário;
value: o valor inicial do semáforo.

Se o valor de retorno da chamada for 0 indica que execução bem sucedida.

Exemplo:

```
#include <semaphore.h>

sem_t s;

if ( sem_init(&s, 0, 1) != 0 )
{
  // Erro !!!
 }
```

A operação *sem_init* inicializa o valor do semáforo *s* com o valor 1, não compartilhado com threads em outros processos. O valor de retorno diferente de zero indica erro na execução da primitiva. As operações P e V em um semáforo são, respectivamente, sem_wait e sem_post, como mostradas a seguir.

*int sem_wait(sem_t *s0);*
*int sem_post(sem_t *s0);*

Exemplo, para o processo Pi, supondo o semáforo s definido e inicializado:

```
Pi () {
   sem_wait(&s);
    " seção crítica"
   sem_post(&s);
   " seção não crítica"
   }
```

O exemplo a seguir apresenta um programa escrito em C que utiliza semáforos para sincronizar o acesso de duas threads a um buffer de 1 posição. Uma thread (a produtora) deposita valores inteiros no buffer, de onde a thread consumidora os retira e imprime.

```
#include <pthread.h>
#include <semaphore.h>
pthread_t tid1,tid2;
sem_t s0, s1 ;

int buffer;

void * produtor(){

  int i ;
  for(i=0; i<100; i++)
     {
       sem_wait(&s0) ;
       buffer = i ;
       sem_post(&s1) ;
     }
}
void * consumidor(){
  int i, k ;
```

```
  for(i=0; i<100; i++)
     {
       sem_wait(&s1) ;
       k = buffer ;
       sem_post(&s0) ;
       printf("Valor consumido: %d\n", k) ;
     }
}

int main(){

  sem_init(&s0, 0, 1) ;
  sem_init(&s1, 0, 0) ;

  pthread_create(&tid1, NULL, produtor, NULL);
  pthread_create(&tid2, NULL, consumidor, NULL)
  pthread_join(tid1,NULL);
  pthread_join (tid2,NULL);

}
```

No programa acima, o semáforo s0 é inicializado com 1 e o s1 com 0. Se o processo consumidor ganhar o processador, irá ficar bloqueado no semáforo s1, e será desbloqueado após o produtor ter depositado um elemento no buffer e executar a operação *sem_post(&s1).* O produtor, por sua vez, ficará bloqueado no semáforo s0 até que o consumidor execute a operação *sem_post(&s0),* o que ocorrerá sempre que o consumidor retirar o elemento do buffer.

Problemas clássicos de sincronização

A seguir será apresentado o problema do buffer limitado, clássico da literatura. Dois processos, um produtor e um consumidor, compartilham um buffer circular de n elementos. Cada elemento tem capacidade de armazenamento de um item de informação. Os semáforos usados são:

- mutex: para exclusão mútua;
- empty: contador do número de buffers vazios;
- full: contador do número de buffers cheios.

O semáforo empty deve ser inicializado com N (número de elementos do buffer), full com 0 e mutex com 1. O código do programa é apresentado a seguir.

Produtor/Consumidor programado com C e pthreads.

```
#include <pthread.h>
#include <semaphore.h>
pthread_t tid1,tid2;
sem_t full, empty, mutex ;
#define N 10

int buffer[N];
int i=0, j=0 ; // Produtor produz na posição i e consumidor consome
da
               //posição j


void * produtor(){


  for(;;)
     {
       sem_wait(&empty) ;
       sem_wait(&mutex) ;
       buffer[i] = 50 ;
       i = ( i + 1 ) % N ;
       sem_post(&mutex) ;
       sem_post(&full) ;
    }
}
void * consumidor(){

  int j, c ;

  for (;;)
    {
       sem_wait(&full);
       sem_wait(&mutex);
       c = buffer[j] ;
       j = ( j + 1 ) % N ;
       sem_post(&mutex);
       sem_post(&empty);
    }
}
int main(){

  sem_init(&mutex, 0, 1) ;
  sem_init(&full, 0, 0) ;
  sem_init(&empty, 0, 10) ;
  pthread_create(&tid1, NULL, produtor, NULL);
  pthread_create(&tid2, NULL, consumidor, NULL)
  pthread_join(tid1,NULL);
  pthread_join (tid2,NULL);

}
```

Problema do jantar dos filósofos

Este problema foi proposto por E. W. Dijkstra em 1965 e é referente a alocação de recursos entre processos. Consiste de um conjunto finito de processos que compartilham um conjunto finito de recursos, cada um podendo ser usado por um processo de dada vez, podendo ocasionar deadlock ou postergação indefinida.

Um grupo de cinco filósofos estão sentados em volta de uma mesa redonda. Existe um garfo entre cada dois filósofos e um filósofo entre cada dois garfos. Cada filósofo tem um prato de espaguete. A vida de um filósofo consiste em pensar e comer e, para isso, precisa de dois garfos. Cada filósofo pode, arbitrariamente, decidir pegar primeiro o garfo da esquerda e depois o da direita, ou vive-versa, mas um garfo somente pode ser usado por um, filósofo de cada vez. Uma solução trivial é a seguinte:

```
void * filosofo (int fil) {
        for(;;) {
           pensar() ;
           pegar_garfo (fil) ;
           pegar_garfo (fil + 1) ;
           comer () ;
           largar_garfo (fil) ;
           largar_garfo (fil + 1) ;
          }
}
```

Esta solução claramente leva a deadlock. Imaginando-se que cada processo pegue o seu garfo da esquerda, todos ficariam bloqueados a espera do garfo da direita. Uma solução simples poderia ser usar um semáforo binário, inicializado com o valor 1. Após a chamada *pensar()* seria colocada a operação P e após a chamada *largar_garfo (fil + 1)* a operação V. No entanto, esta solução não é satisfatória, pois permite que somente um filósofo por vez coma.

Existem inúmeras soluções para o problema:

a) os garfos são numerados e cada filósofo tenta pegar o garfo adjacente de número maior primeiro;
b) os filósofos são coloridos, vermelho e azul. Os vermelhos tentam pegar o garfo da esquerda primeiro, os azuis tentam pegar o garfo da direita primeiro;
c) existe um monitor central que controla a atribuição de garfos aos filósofos;
d) existe uma caixa com n tickets (n é o número de filósofos). Cada filósofo precisa obter o ticket antes de pegar os garfos;

A seguir serão apresentadas duas soluções para este problema, baseadas nas soluções apresentadas por E. W. Dijkstra [Dijkstra, E. W., 1971] e no código escrito por A. S. Tanembaun [A. S. Tanembaun 2000]. Na primeira solução, os filósofos serão classificados em dois grupos (vermelho/azul). Os filósofos vermelhos primeiro tentam pegar o garfo da esquerda, os azuis, o garfo da direita. Esta solução, embora correta, não é simétrica, isto é, os filósofos não executam o mesmo código. É usado um vetor de semáforos, cada um representando um garfo, inicializados com o valor 1, indicando que o garfo está disponível.

A segunda solução que será apresentada é baseada na solução de Dijkstra e permite o máximo de paralelismo para um número qualquer de filósofos. Ela utiliza um vetor para manter o estado corrente dos processos (com fome, comendo, pensando) e um vetor de semáforos no qual, um filósofo com fome tentando adquirir um garfo poderia ficar bloqueado se o garfo estivesse em uso. Nesta solução, um filósofo somente pode passar para o estado comendo se os seus vizinhos não estiverem comendo.

```
//Solução para o problema do jantar dos filósofos na qual os
filósofos // são classificados em vermelhos e azuis

   #include <pthread.h>
   #include <semaphore.h>

   # define N                5
   # define PENSANDO         0
```

```
# define COM_FOME         1
# define COMENDO          2

pthread_t tid[N];
sem_t garfos[N];

void pensar ( int fil) {
   int t ;
   t = rand()%N ;
   printf("O filosofo %d vai pensar %d segundos\n", fil, t) ;
   sleep (t) ;
   }

void comer (int fil) {

     printf("Vai comer o filosofo %d\n", fil) ;
     sleep (2) ;
     }



void pegar_garfos(int fil){

      if ( fil % 2 == 0 )//Vermelhos: pegar o garfo da esquerda
                              e após o da direita
          sem_wait(&garfos[fil]) ;
          sem_wait(&garfos[(fil+1) % N] );
            }
        else {//Azuis: pegar o garfo da direita e após o da
                         esquerda
          sem_wait(&garfos[(fil+1) % N] );
          sem_wait(&garfos[fil]) ;
            }
    }

void depositar_garfos(int fil){

          sem_post(&garfos[fil]) ;
          sem_post(&garfos[(fil+1) % N] );
            }

void * filosofos (void * fil) {

  int i ;

  for (i=0;i<10; i++)
       {
        pensar((int)fil) ;
        pegar_garfos((int)fil) ;
        comer((int)fil) ;
        depositar_garfos((int)fil) ;
        }
}
int main(){

  int  i ;

  for (i=0; i<N;i++) sem_init ( &garfos[i], 0, 1);

  for (i=0; i<N;i++){
      pthread_create(&tid[i], NULL, filosofos, (void *)i);
```

```
      }

  for (i=0; i<N;i++){
      pthread_join(tid[i],NULL);
      }
}


// Problema do jantar dos filósofos que  permite paralelismo
// para um número qualquer de filósofos

#include <pthread.h>
#include <semaphore.h>

# define N               5
# define PENSANDO        0
# define COM_FOME        1
# define COMENDO         2


pthread_t tid[N];
sem_t s[N], mutex ;
int estado [N] ;

void pensar ( int fil) {
   int t ;
   t = rand()%N ;
   printf("O filosofo %d vai pensar %d segundos\n", fil, t) ;
   sleep (t) ;
   }

void comer (int fil) {

     printf("Vai comer o filosofo %d\n", fil) ;
     sleep (2) ;
     }

void teste (int fil) {

   if(estado[fil]==COM_FOME && estado[(fil+N-1)%N]!=COMENDO
      && estado[(fil+1)%N] != COMENDO) {
         estado[fil] = COMENDO;
         printf("O filosofo %d esta comendo\n", fil) ;
         sem_post(&s[fil]) ;
        }
}

void pegar_garfos(int fil){

  int i ;
        sem_wait(&mutex) ;
        estado[fil] = COM_FOME ;
        teste(fil) ;
        sem_post(&mutex) ;
        sem_wait(&s[fil]) ;
 }

void depositar_garfos(int fil){

  int i ;
```

```
        sem_wait(&mutex) ;
        estado[fil] = PENSANDO ;
        teste((fil+N-1)%N) ;
        teste(fil+1%N) ;
        sem_post(&mutex) ;
}

void * filosofos (void * fil){

  int i ;

  for (;;)
       {
        pensar((int)fil) ;
        pegar_garfos((int)fil) ;
        comer((int)fil) ;
        depositar_garfos((int)fil) ;
        }
}


int main(){

int  i ;
 for(i=0; i<5; i++) {
      sem_init (&s[i], 0, 0);
      estado[i] = PENSANDO ;
      }
 sem_init ( &mutex, 0, 1);
 for (i=0; i<=N;i++){
    pthread_create(&tid[i], NULL, filosofos, (void *)i);
    }
    for (i=0; i<N;i++) {
       pthread_join(tid[i],NULL);
       }
}
```

Problema do barbeiro dorminhoco

Também proposto por W. E. Dijkstra, este é um outro problema clássico de comunicação entre processos. Em uma barbearia trabalha apenas um barbeiro e existe um número limitado de cadeiras para os clientes esperarem, se o barbeiro estiver ocupado cortando o cabelo de um cliente. Caso não existam clientes, o barbeiro dorme até que seja acordado pelo próximo cliente e comece a cortar o cabelo. Se chegar um cliente enquanto o barbeiro estiver ocupado ele deve sentar-se em uma das cadeiras livres (se houver) e aguardar até que o barbeiro termine seu trabalho e chame o próximo cliente. Se não houver cadeiras livres o cliente vai embora. O problema consiste em sincronizar a ação do barbeiros e dos clientes.

```
#include <pthread.h>
#include <semaphore.h>
```

```
pthread_t tid1,tid2;
sem_t cliente, barbeiro, mutex ;

#define cadeiras 10

int esperando = 0 ;

void * clientes(void * cli){

       sem_wait(&mutex) ;
          if (esperando <= cadeiras){
             esperando = esperando + 1 ;
             sem_post(&cliente) ;
             sem_post(&mutex) ;
             sem_wait(&barbeiro) ;
                printf("Cliente %d cortando o cabelo\n", cli) ;
                sleep (4) ; //corta o cabelo
             }
          else  sem_post(&mutex) ;
    }
void * barbeiros(){
  int j ;
  for(j=0; j<10; j++)
    {
       sem_wait(&cliente);
           sem_wait(&mutex) ;
               esperando = esperando - 1 ;
               sem_post(&barbeiro) ;
           sem_post(&mutex);
           printf("Barbeiro cortando o cabelo de um cliente\n")
;
           sleep (4) ;
    }
}

int main(){

int i ;
  sem_init(&mutex, 0, 1) ;
  sem_init(&cliente, 0, 0) ;
  sem_init(&barbeiro, 0, 0) ;
  for (i=0; i<10;i++) pthread_create(&tid1, NULL, clientes,
       (void *)i);
  pthread_create(&tid1, NULL, barbeiros, NULL) ;

  for (i=0; i<11;i++) {
       pthread_join(tid1,NULL);
       }
}
```

Problema dos leitores e escritores

Este problema, proposto por Courtois e outros em 1971, é utilizado para modelar o acesso concorrente de processos leitores e escritores a uma base de dados. O acesso pode ser feito por vários leitores simultaneamente. Se um escritor estiver executando uma alteração, nenhum outro leitor ou escritor

poderá acessar a base de dados. Duas soluções, desenvolvidas por Courtois e implementadas com Pthreads, serão apresentadas a seguir. A primeira prioriza os leitores. Os escritores devem acessar a base de dados de forma mutuamente exclusiva, mas se houver um leitor acessando os dados, outros leitores que o desejarem poderão fazê-lo. A segunda solução prioriza os escritores. Da mesma forma que na primeira solução, os escritores devem acessar a base de dados de forma exclusiva e os leitores podem compartilhar o acesso. Porém, se um escritor desejar executar uma operação, deverá fazê-lo o mais rapidamente possível. Em outras palavras, um leitor não deverá acessar a base de dados se houver um escritor esperando.

Primeira solução – prioridade para os leitores: nesta solução, o semáforo r é usado pelos leitores para atualizar de forma mutuamente exclusiva a variável countleitor, que conta o número de leitores querendo acessar a região crítica. O semáforo mutex é usado para exclusão mútua no acesso aos dados compartilhados por leitores e escritores. O semáforo mutex é usado somente pelo primeiro leitor ao entrar na seção crítica e pelo último leitor, ao sair.

```
#include <pthread.h>
#include <semaphore.h>

#define N 10

pthread_t tid1[N], tid2[N] ;
sem_t r, mutex ;

int countleitor=0;

void * leitor(void * i){

     sem_wait(&r) ;
         countleitor = countleitor + 1 ;
         if (countleitor == 1)  sem_wait(&mutex) ;
     sem_post(&r) ;

       //acessa os dados
       sleep (2) ;
       printf("Sou o leitor %d\n", i) ;

       sem_wait(&r) ;
           countleitor = countleitor - 1;
           if (countleitor == 0)  sem_post(&mutex) ;
       sem_post(&r) ;
}
```

```
void * escritor(void * i){

      sem_wait(&mutex);

         //atualiza os dados
         sleep (2) ;
         printf("Sou o escritor %d\n", i) ;

       sem_post(&mutex);
}

int main(){


int i ;

  sem_init(&mutex, 0, 1) ;
  sem_init(&r, 0, 1) ;

   for (i=0; i<N;i++) {
      pthread_create(&tid2[i], NULL, escritor, (void *)i) ;
      pthread_create(&tid1[i], NULL, leitor, (void *)i);
      }
  for (i=0; i<N;i++) {
       pthread_join(tid1[i],NULL);
       }
    for (i=0; i<N;i++) {
       pthread_join(tid2[i],NULL);
       }
 }
```

A segunda solução para o problema dos leitores e escritores, apresentada a seguir, prioriza os escritores. Os semáforos r e w são usados para proteger o acesso aos dados compartilhados. O primeiro escritor que passar pelo semáforo r irá bloquear os leitores que não poderão manipular mutex1 e w. mutex3 garante acesso exclusivo aos leitores ao bloco de código de sem_wait(&r) até sem_post(&r).

```
#include <pthread.h>
#include <semaphore.h>
pthread_t tid1[10], tid2[10];
sem_t r, w, mutex1, mutex2, mutex3 ;

#define N 10

int nleitores = 0, nescritores = 0;

void * leitor(void * i){

       sem_wait(&mutex3) ;
           sem_wait(&r) ;
           sem_wait(&mutex1) ;
                 nleitores = nleitores + 1 ;
```

```
                if (nleitores == 1)  sem_wait(&w) ;
              sem_post(&mutex1) ;
            sem_post(&r) ;
        sem_post(&mutex3) ;

        //acessa os dados
        sleep (1) ;
        printf("Sou o leitor %d\n", i) ;

        sem_wait(&mutex1) ;
              nleitores = nleitores - 1 ;
              if (nleitores == 0)  sem_post(&w) ;
        sem_post(&mutex1) ;
}
void * escritor(void * i){

      sem_wait(&mutex2) ;
          nescritores = nescritores + 1 ;
          if (nescritores == 1)  sem_wait(&r) ;
      sem_post(&mutex2) ;
      sem_wait(&w) ;

      //modifica os dados
      sleep (1) ;
      printf("Sou o escritor %d\n", i) ;

      sem_post(&w) ;
      sem_wait(&mutex2) ;
          nescritores = nescritores - 1 ;
          if (nescritores == 0)  sem_post(&r) ;
      sem_post(&mutex2) ;


}

int main(){


int i ;

  sem_init(&mutex1, 0, 1) ;
  sem_init(&mutex2, 0, 1) ;
  sem_init(&mutex3, 0, 1) ;
  sem_init(&w, 0, 1) ;
  sem_init(&r, 0, 1) ;

  for (i=0; i<N;i++) {
      pthread_create(&tid1[i], NULL, leitor, (void *)i);
      pthread_create(&tid2[i], NULL, escritor, (void *)i) ;
      }
  for (i=0; i<N;i++) {
       pthread_join(tid1[i],NULL);
       }
    for (i=0; i<N;i++) {
       pthread_join(tid2[i],NULL);
       }
}
```

Spin-lock

Spin-lock é um mecanismo não bloqueante de exclusão mútua, utilizado em alguns sistemas para garantir o acesso mutuamente exclusivo a uma área de dados compartilhados. A implementação de Spin-lock necessita da existência de instruções especiais. Duas destas instruções são Test and Set e Swap.

TAS: Test And Set - Testa e modifica o conteúdo de uma variável de forma não interrompível

Swap: Troca o conteúdo de duas variáveis de forma não interrompível

A execução de uma instrução TAS, retorna o valor de uma variável global, inicializada com o valor false, e atribui verdadeiro à variável. Desta forma, se o valor é verdadeiro, retorna verdadeiro e é atribuído o valor verdadeiro. Se o valor é falso, retorna falso e é atribuído o valor verdadeiro. O pseudocódigo a seguir representa a semântica da execução da instrução TAS.

```
int lock = 0; // global

int tas (){

    r = lock ;

    lock = 1 ;

    return (r) ;

 }
```

A instrução swap troca o conteúdo de duas variáveis. Uma variável global é inicializada com o valor falso. Cada processo define uma variável local, inicializada com o valor verdadeiro, cujo conteúdo será trocado com a variável global. Assim, se o valor da variável global é verdadeiro, retorna na variável local o valor verdadeiro e à variável global é atribuído, na troca, o valor verdadeiro. Se a variável global possui o valor falso, retorna falso na variável local ao processo e à variável global é atribuído o valor verdadeiro. O pseudocódigo a seguir representa a semântica desta instrução swap.

```
int lock = 0; % global inicializada com false

swap (lock, key){

    int r ;

     r = lock ;

    lock = key ;

    key = r ;

    }
```

Uma possível forma de oferecer o uso dessas instruções é com dois procedimentos, enter_region e exit_region. A seguir serão apresentados estes procedimentos, programados com uma pseudolinguagem de montagem. A implementação com TAS é a seguinte:

```
int lock = 0; /* variável global

enter_region (){

        L: tas register, #lock  /* o valor de lock é atribuído a register e a lock é

                                                atribuído o valor 1 –  operação atômica */
        cmp register, #0    /* compara com 0 */

        jne L   /* se não for 0 a SC está ocupada, loop */

        ret

    }

leave_region (){

        move lock, #0  /* atribui 0 a variável lock - libera a SC */

       ret

    }
```

A estrutura dos processos que participam da seção crítica é a seguinte:

```
pi() {
    for(;;) {
        enter_region() ;
        “  seção crítica”
        leave_region() ;
        “  seção não crítica”
        }
}
```

Para entrar na seção crítica o processo chama o procedimento enter_region(). Se a seção crítica estiver ocupada, o valor da variável lock é verdadeiro e o processo ficará em loop, dentro do procedimento. No caso do valor ser falso, o processo termina o procedimento enter_region() e executa a seção crítica. Ao terminar a seção critica o processo a libera chamando o procedimento leave_region(), que atribui o valor falso a variável lock.

Com a instrução SWAP os procedimentos são

```
int lock = 0; % global inicializada com false indicando que a SC está livre
enter_region(int key) {
    L: swap #lock, #key /* troca o conteúdo de lock e key -
                           atomicamente */
    cmp #key, 0        /* compara o conteúdo de key com 0 */
    jne L  /* se for 1 a SC está ocupada,  loop */
    rte
}
```

Neste procedimento, o valor das variáveis lock e key são trocados atomicamente. Se o valor de lock é verdadeiro (seção crítica ocupada), o processo fica em loop até que se torne falso (seção crítica livre). O procedimento leave_region():

```
leave_region() {
        move #lock, 0 /* libera a SC */
        rte
}
```

libera a seção crítica, atribuindo o valor 0 à variável lock. O primeiro processo que ganhar o processador e executar o procedimento enter_region() ganhará acesso a seção crítica. A estrutura dos processo que utilizam a seção crítica é

```
pi() {
      for(;;) {
          int key = 1 ;
          enter_region(key) ;
           “  seção crítica”
           leave_region() ;
           “  seção não crítica”
            }
         }
```

de maneira análoga a solução com a instrução TAS, Cada processo para entrar na seção crítica executa o procedimento enter_region(key). Se o valor de lock for verdadeiro, a SC está ocupada e o processo ficará em loop. Quando lock se torna falso, a seção crítica está livre e o processo continua a execução. O procedimento exit_region() libera a seção crítica, atribuindo o valor falso à variável lock.

Instrução xchg

Os processadores da família Intel possuem a instrução xchg, que troca o conteúdo de duas variáveis de forma não interrompível, usada para implementar spin-lock. O programa a seguir apresenta um exemplo de Spin-lock.

```
#include <stdio.h>
#include <pthread.h>
pthread_t tid1,tid2;
int lock = 0, key = 1, ZERO = 0;
int a = 0;
 void  mutex_begin () {
        asm ("ocupado:") ;
         asm ("mov key, %eax") ;
         asm ("xchg lock, %eax") ;// se lock=1, eax recebe 1 e lock recebe 1;
                                  //se lock == 0, eax recebe 0 e lock recebe 1;
         asm("cmp %eax, ZERO") ;
         asm("je ocupado") ; //loop se lock == 1; caso contrario entra na SC
  }

void  mutex_end () {
      lock = 0 ; //libera a SC
  }
void * thread2(){
  int i, k, key;
  for(i=0; i<1000000; i++) {
      mutex_begin () ;
      a = a + 5;
      mutex_end () ;
```

```
    }
}
void * thread1() {
  int i ;
  for (i=0; i<1000000; i++)  {
       mutex_begin () ;
       a = a + 2;
       mutex_end () ;
    }
}
int main(){
      printf("Inicio \n") ;
      pthread_create(&tid2, NULL, thread2, NULL) ;
      pthread_create(&tid1, NULL, thread1, NULL) ;
      pthread_join(tid1,NULL);
      pthread_join(tid2,NULL);
      printf("Valor de a: %d\n", a) ;
      }
```

No programa acima, os procedimentos mutex_begin e mutex_end são usados para entrada e saída da seção crítica. O procedimento mutex_begin, escrito em assembler, troca o conteúdo a variável lock e do registrador eax, que recebeu o valor 1 (variável key). Se lock possui o valor 0, passará a possuir o valor 1, se possui o valor 1, continuará com o valor 1 e neste caso a thread que faz o teste ficará em loop, até que lock se torne 0, o que ocorre na liberação da seção crítica, pelo procedimento mutex_end.

O principal problema do uso de spin lock é a espera ocupada. O processo que deseja entrar em uma seção crítica ocupada ficará em loop, e este tempo do processador é desperdiçado. Um outro problema é que se muitos processos desejam entrar na mesma seção crítica, um processo poderá ficar esperando indefinidamente, pois o processo que entra é o que

faz o teste quando a seção crítica está livre. Conceitualmente, um processo poderá nunca entrar na seção crítica.

Embora estes problemas, na prática o spin-lock é usado em seções críticas pequenas, pois são liberadas muito rapidamente e existe uma probabilidade muito alta dos processos a encontrarem livre. Outra situação em que Spin-lock é utilizado é quando o tempo para executar a seção crítica é menor do que o tempo de salvamento/restauração de contexto, necessários em soluções bloqueantes.

Regiões Críticas Condicionais e Regiões Críticas

Estes mecanismos representam soluções de mais alto nível para o problema da exclusão mútua, sendo mais natural de serem incorporados em uma linguagem de programação. Considerando-se o uso de semáforos, é muito fácil provocar erros em sua utilização. Por exemplo:

- pode-se omitir uma operação P;

- pode-se omitir uma operação V;

- pode-se, no lugar de uma operação P, colocar V e vice-versa;

- pode-se utilizar a seção crítica sem colocar operações P e V.

- etc.

Regiões Críticas Condicionais (Hoare 1972; Brinch Hansen 1972 e 1973)

Região crítica condicional é um mecanismo que garante exclusão mútua no acesso aos dados compartilhados. A exclusão mútua é garantida porque somente é permitido a um processo por vez executar as instruções dentro de uma região crítica condicional. A forma do comando é a seguinte:

region V when b do S;

b: expressão booleana avaliada quando o processo entra na região crítica.

Se b é verdadeira, a lista de comandos S é executada senão o processo libera a exclusão mútua e é suspenso até que a expressão booleana se torne verdadeira. O programa apresentado a seguir ilustra o uso deste mecanismo.

```
CONST N = 10;
    VAR buf : ARRAY [1..N] OF item;
      in, out, count : [1..N] := 0;
    RESOURCE bb : buf, in, out, count;  (* the shared variables *)

    PROCESS producer;
      VAR i : item;
    BEGIN
      LOOP
        produce(i);
        REGION bb WHEN count < N DO
          buf[in] := i;
          in := (in + 1 ) mod N;
          count := count + 1;
        END;
       END;
    END producer;

    PROCESS consumer;
      VAR i : item;
    BEGIN
      LOOP
        REGION bb WHEN count > 0 DO
          i := buf[out];
          out := (out + 1) MOD N;
          count := count - 1;
        END;
        consumer(i);
      END;
    END consumer;
```

O trecho de código acima contém por dois processos, um produtor e outro consumidor, que utilizam região crítica condicional como mecanismo de exclusão mútua. O processo produtor produz elementos no buffer se existe

uma posição disponível. Caso contrário o mesmo ficará bloqueado, no comando region, até que a condição seja verdadeira (count < n), isto é, existam posições disponíveis no buffer. O processo consumidor retira elementos do buffer. Se o buffer está vazio (count = 0), o processo consumidor fica bloqueado no comando region até que a condição se torne verdadeira.

Um inconveniente da região crítica condicional é que os processos necessitam reavaliar continuamente a expressão lógica. Além disso, a avaliação da condição é feita antes do processo entrar na região crítica. Muitas vezes, a avaliação da condição deve ser feita dentro da seção crítica, o que na solução de Brinch Hansen, apresentada a seguir (região crítica mais operações await e cause), é possível de ser feito.

Regiões Críticas (Brinch Hansen 72)

Com este mecanismo, uma variável compartilhada, v, do tipo T é definida como

Var v: shared T ;

Processos concorrentes somente podem acessar a variável compartilhada dentro de um comando estruturado chamado *critical region*.

region v do S ;

esta notação associa a execução do comando S a região crítica v. A variável compartilhada v somente pode ser acessada dentro de um comando region.

Se Processos concorrentes acessam simultaneamente o comando region, o primeiro passa e irá executar o comando S. Os demais ficam bloqueados na entrada da região crítica. Isso significa que o comando region garante exclusão mútua no acesso aos dados definidos como compartilhados.

Processos cooperantes necessitam, em inúmeras situações, esperar por certas condições. Por exemplo, um processo consumidor necessita esperar

que um processo produtor produza os dados que o mesmo irá consumir. A operação

await (e)

[Brinch Hansen 1972] libera a região crítica e bloqueia o processo a espera de uma condição. Segundo Brinch Hansen, o processo fica bloqueado em uma fila de eventos. A primitiva

cause(e)

acorda todos os processos bloqueados na fila de eventos e. O pseudocódigo apresentado a seguir ilustra o uso destas primitivas.

```
// consumidor                                    //produtor

region v do                                      region v do
    begin                                            begin
        while not B do await (e) ;                       S2 ;
        S1 ;                                             cause (e) ;
    end ;                                            end () ;
```

No programa acima, o processo consumidor fica bloqueado, pela execução do comando await(e), se a expressão B é falsa. Será acordado pela operação cause(e), que sinaliza que a expressão B se torna verdadeira para o processo consumidor, que poderá recomeçar a execução.

Embora a biblioteca Ptrhreads não implemente região crítica, tal como definido por Brinch Hansen, é possível utilizar os comandos pthread_mutex_lock() e pthread_mutex_unlock() para obter-se exclusão mútua no acesso a dados compartilhados. As variáveis mutex devem ser globais e uma chamada pthread_mutex_lock que tenha obtido a seção crítica faz com que outras threads que estão tentando adquirir a seção crítica fiquem bloqueadas, até que a thread que detém a região crítica a libere via

pthread_mutex_unlock. Mutex podem ser usados para sincronizar threads no mesmo processo ou em processos diferentes. Para ser possível a utilização de mutex por threads de processos diferentes, as variáveis mutex devem ser alocadas em uma memória compartilhada pelos processos. A primitiva

*int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *attr);*

inicializa a variável mutex mp com os atributos especificados por attr. Se attr é NULL, os atributos padrão são utilizados. Após a inicialização, o estado do mutex é inicializado e é unlocked. Com os atributos padrão, somente threads no mesmo processo podem operar com a variável mutex. A primitiva

*int pthread_mutex_lock(pthread_mutex_t *mp);*

fecha o mutex referenciado por mp. Se o mutex já está fechado, o processo que efetua esta chamada fica bloqueado até que o mutex seja liberado. Se a thread proprietária do mutex executa esta chamada, ocasionará um deadlock. A primitiva

*int pthread_mutex_unlock(pthread_mutex_t *mp);*

é chamada pela thread proprietária do mutex mp para liberá-lo. Se existem threads bloqueadas, o escalonador é chamado para escolher a thread que irá obter o mutex. Se a thread que executa a chamada não é a proprietária do mutex, um erro é retornado e o comportamento do programa é indefinido.

O programa a seguir, que ilustra o uso destes comandos, faz a gerencia de alocação de blocos de disco. O processo que deseja alocar um bloco, chama o procedimento acquire, que retorna um número de bloco. Se o número retornado for superior a 1023, não existia um bloco disponível. Na função

principal deste programa são criadas 5 threads, que executam o código thr, e é inicializada a variável barrier_lock, do tipo mutex.

//Exemplo de Região Crítica programada com pthreads

```
#include <pthread.h>
pthread_t tid;
pthread_mutex_t barrier_lock;
int disk [1024] ;

int acquire () {
     int i ;
     pthread_mutex_lock(&barrier_lock);\
         i = 0 ;
         while(i < 1024 && disk[i] == 1) i++ ;
         if (i < 1024) disk[i] = 1 ;
     pthread_mutex_unlock(&barrier_lock);
     return(i) ;
}

void thr(int i){
     int j, k ;

     printf(" EU SOU A THREAD : %d\n",i);
     for (j=0; j<10; j++){
          k = acquire () ;
          if(k < 1024) printf("THREAD: %d OBTIVE O BLOCO %d \n",i,
k);
          sleep(1);
          }
}

int main(){
int i;
   for(i=0;i<1024;i++) disk[i] = 0 ;
   pthread_mutex_init(&barrier_lock,NULL);
   for (i=0;i<5;i++) pthread_create(&tid, NULL, thr, NULL);

   for (i=0;i<N;i++)  pthread_join(tid, NULL);
   printf("FIM DO MAIN\n");
}
```

A biblioteca Pthreads possui as operações pthread_cond_wait() e pthread_cond_signal(), definidas como segue:

*int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);*

Os parâmetros são a variável condição *cond* e o *mutex.*

A primitiva
*int pthread_cond_signal(pthread_cond_t *cond)* possui como parâmetro a variável condição *cond*.

pthread_cond_wait e pthread_cond_signal apresentam as seguintes propriedades:

- são usadas dentro do comando pthread_mutex_lock(&mutex) e pthread_mutex_unlock(&mutex);
- são associadas a uma região crítica, usadas nos comandos pthread_mutex_lock/pthread_mutex_unlock;
- a operação pthread_cond_wait(&mutex, &cond) bloqueia o processo que a executa na variável condição *cond* e libera a região crítica *mutex* ;
- a operação pthread_cond_signal(&cond) acorda um processo bloqueado na variável condição *cond*;
- a operação pthread_cond_signal não produz efeito se não houver processos bloqueados na variável condição *cond.*

O programa a seguir ilustra o uso destas primitivas.

```
#include <pthread.h>
pthread_cond_t cond0;
pthread_cond_t cond1;
pthread_t tid0;
pthread_t tid1;

pthread_mutex_t mutex;

int buffer ;
void * produtor(){
     int j, k ;
     for (j=0; j<100; j++){
         pthread_mutex_lock(&mutex) ;
             while (x == 1) pthread_cond_wait(&cond1, &mutex) ;
             buffer = j * 5 ;
             X = 1 ;
            pthread_cond_signal(&cond0) ;
```

```
         pthread_mutex_unlock(&mutex)  ;
        }
      }

void * consumidor(){
     int j, k ;
     for (j=0; j<100; j++){
         pthread_mutex_lock(&mutex) ;
             while (x == 0) pthread_cond_wait(&cond0, &mutex) ;
             printf ("Valor do buffer: %d\n", buffer) ;
             X = 0 ;
            pthread_cond_signal(&cond1) ;
         pthread_mutex_unlock(&mutex)  ;
        }
      }


int main(){
   pthread_mutex_init(&mutex,NULL);
   pthread_create(&tid0, NULL, produtor, NULL);
   pthread_create(&tid1, NULL, consumidor, NULL);
   pthread_join(tid0, NULL);
   pthread_join(tid1, NULL);
}
```

No programa acima, se o valor da variável x for igual a zero, o processo consumidor fica bloqueado, na operação pthread_cond_wait(&cond0), até que seja sinalizado pelo processo produtor, o que ocorre após o mesmo ter atribuído um valor à variável buffer e o valor 1 à variável x, pela execução da operação pthread_cond_signal(&cond). De maneira análoga, o processo produtor fica bloqueado na operação pthread_cond_wait(&cond1) até ser sinalizado pelo processo consumidor, o que ocorre pela operação pthread_cond_signal(&cond1). Observe que são utilizadas duas variáveis condição, uma pelo processo produtor e outra pelo processo consumidor.

Monitor [Brinch Hansen 73, Hoare 74]

Um monitor é um tipo abstrato de dados que contém os dados que representam o estado de um determinado objeto, os procedimentos que manipulam este objeto, e um código de inicialização que assinala o estado inicial do objeto. Os procedimentos de um monitor são comuns a todos os processos que acessam o monitor. Porém, a execução destes procedimentos é feita de forma mutuamente exclusiva. Se dois processos chamam ao

mesmo tempo um mesmo procedimento, o primeiro a fazer a chamada executará, enquanto que o segundo ficará a espera que o procedimento termine e que o monitor seja liberado e que a sua chamada seja executada. Variáveis especiais (condição) permitem a um processo se bloquear a espera de uma condição (wait). A condição é sinalizada por um outro processo (operação signal). A figura a seguir apresenta a visão de um monitor.

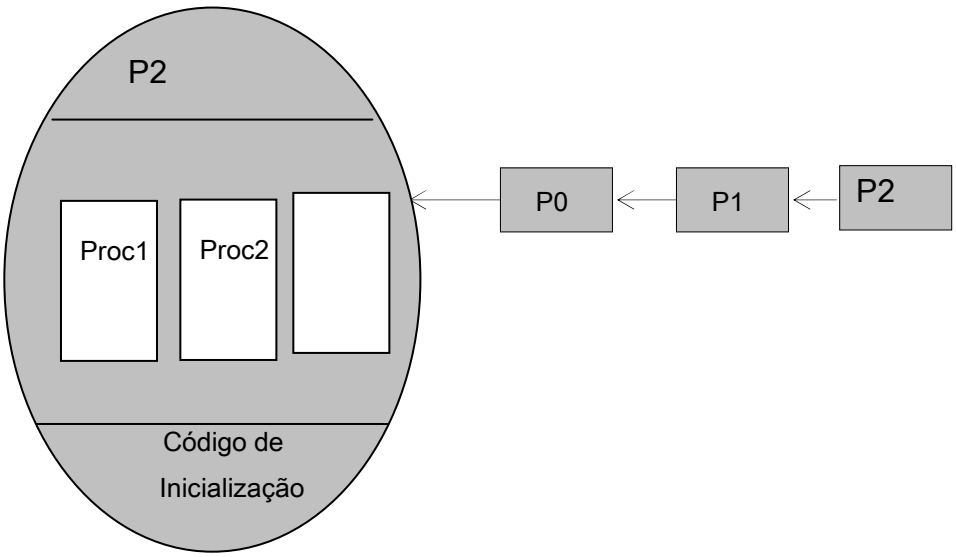

Fig. 5.1 - Visão Esquemática de um Monitor

A figura acima apresenta um monitor formado por três procedimentos, o código de inicialização, que é executado quando o monitor começa a rodar, e os dados globais. Contém também uma fila, formada pelos processos P0, P1 e P2, que estão bloqueados a espera da liberação do monitor para executar um procedimento (Proc0, Proc1 ou Proc2) do monitor. Variáveis condição são associadas ao conceito de monitor, e são utilizadas para bloquear/desbloquear processos que estão a espera de recursos. Por exemplo, pode-se definir as variáveis x e y, do tipo condição com a notação

var x, y : condition ;

A operação

x.wait () ;

bloqueia o processo que a executa na variável condição x. A operação

x.signal () ;

acorda um processo bloqueado na variável condição x. S não existem processos bloqueados, a operação não produz efeitos. Considerando estas operações, um problema que pode surgir é o seguinte:

Se a operação x.signal é executada por um processo P0 e existe um processo P1 bloqueado, P0 e P1 podem executar. Se P1 é acordado, P0 e P1 ficam ativos simultaneamente no monitor, o que viola o principio da exclusão mútua.

Possibilidades de solução:

a) P0 acorda P1 e se bloqueia, liberando o monitor

- P0 recomeça a execução quando P1 liberar o monitor;
- P0 recomeça a execução quando P1 se bloquear a espera de outra condição.

b) P0 sinaliza a condição de P1 e não libera o monitor

- P1 espera que P0 libere o monitor;
- P1 espera que P0 se bloqueie a espera de condição, o que libera o monitor.

A solução adotada por Brinch Hansen, na linguagem Pascal Concorrente [Brinch Hansen 1977] resolve o problema de uma forma simples, a operação signal deve sempre ser o último comando dentro de um procedimento do monitor. Assim, P0 executa signal e libera o monitor (o signal é a última instrução da procedure). A seguir será apresentado um programa Pascal Concorrente que exemplifica o uso de monitor.

Pascal Concorrente é uma extensão da linguagem Pascal, na qual Brinch Hansen acrescentou três tipos abstratos de dados: processos, classes

e monitores, identificados pelas palavras chaves *process, class, monitor*. O programa a seguir é formado por um monitor e por dois processos. O monitor é usado para controlar o acesso exclusivo a um recurso e possui dois procedimentos: *request* e *release*. O processo que chama request é bloqueado, pela operação delay (x), se o recurso está ocupado. Para liberação do recurso o processo que o detém chama o procedimento *release,* que assinala que o recurso é livre e executa a operação *continue (x)*, que acorda um processo bloqueado na variável condição x, se existir. Não existindo processo a ser acordado, o sinal gerado pela operação *continue* é perdido*,* isto é, não é memorizado. Os procedimentos que são visíveis pelos processos são os que contém a palavra reservada *entry*. No exemplo, não existem procedimentos locais ao monitor (sem a palavra reservada *entry*). O monitor possui um código de inicialização, (begin/end do monitor) que é executado quando o monitor é declarado (instanciado).

No main do programa (trecho begin/end.), são declaradas as variáveis do tipo *process* e *monitor,* definidos anteriormente. Os processos recebem o monitor como parâmetro, na operação *init*, que os coloca em execução.

//Programa Pascal Concorrente

```
type resource = monitor
  var
     free : boolean ;
     x :  queue ;
 procedure entry request ;
     begin
        if not free them
            delay ( x ) ;
             free : = false ;
      end
procedure entry release ;
    begin
       free : = true ;
```

```
       continue ( x ) ;
    end
 begin
   free : = true;
end.
Type user = process ( r = resource );
  begin
      r.request ;
         …
      r.release ;
end;
Type user1 = process ( r = resource );
  begin
     r.request ;
         …
     r.release ;
end;
begin
         var
              p0 : user ;
              p1 : user1 ;
              m : resource ;
        init po ( m ) ;
        init p1 ( m ) ;
end;
```

O código a seguir apresenta a implementação de um semáforo binário com monitor.

```
type semaphore = monitor ;
  var   busy = boolean ;
          non_busy : queue ;
procedure entry P
     begin
       if busy then delay (non_busy) ;
```

```
        busy = true ;
    end;
Procedure entry V
    begin
       busy = false ;
       continue (non_busy);
    end ;
begin
   busy = false ;
end.
```

A seguir será apresentada a implementação de um semáforo não binário com monitor.

```
type semaphore = monitor ;
  var   nproc = int ;
          non_busy : queue ;
procedure entry P ()
     begin
        nproc--;
        if nproc<0 then delay (non_busy) ;
     end;
Procedure entry V ()
    begin
       nproc++;
       continue (non_busy) ;
    end ;
begin
   nproc = n;
end;
```

Problema do buffer limitado

O monitor apresentado a seguir implementa uma solução para o problema do buffer limitado.

```
type bounded_buffer = monitor ;
```

```
const     MAX = 10 ;
 var buffer : ARRAY [0..MAX] of integer;
     count, in, out : integer;
    p, c : queue ;

   procedure entry deposita (i: integer) {
                    if (count = MAX)   delay (c) ; //espera consumir, o buffer está cheio
                     (* coloca um novo elemento no buffer *)
                     buffer[in] := i;
                     in := (in + 1) mod MAX;
                    count := count + 1;
                    (* sinaliza o consumidor *)
                    continue(p);
                    }

   procedure entry retira (var i: integer) {
           if (count = 0)   delay (p) ; // espera produzir, o buffer está vazio
           (* coloca um novo elemento no buffer *)
           I = buffer[out]  ;
           out = (out + 1) mod MAX;
          count := count + 1;
          (* sinaliza o produtor *)
          continue(c);
          }
     begin  // do monitor

     end.   //monitor
```

Exercícios

1. Considere a existência de um semáforo para controlar a passagem de automóveis em um cruzamento, que não existe prioridade de uma rua em relação à outra, que o trânsito é nos dois sentidos nas duas ruas e que deve passar, alternadamente, um automóvel de cada rua, em cada um

dos sentidos. Desenvolva uma solução para este problema. Modele os carros como threads e utilize semáforos para sincronização.

2. Considere a existência de uma ponte em um desfiladeiro, com capacidade para passar somente uma pessoa de cada vez. Viajantes da região da Floresta Alta possuem prioridade em relação aos viajantes da região do Grande Rio. Somente após cinco viajantes da Floresta Alta terem atravessado a ponte é que um viajante da região do Grande Rio pode atravessar em sentido contrário. Desenvolva uma solução para este problema. Modele os viajantes como *threads* e utilize semáforos para sincronização.
3. Escreva um programa formado por dois processos concorrentes (threads), **leitor** e **impressor**, que executam um loop infinito, e que sincronizam suas ações com o uso de semáforos. O processo **leitor** fica lendo caracteres do teclado e colocando em um buffer de 132 posições. Quando o buffer está cheio o processo **impressor** deve imprimi-lo.
4. Escreva um monitor, com dois procedimentos, deposita e imprime. O processo leitor chama o procedimento deposita, para inserir caracteres no buffer (132 posições). O processo impressor chama o procedimento imprime, para imprimir o buffer, quando o mesmo estiver cheio.
5. Escreva um programa formado por dois processos concorrentes, **leitor** e **impressor**, que executam um loop infinito, e que sincronizam suas ações com o uso da solução de Peterson, para o problema da exclusão mútua. O processo **leitor** lê caracteres do teclado e os coloca em uma lista encadeada, cujos nodos são alocados dinamicamente. O processo **impressor** fica continuamente retirando os caracteres da lista e imprimindo.
6. Considerando que dois processos, um produtor e um consumidor compartilham um buffer de um elemento, escreva um programa concorrente com o uso da solução de Peterson para sincronizar as ações do produtor e do consumidor.
7. Escreva um programa concorrente formado por três threads, uma consumidora (**c**) e duas produtoras (**p0** e **p1**), que executam um loop eterno. A thread consumidora recebe informações (um valor inteiro) da

thread **p0** no buffer **b0** e da thread **p1** no buffer **b1 (**primeiro consome de **b0** e depois consome de **b1**). Os buffers **b0** e **b1** possuem capacidade de armazenar um único elemento. A thread consumidora somente pode consumir se existirem informações no buffer e as threads produtoras somente podem voltar a produzir depois da thread consumidora haver retirado as informações do buffer. Utilize semáforos para a exclusão mútua no acesso aos buffers.

8. Escreva um programa concorrente formado por dois processos (THREADS) que sincronizam suas ações com o uso de semáforos. Um dos processos conta até 500, e então deve esperar pelo outro que conta até 1000, e vice-versa. Considere que a execução começa pela THREAD que conta até 500.
9. Com o uso de semáforos construa uma implementação dos procedimentos ***enqueue*** e ***dequeue,*** que controlam o uso de recursos de forma mutuamente exclusiva, definidos como segue:

```
enq(r): if inuse (r){
            insere p na fila de r ;
          block(p) ; }
        else inuse = true ;
deq(r): p = primeiro da  fila de r ;
      if p != null ativar p ;
        else inuse (r) == false ;
```

10. Região crítica é implementada em Pthreads com os comandos:

pthread_mutex_t lock; /* define lock como uma variável do tipo mutex */
pthread_mutex_lock(&lock); /* Previne múltiplas threads de executar a seção crítica*/
pthread_mutex_unlock(&lock); /* Libera a seção crítica*/
pthread_cond_t c ; /* define uma variável condição ***c*** */
pthread_cond_wait (&lock, c) ; /* bloqueia o processo na variável condição ***c,*** associada a região crítica ***lock***/
pthread_cond_t signal (c) ; /* acorda um processo bloqueado na variável condição ***c*** */

Escreva os procedimentos *alocaBuffer ( )* e *liberaBuffer ( ),* que acessam um vetor compartilhado de 100 posições, e que mantém um mapa de ocupação de buffers de memória. As posições disponíveis são marcadas com *0* e ocupadas com o valor 1.

11) Escreva um monitor que implementa um buffer de 132 elementos e é composto por duas procedures, **getBuf e putBuf**. O procedimento **putBuf** recebe como parâmetro um valor que deve ser armazenado no buffer. O procedimento **getBuf** retorna um elemento do buffer. Considere a necessidade de sincronização nas situações de buffer cheio e buffer vazio.

12) Região crítica é implementada em Pthreads com os comandos:

- mutex_t lock; /* define uma variável *lock* do tipo mutex */
- mutex_lock(&lock); /* Previne múltiplas threads de executar a seção crítica*/
- mutex_unlock(&lock); /* Libera a seção crítica*/

Supondo que uma thread é criada com a primitiva *pthread_create(nomefunc)* e que a *função pthread_ join ()* bloqueia a thread que a executa até que uma thread filha termine, escreva um programa formado por duas threads, uma produtora e uma consumidora que compartilham um buffer de um único elemento. A solução poderia ser melhorada com o uso de alguma(s) primitiva(s)? Qual(is)?

6 Comunicação entre Processos por Troca de Mensagens

Neste capítulo são estudadas a comunicação entre processos por troca de mensagens. São apresentadas as primitivas básicas de comunicação, send e receive, a comunicação em grupo a chamada remota de procedimento. O capítulo também contém um estudo sobre MPI: Message Passing Interface.

6.1 Princípios básicos

Processos cooperantes, componentes de uma aplicação, em inúmeras situações necessitam trocar informações. Com memória compartilhada, os processo compartilham variáveis e trocam informações através do uso de variáveis compartilhadas. Sem memória compartilhada, os processos compartilham informações através de troca de mensagens (passing messages). Neste modelo, o S.O é responsável pelo mecanismo de comunicação entre os processos.

A comunicação de Processos por Troca de Mensagens permite a troca de informações entre processos que não compartilham memória. As duas primitivas básicas são *send* e *receive*.

send: utilizada por um processo para enviar uma msg. Pode ser bloqueante ou não bloqueante.

receive: utilizada por um processo para receber uma msg. Essa operação é bloqueante, o processo que a executa fica bloqueado até o recebimento de uma mensagem.

A sincronização entre processos nos sistemas que implementam troca de mensagens, é implícita às operações. A operação send pode ser bloqueante ou não bloqueante, e a operação receive, normalmente, é bloqueante. Desta forma, um processo que executa uma operação receive permanece bloqueado até o recebimento de uma mensagem, enviada com uma operação send. Uma outra operação existente é reply, usada por um processo para responder a uma mensagem recebida. Pode conter uma informação de resposta ou simplesmente pode indicar que uma mensagem enviada com uma operação send foi recebida sem erros. A operação reply é não bloqueante.

A comunicação por troca de mensagens pode ser direta ou indireta. Na comunicação direta simétrica os processos se identificam mutuamente, isto é, o remetente identifica o recebedor e o recebedor identifica o remetente. Ex.:

```
/ * processo 0 * /
main ( ) {
  -
  send (t1, &m);
 }
```

```
// processo 1
main ( ) {
      -
   receive (t0, &m);
 }
```

Na comunicação direta assimétrica o remetente identifica o recebedor e o recebedor recebe de qualquer remetente, o que é indicado por -1 no campo que identifica o remetente, na operação *receive*.

```
// processo 0
```

```
main ( ) {
```

```
  -                                    main ( ) {
  send (t1, &m);                             -
 }                                        receive (-1, &m);
                                         }
// processo 1
```

Quando a comunicação é indireta, existe uma entidade intermediária através da qual os processos se comunicam, denominada porta ou mailbox. Exs.: sistema operacional MACH, sockets. O direito de receber mensagens enviadas sobre uma porta pode ser do processo criador ou de todos os processos que possuem o direito de acesso à porta. As operações sobre portas são:

p = port-create ( );

Esta operação é utilizada para criar uma porta de comunicação. Ao término da operação, p contém a identificação da porta criada.

port-destroy (p);

Esta operação elimina uma porta. Ao término da operação, a porta p deixa de existir. Mensagens associadas a porta p não recebidas serão perdidas.

send (p, m);

Permite a um processo enviar uma mensagem m para uma porta p. As mensagens serão colocadas em uma fila de mensagens ainda não consumidas, associadas à porta p.

receive (p, m);

Utilizada por um processo para receber uma mensagem. Ao término da operação, m conterá a primeira mensagem da fila de mensagens associada à porta p. Se não existirem mensagens na porta p, o processo ficará bloqueado, na fila de processos bloqueados a espera de mensagens. O pseudocódigo a seguir ilustra o uso destas primitivas.

```
port_t p;
// processo 0
p0 ( ) {
   while(1){
        -
        send (p, &m);
        }
 }

// processo 1
p1 ( ) {
   while(1){
         -
         receive (p, &m);
        }
  }
main() {
    p=port_create();
   create (p0);
   create (p1) ;
   }
```

No sistema operacional desenvolvido para o computador RC4000[The Nucleus of a Multiprogramming System, Brinch Hansen 1970], Brinch Hansen utilizou troca de mensagens para implementar a comunicação entre processos.

O sistema possui um núcleo, com as funções mais básicas de salvamento e restauração de contexto, tratamento de interrupções, escalonamento, as funções de criação e gerenciamento de processos que podem ser chamadas por outros processos e as funções de comunicação entre processos. As demais funcionalidades do sistema foram implementadas como processos concorrentes.

O sistema possui dois tipos de processos: internos e externos. Processos internos são os programas em execução e processos externos são relacionados às operações de entrada e saída.

O núcleo do sistema gerência um conjunto de buffers de mensagens do sistema e uma fila de mensagens para cada processo. A comunicação entre os processos internos no sistema pode ser feita com o uso das seguintes primitivas:

send message (receiver, message, buffer): copia a mensagem *message* para o primeiro buffer disponível dentro do pool de buffers e coloca na fila do receptor *receiver*. O receptor é ativado se estiver esperando uma mensagem. O remetente continua após receber a identificação do buffer da mensagem.

wait message (sender, message, buffer): o processo fica bloqueado até que uma mensagem seja depositada em sua fila. Quando isso acontece, o processo é acordado e está de posse do nome do processo remetente (*sender)*, do conteúdo da mensagem (*message)*, e da identidade do buffer de mensagem. O buffer é removido da fila e tornado disponível para transmitir uma resposta.

send answer (result, answer, buffer): copia a resposta *answer* no buffer utilizado pela mensagem recebida e o coloca na fila do processo que enviou a mensagem. O sender da mensagem é ativado se está esperando por uma resposta. O processo que executa a operação *send answer* continua imediatamente após a execução da primitiva.

wait answer (result, answer, buffer): bloqueia o processo que a executa até que uma resposta chegue no *buffer* especificado. Quando a resposta chega é copiada para o espaço de endereçamento do processo (*answer*) e o buffer é retornado para o pool de buffers do sistema. *result* especifica se a resposta foi enviada por um outro processo ou pelo núcleo do sistema, se a mensagem foi enviada a um processo não existente.

6.2 Comunicação em Grupo

Comunicação em Grupo é um modelo de comunicação que envolve um **grupo de processos**. Neste modelo, várias formas de comunicação são possíveis, podendo-se destacar a comunicação um-para-todos, todos-para-um e todos-para-todos.

Na comunicação **um-para-todos**, também chamada de difusão, um dos processos membros do grupo envia dados para o restante do grupo. Na comunicação **todos-para-um** os processos pertencentes a um mesmo grupo enviam uma mensagem para um único processo destinatário. Na comunicação **todos-para-todos**, cada membro de um grupo envia uma mensagem para os demais componentes do grupo. Da mesma forma que na troca de mensagens **ponto-a-ponto**, a difusão de mensagens para um grupo e a recepção de mensagens produzidas por membros de um grupo podem ser bloqueantes ou não-bloqueantes.

Entre as principais vantagens do modelo de comunicação em grupo pode-se citar a maior facilidade de expressão da comunicação envolvendo múltiplos processos, assim como o melhor desempenho que este tipo de comunicação pode oferecer em comparação à troca de mensagens ponto-a-ponto (a que envolve um processo emissor e um processo receptor). A primeira vantagem pode ser evidenciada supondo-se que uma mensagem deva ser enviada para múltiplos processos e que não existam primitivas de comunicação em grupo, o processo emissor, terá que enviar uma mensagem para cada processo receptor. Assim, o processo remetente deverá enviar tantas mensagens quantos forem os processos destinatários. No entanto, se os processos destino forem incluídos em um grupo, basta para o emissor enviar uma única mensagem (de difusão) à este grupo para que todos os processos a recebam. A segunda vantagem da comunicação em grupo, diz respeito ao maior desempenho que pode ser oferecido por este tipo de comunicação, comparado ao da troca de mensagens ponto-a-ponto. Como a comunicação em grupo é um mecanismo de mais alto nível, os detalhes de sua implementação são totalmente transparentes ao usuário, o que permite que protocolos mais eficientes possam ser utilizados para a implementação da comunicação. Considerando ainda o exemplo da

difusão de uma mensagem, a implementação deste tipo de interação entre processos pode levar em conta o nodo onde os mesmos residem, de forma que apenas uma mensagem seja endereçada a cada nodo contendo processos que devam receber a mensagem. Deste modo, quando a mensagem chega em um nodo destino será encaminhada a todos os processos que devem receber a mensagem no nodo.

Na comunicação em grupo, vários aspectos, especialmente relacionados à organização interna dos grupos, devem ser levados em consideração. Conforme sua organização interna, grupos podem ser classificados em *estáticos ou dinâmicos*, *abertos ou fechados*, *pares ou hierárquicos*

Um grupo é **estático** se, após ter sido formado, permanece sempre com o mesmo conjunto de membros. O tamanho, isto é, o número de elementos deste tipo de grupo é constante. Em um grupo **dinâmico**, processos podem entrar e sair do grupo sempre que necessário. O tamanho deste tipo de grupo portanto é variável. Grupos **fechados** não permitem que processos de fora se comuniquem com o grupo. Somente a comunicação entre os membros do grupo é permitida. Nos grupos **abertos** é possível que processos de fora do grupo se comuniquem com seus membros. Um grupo é **par** quando não existe hierarquia entre os seus membros, isto é, todos os processos que pertencem ao grupo são iguais. Existindo hierarquia o grupo é classificado como **hierárquico**. Neste caso, um processo é o coordenador e os demais são os trabalhadores. É o coordenador quem recebe as mensagens endereçadas ao grupo e escolhe o trabalhador que deverá executar a ação. As figuras a seguir apresentam dois grupos, um aberto não hierárquico e um fechado hierárquico.

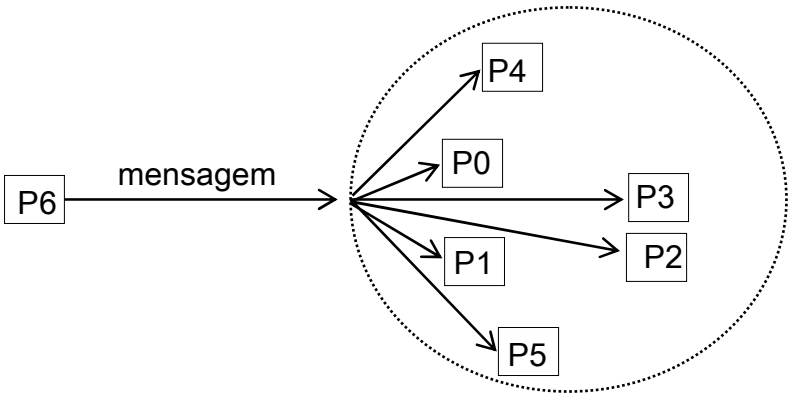

Fig. 6.1 - Grupo aberto não hierárquico

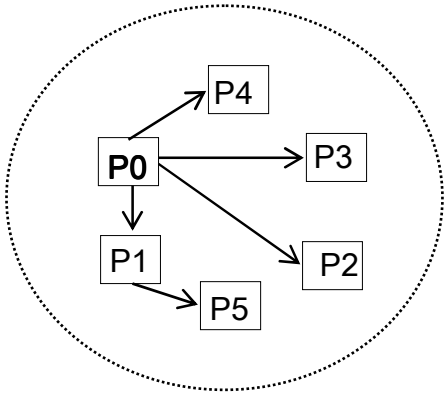

Fig. 6.2 - Grupo hierárquico fechado

No caso do grupo aberto não hierárquico, a mensagem enviada pelo processo P6, não pertencente ao grupo, será recebida por todos os membros do grupo. No grupo fechado hierárquico, o coordenador do grupo, P0, encaminha uma mensagem a todos os demais participantes do grupo.

6.3 Chamada remota de procedimento (Remote Procedure call)

O modelo cliente/servidor corresponde a idéia de estruturar um sistema operacional em um grupo de processos cooperantes, os servidores que

fornecem serviços a outros processos, os clientes. Esse modelo foi usado em vários e em sistemas operacionais para máquinas paralelas. Nesse modelo, um cliente envia uma mensagem a um servidor solicitando um serviço e fica bloqueado esperando a resposta: é a chamada de procedimento a distância (RPC).

O modelo RPC

As chamadas de procedimentos a distância (RPC) são um eficiente mecanismo de comunicação usados nos sistemas distribuídos e nos sistemas para máquinas paralelas sem memória comum, organizados de acordo com o modelo cliente/servidor.

O modelo RPC [Bi84] é uma extensão distribuída do modelo procedural existente nas linguagens de programação tradicionais. A idéia principal é permitir a chamada de um procedimento em uma máquina por um programa sendo executado em outra máquina. Em um RPC, um programa em uma máquina P, (o cliente) chama um procedimento em uma máquina Q, (o servidor) enviando os parâmetros adequados. O cliente é suspenso e a execução do procedimento começa. Os resultados são enviados da máquina Q para a máquina P e o cliente é acordado.

Nos sistemas distribuídos o uso de RPC permite obter um comportamento idêntico a uma chamada de procedimento nos sistemas operacionais tradicionais. Por exemplo, nos sistemas operacionais tradicionais as chamadas de sistema são implementadas por procedimentos que são chamados pelos programas dos usuários. Essas procedures recebem os parâmetros (ex. nos registradores da máquina) e são invocadas com o uso de recursos específicos da arquitetura física (ex. Trap).

No caso de um RPC, o funcionamento de uma chamada não é assim tão simples. O esquema de funcionamento proposto por Nelson[Bi84] é o seguinte:

Toda função que pode ser chamada à distância é definida no lado do cliente por uma procedure chamada stub client. Essa procedure recebe os

parâmetros e os coloca em uma mensagem, adicionando a identificação da procedure chamada e transmite essa mensagem ao processo/processador servidor. O servidor, quando recebe a mensagem, procede a extração dos parâmetros e chama a procedure especificada na mensagem. No fim da execução da procedure, é realizada a operação inversa: Colocação dos resultados e envio da mensagem de resposta ao processo cliente. A procedure que realiza essa operação é

chamada de stub server. Finalmente, a procedure stub client recebe os resultados, termina e retorna o controle ao cliente .

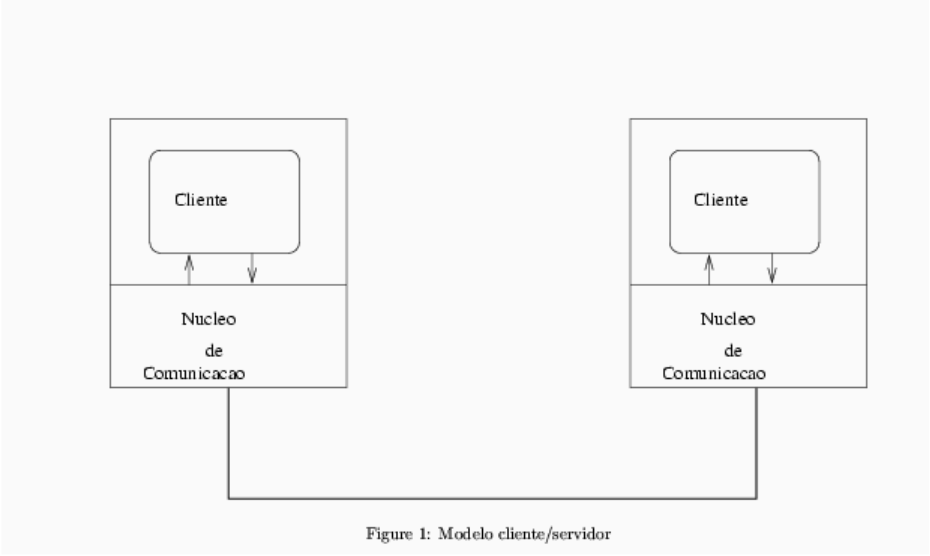

Fig. 6.3 – Modelo cliente/servidor

Uma qualidade importante deste esquema é que ele repousa unicamente no modelo de processos comunicantes. Por isso, ele funciona independentemente da localização do cliente e do servidor. Se eles são localizados na mesma máquina a comunicação se fará localmente, senão ela se fará através da rede.

O funcionamento desse modelo coloca em jogo cinco elementos:

- O processo cliente;
- O processo servidor;
- O procedimento chamado à distância;
- O procedimento stub client ;
- O procedimento stub server.

Os modelos síncrono e assíncrono

Em um RPC síncrono o cliente é suspenso até o momento que a resposta chegue. Uma crítica a esse modelo é que ele não explora o paralelismo existente entre cliente e servidor nas arquiteturas paralelas ou distribuídas. Em um RPC assíncrono, a chamada não suspende o cliente e as respostas são recebidas quando elas são necessária. Assim, o cliente continua a sua execução em paralelo com o servidor.

Independentemente do assincronísmo entre cliente e servidor, existe também o paralelismo possível na execução de uma procedure. Essa execução pode ser exclusiva e as chamadas serializadas. Em um outro extremo pode-se permitir tantas execuções concorrentes quantas forem as chamadas em um dado momento. Por exemplo, para cada chamada de procedimento recebida por um servidor ele pode criar uma thread para tratá-la e voltar a aceitar novas requisições. Uma nova requisição que chega pode ser de um procedimento já em execução por uma thread, o que não impedirá a criação de uma nova thread para executar uma nova instância desse procedimento.

O princípio da solução

Um RPC síncrono trata as chamadas locais e distantes de maneira idêntica. O princípio da solução adotada é o seguinte: o cliente faz a chamada à uma procedure de biblioteca (stub client). Essa procedure pega os parâmetros, junta a identificação da procedure e do servidor que deve tratar a chamada,

coloca essas informações em uma mensagem e a envia. O cliente é suspenso.

A localização do servidor destinatário da mensagem pode ser feita pelo sistema operacional, com um servidor de nomes, ou o usuário pode especificar o servidor ao qual a mensagem é dirigida. O servidor pode ser local ou distante. As etapas de uma chamada local são as seguintes:

1. O cliente chama a procedure e envia os parâmetros;

2. O cliente é suspenso;

3. A procedure constrói a mensagem;

4. Envia a mensagem para o núcleo local de comunicação;

5. O núcleo de comunicação analisa o header da mensagem e descobre que o servidor é local (mesmo processador);

6. Aloca um buffer;

7. Faz uma cópia da mensagem para o buffer;

8. Coloca o endereço do buffer na fila de mensagens do servidor;

9. Acorda o servidor;

10. O servidor pega a mensagem;

11. Analisa os parâmetros;

12. Dispara a procedure correspondente ao serviço;

13. Fabrica a mensagem de resposta;

14. Envia a mensagem de resposta ao núcleo de comunicação;

15. O núcleo de comunicação acorda o cliente;

16. O stub client extrai os resultados;

17. O cliente que fez a chamada inicial recomeça;

A figura a seguir mostra uma chamada à um servidor local.

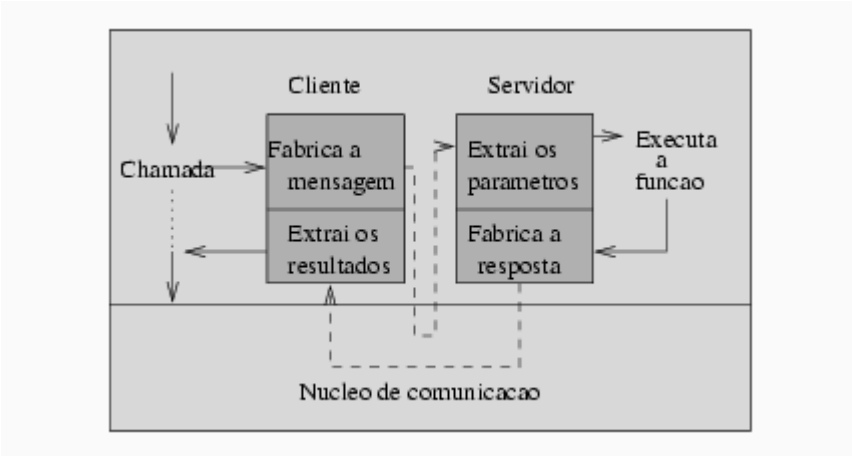

Fig. 6.4 - Troca de mensagens entre cliente e servidor local

No que diz respeito as chamadas de servidores distantes, a diferença é a ação do núcleo de comunicação:

1. Procura em sua tabela de servidores a identificação do servidor;

2. Envia a mensagem endereçada a este servidor;

3. O núcleo de comunicação do processador destinatário recebe a mensagem;

4. Analisa seu header e determina o servidor destinatário;

5. Aloca um buffer e copia a mensagem;

6. Coloca o endereço do buffer na fila de mensagens do servidor e o acorda.

A figura abaixo apresenta uma troca de mensagens entre cliente e servidor distantes.

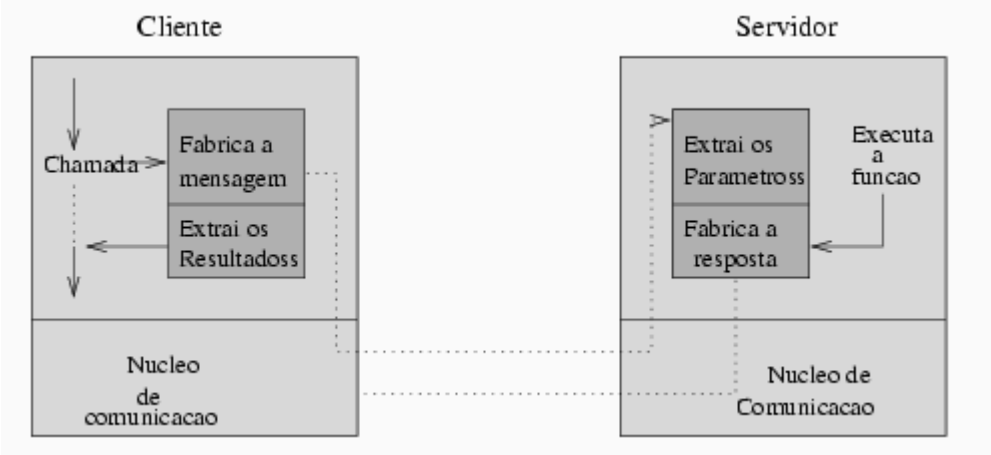

Fig. 6.5 - Troca de mensagens entre cliente e servidor distantes

Implementação

A seguir será feita a descrição da implementação de um RPC síncrono, considerando-se um ambiente formado por um conjunto de nodos processadores que não compartilham memória, interligados por uma rede física. Em cada nodo processador executa um Microkernel, que implementa as funções de mais baixo nível e um conjunto de servidores especializados, que implementam os serviços.

O núcleo local

O núcleo que roda em cada nodo processador possui três níveis lógicos: O núcleo de comunicação, o núcleo local e o nível dos usuários (figura a seguir).

Fig. 6.6 - Núcleo local

O núcleo de comunicação é responsável pela recepção e encaminhamento de mensagens locais ou entre processadores. No caso de recepção de uma mensagem, duas situações podem se produzir: na primeira, a mensagem lhe pertence e ele deve transmití-la ao processo destino; na segunda, a mensagem pertence a um outro processador e ele deve encaminhá-la propriamente para que ela chegue ao seu destino. Este encaminhamento progressivo é assegurado em cada nodo por uma função de roteamento local, definida por uma tabela de roteamento. Esta tabela também é utilizada para o envio de mensagens distantes. Quando uma mensagem é gerada localmente, o núcleo de comunicação é ativado. Se o processador destino é um outro, ele envia a mensagem com a identificação do nodo destino, em caso contrário, ele a envia ao processo local. O nível chamado de núcleo local é responsável pelas funções de gerência de processos e de gerência de memória. Ele implementa operações de:

- Criação/destruição de processos;
- Sincronização entre processos;
- Alocação/liberação de memória.

No nível usuários é que executam todos os servidores do sistema. É também neste nível que rodam as aplicações dos usuários.

Descrição do RPC implementado

A implementação do modelo RPC é caracterizada pela estrutura e troca de mensagens, pelo stub client, pelo servidor, pelo stub server e pelo mecanismo de troca de mensagens.

A mensagem

A mensagem (figura a seguir) pode ser dividida em duas partes: Um header e um corpo. O header possui a identificação tipo da mensagem, do cliente e do servidor e o corpo contém a mensagem propriamente dita. O tipo pode ser um SEND (pedido de serviço) ou um REPLY (resultados da execução de uma função). O cliente é identificado por (processador, &message). Processador é

onde ele reside e &message contém o semáforo no qual ele está bloqueado a espera da resposta. O servidor é identificado por (Processador, Servidor, Numero_função). Processador é o processador onde o servidor reside. Servidor identifica o servidor. A cada servidor existente no sistema é associado um número inteiro, conhecido globalmente. Numero_função identifica a função que o servidor deverá executar. Da mesma maneira que um servidor é identificado por um número inteiro, as funções também o são. Por exemplo, se ao servidor FILE-SYSTEM é associado o número 1 e a procedure READ o número 7, considerando que esse servidor resida no processador 1, a chamada:

n = read (file_descriptor, &buffer, nbytes);

fará com que seja construída uma mensagem endereçada ao processador número 1, servidor número 1 que deverá executar a procedure de número 7. O corpo da mensagem pode ser formado por palavras, registros, ou ambos e constituem os parâmetros de chamada da função. Em qualquer caso, é especificado na mensagem o número de palavras e o número de registros que ela contém. A figura a seguir mostra a organização de uma mensagem.

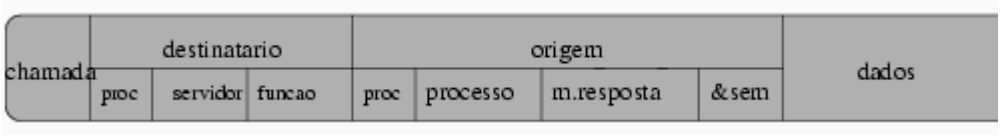

Fig. 6.7 - Estrutura de uma mensagem

O stub client

Quando um cliente faz um RPC, ele chama uma procedure de biblioteca, stub client, com os parâmetros apropriados. Essa procedure fabrica a mensagem e a envia ao núcleo local de comunicação que a envia ao servidor local ou ao processador destinatário, se o servidor é distante. O cliente fica bloqueado esperando a resposta.

O servidor

A função de um servidor é de tratar demandas de clientes. Cada servidor é responsável por um conjunto de serviços, por exemplo, o File System é

responsável pelas funções de gerência de arquivos, o Memory Manager pelas funções de gerência de memória, etc. Os servidores executam o seguinte procedimento:

- Pegar a primeira mensagem da fila;
- Identificar a função;
- Verificar o número de parâmetros;
- Chamar a procedure correspondente;
- A procedure se executa;
- Chamar o stub server para fabricar a resposta;
- Ativar o núcleo para retransmitir a resposta.

A estrutura geral de um servidor (ex. file system) é a seguinte:

```
int main() {
        for(;;) {
            P(message);
             switch (message.fonction)
                    --
                case READ: do_read() ;
                case WRITE: do_write() ;
              }
}
```

O stub server

A função do stub server é construir a mensagem de resposta. Para isso, ele pega os resultados e os coloca em uma mensagem. Os resultados podem ser

palavras ou registros e, em qualquer caso, a respectiva quantidade deve ser especificada na mensagem. Após, ele inverte o header da mensagem, isso é, ele pega processador origem e o coloca como destinatário e destinatário como origem. Como identificação de processo destinatário ele coloca o semáforo no qual o cliente está bloqueado a espera da resposta. O núcleo de comunicação do processador origem da mensagem (ping por ex.), quando receber essa mensagem a identificará como resposta e, nesse caso, desbloqueará o processo cliente (operação V no semáforo).

O mecanismo de troca de mensagens

O mecanismo de base que suporta a troca de mensagens no sistema é o núcleo de comunicação. Esse núcleo é organizado em torno de threads. Existem dois tipos de threads: Um que recebe as mensagens do meio físico e que permite a comunicação entre processadores diferentes. O outro tipo é formado por threads de controle que são associadas aos processos existentes no sistema, existindo uma para cada processo. Esse último grupo é encarregado de enviar as mensagens, locais ou distantes, que são geradas pelo processo. Esse núcleo é o suporte básico da implementação do protocolo de comunicação Cliente/servidor implementado.

Um exemplo de RPC

Para mostrar mais em detalhes a execução de um RPC, será apresentada uma chamada simples, ping, que permite saber se um processador está ativo. Considere um cliente que executa no processador 5 o comando

```
i = ping ( 12 ) ;
```

onde i é um inteiro e 12 é o processador que se quer saber se está ativo. A procedure

```
int ping ( int proc ) {

--
```

```
message m ;

m.origem.client = &m ;

m.dest.proc = 12 ;

m.dest.server = SYSTEM ;

m.dest.fonction = PING ;

m.message_type = SEND ;

out ( channel , &m ) ;

P ( m.semaforo ) ;

}
```

fabrica a mensagem. Ela aloca uma estrutura mensagem (m), a qual completa com os parâmetros recebidos e com outras informações. Em m.origem.client ela coloca o endereço da mensagem, depois ela completa os campos que identificam o processador, o servidor e o tipo da mensagem (SEND). Então, ela ativa o núcleo de comunicação com a operação out . Cada processo no sistema possui uma thread de controle que lhe é associada quando de sua criação e que pertence ao núcleo de comunicação. A comunicação entre o processo e a sua thread de controle é feita através de um canal lógico global a essas duas entidades. A thread fica sempre a espera de uma mensagem nesse canal (operação in) e ela é ativada pela operação out nesse canal. A instrução out (channel ,&m) coloca no canal lógico o endereço da mensagem e ativa a thread. Assim, a mensagem é enviada do processo cliente para o núcleo de comunicação. A seguir, o cliente é bloqueado, com a operação P no semáforo que existe na mensagem. A thread que é ativada executa

```
in ( channel, m ) ;

m.origem.proc = my-number ; /* 5 neste exemplo */

canal_fisico = get_route( m->dest.proc ) ;
```

out (canal-fisico, m) ;

A operação in recebe a mensagem. A thread completa a mensagem com a identificação do processador origem, número lógico atribuído a cada processador quando da fase de boot e conhecido do núcleo. Após, com o uso da função get-route obtém o endereço do processador destino (12) e envia a mensagem com a operação out(canal_fisico, m). A mensagem atravessa a rede de processadores e chega no núcleo de comunicação do processador 12. A thread do núcleo de comunicação que recebe as mensagens que chegam no processador 12 executa:

```
buffer = in ( canal_fisico, m ) ;
switch ( buffer.dest.server )
        --
        case SYSTEM: --
                        coloca a mensagem na fila do servidor
                        V( descripteur_server[SYSTEM].semaforo ) ;
                        break ;
         case MEMORY_MANAGER:
         case FILE_SYSTEM:
          --
          --
```

Inicialmente a thread lê a mensagem do canal físico e a coloca em um buffer. Cada servidor no sistema é descrito por um registro descritor que contém, entre outras informações, o endereço de uma fila de mensagens a ele endereçadas e um semáforo no qual ele fica bloqueado a espera de uma mensagem. A entrada descriptor_server[SYSTEM] no array de descritores

descreve o servidor SYSTEM. A thread receptora coloca a mensagem na fila do servidor SYSTEM e o acorda com a operação V em seu semáforo.

O código do servidor é o seguinte:

```
system () {
        --
        for (;;) {
                P( descriptor_server[SYSTEM].semaforo ) ;
                message = descriptor_server[SYSTEM].m ;
                switch (message.fonction )
                        --
                        case PING:    message.type = REPLY ;
                                      message.msg_reply = 1 ;
                                      message.dest.proc =
                                      message.m.origem.proc ;
                                      message.dest.server = &message ;
                                      out ( channel, &message ) ;
                                      break ;
           }
}
```

Quando o servidor SYSTEM recomeça, pela ação da operação V, ele pega a mensagem, simbolizado pela instrução

message = descriptor_server[SYSTEM].m)

e identifica a função. No caso de um PING a função é muito simples, consiste unicamente em enviar a resposta. Então, ele fabrica a mensagem resposta

colocando REPLY como tipo, 1 no campo msg_reply indicando sucesso na execução do PING e inverte os campos que identificam o processador na mensagem original. Assim, o processador origem se torna destinatário e o destinatário origem. Como servidor destinatário ele coloca o endereço da mensagem e a envia ao núcleo de comunicação com a operação out que desbloqueia a sua thread de controle. A thread de controle completa a mensagem colocando a identificação do processador como origem e analisa o processador destinatário. Como trata-se de um outro (5), ele obtém o endereço deste processador com a função

canalfisico = get_route(m->dest.proc)

e envia a mensagem. A seguir é apresentado o fragmento de código da thread que executa essas ações.

```
in ( channel, message ) ;

message.origem.proc = my_number ; /* 12 neste exemplo */

if ( message.dest.proc ! = my_number ) {

        canal_fisico = get_route( message->dest.proc ) ;

        out ( canal_fisico, message ) ;

        }

else

    “  para uma mensagem local”
```

O núcleo de comunicação do processador originário da mensagem (5) recebe a mensagem e a reconhece como resposta. Neste caso, ele acorda o cliente com a operação V no semáforo que ele estava bloqueado. O código a seguir apresenta a recepção da resposta.

```
in ( canal-físico, m ) ;

if ( m.type == REPLY )
```

V (m.semaforo) ;

else

“ É um pedido de serviço para um servidor”

6.4 Estudo de caso: Message Passing Interface - MPI

MPI é uma biblioteca de comunicação que permite a programação paralela baseada em troca de mensagens. Foi definida pelo MPI Fórum (www.mpi-forum.org), com a participação de Universidades, empresas, laboratórios de pesquisa. A versão 1.0, definida no MPI Fórum, se tornou disponível em maio de 1994, e contém as especificações técnicas da interface de programação.

Em MPI uma execução compreende um ou mais processos que se comunicam chamando rotinas da biblioteca para enviar e receber mensagens. Este conjunto de rotinas pode ser utilizado a partir de programas escritos em ANSI C, Fortran, C++ e Java.

Um programa MPI é formado por um conjunto de processos, criados no momento da inicialização, sendo que pode existir mais de um processo por processador. Cada um desses processos pode executar um programa diferente, o que caracteriza o modelo de programação MPMD. No entanto, uma forma natural de programar é utilizando o modelo SPMD, no qual um mesmo programa é disparado em cada um dos processadores participantes da execução e em cada processador é selecionado para execução um trecho do programa.

O padrão MPI define funções para:

- Comunicação ponto a ponto;
- Operações coletivas;
- Grupos de processos;
- Contextos de comunicação;
- Ligação para programas ANSI C e Fortran 77;
- Topologia de processos.

A versão 1.1 de MPI possui um enorme conjunto de funções. No entanto, com um número reduzido (apenas 6) é possível resolver uma grande variedade de problemas. Nesta seção serão apresentadas as funções básicas de comunicação ponto a ponto que, juntamente com mais um número reduzido de funções, permite o desenvolvimento de programas. Serão também apresentados exemplos de programas escritos em MPI e as principais características de MPI-2, versão definida em 1997, que incorpora novas características de programação paralela à biblioteca MPI.

Conceitos básicos

A seguir serão apresentados alguns conceitos básicos MPI.

Processo

Cada programa em execução se constitui um processo. Desta forma, o número de processadores especificado pelo usuário quando dispara a execução do programa indica o número de processos (programas) em execução. Se o número de processadores físicos é menor que o número especificado, os processos são criados, circularmente, de acordo com a lista de processadores especificada na configuração do ambiente.

Mensagem

É o conteúdo de uma comunicação, formada por duas partes:

- **Envelope**
  Endereço (origem ou destino). É composto por três parâmetros:

- Identificação dos processos (transmissor e receptor);
- Rótulo da mensagem;
- Communicator.

- **Dado**

Informação que se deseja enviar ou receber. É representado por três argumentos:

- Endereço onde o dado se localiza;
- Número de elementos do dado na mensagem;
- Tipo do dado. Os tipos de dados na linguagem C são:

Tipos de Dados Básicos no C	
Definição no MPI	Definição no C
MPI_CHAR	signed char
MPI_INT	signed int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_LONG_DOUBLE	long double
MPI_LONG	Signed long int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_SHORT	Signed short int
MPI_BYTE	-
MPI_PACKED	-

Rank

Todo o processo tem uma identificação única atribuída pelo sistema quando o processo é inicializado. Essa identificação é contínua representada por um número inteiro, começando de zero até N-1, onde N é o número de processos. É utilizado para identificar o processo destinatário de uma mensagem, na operação send, e o processo remetente de uma mensagem, na operação receive.

Group

Group é um conjunto ordenado de N processos. Todo e qualquer group é associado a um communicator muitas vezes já predefinido como

"MPI_COMM_WORLD". Inicialmente, todos os processos são membros de um group com um communicator.

Communicator

O communicator é um objeto local que representa o domínio (contexto) de uma comunicação (conjunto de processos que podem ser endereçados).

Primitivas Básicas MPI

A seguir serão apresentadas as primitivas básicas de MPI.

MPI_Init(&argc , &argv)

Inicializa uma execução em MPI, é responsável por copiar o código do programa em todos os processadores que participam da execução. Nenhuma outra função MPI pode aparecer antes de MPI_INIT. argc, argv são variáveis utilizadas em C para recebimento de parâmetros.

MPI_Finalize()

Termina uma execução MPI. Deve ser a última função em um programa MPI.

MPI_Comm_Size(*communicator* , *&size*)

Determina o número de processos em uma execução. *communicator* indica o grupo de comunicação e &size contém, ao término da execução da primitiva, o número de processos no grupo.

MPI_Comm_Rank(*communicator , &pid*)

Determina o identificador do processo corrente. *communicator* indica o grupo de comunicação e &*pid* identifica o processo no grupo.

MPI_Send (*&buf, count, datatype, dest, tag, comm*)

Permite a um processo enviar uma mensagem para um outro. É uma operação não bloqueante. O processo que a realiza continua sua execução. Os parâmetros são:

&buf: endereço do buffer de envio

count: número de elementos a enviar
datatype: tipo dos elementos a serem enviados
dest: identificador do processo destino da mensagem
tag: tipo da mensagem
comm:grupo de comunicação

MPI_Recv (*&buf, count, datatype, dest, tag, comm*)
Função responsável pelo recebimento de mensagens. É uma operação bloqueante. O processo que a executa fica bloqueado até o recebimento da mensagem. Os parâmetros são:

&buf: endereço do buffer de recebimento
count: número de elementos a enviar
datatype: tipo dos elementos a serem enviados
dest: identificador do processo remetente da mensagem
tag: tipo da mensagem
comm:grupo de comunicação
status:status de operação

Exemplo de programa

A seguir será apresentado um programa simples, no qual um processo envia uma mensagem para um outro, que a imprime.

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv) {
  int my_rank; /* Identificador do processo */
  int n;     /* Número de processos */
  char c[5] ;
  MPI_Status status ;

    MPI_Init(&argc , & argv);
```

```
  MPI_Comm_Rank(MPI_COMM_WORLD, &my_rank) ;
  MPI_Comm_Size(MPI_COMM_WORLD, &n) ;
  if (n != 2) exit() ;
  if (my_rank == 0) {
       strcpy (c, “ alo” ) ;
       MPI_Send (c, strlen (c),  MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  else {
       MPI_Recv (c, 5,  MPI_CHAR, 0, 99, MPI_COMM_WORLD, status) ;
       printf(“ %s\n” , c) ;
       }
  MPI_Finalize();
 }
```

O programa acima é formado por dois processos. Cada processo obtém sua identificação (my_rank) e se o numero de processos for diferente de 2 o programa termina (é para ser executado por dois processos). O processo 0 envia uma mensagem contendo “ alo” para o processo 1 com a primitiva Send. O processo 1 recebe a mensagem, primitiva Recv, e a imprime.

Modelos de programas paralelos

Um programa paralelo é composto por processos comunicantes que executam em processadores diferentes e que cooperam para a resolução de um cálculo. Os modelos que podem ser utilizados pelo programador para o desenvolvimento de suas aplicações são:

Divisão e conquista

Esta técnica consiste na criação de processos filhos para executar partes menores de uma tarefa. Os filhos executam e devolvem os resultados ao processo pai.

Pipeline

Os processos cooperam por troca de mensagens. Um conjunto de processos formam um pipeline, com a troca de informações em um fluxo contínuo. Para enviar dados a um processo sobre um outro processador, o processo remetente deve agrupar os dados e remeter. O processo recebedor extrai os dados de uma mensagem recebida, processa e envia para o processo seguinte no pipeline. Neste modelo, o paralelismo na execução é obtido quando cada um dos processos possui tarefas para processar.

Mestre/escravo

O programa é organizado como sendo formado por um processo (Mestre) que executa parte da tarefa e divide o restante entre os demais processos (Escravos). Cada escravo executa sua tarefa e envia os resultados ao mestre, que envia uma nova tarefa para o escravo.

Pool de trabalho

Um conjunto de tarefas é depositado em uma área acessível aos processos componentes do programa paralelo. Cada processo retira uma parte de uma tarefa e executa. Esta fase se repete até que o conjunto de tarefas seja executado.

Fases paralelas

O programa paralelo é formado por fases, sendo necessário que todos os processos terminem uma fase para passar a fase seguinte. Mecanismos de sincronização (ex. barreiras) são usados para que um processo espere pelos demais para passar à fase seguinte.

Com MPI é possível elaborar programas utilizando-se dos modelos acima apresentados. A seguir serão apresentados dois programas, um com o modelo Mestre/Escravo e outro um Pipeline.

Programa mestre/escravo

No programa a seguir um processo (o mestre) obtém valores, envia para os escravos calcularem a fatorial, recebe os cálculos e os imprime.

```
include "mpi.h"
int main(argc,argv)
int argc;
char **argv;
{
int numero, i, fat=1  ;
int myrank, size;
MPI_Status status;
MPI_Init (&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD,&size);

if (myrank==0){
      printf("Sou o processo 0 \n");
      for(i=1; i<4; i++){
            scanf(“%d”, &numero);
            printf("Numero: %d \n",numero);
      MPI_Send(&numero,1,MPI_INT,i,99,MPI_COMM_WORLD);
      }
      for(i=1; i<4; i++){
            MPI_Recv(&numero,1,MPI_INT,MPI_ANY_SOURCE,99,MPI_COMM_WOR
            LD,s&tatus);
            printf("resultado: %d \n",numero);
      }
      }
else
      {
      if (myrank==1)
            {
            printf("Eu sou o processo 1 \n");
            MPI_Recv(&numero,sizeof(int),MPI_INT,0,99,MPI_COMM_WORLD,
            &status);
            for(i=1; i<numero; i++)
                  fat = fat*i ;
            MPI_Send(&fat,1,MPI_INT,0,99,MPI_COMM_WORLD);
            }
      else {
            if (myrank==2)
            {
            printf("Eu sou o processo 2 \n");
            MPI_Recv(&numero,sizeof(int),MPI_INT,0,99,MPI_COMM_WORLD,
            &status);
            for(i=1; i<numero; i++)
                  fat = fat*i ;
            MPI_Send(&fat,1,MPI_INT,0,99,MPI_COMM_WORLD);
            }
            else{
                  printf("Eu sou o processo 3 \n");
                  MPI_Recv(&numero,sizeof(int),MPI_INT,0,99,MPI_COMM_
                  WORLD,&status);
                  for(i=1; i<numero; i++)
                        fat = fat*i ;
                  MPI_Send(&fat,1,MPI_INT,0,99,MPI_COMM_WORLD);
                  }
            }
}
MPI_Finalize();
```

```
}
```

O programa acima é formado por quatro processos: o mestre e três escravos. O mestre faz a leitura de três valores e envia um para cada escravo, com a primitiva Send, a seguir fica em um laço esperando pelos cálculos dos escravos. Para o recebimento dos resultados, primitiva Recv, a identificação do remetente é feita com MPI_ANY_SOURCE, que permite o recebimento de mensagens de qualquer processo. Cada escravo recebe o valor do mestre (primitiva Recv), calcula a fatorial do número recebido e envia este resultado para o mestre.

Programa Pipeline

O programa a seguir apresenta o exemplo de um pipeline.

```
#include "mpi.h"
main(argc,argv)
int argc;
char **argv;
{
int numero;
int myrank, size;
MPI_Status status;
MPI_Init (&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD,&size);

printf("Ola... \n");

if (myrank==0)
      {
      printf("Sou o processo 0 \n");
      scanf(“%d”, &numero);
      MPI_Send(&numero,1,MPI_INT,1,99,MPI_COMM_WORLD);
      MPI_Recv(&numero,1,MPI_INT,2,99,MPI_COMM_WORLD,&status);
      printf("Sou o processo 0, recebi do processo 2 o valor%d \n",
      numero);

      }
else
      {
      if (myrank==1)
            {
            printf("Eu sou o processo 1 \n");
            MPI_Recv(&numero,sizeof(int),MPI_INT,0,99,MPI_COMM_WORLD,
            &status);
            numero = numero + 10 ;
            MPI_Send(&numero,1,MPI_INT,2,99,MPI_COMM_WORLD);
            }
      else
            {
```

```
            printf("Eu sou o processo 2 \n");

            MPI_Recv(&numero,sizeof(int),MPI_INT,1,99,MPI_COMM_WORLD,
            &status);
            numero = numero + 10 ;
            MPI_Send(&numero,1,MPI_INT,0,99,MPI_COMM_WORLD);
            }
      }
MPI_Finalize();
}
```

No programa acima são criados três processos. O processo 0 obtém um valor e o envia para o processo 1. O processo 1 adiciona um outro valor ao recebido e envia ao processo 2, que adiciona um valor ao recebido e envia para o processo seguinte no pipeline, que no caso é o processo inicial (0).

Compilação e execução de programas

A compilação de programas escritos na linguagem C (C++) é feita com o comando

mpicc [fonte.c] -o [executável] [parâmetros]

onde o comando mpicc aceita todos os argumentos de compilação do compilador C. A execução é feita com o comando

mpirun -[argumentos] [executável]

Os argumentos são:

h - Mostra todas as opções disponíveis

arch - Especifica a arquitetura da(s) máquina(s)

machine - Especifica a(s) máquina(s)

machinefile - Especifica o arquivo que contém o nome das máquinas

np - Especifica o número de processos

leave_pg - Registra onde os processos estão sendo executados

nolocal - Não executa na máquina local

t - Testa sem executar o programa, apenas imprime o que será executado

dbx - Inicializa o primeiro processo sobre o dbx

Exemplos:

mpirun -np 5 teste

Executa o programa teste com 5 processos

mpirun -arch sun4 -np 5 teste

Executa o programa teste com 5 processos na arquitetura sun4.

MPI-2

Desde 1995 o Fórum MPI começou a considerar correções e extensões ao MPI padrão. Em julho de 1997 foi publicado um documento que, além de descrever MPI 1.2 e apresentar seus padrões, define MPI-2. MPI-2 adiciona novas funcionalidades à MPI, permitindo uma extensão aos modelos de computação. Estas novas funcionalidades permitem:

- **Criação e gerência de processos:** são definidas primitivas que permitem a criação dinâmica de processos.

- **Comunicação com um único participante:** são definidas operações de comunicação que são realizadas por um único processo. Estas incluem operações de memória compartilhada (get/put) e operações remotas de acumulação.

- **Extensões às operações coletivas:** novos métodos de construir intercomunicators e novas operações coletivas.

- **Interfaces Externas:** é definido um novo nível, acima de MPI, para tornar mais transparente objetos MPI.

- **I/O Paralelo:** são definidos mecanismos de suporte para I/O paralelo no MPI.

- **Linguagens**: são definidas novas características de ligação de MPI com C++ e com Fortran-90.

Conclusão sobre MPI

O estudo de MPI apresentado nesta seção tem como objetivo iniciar o leitor no desenvolvimento de programas paralelos com o uso desta biblioteca de comunicação. Foram apresentados conceitos básicos utilizados em MPI, as principais funções utilizadas para troca de mensagens e para inicializar e finalizar o sistema. Foram apresentados exemplos de programas em dois modelos de programação (mestre/escravo e pipeline) de maneira a permitir uma fácil compreensão.

Exercícios

1. Considerando um sistema em que os processos não compartilham memória, o mais indicado para troca de informações e sincronização são semáforos ou send/receive? Justifique.

2. Descreva uma possível forma de implementação das primitivas send/receive.

3. Descreva o funcionamento do mecanismo RPC (Remote Procedure Call).

4. Em um sistema implementado com RPC, sendo permitida a comunicação entre servidores é possível a ocorrência de deadlock? Como poderia ser evitado?

5. Considerando as primitiva *receive(-1, msg)* , que permite ao processo que a executa receber mensagens de qualquer processo do sistema e a primiva *send (proc, msg),* usada para enviar uma mensagem ao processo especificado por *proc*, apresente a forma geral de um processo servidor e de um processo cliente.

6. Apresente três situações em que a comunicação em grupo é a mais adequada?

7. Defina um problema e escreva um programa em MPI para solucioná-lo utilizando o modelo mestre/escravo.

7 Deadlock

São estudados neste capítulo princípios de deadlock, detecção e recuperação de deadlock, prevenção de Deadlock e algoritmos para evitar Deadlock.

7.1 Princípios de Deadlock

Modelo de sistema

Um sistema consiste de um número finito de recursos a ser distribuído entre um número de processos competindo pelos mesmos.

Recursos

São quaisquer entidades que puderem ocasionar bloqueio de processos. Um bloqueio ocorre quando um recurso não está disponível e é requisitado por um processo. Ex. de recursos:

Dispositivos de E/S

Memória

região crítica

Deadlock

É uma situação na qual processos estão bloqueados à espera de recursos que jamais serão liberados

Exemplo clássico de deadlock:

Supondo que o processo P1 requisite o recurso R1, execute algumas ações e requisite o recurso R2 e que o processo R2 requisite o recurso R2, execute algumas ações e requisite os recursos R1, como mostrado abaixo.

P1: ... R (R1); ... R(R2); ... L(R1,R2)

P2: ... R (R2); ... R(R1); ... L(R1,R2)

O deadlock ocorre quando os dois processos adquirem o primeiro recurso que necessitam. No exemplo, se o processo P1 adquire o primeiro recurso R1 (R(R1)) e antes de obter o segundo recurso, o processo P2 adquire R2 (R(R2)), o sistema entra em deadlock, pois tanto P1 quanto P2 ficarão bloqueados a espera de um recurso que nunca será liberado.

Tipos de recursos

Serialmente reusáveis

Constituídos de um número fixo de unidades idênticas, cada uma podendo ser utilizada de forma independente das demais. Após uma unidade ter sido utilizada por um processo, pode ser reutilizada por outro, de forma serial. Cada unidade pode estar em um dos seguintes estados: alocada, disponível.

Recursos consumíveis

São constituídos por um número variado de unidades que são criadas e consumidas dinamicamente pelos processos. Os processos que produzem as unidades são denominados produtores e os processos que consomem as unidades são denominados consumidores. São características dos sistemas de processos cooperantes (trocas de mensagens, sinas de sincronização)

Condições necessárias para a ocorrência de deadlock

1) Exclusão mútua: o processo solicita o recurso para uso de forma mutuamente exclusiva.

2) Espera por recursos: Processos possuem recursos enquanto esperam por recursos adicionais.

3) Não preempção: Quando os recursos não puderem ser confiscados temporariamente para serem alocados a outros processos.

4) Espera circular: Quando for possível a formação de um ciclo no qual cada processo está bloqueado à espera de recursos que estão alocados para outros processos de mesmo ciclo.

Métodos para tratamento de deadlock

1- Detectar e recuperar

2- Prevenir

3- Evitar deadlocks

7.2 Detectar e recuperar deadlocks

Representação

Uma aresta de um processo Pi para um recurso Ri significa a solicitação de uma unidade do recurso Ri. Uma aresta do recurso Ri para um processo Pi significa que o processo Pi possui alocada uma unidade do recurso Ri.

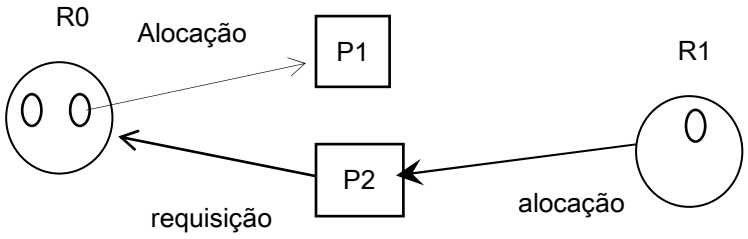

Fig. 7.1 – Grafo de alocação e requisição de recursos

Na figura acima, o processo P1 possui uma unidade do recurso R0 alocada. O processo P2 possui uma unidade do recurso R1 e está solicitando uma unidade do recurso R0. Na figura 7.2 abaixo,

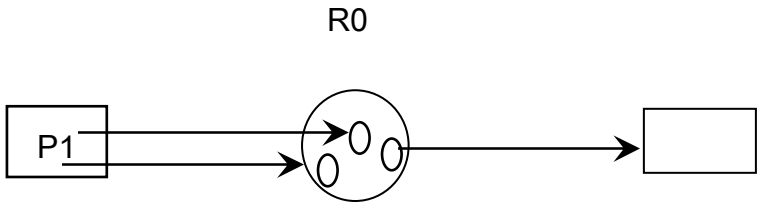

Fig. 7.2 – Alocação de R0 por P2 e requisição de R0 por P1

P2 possui uma unidade do recurso R0 alocada, e P1 está solicitando duas unidades deste mesmo recurso.

É possível, pela análise dos grafos, determinar se existe um deadlock.

“ Existe deadlock se as requisições para alguns processos nunca serão atendidas ”

Redução de grafos

É uma maneira simples de examinar grafos para determinar se existe deadlock “ Um grafo pode ser reduzido em relação a um processo se todas as requisições do processo podem ser garantidas ”

“ Os processos que não podem ser reduzidos são os que estão em deadlock“

Teorema:

“ Não existe processos em deadlock se e somente se o grafo é completamente redutível “

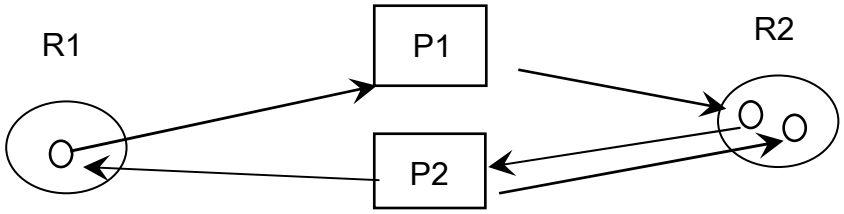

Fig. 7.3 – P1 possui uma unidade de R1 e requisita uma unidade de R2

Podemos reduzir o grafo por P1 porque suas requisições podem ser garantidas a redução é feita retirando-se as arestas de/para o processo

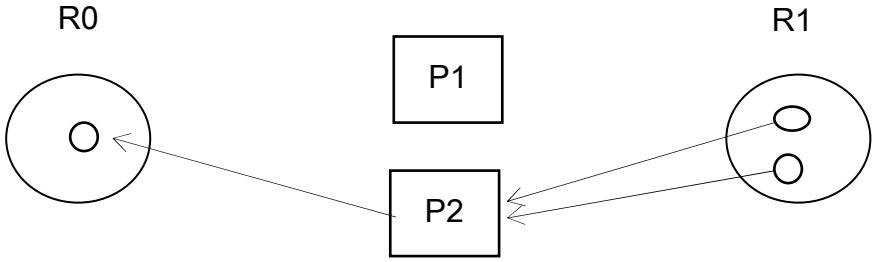

Fig. 7.4 – Grafo reduzido em relação a P1

P2 pode agora ter suas requisições atendidas. Portanto, os processos podem adquirir os recursos de que necessitam e o sistema não entra em deadlock.

Na figura 7.5 abaixo, temos uma situação de deadlock. O processo P1 possui uma unidade do recurso R1 e está solicitando uma unidade do recurso R2. Este pedido não pode ser atendido, pois as duas unidades de R2 estão alocadas para o processo P2. O processo P2 está solicitando uma unidade do recurso R1. Este pedido não pode ser atendido. Portanto, os pedidos de P1 e P2 não podem ser atendidos e estes processos estão em deadlock.

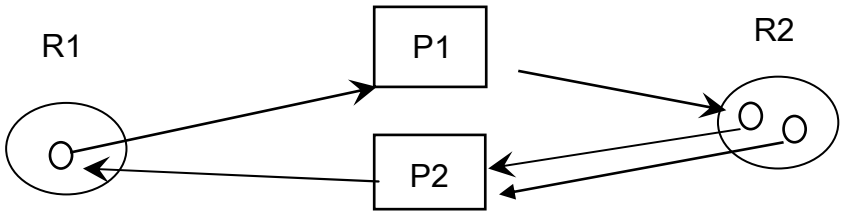

Fig. 7.5 – Grafo que mostra P1 e P2 em deadlock

Algoritmo de detecção de deadlock

Para um único recurso com muitas unidades

estruturas de dados

- vetor que representa o número de unidades necessárias para cada processo;
- variável inteira, que representa o número de unidades do recurso disponíveis.

Algoritmo

AVAIL: número de unidades presentemente disponíveis

```
begin
        for (i=1 to numberofprocess)
                reduced(i) := false,
        reducedprocess:=0;
“  Repetir até que não seja mais possível reduzir “
        loop
                exit when not reduction
                reduction := false;
```

```
            for ( P = 1 to numberofprocess) {
                if (not reduced(P)) {
                    if (req(P) <= avail) then {
                        reduced(P) := true;
                        reducedprocess := reducedprocess +1;
                        reduction := true;
                        avail := avail + aloc(P);
                    }
                }
            }
        end loop
        completelyreduced := (reducedprocess = numberofprocess);
end;
```

Algoritmo para vários recursos com várias unidades por recurso

req : matriz que representa as requisições dos processos

alloc : matriz que representa as unidades, de cada recurso, presentemente alocadas a cada processo

avail : vetor que representa o número de unidades, de cada recurso, presentemente disponíveis.

A figura 7.6 abaixo mostra as estruturas de dados utilizadas pelo algoritmo.

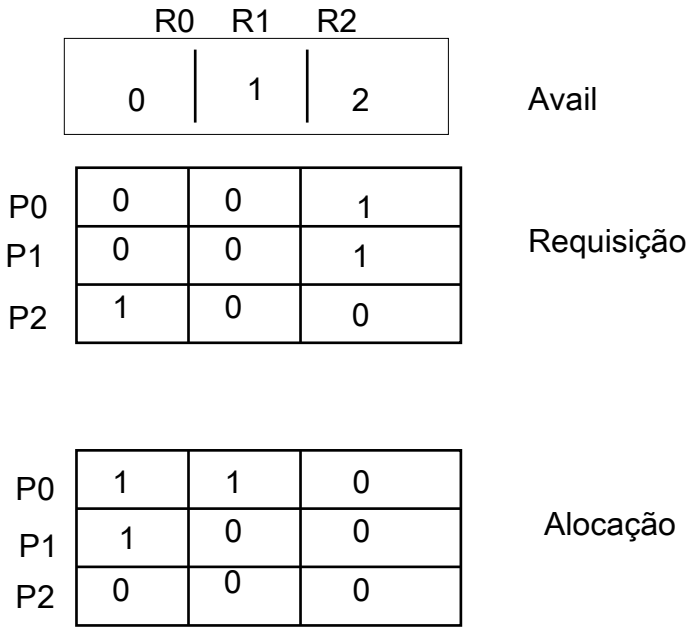

Fig. 7.6 – Estruturas de dados do algoritmo

Algoritmo para vários recursos com várias unidades por recurso

O algoritmo a seguir é uma extensão do apresentado anteriormente, e detecta a ocorrência de deadlock considerando um sistema com N recursos e cada recurso possuindo N unidades.

```
begin
        for (i=1 to numberofprocess)
                reduced(i) := false,
        reducedprocess:=0;
“  Repetir até que não seja mais possível reduzir “
        loop
                exit when not reduction;
                reduction := false;
                for ( P = 1 to numberofprocess) {
                        if (not reduced(P)) {
```

```
                    reducedbyp := true
                    for (R = 1; R <=  numberofressources; R++) {
                        if req(P)(R) > avail(R) then {
                        reducedbyp := false;
                        }
                    if (reducedbyp) {
                        reduced(P) := true;
                        reducedprocess := reducedprocess +1;
                        reduction := true;
                        for (R = 1; R <=  numberofressources; R++)
                                    avail(R) =  avail(R) + aloc(P)(R);

                    }
                }
        end loop
        completelyreduced := (reducedprocess = numberofprocess);
end;
```

Quando executar o algoritmo:

- Quando uma requisição não pode ser satisfeita;
- Quando existe suspeita de deadlock Ex: Um processo está bloqueado há muito tempo a espera de recursos.

Recuperação de deadlock

Quando o algoritmo de detecção de deadlock determina a existência de um deadlock, muitas ações são possíveis:

a) Terminação de processos

- Terminar todos os processos. Esta ação implica em reexecução do processo eliminado. Assim:
    - escolher uma vítima por vez até que o deadlock seja eliminado;
    - escolher criteriosamente o processo a ser terminado;
    - reexecutar o algoritmo de detecção.
    - Problema da reexecução de um processo:
        - nem sempre é possível ex. atualização de base de dados.

b) Preempção de recursos

Recursos são retirados de processos no ciclo e entregues a outros, no mesmo ciclo, até que o deadlock seja eliminado. O problema a ser resolvido é a definição de critérios para a escolha da vítima.

c) Rollback

- Os processos possuem checkpoints em que, periodicamente, o estado é gravado em um arquivo (imagem de memória, recursos);
- quando um deadlock é detectado, o processo é rolledback até antes de pedir um recurso;
- o recurso é atribuído a um outro processo no ciclo.

7.3 Prevenir a ocorrência de deadlocks

Segundo Coffman, as quatro condições devem estar presentes para um deadlock ocorrer. A ausência de uma condição impede a ocorrência de

deadlock. Portanto, previne-se situações de deadlock eliminando-se pelo menos uma das condições necessárias para sua ocorrência.

1) Exclusão mútua: o processo solicita o recurso para uso de forma mutuamente exclusiva. Esta condição é eliminada se o processo solicita todos os recursos que necessita em uma única vez.

2) Espera por recursos:. Processos possuem recursos enquanto esperam por recursos adicionais. Um processo pode requisitar e liberar recursos, mas quando requisitar um recurso não disponível deve liberar os que estava utilizando e, então solicitar todos coletivamente

3) Não preempção: Quando os recursos não puderem ser confiscados temporariamente para serem alocados a outros processos. Elimina-se esta condição se os recursos forem ordenados e os processos devem requisitar os recursos em uma seqüência que respeite esta ordenação.

4) Espera circular: Quando for possível a formação de um ciclo no qual cada processo está bloqueado à espera de recursos que estão alocados para outros processos de mesmo ciclo. Esta condição é eliminada se for construído um grafo e se for verificado, para cada requisição, se o atendimento não levará o sistema a um estado não seguro. As requisições que levarem o sistema a um estado não seguro ou de deadlock não deverão ser atendidas.

7.4 Evitar deadlocks

Os recursos são requisitados quando necessários e o sistema deve decidir se a requisição pode ser atendida sem gerar deadlock. Os algoritmos necessitam que os processos declarem o máximo de recursos, de cada tipo, necessários.

Estado seguro/não seguro

Se o sistema consegue alocar recursos para cada processo, em alguma ordem, e ainda evitar deadlock, o estado é seguro.

“ O estado é seguro se existe uma seqüência segura “

“ O estado é não seguro se não existe uma seqüência segura “

“ O estado de deadlock é um estado não seguro “

“ Nem todo estado não seguro é um estado de deadlock “

Com a informação de número máximo de recursos necessários para cada processo é possível construir um algoritmo que assegura que o sistema nunca entrará em deadlock. O algoritmo examina dinamicamente o estado de alocação de recursos para assegurar que não existe uma espera circular. O estado de alocação é definido pelo número de disponíveis, alocados e demanda máxima de cada processo.

Algoritmo para evitar deadlocks

O algoritmo que será apresentado (Banker`s algorithm) é baseado no estado e foi proposto pelo Dijkstra 1965.

Este algoritmo utiliza quatro estruturas de dados:

Disponível: Vetor que indica o número de unidades disponíveis de cada tipo de recurso

Máximo: Matriz que define, para cada processo, o número máximo de unidades, de cada tipo de recurso, que o mesmo pode alocar.

Alocação: Matriz que define o número de unidade, de cada tipo de recurso, alocadas presentemente para cada processo.

Necessidade: Matriz que indica o número de unidades, de cada recurso, necessárias para cada processo (Max_Allocation)

As figuras que seguem apresentam estas estruturas de dados.

Matriz Alocação

Processo	Fita	Impressora	CD
P0	2	0	0
P1	0	0	0
P2	0	1	1
P3	1	1	0

Fig. 7.7 – Matriz alocação

A figura acima mostra que P0 possui duas unidades de fita alocadas. P1 não possui recursos alocados, P2 possui uma unidade de cada recurso e P3 somente não possui a unidade de CD alocada. A matriz abaixo, figura 7.8, apresenta as necessidades de cada processo.

Matriz Necessidades

Processo	Fita	Impressora	CD
P0	1	1	1
P1	2	1	1
P2	0	0	0
P3	1	1	0

Fig. 7.8 – Matriz necessidades

A matriz necessidades contém o número de unidades de cada recurso que cada processo está solicitando. No exemplo, P0 está solicitando uma unidade de cada recurso. P1 está solicitando duas unidades de fita e uma impressora e a unidade de CD. P2 não está solicitando recursos e P3 está solicitando uma unidade de fita e uma impressora.

Matriz Máximo

Processo	Fita	Impressora	CD
P0	4	1	1
P1	2	1	1
P2	1	1	1
P3	2	1	1

Fig. 7.9 – Matriz máximo

A matriz máximo acima mostra o número máximo de recursos que cada processo pode alocar. Para cada processo, a soma de suas necessidades com a alocação, para cada recurso, não pode exceder o número máximo permitido. O vetor disponível, figura abaixo, contém o número de unidades disponíveis de cada recurso.

Vetor disponível

Fitas	Impressoras	CD
1	1	0

Fig. 7.10 – Vetor disponível

Considerando as estruturas de dados apresentadas anteriormente, temos que existem quatro unidades de fita (três alocadas e uma disponível), três impressoras (duas alocadas e uma disponível) e uma unidade de CD.

Sabendo que quando um processo tem suas necessidades atendidas executa por um período e libera os recursos que possui alocados, no exemplo mostrado anteriormente, as requisições podem ser atendidas e o sistema não entrará em deadlock: P0 não pode ter o pedido da unidade de CD atendido, porém, quando esta unidade for liberada por P2, que não está solicitando recursos neste momento, poderá ser alocada a P0, que portanto poderá ter seus pedidos atendidos. De maneira análoga, P1 e P3 poderão ter

seus pedidos atendidos, considerando a liberação dos recursos pelos demais processos.

O funcionamento do algoritmo é o seguinte: para cada processo é verificado se as suas requisições podem ser atendidas (se o número de unidade de cada recurso que o mesmo está solicitando é menor ou igual ao número de unidades disponíveis de cada recurso). Se as requisições puderem ser atendidas, o algoritmo verifica se o estado permanecerá seguro, isto é, as requisições podem ser atendidas e o sistema não entrará em deadlock. O corpo principal deste algoritmo é mostrado a seguir.

Considerando que para uma requisição request (P,i) = k o processo P esta solicitando k unidades do recurso i

```
if (request(P,i) <= need(P,i)) {
        if (request(P,i) <= available(i)) {
                available(i) = available(i) - k;
                allocation(P,i) = allocation(P,i) + k;
                need(P,i) = need(P,i) - k;
        }
        safe_state();
} else
        erro;
```

Safe_state: Estrutura de dados:

temp: vetor

Finish: vetor

temp[i] = available[i];

finish[i] = false; “ para todo i “

```
continue = true;
while(continue) {
        continue = false;
        if (need(i) <= temp(i) and finish(i) = false) {
                temp(i) = temp(i) + allocation(i);
                finish(i) = true;
                continue = true;
        }
}
if finish(i) = true para todo i then
        o sistema está em um estado safe
```

Exercícios

1. Explique as condições necessárias para a ocorrência de deadlock.
2. “ *Quando existe suspeita de deadlock deve ser executado o algoritmo que evita a ocorrência de deadlock*” . Esta afirmação está correta? Justifique.
3. Mostre, com o uso de vetores e matrizes contendo os recursos alocados, requisitados e disponíveis, uma situação em que existe deadlock.
4. Apresente um grafo de alocação e requisição de recursos, com no mínimo três recursos e três processos, no qual existe deadlock.

5. Considerando a existência de três processos (P0, P1 e P2), compartilhando três arquivos (F0, F1 e F2) e que os arquivos são acessados de forma exclusiva, apresente uma situação em que os processos estão em deadlock.

6. Explique como o deadlock pode ser eliminado confiscando recursos de um processo que está no ciclo e entregando a outro que também está no ciclo.

8 Gerência de Memória

Neste capítulo são estudados conceitos básicos de gerência de memória e aspectos de hardware e as políticas de partições fixas, partições variáveis, swapping, paginação, segmentação. O capítulo possui também um estudo sobre a gerência de memória no Linux e no Windows NT .

8.1 Conceitos Básicos e aspectos de hardware

Memória

- central em sistemas de computador
- CPU e sistema de I/O interagem com a memória
- é um conjunto (array) de bytes ou palavras, cada um com seu próprio endereço
- CPU faz busca e armazenamento na memória
- um programa deve ser carregado na memória para ser executado

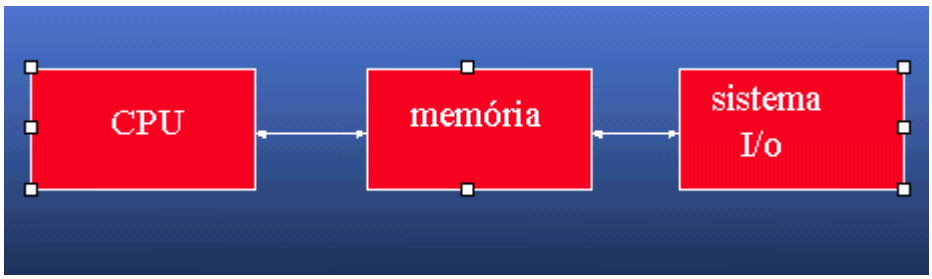

Fig. 8.1 - Interação CPU/ES/Memória

Memória física e Memória virtual

Memória física é a memória do hardware. Começa geralmente no endereço físico 0 e continua até o maior endereço, que indica o tamanho da memória. Certas posições são reservadas pelo hardware para objetivos especiais (ex. vetor de interrupções)

Memória virtual é a memória que o processo enxerga. É o espaço virtual de um processo. O maior endereço virtual é limitado pela arquitetura da máquina (No. de bits usados para endereçar). O espaço virtual pode ser maior que o espaço físico, especialmente se é usada paginação/segmentação e um processo pode acessar qualquer parte de seu espaço virtual.

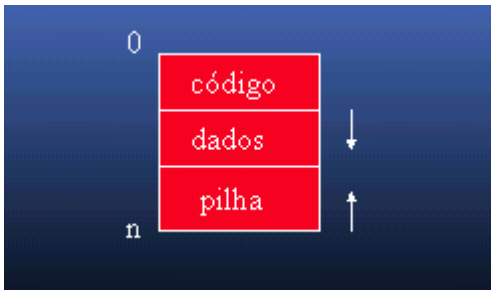

Fig. 8.2 - Espaço de endereçamento de um processo

Tradução de endereço

O processo trata com endereços virtuais. Em todo acesso deve haver uma tradução de endereços, o que significa que a tradução não deve ser feita por software, o que inviabilizaria o desempenho do sistema. A figura a baixo representa o mapeamento entre a memória virtual e a memória física.

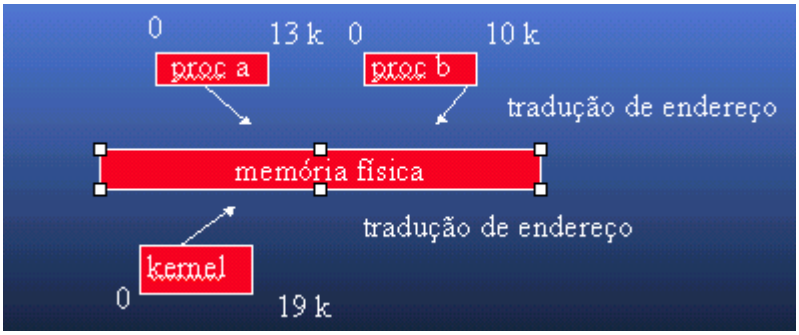

Fig. 8.3 - Mapeamento entre a memória virtual e a memória física

Monoprogramação e multiprogramação

Nos sistemas monoprogramados, existe um único processo na memória em execução e ao mesmo é permitido usar toda a memória. Com multiprogramação

existem vários processos na memória aptos à executar e um em execução.

Monoprogramação é o esquema mais simples possível:

- um processo por vez na memória;
- o processo pode usar toda a memória;
- a memória é dividida entre o sistema operacional e o processo do usuário (figura a seguir).

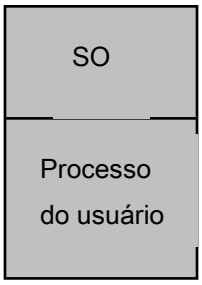

Fig. 8.4 - Organização da memória em um sistema operacional monoprogramado

Multiprogramação

Com multiprogramação, vários programas podem residir simultaneamente na memória, sendo que cada um executa por um determinado tempo. São necessários mecanismos de proteção (hardware) para que um processo não acesse indevidamente a área de memória de outro processo. Soluções possíveis para a implementação de proteção de hardware é a utilização de registradores para indicar o limite inferior e o limite superior da memória de um processo e de registradores base e limite. A figura a seguir ilustra o uso de registradores limite inferior e limite superior.

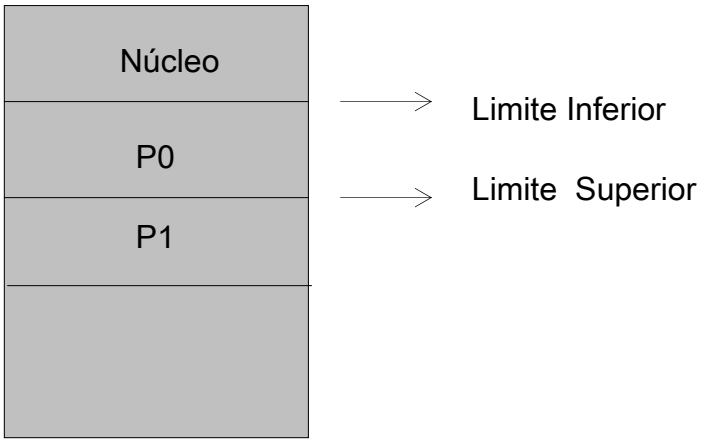

Fig. 8.5 - Uso de registradores Limite Inferior e Limite Superior

Na figura acima são representados o núcleo do sistema operacional e dois processos, P0 e P1. Supondo que P0 esteja em execução, o registradores Limite Inferior contém o valor do endereço mais baixo de memória ocupado por este processo e o registrador Limite Superior, o mais alto. A todo acesso a memória o hardware verifica se o endereço a ser acessado está entre estes dois valores. Em caso positivo, o acesso é realizado. Caso contrário é gerada uma exceção, endereço inválido, e o processo é terminado. Quando o processo P1 é selecionado para execução, estes registradores são carregados com os endereços de memória (inferior e superior) que o mesmo ocupa.

Registradores base e limite

O uso de registradores base e limite é uma solução superior ao método anterior (registradores Limite Inferior/Limite Superior), pois diminui o número de comparações. Quando o processo é selecionado, o endereço inferior de memória é atribuído ao registrador base. A todo acesso a memória, o endereço de leitura ou escrita é adicionado ao valor do registrador base. Se o endereço resultante é menor ou igual ao valor do endereço limite (registrador Limite), o acesso é realizado (endereço válido). Caso contrário, é gerada uma exceção: endereço fora do espaço de endereçamento do processo. A figura a seguir exemplifica o uso de registradores base e limite.

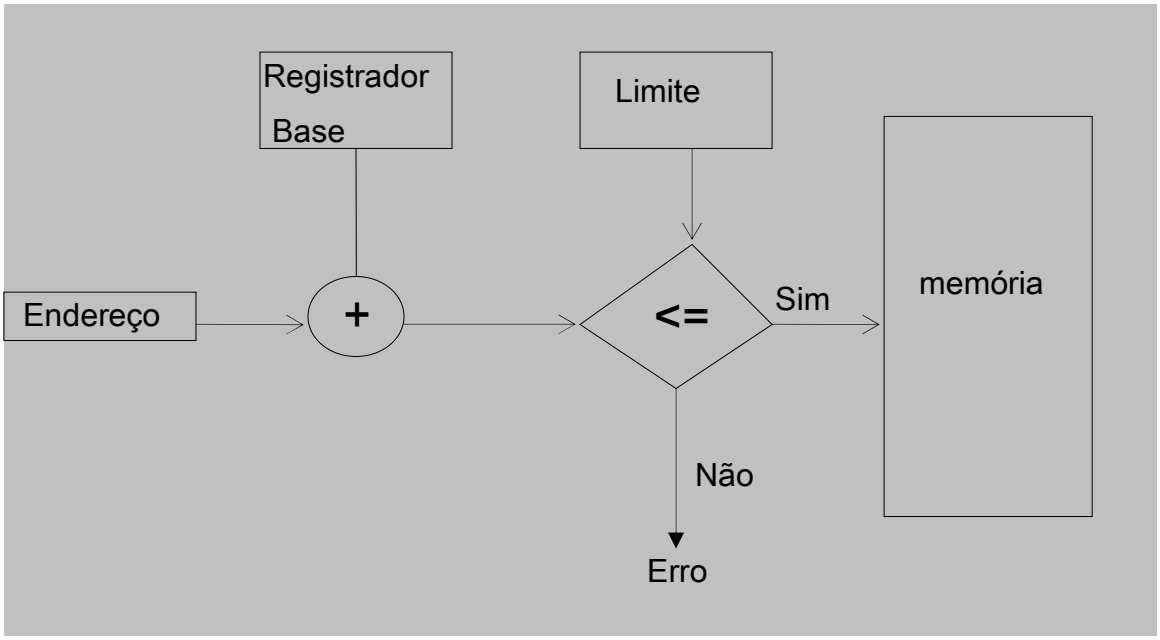

Fig. 8.6 - Uso de registradores base e limite na tradução de endereço

Com registradores limite muitos processos podem residir na memória simultaneamente. Duas políticas de gerência de memória são Partições Fixas e Partições Variáveis, apresentadas a seguir, podem ser implementadas com os mecanismos de hardware apresentados anteriormente.

8.2 Partições Fixas e Partições Variáveis

Com o uso de Partições Fixas o sistema operacional divide a memória em um certo número de partições de tamanha fixo. Uma das partições é ocupada pelo código do sistema operacional, e as demais pelos programas de aplicação. O tamanho das partições somente pode ser alterado por um usuário com privilégios especiais (super usuário), na carga do sistema. Considerando uma memória de 128KBytes, a mesma pode ser dividida em 4 partições de mesmo de 32K. Neste caso, todas as partições possuem o mesmo tamanho. Outra alternativa é dividir a memória existente em um certo número de partições de tamanhos diferentes. Por exemplo, pode-se ter uma partição de 32K para o SO, uma de 32 para processos pequenos (P0) e 1 de 64K para processos maiores (P1), figura abaixo.

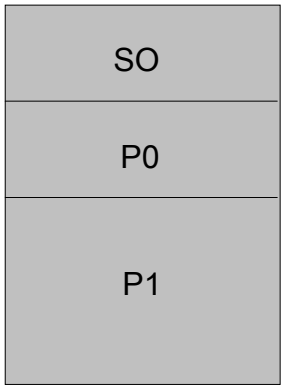

Fig. 8.7 - Memória física organizada em três partições de tamanha fixo

Para carregar um programa para execução, o scheduler verifica sua necessidade de memória, que partições estão disponíveis e carrega o processo em uma das partições com tamanho suficiente para contê-lo. O grau de multiprogramação, isto é, o número máximo de processos em execução simultânea, é determinado pelo número de partições existentes.

Em sistemas de partição fixa existem dois tipos de fragmentação:

interna: tamanho da partição maior que o tamanho do processo.

externa: partição não usada é menor que o tamanho do processo esperando para ser executado.

Em sistemas que utilizam partições fixas, quando os processos são submetidos para execução, alternativas para alocação de memória são:

a) Classificá-los de acordo com suas necessidades de memória (especificada pelo usuário ou pelo sistema) e colocá-los em uma única fila de processos aptos a rodar, de onde o escalonador os seleciona para execução. Como as necessidades de memória e os tamanhos das partições são diferentes, problemas que podem ocorrer são:

- uma partição se torna disponível e o primeiro processo da fila necessita de memória maior do que o tamanho da partição. Neste caso, o escalonador pode decidir selecionar um outro processo na fila (que deixa de ser uma fila). Esta solução pode levar a postergação indefinida.
- Um outro problema é que pode haver aumento de fragmentação interna a partição, na medida que processos pequenos podem ser carregados em partições grandes.

b) Cada partição tem sua própria fila de processos. Neste caso, pode-se ter uma fila com vários processos esperando ser carregados para execução em uma partição enquanto outra(s) partições de memória podem estar disponíveis (fila vazia).

Partições variáveis: O sistema operacional mantém uma tabela indicando que partes da memória estão disponíveis e quais estão ocupadas. Inicialmente toda a memória esta disponível, considerada como um bloco único. Quando um processo chega e necessita memória é pesquisado um bloco com tamanho suficiente para contê-lo. Quando um processo termina libera sua memória, que é reincorporada ao conjunto disponível. A figura a seguir ilustra uma memória organizada com o uso de partições variáveis, com um bloco único de 128 K disponível.

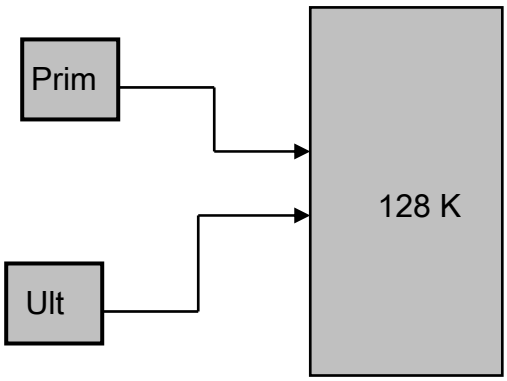

Fig. 8.8 - Partições variáveis com um único bloco disponível

Supondo que um processo P0 foi submetido para execução e que necessita de 12K, que um processo P1 necessita de 28K e que um processo P2 necessita de 8K, o mapa de alocação de memória é mostrado na figura abaixo.

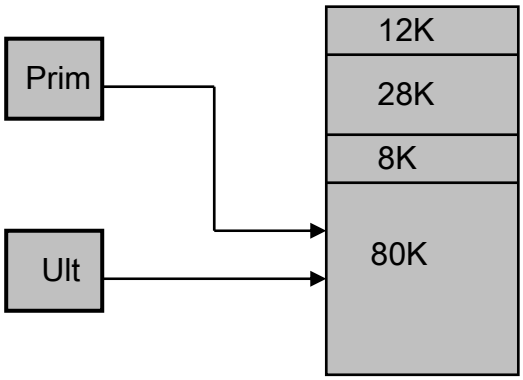

Fig. 8.9 - Partições variáveis com blocos de diferentes tamanhos ocupados

Supondo que os processos P0 e P2 terminem e liberem a memória que possuíam, a lista de blocos disponíveis fica composta por um bloco de 80K, que é o primeiro, que aponta para um bloco de 12K, que aponta para um bloco de 8K, que é o último. O mapa de memória resultante é mostrado na figura abaixo:

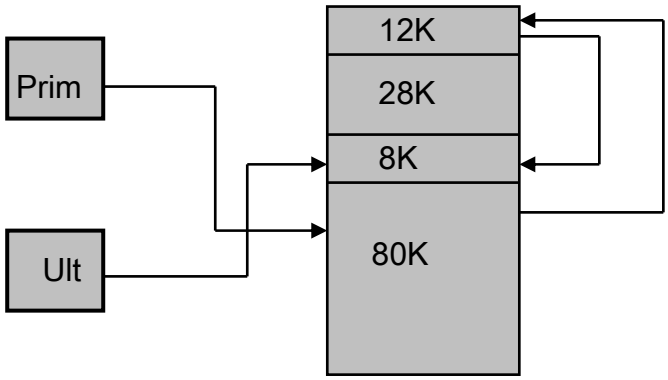

Fig. 8.10 - Partições variáveis com blocos de diferentes tamanhos livres e ocupados

Características da política de partições variáveis:

- existem blocos livres de diferentes tamanhos;
- se um bloco é muito grande para um processo, é dividido em dois, um alocado para o processo e o outro retorna para os blocos livres;
- quando um processo termina libera a memória, sendo esta colocada no conjunto de blocos livres;
- se o bloco liberado é adjacente à outro livre, podem ser agrupados formando um único bloco.

Em sistemas de partição variável, existem três alternativas para alocação de memória para os processos:

- first-fit: alocar o primeiro bloco da lista de blocos disponíveis com tamanho suficiente para atender a requisição. O bloco alocado é

dividido em dois: o primeiro cujo endereço é retornado ao requisitante e que contém o tamanho solicitado e um segundo bloco que deve ser acrescentado a lista de blocos disponíveis, cujo tamanho é a diferença do tamanho do bloco original e o tamanho de memória solicitada na requisição.

- best-fit: alocar o bloco de memória com o tamanho mais próximo do requisitado. O problema desta solução é que existe uma tendência a geração de fragmentos de memória pequenos.
- worst-fit: alocar o maior bloco de memória disponível existente. Com esta técnica, a idéia é gerar fragmentos grandes.

A solução mais utilizada para alocação de memória e que apresenta melhores resultados é a first-fit. A pior é a best-fit, pelo problema citado anteriormente.

Um problema importante nos sistemas de partições variáveis é a fragmentação. Uma solução utilizada é a compactação, que consiste em unir os blocos de memórias disponíveis e de endereços adjacentes em um único bloco, de tamanho maior.

Considerando-se as políticas de partições fixas e partições variáveis, apresentadas anteriormente, o hardware necessário (registradores limite inferior/superior, registradores base/limite) permite a implementação de qualquer uma das políticas. O software é que irá determinar o uso de uma ou outra. Para estas políticas, quando um processo é selecionado, o dispatcher carrega os endereços de memória nos registradores. Ainda, a utilização de memória é geralmente melhor nos sistemas de partições variáveis do que nos de partições fixas.

8.3 Swapping, Paginação e Segmentação

Nos políticas apresentadas anteriormente, partições fixas e partições variáveis, os processos são carregados para execução em uma área contígua de memória, e somente ao seu término é que esta memória é liberada para

ser alocada a um outro processo. Políticas que movimentam processos entre a memória e o disco, em tempo de execução, são Swapping, Paginação e segmentação.

Swapping: Com esta política, os processos em execução podem ser transferidos para disco para liberar memória, possibilitando desta forma a carga de outros processos para execução. Desta forma, o número de processos em execução simultânea é formado pelos processos que estão na memória principal e pelos que estão na área de Swap.

O funcionamento desta política é o seguinte: quando um processo é submetido para execução o sistema verifica a memória disponível. No caso de não existir, um processo é escolhido para ser levado para a área de Swap, é copiado da memória principal para os endereços do disco que correspondem a área de Swap (Swap out), é assinalado no seu registro descritor esta situação e o novo processo é carregado para a memória e introduzido na ready list. Quando um processo na área de Swap é selecionado para execução deve ser carregado na memória principal (Swap in). Se necessário, um outro deverá ser levado para a área de Swap para liberar a memória necessária. Desta forma, a ready list é formada por processos na memória principal e na área de swap.

A área de Swap ocupa uma localização definida, alocada durante a formatação do disco e de conhecimento do sistema operacional. Deve ser estabelecida de maneira a poder armazenar um grande número de processos, permitindo um alto grau de multiprogramação.

Quando o scheduling decide executar um processo chama o select que verifica se o processo selecionado está na memória. Se não estiver, verifica se existe memória disponível para acomodá-lo. Se houver, dispara swap in do processo selecionado. Caso contrário, dispara swap out de um processo escolhido para sair da memória e swap in do processo selecionado. A seguir, seleciona um outro processo que esteja na memória, restaura os registradores e transfere o controle para o processo selecionado.

Nesta política, um aspecto importante é o tempo de swap, que é proporcional ao tamanho da memória a ser transferida. A medida que aumenta a velocidade de transferência dos dispositivo, este tempo de transferência diminui.

Paginação

Com o uso desta política de gerência de memória, a memória física é dividida em frames e a memória lógica é divida em pages, de igual tamanho. Um programa para ser executado tem suas pages carregadas em frames disponíveis na memória principal. O tamanho da página é definido pelo hardware. Tamanhos característicos podem ser 1Kbytes, 2Kbytes, 4Kbytes.

O funcionamento desta política é o seguinte: quando um processo deve ser carregado para execução, o scheduling verifica o número de páginas que ele precisa, verifica na lista de frames disponíveis se existe um número suficiente, existindo, aloca para o processo. A seguir, o processo é carregado para as páginas de memória física alocadas, e a sua tabela de páginas é atualizada, de maneira a conter estes endereços. O endereço contido na instrução é lógico e a cada acesso à memória é feita uma transformação do endereço lógico no endereço físico correspondente.

Transformação do endereço lógico para o endereço físico

O endereço gerado pela CPU contém duas partes:

- No. da página: (p) - endereça uma tabela de páginas;
- deslocamento: (d) - combinado com o endereço base da página define o endereço físico de memória.

Para a transformação de endereço lógico em endereço físico (figura a seguir), a parte do endereço que contém a identificação da página é usada para indexar uma tabela de páginas, que contém o endereço base da página. O deslocamento, combinado com o endereço base da página, indica a posição de memória a ser acessada.

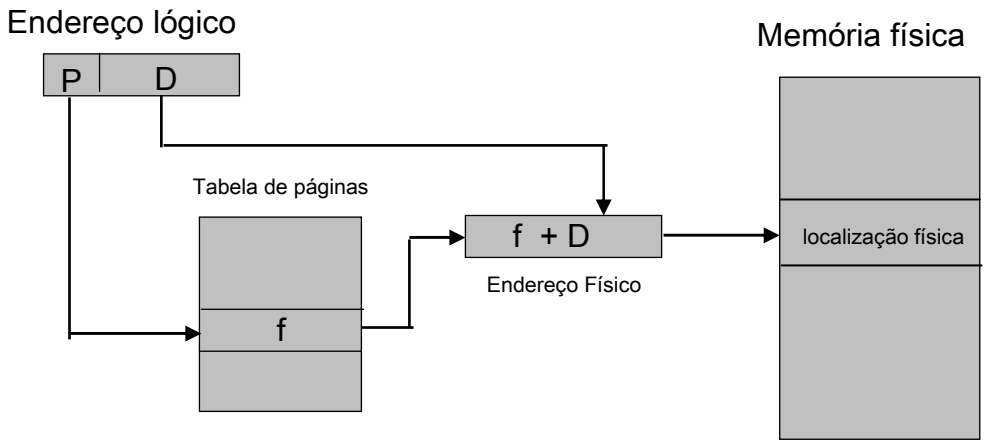

Fig. 8.11 - Tradução de endereço com paginação

Alternativas para a implementação da tabela de páginas são:

a) uso de um conjunto de registradores:

- recarregados como qualquer outro;
- carregados/modificados por instruções especiais.

b) na memória principal

- um registrador -page table base register-(ptbr) aponta para a tabela de páginas;

c) uso de memória associativa (memória endereçável pelo conteúdo).

A primeira solução somente é possível de ser adotada se a tabela de páginas é pequena. Para a solução b, o tempo de acesso a uma posição de memória é o problema. Para acessar endereço i os passos são os seguintes:

1. acessar tabela de páginas (uso do ptbr+ No.page desloc.) (1o. acesso à memória);
2. o passo 1 produz um No. de frame que deve ser combinado com o deslocamento e que produz o end i;

3. acessar a posição i na memória física (2o. acesso).

Portanto, são necessários dois acessos a memória para obter a palavra desejada.

A adoção de memória associativa otimiza os tempos de acesso a memória. Trata-se de uma memória mais rápida que a memória tradicional, porém, devido aos seus custos, normalmente são pequenas. Assim, contém poucas entradas da tabela de páginas. Um endereço lógico gerado pela CPU é comparado com todas as entradas da tabela de páginas simultaneamente, a qual contém número de página/número de frame (na memória associativa). Se o número da página é encontrado, o número da frame é usado imediatamente para acessar a memória. Se o número da página não é encontrado, é feito um acesso a tabela de páginas, na memória principal, para obter o número da frame que é usado para acessar a memória. Neste caso, o número da página e o número da frame são colocados na memória associativa, de modo a ser encontrado rapidamente na próxima referência

Compartilhamento de Páginas

É uma técnica particularmente importante pois permite compartilhamento de memória e conseqüentemente otimiza espaço de armazenamento. O código pode ser compartilhado entre diversos processos, sendo que os dados são privativos. Com esta técnica, as tabelas de páginas dos processos que compartilham o código possuem os mesmos endereços (apontam para as mesmas páginas) de memória principal. Exemplos de programas que compartilham código são compiladores e editores de texto.

Segmentação

Um programa é um conjunto de sub-rotinas, funções, estruturas de dados (tabelas, etc.) que são referidos pelo nome. Cada um é um segmento de tamanho variável. Segmentação é um esquema de gerência de memória que

suporta esta visão, sendo que cada segmento tem um nome e um tamanho. O endereço é especificado pelo nome do segmento e pelo deslocamento, em relação ao início do segmento.

Com o uso de segmentação, um endereço lógico é formado por duas partes: uma é usada para indexar uma tabela de segmentos, que contém o endereço base do segmento e a outra o deslocamento, que combinado com o endereço base indica a posição física de memória.

Implementação da tabela de Segmentos

A implementação da tabela de segmentos pode ser feita por:

a) conjunto de registradores;

b) tabela de segmentos na memória;

c) uso de memória associativa para manter as entradas da tabela de segmentos mais recentemente usadas.

A primeira alternativa, a exemplo de paginação, somente é viável se a tabela de segmentos é pequena. Para a segunda possibilidade, utiliza-se dois registradores:

- STBR (segment table base register): aponta para o endereço base da tabela de segmentos do processo;
- STLR (segment table length register): contém o tamanho da tabela de segmentos.

A transformação de um endereço lógico (s,d) em um endereço físico é feita da seguinte maneira:

Se (s < STLR) então

S = s + STBR;

endereço = S + d;

senão

erro.

A tabela de segmentos na memória requer dois acessos, como na paginação, para a transformação de um endereço lógico em um endereço físico.

8.4 Memória Virtual

A idéia com memória virtual é dar ao programador a ilusão de que ele possui uma memória principal muito grande, maior do que a existente. O aspecto principal nesta idéia é de que o endereço existente no programa é distinto de localização física. O programa em execução possui uma memória virtual, na qual os endereços são lógicos. A tradução do endereço (lógico) para a localização na memória física (endereço físico) é feita pelo sistema operacional, com o auxílio do hardware. Como a memória virtual do processo pode ser maior que a memória real, partes do programa estão na memória e partes estão em disco. Isto para o usuário é transparente, pois é o SO que trata da transferência entre disco e memória. A implementação de memória virtual pode ser feita usando-se os mecanismos de paginação e segmentação. Os conceitos e algoritmos apresentados a seguir se referem a paginação. Para os algoritmos serem utilizados com segmentação, necessitariam tratar do tamanho do segmento, que é variável, sendo portanto mais complexos.

Com paginação, o programa para ser executado necessita que somente uma página, a que contém o código inicial do programa, esteja na memória. Somente quando necessária à página será transferida do disco para a memória. A cada página do processo são associados cinco bits: um que indica se a página está presente na memória cache, um de página referenciada, outro que indica página modificada, o bit de presença na

memória principal e um bit de proteção, que indica se a página pode ser escrita ou é somente de leitura. (figura abaixo).

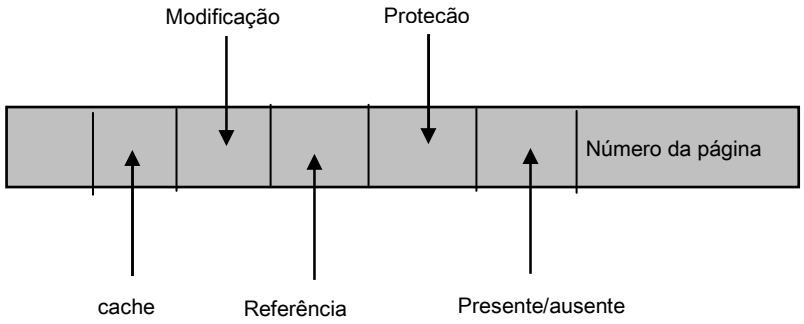

Fig. 8.12 - Estrutura de uma entrada na tabela de páginas

Os bits de página referenciada e modificada são atualizados a cada acesso a memória.

Quando a tradução do endereço lógico para o endereço físico é feita, a página referenciada pode se encontrar na memória física ou não. Se a página está na memória, o tradução se completa. Se a página está no disco, é gerada uma interrupção, o processo que gerou a falta de página fica bloqueado a espera que a página seja transferida para a memória. O sistema operacional deverá selecionar uma página, que deverá ser removida da memória, para dar lugar à página necessária. Se a página que será removida tiver sofrido alteração, a mesma deverá ser gravada em disco. Caso contrário, a nova página poderá ser gravada sobre a página escolhida para ceder lugar na memória principal. Após esta transferência, o bit de presença da página na memória é alterado para indicar esta situação e o processo é inserido na lista de processos aptos a rodar. Por exemplo considerando dois processos , P0 e P1, cada um com cinco páginas lógicas e uma memória principal formada por quatro páginas, poderemos ter a seguinte situação:

P0

0	A
1	B
2	C
3	D
4	E

Tabela de páginas de P0

0	1	V
1	3	V
2		I
3		I
4		i

P1

0	F
1	G
2	H
3	I
4	J

Tabela de página de P1

0		i
1		i
2	0	v
3		i
4	2	v

Memória principal

0	h
1	a
2	j
3	b

Fig. 8.13 - Exemplos de Tabelas de páginas

A figura acima mostra que o processo P0 possui as páginas 0 e 1 presentes na memória principal (v na tabela de páginas) e as páginas 2, 3 e 4 não presentes (bit de presença i – inválido). O processo P1 possui na memória principal as páginas 2 e 4.

Se o endereço referenciado por P0 se refere a página 2, esta página não está presente na memória. Neste caso, o sistema operacional deverá selecionar uma página da memória principal para ser substituída.

Algoritmos de substituição de páginas

O problema para o sistema operacional é a escolha da página a ser substituída, que deverá minimizar as interrupções por falta de páginas. A política de escolha da página a ser substituída (página vítima), pode ser local

ou global. É local quando a escolha é entre as páginas que pertencem ao processo que gerou a falta de páginas. É global quando a escolha é feita considerando o conjunto completo de páginas existentes na memória.

Algoritmo Ótimo

Considerando como critério de algoritmo ótimo a minimização do número de falta de páginas, o algoritmo Ótimo necessita que se conheça o string de referência (seqüência de referência) futuro, para selecionar para substituição a página que será necessária no tempo mais longínquo. Com isso, estaria se aumentando o tempo para a ocorrência de uma falta de páginas. Para se obter esta informação, pode-se executar o programa e obter o string de referência, mas a seqüência poderá não se repetir, dependendo dos dados e das decisões existentes no código do programa. O algoritmo Ótimo é não implementável, pois não tem como o sistema operacional saber quando uma página será novamente necessária.

Algoritmo FIFO (primeira página a entrar na memória principal é a primeira a sair)

Com este algoritmo, a página a ser substituída é a que está na memória há mais tempo. Para implementar este algoritmo é necessária uma fila contendo as páginas de acordo com a ordem de chegada na memória principal. Quando ocorrer uma falta de páginas, a primeira página da fila será substituída, independentemente se está sendo utilizada ou não. A nova página será colocada no final da fila. Esta escolha poderá interferir negativamente na performance do sistema, bastando para tal que a página escolhida esteja sendo referenciada. Este algoritmo é raramente utilizado.

Algoritmo da Segunda Chance

Este algoritmo é uma combinação do FIFO com o bit R. A página mais velha na memória, a primeira da fila, é candidata a ser substituída. Se o bit R possui o valor 1 indicando que a página foi acessada no último intervalo de tempo considerado, é zerado, e a página vai para o final da fila, recebendo

uma segunda chance. O algoritmo recomeça a examinar novamente o bit R da página que agora é a primeira da fila. No pior caso, todas as páginas da fila têm o bit R com o valor 1 e o algoritmo se transforma em FIFO.

Algoritmo de substituição da página Não Usada Recentemente (NUR)

Este algoritmo considera os bits R e M para a escolha da página que deverá ser substituída. Considerando as classes mostradas na figura 8.14 abaixo:

Classe	Bit R	Bit M	Significado
0	0	0	Não referenciada, não modificada
1	0	1	Não referenciada, modificada
2	1	0	Referenciada, não modificada
3	1	1	Referenciada, modificada

Fig. 8.14 – Classes de páginas

Será escolhida aleatoriamente uma página de menor classe existente e que não tenha sido referenciada no intervalo de tempo considerado.

Algoritmo do Relógio

As páginas na memória fazem parte de uma fila circular, que pode ser vista como um relógio. Quando ocorre uma falta de página, a página apontada (cabeça da fila) é avaliada. Se o bit R é zero, a página sai da memória, cedendo lugar à página que necessita ser carregada e o apontador passa a apontar para a página seguinte. Se o bit R possui o valor 1, recebe o valor 0 e o apontador de páginas avança para a próxima página que é então avaliada. Portanto, o algoritmo funciona da seguinte maneira:

Se R = 0 então retira a página e avança o ponteiro

Se R = 1 faz R = 0 e avança o ponteiro.

Algoritmo de substituição da página Menos recentemente usada (LRU)

Este algoritmo parte do princípio de que as páginas usadas recentemente voltarão novamente a ser usadas. Mantém uma lista contendo as páginas mais usadas na frente, e as menos utilizadas no final da lista. Em todo o acesso a memória esta lista é atualizada. Será substituída a página no final da lista, isto é, a página que está a mais tempo sem ser acessada. Para implementar este algoritmo, pode ser considerado o bit R. A intervalos regulares de tempo (por exemplo a cada interrupção), os bits R são zerados. Um contador do número de vezes que o bit foi resetado indica a utilização da página.

O conceito de conjunto de trabalho (Working Set)

Com o uso de memória virtual, para iniciar a execução de um novo programa somente a página que contém o ponto de início de execução é necessária. A medida que a execução avança, páginas não presentes na memória são necessárias, originando a execução do algoritmo que trata essas falta de páginas. Localidade de referência é um conceito que considera que um processo referencia somente um conjunto de páginas em uma determinada fase de sua execução. O conjunto de trabalho (working set) de um processo é o conjunto das páginas mais recentemente referenciadas no último intervalo de tempo *t.*

Por exemplo, considerando a figura 8.15 abaixo,

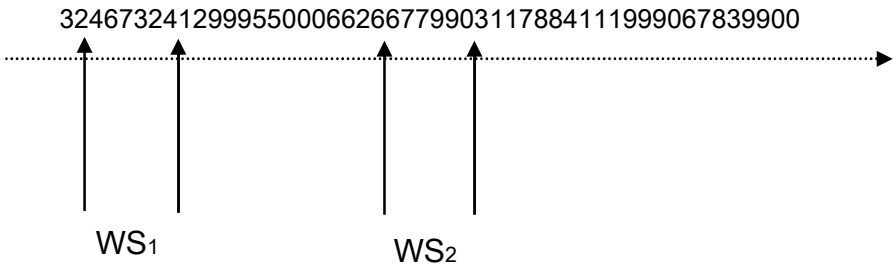

Fig. 8.15 – Exemplos de Working Set

O primeiro working set ($WS_{1)}$ é formado pelas páginas {23467} e o WS_2 pelas páginas {6790}. Observe que as páginas 2,3 e 4, do WS_1 não estão mais presentes no $WS_{2,}$ pois deixaram de ser acessadas.

Em outras palavras, o working set de um programa *W(k, h),* é o conteúdo de uma janela de tamanho *h* no string de referência *rK.* O working set no tempo *t* é *W(t, r),* é o conjunto de páginas referenciadas pelo processo no tempo *t.*

O problema da utilização do working set para minimizar as transferências de páginas é determinar o tamanho da janela. Deverão permanecer na memória as páginas pertencentes ao working set, isto é, pertencentes a janela. Quando uma falta de página ocorre deve ser encontrada uma página que não pertence ao working set para ser substituída. Uma forma que pode ser empregada é com a utilização do bit de referência da página. A janela, isto é, o tamanho do working set pode ser o número de referências a memória. O bit é colocado em 1 quando a página é referenciada e zerado a cada intervalo de tempo (interrupção). Quando ocorre uma falta de páginas, se o bit de referência da página possui o valor 1, esta pertence ao working set e não deve ser removida. Se o bit de referência é 0, a página não foi referenciada durante o último intervalo de tempo e é candidata a remoção.

8.5 Gerência de Memória no Linux e no Windows NT

O sistema operacional Linux gerencia memória utilizando paginação por demanda. Quando o usuário submete um programa para execução, o arquivo contendo o código executável é aberto e uma parte vai para a memória. As estruturas de dados são ajustadas para indicar que os endereços da parte do programa executável que foram copiados para a memória se referem a memória física. O restante do programa fica em disco. Quando o programa em execução necessita acessar um endereço não mapeado na memória física, o Linux transfere a página do disco para memória e o acesso pode então ser realizado.

Quando o processo necessita de uma página virtual (que está em disco) e não existe memória disponível, o Linux necessita criar um espaço para acomodar a página necessária. Se a página escolhida como vítima não foi alterada, pode simplesmente ser utilizada para armazenar a página virtual. Quando for necessária, poderá ser carregada da imagem do processo em disco. Se a página foi alterada, o Linux a escreve no arquivo de swap. O Linux usa o algoritmo LRU (Menos Recentemente Usada) para escolha da página a ser substituída. Toda a página no sistema possui uma idade associada. Quanto mais a página é acessada, mais jovem a página se torna. As páginas mais velhas (acessadas menos recentemente) são boas candidatas para swapping.

Para o Linux, a tabela de páginas possui três níveis. Uma entrada na tabela de páginas de nível 1contém o deslocamento da tabela de páginas de nível 2, que contém o deslocamento da tabela de páginas de nível três, que contém o endereço base da página de memória que, combinado com o deslocamento que faz parte da instrução, indica o endereço físico. A implementação do Linux nas diferentes plataformas deve prover mecanismos de tradução entre os três níveis de tabela de páginas. Para o kernel esta tradução deverá ser transparente.

Para alocação de páginas, o Linux usa o algoritmo Buddy. Este algoritmo tenta alocar um bloco com uma ou mais páginas. Os blocos de páginas possuem um tamanho que é uma potência de 2 (os blocos possuem tamanho de 1 página, 2 páginas, 4 páginas, ...) e existe uma lista de blocos disponíveis para cada tamanho de bloco. Quando uma requisição é feita, o algoritmo procura por um bloco de páginas com o tamanho pedido. Se não encontra, procura alocar então um bloco que possui o dobro do tamanho solicitado, e assim por diante, até alocar um bloco. Se o bloco alocado é maior do que o solicitado, é dividido de maneira a atender o pedido e o restante é mantido como disponível e inserido na lista correspondente ao seu tamanho. O bloco alocado é então retornado ao processo solicitante.

A alocação de blocos de páginas pode levar a fragmentação de memória. O algoritmo de liberação de páginas tenta combinar os blocos liberados com blocos disponíveis, de maneira a formar blocos maiores. Quando um bloco é liberado, o algoritmo verifica se existe um bloco de mesmo tamanho com endereços adjacentes. Existindo são combinados de maneira a formar um único bloco de tamanho maior. Este processo se repete agora com o bloco de tamanho maior, na lista correspondente.

No Windows NT o mecanismo de gerência de memória considera um espaço linear de 32 bits, o que permite endereçar 4 GB de memória virtual. Ao processo do usuário normalmente é atribuído 2 GB, ficando os 2GB restantes para o sistema operacional.

Quando um processo é carregado, um certo número de páginas de memória virtual é reservado. Quando o processo necessita de páginas, estas páginas que já estavam reservadas são utilizadas. As páginas possuem três estados:
livre: não utilizadas pelo processo;
reservadas: alocadas a algum processo, porém não mapeadas em disco físico; dedicadas: já mapeadas para o processo.

Para a tradução de um endereço virtual em um endereço físico é usada uma tabela de paginação em dois níveis. Um endereço virtual de 32 bits é formado por três componentes: índice de diretório de páginas, índice de tabelas de páginas e deslocamento dentro da página. O endereço de páginas é usado para determinar a entrada da tabela de páginas correspondente ao endereço virtual. Cada entrada da tabela de páginas possui o endereço base da página a ser acessada que, somado ao deslocamento indica o endereço físico a ser acessado.

O Windows NT utiliza paginação por demanda com clustering. Quando ocorre uma falta de página, o gerenciador de memória carrega a página que faltava e algumas ao redor. Com isso, tenta minimizar o número de falta de páginas, objetivando diminuir o tempo de execução dos processos.

O algoritmo de substituição de páginas utilizada no Windows NT é o LRU, implementado através do algoritmo do relógio (clock). Para máquinas multiprocessadoras, o algoritmo é o FIFO. Outro aspecto considerado ainda é o working set do processo. O sistema defini um número mínimo e máximo de páginas presentes na memória, valores estes que são definidos em função do tamanho do processo.

Exercícios

1. Considere um sistema em que a memória e gerenciada por uma lista encadeada de blocos disponíveis, de diferentes tamanhos. Essa lista é definida por uma estrutura de dados contendo o endereço base do bloco, o tamanho de cada bloco e um apontador para o próximo elemento da lista. Existem dois apontadores, um para o primeiro elemento da lista, e outro para o último. Escreva o procedimento addr=allocmem (n), onde n é o tamanho do bloco que deve ser alocado e addr contém, no retorno da chamada, o endereço base do bloco alocado. A técnica de alocação a ser utilizada é First-Fit (o primeiro bloco com tamanho suficiente). Escreva também o procedimento free(addr), sabendo que se o bloco a ser liberado for adjacente a um outro, os mesmos devem ser agrupados.
2. Compare Partições Fixas, Partições variáveis e Swapping.
3. Discuta o escalonamento de processos em sistemas de gerência de memória com partições fixas.
4. Justifique o uso de compartilhamento de páginas, e apresente um exemplo de sua utilização.
5. Considere um sistema em que uma memória de 2MB é gerenciada em dez partições de tamanho fixo cada uma. Sabendo que não existem prioridades entre as partições, e que existem registradores base e deslocamento, defina a(s) estrutura(s) de dados necessária e descreva a carga um programa em uma partição livre.
6. A afirmação " *Partições fixas são mais eficientes que partições variáveis, pois aproveitam melhor o espaço de memória existente e permitem que, definindo-se um número elevado de partições, se tenha mais processos na memória*" está correta? Justifique sua resposta.

7. Considere um sistema em que uma memória de 64MB é gerenciada com o uso de partições variáveis. Faça uma figura representando três processos na memória, com, respectivamente, 12MB, 8MB e 6MB e a lista de disponíveis contendo dois blocos (defina os tamanhos). A seguir, faça uma nova figura considerando o término de um dos processos em execução.
8. Compare partições fixas e paginação.

9 Gerência de Arquivos

Conceitos Básicos. Armazenamento e recuperação de arquivos. Compartilhamento de arquivos e sinônimos. Métodos de acesso. Estudo de casos: Gerência de arquivos no Linux e no Windows NT .

9.1 Conceitos Básicos

Arquivos são entidades lógicas mapeadas pelo sistema operacional em dispositivos físicos, que possuem informações definidas pelo proprietário (dados e programas). Cada arquivo possui um nome e é referenciado pelo mesmo. O sistema operacional possui suporte (chamadas de sistema) para manipulação de arquivos.

Tipos de Arquivos

Os arquivos podem ser:

- Regulares: contém informações dos usuários. Pode ser um arquivo texto, o código de um programa, etc., bem como pode conter também um código binário.
- Diretórios: são arquivos mantidos pelo sistema e que implementam a estrutura do sistema de arquivos. Contém nomes de arquivos e os endereços nos quais cada arquivo está armazenado nos periféricos.

Acesso aos arquivos regulares

O sistema operacional oferece ao usuário uma interface, sob a forma de procedimentos de biblioteca, que permite a manipulação de arquivos. Os principais procedimentos são:

- Create: permite a criação de um novo arquivo.
- Delete: elimina um arquivo existente.
- Open: abre um arquivo para posterior utilização.
- Close: fecha um arquivo que estava em uso.
- Read: acessa um arquivo para leitura de dados.
- Write: escreve dados em um arquivo.
- Append: acrescenta, no final de um arquivo A, dados armazenados em um arquivo B.
- Rename: troca o nome de um arquivo.
- Link: cria um nome alternativo para um arquivo.

Diretórios

Diretórios são arquivos especiais, mantidos pelo sistema operacional, que contém informações que permitem acessar os arquivos regulares. Informações típicas mantidas em um diretório são:

- Proprietário do arquivo
- Data de criação;
- Data de modificação;
- Data de acesso;
- Direitos de acesso;
- Existência de sinônimos;
- Contador de uso;
- Endereço dos dados no disco;
- Etc.

Um diretório possui várias entradas, uma por arquivo que pertence aquele diretório. Cada entrada do diretório contém:

- O nome e os atributos do arquivo; ou
- O nome do arquivo e um ponteiro para uma estrutura de dados com os atributos do arquivo.

Estas alternativas são esquematizadas nas figuras a seguir.

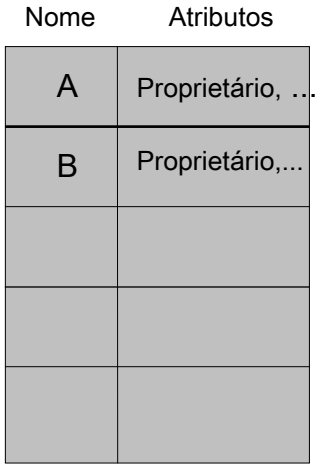

Fig. 9.1 Diretório contendo o Nome do arquivo e seus atributos

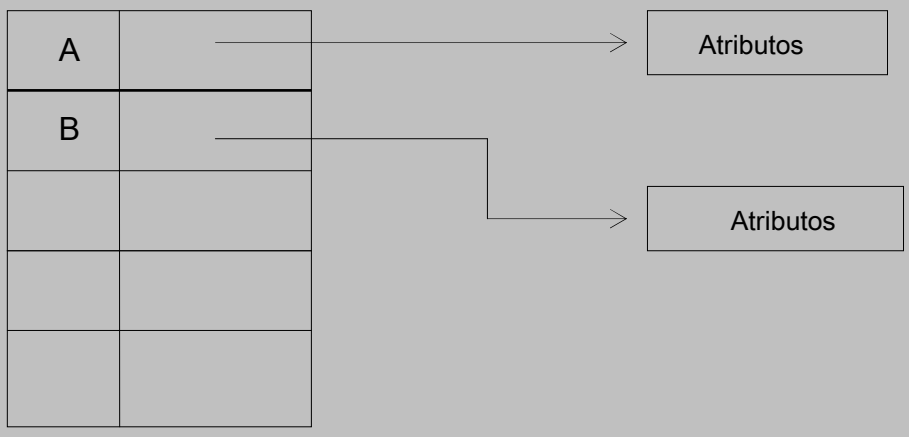

Fig. 9.2 - Diretório contendo nomes de arquivos e um apontador para os atributos

Um sistema de arquivos pode ser organizado como possuindo um único diretório contendo os arquivos de todos os usuário ou com um diretório por usuário. A organização mais natural é haver um diretório por usuário, sendo que somente o proprietário do diretório deve ter direitos de acesso, e que pode estender estes direitos a outros usuários Com isso, os usuários têm seus arquivos protegidos contra acessos não autorizados. As figuras a seguir exemplificam estas alternativas.

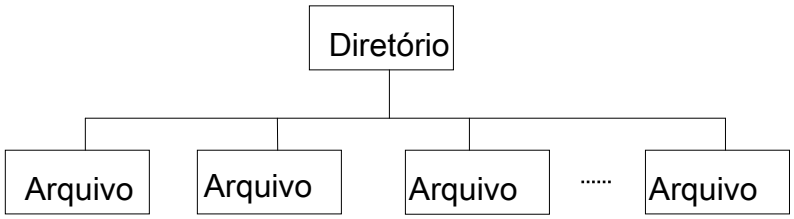

Fig 9.3 - Sistema de arquivos com um único diretório

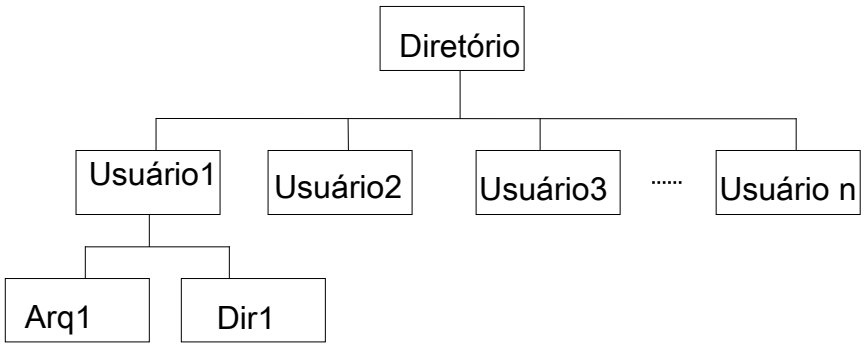

Fig. 9.4 - Sistema de arquivos com um diretório por usuário

Na organização com um diretório por usuário, cada usuário pode criar arquivos regulares ou diretórios, formando uma estrutura hierárquica de arquivos (uma árvore de arquivos).

Numa estrutura hierárquica, os nomes dos arquivos podem ser absolutos ou relativos.

- Absolutos: consiste do caminho desde a raiz até o arquivo. Por exemplo, /usr/home/cmc/livro/cap5 é o nome completo do arquivo cap5.
- Relativos: utilizados juntamente com o conceito de diretório corrente. Todos os nomes de arquivo que não começam com o separador (/,\), são considerados relativos ao diretório corrente. Ex. Se o diretório corrente é mail, o comando Linux cp usr1 usr1_message copia o arquivo /usr/spool/mail/usu1 para o arquivo /usr/spool/mail/usu1_message.

Um conjunto de primitivas do sistema operacional permite aos usuários a manipulação de diretórios. As principais são:

- Create: cria um novo diretório;
- Delete: elimina um diretório que não contém arquivos;
- List: lista o conteúdo de um diretório (nomes de arquivos que pertencem ao diretório);
- Rename: troca o nome de um diretório.

9.2 Armazenamento e Recuperação de Arquivos

Os dados pertencentes aos arquivos são armazenados em dispositivos físicos, não voláteis. A alocação de espaço pode ser:

Alocação Contígua: O sistema operacional aloca uma área contígua no dispositivo para conter os dados do arquivo. Assim, para um arquivo de 230Kbytes seriam alocados 230 blocos consecutivos de um Kbyte. Esta alternativa é simples de implementar e o arquivo pode ser lido de maneira eficiente. Os principais problemas desta solução são a determinação do

tamanho do arquivo a priori e a fragmentação, isto é, a área alocada não utilizada.

Lista encadeada: Os dados pertencentes a um arquivo são armazenados em uma lista encadeada de blocos do dispositivo. Por exemplo, com blocos de 1k, 1022 bytes armazenam informações, dois bytes endereçam o próximo bloco. Nesta solução, para acessar uma determinada informação no bloco i (acesso randômico) é necessário percorrer a lista encadeada até o bloco desejado, isto é, percorrer os i - n blocos de dados do arquivo.

Lista de blocos com tabela na memória: Cada apontador de bloco de disco é armazenado em uma tabela, no descritor do arquivo. Quando o arquivo é aberto, esta tabela vai para a memória (juntamente com o descritor). Para identificar um bloco não é necessário nenhum acesso a disco (acesso randômico). O acesso é feito à tabela na memória, o endereço do bloco é recuperado e então pode ser realizado o acesso ao bloco de dados, no periférico.

Gerência de blocos livres

O sistema operacional possui um conjunto de procedimentos utilizados para a gerência do espaço disponível nos periféricos, utilizados para o armazenamento de arquivos. Duas soluções utilizadas são:

Lista encadeada de blocos livres: O sistema operacional mantém uma lista encadeada de blocos livres. A alocação de um novo bloco para um arquivo é feita retirando um bloco da lista de disponíveis, e a liberação de um bloco o reincorpora a lista de disponíveis. O inconveniente desta solução é a quantidade de blocos necessária para conter a lista. Por exemplo, com blocos de 1Kbyte e dois bytes para identificar um bloco livre, pode ser armazenado em um bloco 512 endereços de blocos livres. Logo, são necessários dois blocos para endereçar um Mbyte.

Mapa de bits: Uma tabela (mapa de bits) contém o mapa de ocupação do disco, na qual os blocos livres são representados por um e os ocupados por zero. Assim, para um disco com n blocos será necessário um mapa de n bits.

9.3 Métodos de Acesso

Os dados dos arquivos são armazenados em blocos de disco. O acesso aos dado, nas operações de leitura e gravação, depende da forma de organização utilizada.

Acesso seqüencial: os dados armazenados somente podem ser acessados seqüencialmente. Para acessar o registro i é necessário ler os (i – 1) registros anteriores. Por exemplo, se a posição corrente do arquivo é n, a leitura do registro (n + 8) é feita lendo todos os registros anteriores. Neste caso, a posição corrente passa a ser (n + nove), o que indica o próximo registro a ser acessado é o número 9. Se for necessário ler um registro anterior, o arquivo necessitará ser lido a partir no primeiro registro do arquivo.

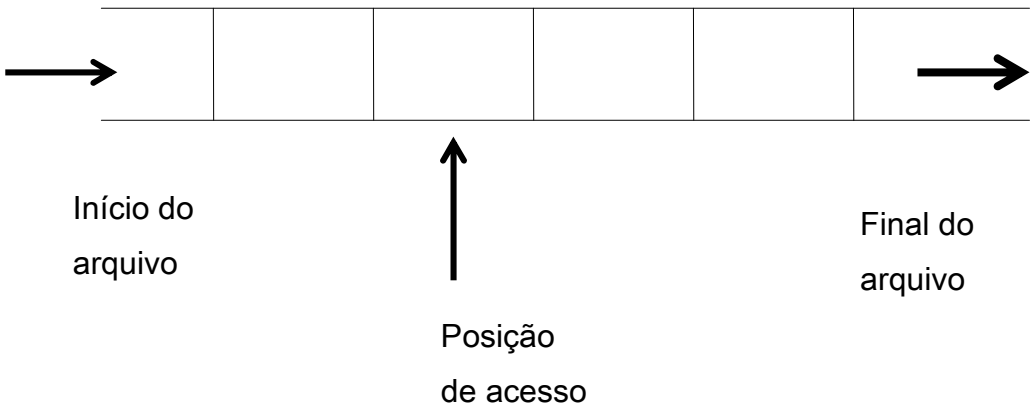

Fig. 9.5 - Arquivo de Acesso Seqüencial

Acesso direto: Os dados pertencentes ao arquivo são organizados logicamente em registros que podem ser acessados diretamente, independentemente da sua localização física no disco. Uma função de transformação, executada pelo sistema operacional, associa um endereço físico a um número de registro lógico. Considerando um disco com blocos de um Kbyte e um arquivo com registros lógicos de 100 Bytes, no primeiro bloco de dados do arquivo podem ser armazenados os 10 primeiros registros lógicos. Assim, a leitura do registro lógico três é transformado em uma leitura, a partir do byte 200 e até o byte 299, no primeiro bloco de dados do arquivo.

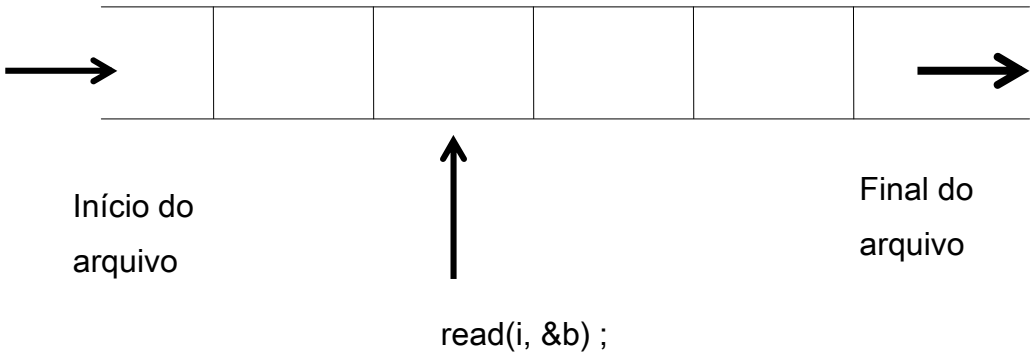

Fig. 9.6 - Método de Acesso Direto

Na figura acima, na operação read, a posição de acesso i é o resultado da transformação do número lógico do registro em número de bloco e deslocamento dentro do bloco.

9.4 Gerência de Arquivos no Linux

Em sua origem, o Linux foi implementado baseado no Minix, sistema operacional desenvolvido por Tanembaun em 1987. Em sua primeira versão, suportava somente o sistema de arquivos do Minix, que possuía sérias restrições, referentes ao tamanho dos nomes dos arquivos (máximo de 14 caracteres) e ao tamanho dos arquivos (64MBytes). Devido a estas

limitações, outros sistemas de arquivos foram desenvolvidos e incorporados ao Linux. Para que isso fosse possível, foi criado o Virtual File System (VFS).

O VFS é um nível do software que foi incorporado ao código do Linux e permite a existência de diferentes sistemas de arquivos no kernel do Linux. O VFS implementa a interface do sistema de arquivos com os programas de usuários. Trata todas as chamadas do sistema relacionadas ao sistema de arquivos, oferecendo uma interface uniforme, independentemente do tipo de sistema de arquivos. Uma chamada de sistema, por exemplo open, será endereçada pelo VFS ao sistema de arquivos correspondente. Tratando-se da abertura de um arquivo gerado no MS-DOS, será executada a função correspondente ao MS-DOS. Se for um arquivo gerado no Minix, será executada a função implementada no sistema de arquivos Minix. Alguns sistemas de arquivos suportados pelo Linux são:

- Sistema de arquivos do Minix.
- MS-DOS, VFAT, NTFS.
- Ext2, Ext3, ReiserFS;
- Etc.

A figura a seguir ilustra o funcionamento do VFS.

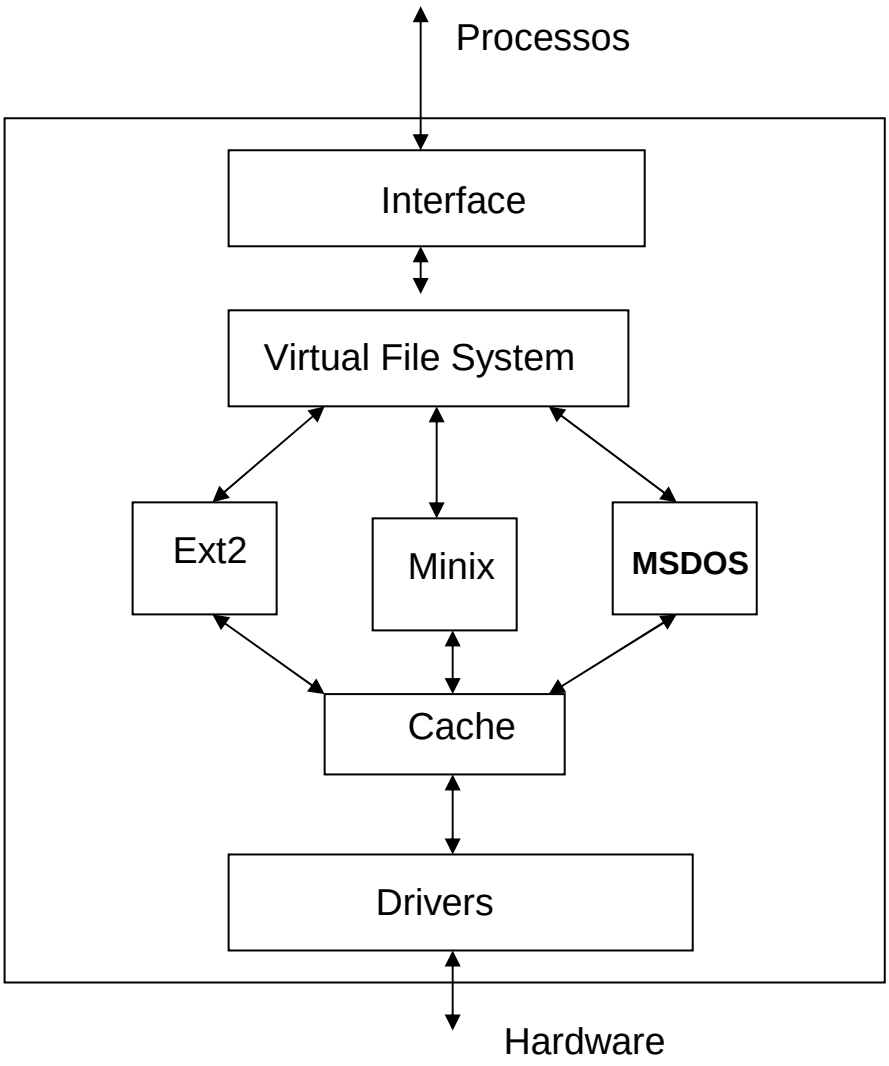

Fig. 9.7 - Estrututra do VFS

O VFS implementa um modelo comum de sistema de arquivos. Com este modelo, um arquivo é representado por uma estrutura de dados chamada *file,* que contém um campo chamado *f_op* que possui apontadores para as funções do sistema de arquivo específico. Assim, uma chamada de sistema read (*sys_read*) localiza o apontador para a função read do sistema específico, e se transforma numa chamada indireta *file_op_read (parâmetros).*

A implementação do VFS foi pensada com o modelo de orientação a objetos, com as estruturas de dados e os métodos que manipulam as estruturas de dados. Como não foi implementado em uma linguagem orientada a objetos, as estruturas de dados possuem alguns campos que contém os endereços das funções que correspondem aos métodos do objeto.

Os objetos que são utilizados pelo VFS são:

- File systems
- File
- Inode
- Dentry

File systems

Todo o sistema de arquivos deve ser montado, antes que possa ser utilizado. O Linux mantém uma lista encadeada de sistema de arquivos (figura abaixo), onde cada nodo da lista contém o nome do sistema de arquivos (ex. ext2) e a função read_super(), que é chamada quando uma nova instância do sistema de arquivos deve ser montada. O superbloco (estrutura superblock) é inicializado pela função read_super(). A estrutura superblock contém informações sobre o sistema de arquivos, tais como tamanho do bloco em bytes, tamanho do bloco em bits, identificador do dispositivo, direitos de acesso, data da última modificação, diretório no qual o sistema de arquivos foi montado e o i-node raiz do sistema de arquivos.

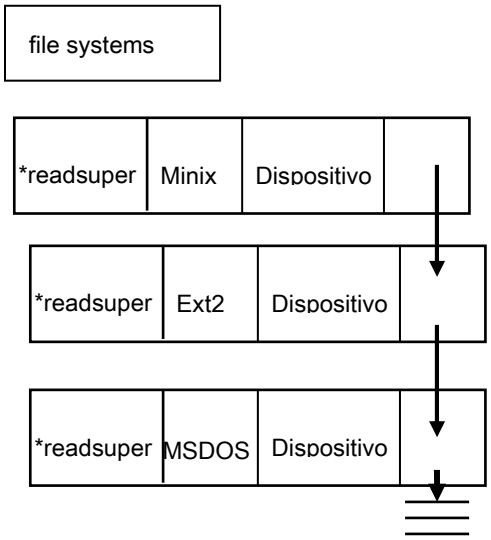

Fig. 9.8 - Lista encadeada de file systems

File

Esta estrutura existe para permitir o acesso compartilhado a um arquivo por diferentes processos. Quando um certo número de processos deseja acessar um mesmo arquivo para executar operações de leitura/escrita em diferentes pontos do arquivo, a estrutura file mantém, para cada processo, a posição corrente no arquivo a partir da qual a operação deverá ser realizada. Fazem parte desta estrutura as seguintes informações: posição corrente do arquivo (último byte acessado), direitos de acesso, apontador para o i-node que representa o arquivo, o tipo de acesso, o número de acessos ao arquivo, etc.

Inode

O Inode é o descritor do arquivo. Contém informações sobre o arquivo e sobre o sistema de arquivos. As informações são

- Proprietário;
- Tipo de arquivo (ordinário, diretório, bloco especial, caractere especial, fifo(pipe));
- Direitos de acesso;
- Datas de acesso, modificação;
- Número de links;
- Lista de blocos (direto, indireção, dupla indireção);

Dentry

O VFS mantém uma cache de diretórios, que mantém os diretórios mais comumente acessados e que existe somente por questões de performance. Somente nomes pequenos de diretórios, até 15 caracteres, são mantidos nesta cache. Quando uma entrada de um diretório é lida para a memória, é transformado em uma estrutura dentry. O sistema cria uma estrutura dentry para todo o nome de arquivo ou diretório que um processo acessa. Uma estrutura dentry está associada a um inode. Supondo que um processo deseje acessar o arquivo /user/home/aluno/aluno1, o sistema irá criar uma estrutura dentry para cada componente do pathname, ou seja, uma para o diretório /, uma para

o diretório user, uma para o diretório home, uma para o diretório aluno e uma para o diretório aluno1. A próxima vez que este arquivo for acessado, irá ser encontrado na cache.

A cache de diretórios consiste de uma tabela hash, onde cada entrada aponta para uma lista de entradas que possuem o mesmo valor de chave hash. Para o cálculo da chave hash o VFS utiliza o número do dispositivo e o nome do diretório.

Além de manter na memória uma cache dos diretórios mais utilizados, o VFS possui também uma cache de inodes. Cada vez que um inode necessita ser acessado e é encontrado na cache, o sistema ganha em performance. A cache de inodes é implementada como uma tabela hash, com cada entrada tendo um apontador para uma lista de inodes de mesmo valor de chave hash. O cálculo desta chave hash é feito com o número do inode e com o identificador do dispositivo.

Para localizar um inode de um arquivo, o VFS necessita percorrer o caminho até o arquivo, acessando todos os diretórios intermediários. Para isso, o procedimento de busca de um inode, que é específico para cada tipo de sistema de arquivos, deve ser executado. A estrutura superbloco de cada sistema de arquivos possui um apontador para o diretório raiz e para o procedimento de busca de um inode. Cada vez que um inode deve ser acessado, este procedimento verifica na cache de diretórios. Não encontrando, a cache de inodes é pesquisada. Se o inode é encontrado, o contador de uso é incrementado, indicando que um outro usuário o está utilizando. Para acessar um inode os inodes que representam os diretórios intermediários também necessitam ser acessados e são colocados na cache de inodes. O VFS utiliza u algoritmo LRU, para manter na cache somente os inodes mais utilizados.

O sistema de arquivos ext2

O primeiro sistema de arquivos desenvolvido para o Linux foi o ext, porém apresentava problemas de performance. Em 1993, o ext2 foi incorporado ao Linux. Para tornar possível a incorporação do ext2 ao Linux, foi também

incorporado o VFS, que separa os sistemas de arquivos do sistema operacional, que permite ao Linux suportar muitos diferentes sistemas de arquivos. A estrutura do sistema de arquivos é definida por inodes. Um inode possui os blocos de dados do arquivo, direitos de acesso ao arquivos e as datas de acesso e o tipo do arquivo. Cada arquivo é descrito por somente um inode. Os diretórios são arquivos especiais usados para manter o caminho de acesso aos arquivos em um sistema de arquivos. Um diretório possui uma lista de entradas e cada entrada possui as seguintes informações:

- Inode: apontador para o inode correspondente a entrada no diretório;
- Tamanho do nome: tamanho da entrada do diretório, em bytes;
- Nome: nome do arquivo que aparece na entrada do diretório.

O ext2 trata com blocos lógicos. É função do driver fazer a transformação do bloco lógico no correspondente bloco físico. O ext2 divide a partição lógica que ocupa em grupos de blocos. Em c da grupo são replicadas as informações críticas do sistema de arquivos, para recuperação em caso de falha.

O superbloco do ext2

O superbloco contém informações sobre o sistema de arquivos. Normalmente, o superbloco do bloco 0 é lido para a memória quando o sistema de arquivos é montado, mas cada grupo de blocos possui uma cópia do superbloco, para ser usada em caso de falha. O superbloco possui, entre outras, as seguintes informações:

- Número mágico: permite ao procedimento que monta o sistema de arquivos verificar que se trata do superbloco de um sistema de arquivos ext2.

- Número do grupo de blocos: contém o número do grupo de blocos que contém o presente superbloco;
- Tamanho do bloco: contém tamanho do bloco, em bytes.
- Blocos por grupo: contém número de blocos por grupo de blocos;
- Free blocos: contém o número de blocos livres no sistema de arquivos;
- Free inodes: contém o número de inodes livres no sistema de arquivos;
- First inode: contém um apontador para o primeiro inode do sistema de arquivos.

Descritor de grupo

É utilizado para descrever um grupo de blocos. Todos os descritores de grupo de blocos são armazenados seqüencialmente, constituindo a tabela de grupos de blocos. Esta tabela é replicada em cada grupo, para recuperação em caso de falhas. O descritor de grupo de blocos contém as seguintes informações:

- Mapa de bits de blocos: contém o mapa de alocação de blocos para o grupo de blocos. É utilizado para alocação e liberação de blocos;
- Mapa de bits de inodes: armazena o mapa de bits de alocação de inodes para o grupo de blocos. É usado na alocação/liberação de inodes;
- Tabela de inodes: contém o número do primeiro bloco que armazena a tabela de inodes do grupo de blocos;
- Número de blocos livres: contém o número de blocos livres no grupo de blocos;

- Número de inodes livres: contém o número de inodes livres no grupo de blocos.

A figura abaixo exemplifica um descritor de grupo de blocos.

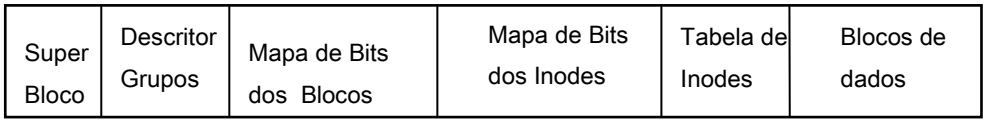

Fig. 9.9 - Descritor de grupo de blocos

O inode no ext2

O inode possui as informações sobre uma arquivo no ext2. Os inodes de um grupo de blocos são gerenciados por um mapa de bits, que permite ao sistema identificar inodes livre e ocupados. Um inode ext2 possui, entre outras, as seguintes informações:

- Proprietário;
- Permissões;
- Tipo de arquivo;
- Tamanho do arquivo;
- Datas: criação, modificação, acesso;
- Blocos de dados: endereços dos 12 primeiros blocos de dados (direto);
- Bloco de indireção: aponta para um bloco que contém apontadores de blocos de dados
- Bloco de dupla indireção: dois níveis de indireção.
- Bloco de tripla indireção: três níveis de indireção.

A figura a seguir apresenta um inode ext2.

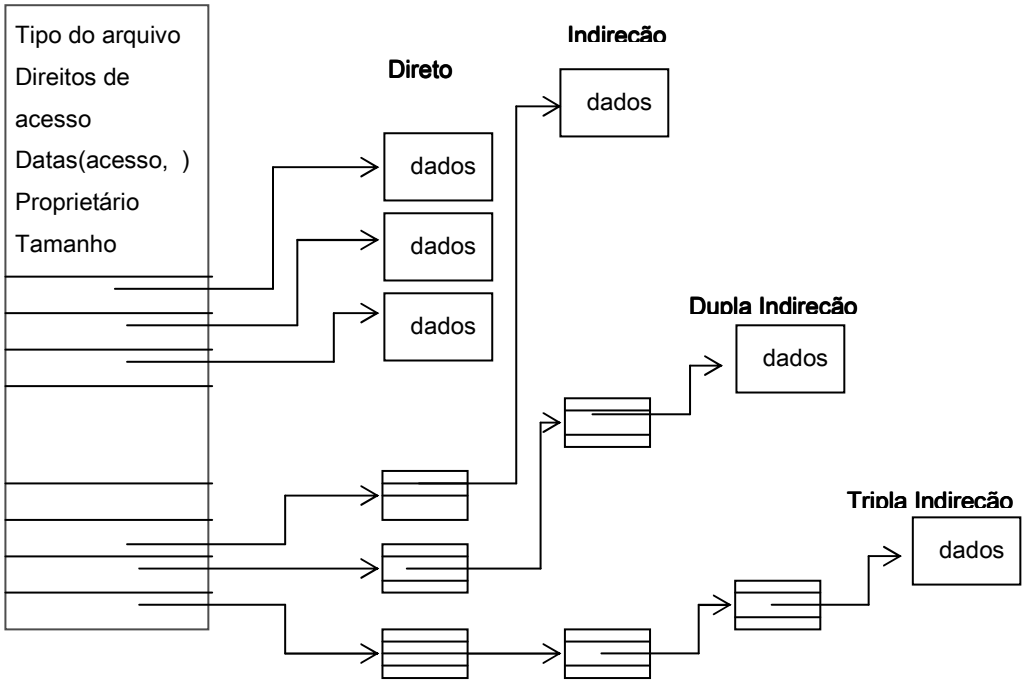

Fig. 9.10 - Representação de um I-node

As últimas versões do Linux possuem o sistema de arquivos ext3. ext3 é essencialmente um sistema de arquivos ext2, com capacidade de journaling. Journaling é uma técnica que armazena as alterações realizadas no sistema de arquivos em um arquivo especial, chamado arquivo de log. Estas alterações são efetuadas no sistema de arquivos e então o log (journal) é removido.

O jounaling evita que alterações que estão sendo realizadas e que ainda não foram efetivamente gravadas em disco se percam se, por exemplo, faltar energia. Quando o sistema voltar a operar o log é analisado. Operações que se encontram no log marcadas como não

realizadas são então gravadas em disco. Desta forma, o sistema de arquivos diminui a probabilidade de perda de dados.

9.5 Gerência de Arquivos no Windows NT

O NTFS (New Technology File System) é o sistema de arquivos desenvolvido para o Windows NT e utilizado em todos os sistemas operacionais Windows NT modernos. Foi criado pela Microsoft nos anos 1990 e apresenta alta performance, flexibilidade e segurança.

O NTFS divide o disco em clusters lógicos, isto é, a quantidade de blocos utilizada de cada vez, e o tamanho dos cluster é definido na formatação do disco. Se o usuário não especifica um tamanho diferente, o padrão para cluster em um disco novo formatado é de 4KB. Um cluster é a unidade de alocação, usada pelo NTFS. Arquivos com menos de 4KB também terão um cluster alocado, o que implica em desperdício de espaço de armazenamento.

O NTFS é um sistema hierárquico, formado por uma parte, denominada Master File Table (MFT) e uma área geral de armazenamento. Quando um disco é formatado,12% do espaço é reservado para a MFT, ficando os 88% restantes para a área geral de armazenamento. Se a MFT ocupar totalmente o seu espaço, outras áreas do disco serão alocadas para ela.

A MFT é uma tabela que contém, em cada entrada um registro que contém os atributos de um arquivo, que pode ser arquivo plano ou um diretório, armazenado na área de armazenamento. Alguns atributos de um arquivo armazenados na MFT são o nome do arquivo (no máximo 255 caracteres), direitos de acesso, a data de criação do arquivo, data de modificação, data do último acesso, etc. Nos registros da MFT são armazenados também dados dos arquivos.

Os atributos podem ser residentes ou não residentes. Atributos residentes são armazenados na MFT e os não residentes são armazenados

na área de armazenamento. O tamanho de um registro na tabela MFT é tipicamente o tamanho de um cluster. Se todos atributos de um arquivo, mais os seus dados tiverem tamanho inferior ao de um registro MFT, este arquivo será armazenado na MFT. Isto somente é possível para arquivos pequenos. Na maioria dos casos, uma parte dos dados está armazenado na MFT, juntamente com um apontador para a área de armazenamento geral, onde estão armazenados os demais dados do arquivo.

O NTFS é um sistema de arquivos transacional, implementa tolerância a falhas utilizando o conceito de transações. Uma transação é um conjunto de operações que devem ser executadas atomicamente, de maneira a manter os arquivos consistentes. Com o conceito de atomicidade, uma transação é executada completamente ou não é executada.

O funcionamento do mecanismo é o seguinte: cada vez que uma parte da transação é executada ela é gravada em um arquivo log. Se todas as operações referentes a transação são registradas no log, o sistema operacional registra (no log) que a transação se completou. No caso da transação ser interrompida por uma falha, quando o sistema operacional começar novamente a executar irá analisar o arquivo log. Transações que não se completaram terão as operações realizadas desfeitas, o que vai tornar o sistema novamente consistente.

De maneira a diminuir o overhead, o Windows NT possui uma memória cache na qual são gravadas as marcas de final de transação, que são gravadas no arquivo log, de tempos em tempos, por um processo de baixa prioridade.

O NTFS também mantém um arquivo especial, o Change Journal, no qual armazena as alterações ocorridas no sistema de arquivos. Cada entrada no arquivo é identificada por um número (Update Sequence Number) e contém o nome do arquivo e as alterações ocorridas. Sempre que um arquivo é criado, deletado ou modificado é adicionado a este arquivo. Programas de backup, por exemplo, se utilizam do Change Journal para salvar arquivos modificados desde a última operação de backup.

O NTFS possui um mecanismo de proteção de acesso aos arquivos que permite a definição de permissões a nível de diretório e a nível de arquivo plano. As permissões são para o proprietário do arquivo, o grupo e o administrador do sistema. Para diretório as permissões são:

Leitura: Listar conteúdo de um diretório e exibir permissões, donos e atributos do diretório

Gravação: Criar novos arquivos dentro do diretório, alterar atributos do diretório e visualizar o dono e as permissões do diretório.

Listar Conteúdo: Ver o nome dos arquivos pertencentes ao diretório.

Ler e executar : É permitido navegar em no diretório para chegar a outros diretórios e arquivos, mesmo que o usuário não tenha permissão de acesso aos diretórios nos quais está navegando. Ele possui os mesmos direitos que as permissões Leitura e Listar Conteúdo de diretórios.

As permissões para arquivos são:

Leitura: Ler o arquivo, exibir permissões, dono e atributos.

Gravação: O usuário pode gravar um arquivo com o mesmo nome sobre o arquivo, alterar atributos da pasta e visualizar o dono e as permissões da pasta.

Leitura e execução: Executar programas e realizar todas as ações da permissão Leitura.

Modificação: Modificar e eliminar arquivo, além de todas as ações das permissões Gravar e Ler e executar

Controle total: O usuário pode alterar permissões e tornar-se dono do arquivo, além de poder realizar todas as demais ações concedidas as outras permissões.

O NTFS mantém alguns arquivos especiais, chamados de Metadados, que possuem informações relacionadas a gerência do NTFS. Estes arquivos são criados quando um disco é formatado, e ocupam as primeiras 16 entradas (registros) da MFT. Os arquivos MetaDados são:

- Registro 0: $MFT: a MFT é um arquivo, que contém uma entrada, isto é, um registro para cada arquivo existente no volume. No caso em que as informações sobre a localização dos blocos de dados do arquivo é maior que o espaço existente em um registro, outro registro é alocado para o arquivo.

- Registro 1: $MFTMirr: armazena uma cópia das 16 primeiras entradas da MFT, para ser usada em caso de falha na MFT original.

- Registro 2: $LogFile: Contém o log de transações do NTFS.

- Registro 3: $Volume: contém informações sobre o volume em si, como versão do NTFS e data de criação do volume.

- Registro 4: $AttrDef: Contém a definição dos tipos de atributos disponíveis para serem utilizados em arquivos e pastas NTFS.

- Registro 5: $.: Contém um indicador para o diretório raiz (topo da hierarquia) do volume.

- Registro 6: $Bitmap: contém o mapa de bits que indica quais clusters estão utilizados no momento e quais estão disponíveis para alocação.

- Registro 7: $Boot: Contém cópia do código de boot do volume ou um apontador para ela.

- Registro 8: $BadClus: Contém uma lista dos clusters defeituosos do volume, para garantir que o sistema de arquivos não os utilize.

- Registro 09: $Secure: contém os descritores de segurança para todos os arquivos no volume.

- Registro 10: $UpCase: Tabela que contém informações de conversão de nomes de arquivos para o formato Unicode

- Registro 11: $Extend: usado para cotas de disco, identificadores de objetos, etc.

- Registros 12 a 15: reservados para uso futuro.

Exercícios

1. Descreva, exemplificando graficamente, a abertura do arquivo /usr/home/profs/sisop.doc em um sistema de arquivos hierárquico. Considere a existência de um I-node, que é o descritor de arquivos.
2. Considerando um sistema de arquivos hierárquico, defina uma struct com as informações que devem ser mantidas pelo sistema, referentes aos arquivos, e descreva a operação de abertura de um arquivo.
3. Descreva o funcionamento de duas operações executadas em arquivos do tipo diretório.
4. Supondo que uma arquivo de acesso direto possua blocos de 1024 bytes e que os registros lógicos sejam de 128 bytes. Em quais blocos lógicos estarão armazenados os registros lógicos 5, 21,47, 132?
5. Justifique a necessidade de armazenar na memória principal o descritor dos arquivos abertos pelos usuários.
6. Descreva a execução de uma operação read no sistema operacional Linux.

Anexo 1

Programação Concorrente com Pthreads

Neste anexo serão apresentadas as principais primitivas Pthreads de criação, espera pelo término, destruição de threads e de sincronização. Serão também apresentados, ao longo do texto exemplos de programas multithread. O objetivo do texto é servir de um guia rápido para o auxílio à construção de programas multithread no sistema operacional Linux.

1. Introdução

A biblioteca, Pthreads incorporada ao sistema operacional Linux, permite a construção de programas concorrentes escritos em C, C++. Para poder fazer uso das primitivas Pthreads, o programa deverá incluir a biblioteca Pthreads com o comando

#include <pthread.h>

A compilação de um programa escrito em C pode ser feita como mostrado a seguir:

gcc – lpthread prog.c – o prog

O compilador gcc compila o programa fonte prog.c e gera o executável prog.

2. Primitivas Pthreads de criação, de espera de término e de destruição de threads

O corpo de uma thread (o código que ela executa) é o de uma função C/C++. A criação de uma thread é feita com a primitiva

*int pthread_create(pthread_t *thid, const pthread_attr_t *atrib,*
*void *(*funcao), void *args);*

que pode aparecer em um bloco qualquer de comandos do programa. O parâmetro *thid,* do tipo `pthread_t,` contém, no retorno da chamada, o identificador da thread criada, *atrib* permite ao programador definir alguns atributos especiais (tamanho da pilha, política de escalonamento da *thread*., escopo de execução da thread, isto é, modelo 1:1 ou modelo N:1, etc.). Se for passado o argumento `NULL`, são utilizados os atributos *default*. *args* `contém` o endereço de memória dos dados passados como parâmetros para a *thread* criada. Se a operação falha, a thread não é criada e o valor retornado indica a natureza do erro:

- EAGAIN: O sistema não possui os recursos necessários para criar a nova thread.
- EFAULT: o nome da função a ser executada (thread) ou attr não é um ponteiro válido.
- EINVAL: attr não é um atributo inicializado.

No caso de sucesso da operação é retornado o valor 0.

A primitiva

*pthread_join(pthread_t thid, void **args);*

permite a uma *thread* se bloquear até que a *thread* identificada por `thid` termine. A thread pode retornar valores em *args*: *Zero* em caso de operação bem sucedida, um valor negativo em caso de falha, e um valor de retorno

calculado pela thread. Se o programador não deseja valor de retorno, pode passar NULL como segundo argumento.

O programa a seguir, escrito em C, mostra o uso destas primitivas para a implementação de um programa concorrente.

```
#include <pthread.h>
#include <stdio.h>


void * Thread0 ()
        {
          int i;
          for(i=0;i<100;i++)
                printf(" Thread0 - %d\n",i);

        }
void * Thread1 ()
        {
          int i;
          for(i=100;i<200;i++)
                printf(" Thread1 - %d\n",i);

        }

int main(){
   pthread_t t0, t1;

   pthread_create(&t0, NULL, Thread0, NULL) ;
   pthread_create(&t1, NULL, Thread1, NULL) ;
   pthread_join(t0, NULL);
   pthread_join(t1, NULL);
}
```

O programa acima é formado por três threads, uma thread t0, que executa o código da função Thread0 que imprime os valores entre 0 e 99, uma thread t1, que executa o código da função Thread1 que imprime os valores entre 100 e 199, e a thread main, que executa o código da função main do programa. A execução começa na thread main, que cria a thread to, com a chamada da primitiva pthread_create. A partir da execução desta primitiva, a thread main e a thread criada passam a compartilhar o uso do processador. A seguir, a thread main cria a thread t1 (passando então a existir três threads em execução) e se bloqueia, com a primitiva pthread_join(t0,NULL), a espera que a thread t0 termine. Após o término da thread t0, a thread main se bloqueia, com a operação pthread_join(t1,NULL), e será acordada com o término da thread t1. O uso da primitiva pthread_join é indispensável, pois impede que a

thread main continue a execução e termine, pois as threads definidas em um processo pesado são eliminadas se o processo pesado termina, o que ocorre com o término da thread main.

void pthread_exit (void *status);

Esta função termina a thread que a executa, retornando *status*.
O valor retornado é um apontador para uma variável, que pode ser o endereço de uma struct. O trecho de código a seguir ilustra o uso desta primitiva, retornando um valor.

```
int a = 0 ;
void * t0(){
    int i , x;
    for (i=0; i<10000; i++)
       {
        x= a ;
        x = x + 2 ;
        a = x ;
        }
       pthread_exit ((void *) a) ;
    }
int main(){
  int result ;
   pthread_create(&tid1, NULL,t0,NULL);
   pthread_join(tid1, ((void *)&result);
   printf("O valor de result eh: %d\n",result);
  }
```

No programa acima, no último comando executado pela função main será o printf, que exibirá o valor da variável a, calculado na thread t0.

```
pthread_kill(pthread_t tid, int signo)
```

Esta primitiva é usada para enviar sinais para threads específicas. Os sinais usados são os mesmos definidos no Unix. O programa a seguir contém a thread t0 em loop eterno, e será eliminada com o uso da primitiva *pthread_kill.*

```
#include <pthread.h>
#include <signal.h>
pthread_t tid;

long a = 0 ;

void * t0(){
     int x ;

     while(1){
          x = a ;
          x = x + 2 ;
          a = x ;
          }
     }


int main(){

   pthread_create(&tid, NULL,t0,NULL);
   sleep (1) ;
   pthread_kill(tid,  SIGKILL);
   printf("O valor de a eh: %d\n",a);
  }
```

O identificador único de uma thread pode ser obtido com a primitiva *pthread_self*, como mostrado no programa a seguir:

```
pthread_t t ;

t = pthread_self () ;
```

No final da execução da primitiva, o valor de *t* não é um inteiro, não podendo portanto ser impresso com um *printf*.

3. Primitivas de Sincronização

Pthreads suporta dois mecanismos de sincronização: semáforos e mutexes. Semáforos são primitivas usadas para sincronização e exclusão mútua. Em Pthreads os semáforos podem assumir valores maiores que 1 (semáforos

contadores). Uma variável semáforo é um tipo especial, usado para definir variáveis semáforos, como mostrado abaixo.

pthread_sem_t s0 ;

No comando acima é criada a variável semáforo s0. As operações possíveis no semáforo s0 são:

pthread_sem_init: utilizada para inicializar o valor do semáforo. O valor inicial pode ser zero ou maior que zero.

pthread_sem_wait (): se o valor do semáforo é maior que zero, o valor é decrementado e a thread continua. Caso contrário, isto é, se o valor do semáforo é zero, a thread fica bloqueada na fila do semáforo.

pthread_sem_post(): utilizada para acordar uma thread bloqueada no semáforo. Se a fila do semáforo está vazia, o valor do semáforo é incrementado.

O programa a seguir ilustra o uso de semáforos para a exclusão mútua.

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>

pthread_sem_t s0;

int a = 0 ;

void * Thread0 (){
          int i, x;
          for(i=0;i<100000;i++){
             pthread_sem_wait (&s0) ;
              x= a ;
              x = x + 2 ;
              a = x ;
             pthread_sem_post (&s0) ;
              }
         }
void * Thread1 (){
        int i, x;
        for(i=0;i<1000000;i++){
           pthread_sem_wait (&s0) ;
            x= a ;
            x = x + 5 ;
            a = x ;
           pthread_sem_post (&s0) ;
```

```
          }
        }

int main(){
   pthread_t t0, t1;
   pthread_sem_init (&s0, 0, 1) ;
   pthread_create(&t0, NULL, Thread0,NULL) ;
   pthread_create(&t1, NULL, Thread1,NULL) ;
   pthread_join(t0,NULL);
   pthread_join(t1,NULL);
}
```

No exemplo acima, o semáforo s0 é usado pelas threads Thread0 e Thread1 para controle de acesso a seção crítica de código. Na primitiva `pthread_sem_init` o semáforo s0 é inicializado com o valor 1 (terceiro argumento). O valor zero (segundo argumento) indica que as threads que irão sincronizar com o semáforo s0 fazem parte de um mesmo programa.

Mutex

Mutexes são mecanismos especiais utilizados para garantir acesso mutuamente exclusivo a dados compartilhados por diferentes threads. As funções utilizadas são:

int pthread_mutex_lock(pthread_mutex_t *mutex)

Esta primitiva é bloqueante. Se a seção crítica está ocupada, a thread requisitante ficará bloqueada até a obtenção do lock.

int pthread_mutex_trylock(pthread_mutex_t *mutex):

Esta primitiva retorna imediatamente se o lock não foi obtido.

int pthread_mutex_unlock(pthread_mutex_t *mutex)

Esta primitiva libera o lock. Se existe thread bloqueada, será liberada.

Variáveis condição

Variáveis condição são usadas nas seções críticas (lock/unlock) para sincronização de threads.

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)

Esta função bloqueia a thread que a executa na variável condição *cond,* associada a seção crítica identificada por *mutex*.

int pthread_cond_signal(pthread_cond_t *cond)

Sinaliza uma thread (acorda) entre as possíveis threads que estão bloqueadas na variável condição *cond*.

int pthread_cond_broadcast(pthread_cond_t *cond)

Acorda todas as threads bloqueadas na variável condição *cond.*

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);

Esta primitiva inicializa *mutex* com os atributos definidos por *mutexattr.* Se *attr* é NULL, os valores default são utilizados. Neste caso, se uma thread que já possui um *mutex* executar novamente *pthread_mutex_lock*, no mesmo mutex, ficará bloqueada eternamente.

int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);

Inicializa a variável condição *cond* com os atributos definidos em *cond_attr.* Se é usado NULL,os valores default são usados. Na implementação Linux threads, não são suportados atributos para as variáveis condição.

O programa a seguir apresenta um exemplo de uso de mutex.

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>


#define  N 100

pthread_mutex_t lock ;
pthread_cond_t cond, cond1 ;

int produzir = 0, consumir = 0 ;
struct node {
         int chave;
        char nome [30] ;
          struct node * prox;
       } ;
```

```
struct node *pilhaptr ;



void initPilha() {
   pilhaptr = NULL ;
   }

int pilhaVazia(){
    if (pilhaptr == NULL) return 1 ;
    else return 0 ;
    }


void empilha(int chave, char nome[30]) {

    struct node * p ;
    p = malloc (sizeof (struct node)) ;
    p->chave = chave ;
    strcpy(p->nome, nome) ;
    p->prox = pilhaptr ;
    pilhaptr = p ;
    }

void desempilha( ) {
    struct node *p ;
    p = pilhaptr ;
    if (p == 0) printf ("Pilha vazia \n") ;
    else{
         pilhaptr = p->prox ;
       printf ("Desempilhado\n");
       printf ( "chave:  %d\t Nome:  %s\n",  p->chave, p->nome) ;
        }
    }

void  * produtor (){
        int chave ;
        char n[30] ;
        int i = 0 ;
          while (i < N) {
            pthread_mutex_lock(&lock) ;
            while (!produzir) pthread_cond_wait(&cond, &lock) ;
            printf ("Produtor\n") ;
            printf ("Digite um valor inteiro\n");
            scanf ("%d", &chave) ;
            printf ("Digite o nome\n");
            scanf ("%s", &n) ;
              empilha(chave, n) ;
            produzir = 0 ;
            pthread_mutex_unlock(&lock);
           }
    }

void * consumidor (){
         int i = 0 ;
         while (i < N) {
            pthread_mutex_lock(&lock) ;
              while (!consumir) pthread_cond_wait(&cond1, &lock);
            desempilha () ;
            consumir = 0 ;
            pthread_mutex_unlock(&lock);
```

```
            }
     }

void * callProc () {

 int cont = 0, op;

 while(cont< N){
     op = rand ()%2;
     switch(op){
         case 0:
            pthread_mutex_lock(&lock) ;
              produzir = 1 ;
                pthread_cond_signal(&cond);
              pthread_mutex_unlock(&lock);
            break ;
         case 1:
            pthread_mutex_lock(&lock) ;
              consumir = 1 ;
                pthread_cond_signal(&cond1);
              pthread_mutex_unlock(&lock);
              break;
       default: break;
         }
     }
}


int main(){

   pthread_t t0, t1, t2;

   initPilha () ;

   pthread_mutex_init(&lock, NULL);
   pthread_cond_init(&cond, NULL) ;
   pthread_cond_init(&cond1, NULL) ;
   pthread_create(&t0, NULL, produtor, NULL) ;
   pthread_create(&t1, NULL, consumidor, NULL) ;
   pthread_create(&t2, NULL, callProc, NULL) ;
   pthread_join(t0,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);

}
```

O programa acima é formado por três threads: produtor,consumidor e calcproc. A thread calcproc gera um valor aleatório que indica a thread que deverá ser acordada (produtor ou consumidor). As três threads compartilham um *mutex* . Somente as threads produtora e consumidora se bloqueiam. A thread calcproc é a sinalizadora.

BIBLIOGRAFIA

[Accetta e al. 1986] Accetta, R. B., Bolosky, W., Golub, D. B., Rashid, R., Tevanian, A. E Young, M. Mach: A New Kernel Foundation for UNIX Development. Proceedings of the Summer 1986 USENIX Conference (June de 1986), páginas 93-112.

[Andrews 1983] Andrews, Gregory R. Concepts and Notations for Concurrent Programs. ACM Computing Surveys, volume 15, Número 1 (March de 1983).

[Bach 1990] Bach, M. The design of the Unix Operating System. Englewood Cliffs, N.J., Prentice-Hall (1990).

[Beck e al. 1998] Beck, M., Böhme, H., Dziadzka, M., Kunitz, U., Magnus, R., Verworner. Linux Kernel Internals. Addison-Wesley Longman (1998).

[Ben-Ari 1990] Ben-Ari, M. Principles of concurrent and distributed programming. New York, NY, Prentice-Hall (1990).

[Bernstein 1966] Bernstein A. J. Program Analysis for Parallel Processing. IEEE Trans. on Electronic Computers, EC-15, Oct 66, 757-762. (1966).

[Birrel e Nelson 1984] Birrel, A. D. e Nelson, B. J. Implementing Remote Procedure Call. ACM Transactions on Computer Systems, Volume 2, Number (February 1984), páginas 39-59.

[Black 90] Black, D. L. Scheduling support for concurrency and parallelism in the Mach operating system. Computer. V. 5(23). May, 1990.

[Bovet 2003] Bovet, D. P. e Cesati, M. Understanding the Linux Kernel. O´Reilly&Associates, CA (2003).

[Brinch Hansen 1970] Hansen, Per Brinch. The Nucleus of a Multiprogramming Systems. Communications of the ACM, volume 13, Número 4 (April de 1970), páginas 238-241.

[Brinch Hansen 1972] Hansen, Per Brinch. Structured Multiprogramming. Communications of the ACM, volume 15, Número 7 (July de 1972), páginas 574-578.

[Brinch Hansen 1977] Hansen, Per Brinch. The Architecture of Concurrente Programs. Prentice-Hall, Inc. New Jersey(1977).

[Brinch Hansen 1999] Hansen, Per Brinch. Java's Insecure Parallelism. ACM SIGPLAN Notices, volume 34 (4), (April de 1999), páginas 574-578.

[Casavant, T. L. 1988] Casavant , T.L., Khul, J.G. A taxonomy of schedulling in general-purpose distributed computing systems. IEEE Trans. Software Eng., v.l, SE-14, n.2, p. 141-154, February. 1988.

[Coffman e al. 1971] Coffman, E.G., Elphick, M. J. Shoshani, A. System Deadlocks. ACM Computing Surveys. Volume 3, Número 2 (June de 1971), Páginas 67 – 78.

[Conway 1963] Conway, M. "A Multiprocessor System Design", Proceedings AFIPS Fall Joint Computer Conference, 1963, pp.139-146.

[Courtois et al. 1971] Courtois, P. J., Heymans, F. e Parnas, D. L. Concurrent Control with Readers and Writers. Communications of the ACM, volume 14, Número 10 (October de 1971), páginas 667-668.

[Creasy, 1981] Creasy, R. J. The origin of the VM/370 time-sharing system. IBM Journal of Research & Development, Vol. 25, No. 5 (September 1981), pp. 483–90.

[Custer 1994] Custer, Helen. Windows NT. Makron Books do Brasil Editora. São Paulo (1994).

[Davis 1996] Davis, W. S. Sistemas operacionais: uma visão sistemática. Rio de Janeiro, RJ, Campus (1991).

[Deitel 1990] Deitel, H. M. Operating Systems. Second Edition, Addison-Wesley, MA (1990).

[Denning 1968] Denning, P. J. The Working Set Model for Programming Behavior. Communications of the ACM, volume 11, Número 5 (May de 1968), páginas 323-333.

[Denning 1970] Denning, P. J. Virtual Memory. ACM Comuting Surveys, volume 2 (September de 1970), páginas 153 - 189.

[Dennis e Van Horn 1966] Dennis, J. B. e Van Horn, E. C. Programming Semantics for Multiprogrammed Computations. Communications of the ACM, volume 9, Número 3 (March de 1966), páginas 143-155.

[Dijkstra1965] Dijkstra, E. W. Cooperating Sequential Process. Technical Report EWD-123. Technological University, Eindhoven, Holanda (1965).

[Dijkstra1968a] Dijkstra, E. W. The Structure of the THE Multiprogramming System. Communications of the ACM, volume 11, Número 5 (May de 1968), páginas 341-346.

[Dijkstra1968b] Dijkstra, E. W. Solution of a Problem in Concurrent Programming Control. Communications of the ACM, volume 8, Número 9 (September de 1965), página 569.

[Finkel 1988] Finkel, R. A. Operating Systems Vade Mecum. Second Edition, Prentice Hall, Englewood Cliffs, N.J. (1988).

[Flyn 72] Flynn, M.J., "Some Computer Organizations and Their Effectiveness", IEEE Transactions on Computers, Vol. 24, No. 9, pp. 948-960 (Sept. 1972).

[Foster 95] Foster, Yan. Designing and Building Parallel Programs. Addison Wesley (1995).

[Geist 94] GEIST, Al e al. PVM: Parallel and Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing. London: MIT (1994).

[Holt 1972] Holt, Richard C. Some Deadlock Properties of Computer Systems. ACM Computing Surveys, volume 4, Número 3 (September de 1972).

[Holt 1983] Holt, Richard C. Concurrent Euclid, The Unix System and Tunis. Addison-Wesley, Reading, MA (1983).

[Hoare1974] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. Communications of the ACM, volume 17, Número 10 (October de 1974), páginas 549-557.

[Kernighan e Ritchie 1988] Kernighan, B. W. e Ritchie, D. M. The C Programming Language. Second Edition, Prentice Hall, Englewood Cliffs, N.J. (1988).

[Krakowiak 1987] Krakowiak, Sacha. Principes des Systèmes d´exploitation des ordinateurs. Dunois Informatique. Paris (1987).

[**Lamport 74**] Lamport, L., A new solution of Dijkstra's concurrent programming problem. Communications of the ACM, Volume 17, Número 8, páginas 453-455, (August de 1974).

[Lampson e Redell 1980] Lampson, B. W. e Redell, D. D. Experiences with Process and Monitors in Mesa. Communications of the ACM, volume 23, Número 2 (February de 1980), páginas 105-117.

[LEWIS 1996] LEWIS, B.; BERG, D. J. Threads primer: a guide to multithreaded programming. New Jersey, Prentice-Hall (1996).

[MPI-2.1] MPI-2.1 disponível em http://www.mpi-forum.org. Acessada em 11 junho de 2006.

[Pacheco 1997] PACHECO, PETER. Paralllel Programming with MPI. Morgan Kaufmann Publischers, 1997.

[Peterson 1981] Peterson, G. L. Miths About the Mutual Exclusion Problem. Information Processing Letters, volume 12, Número 3 (June de 1981).

[PVM] PVM – Parallel Virtual Machine . Home Page disponível em http://www.epm.ornl.gov/pvm/pvm_home.html . Acessada em maio de 2007.

[Rubini e Corbet 2001] Rubini, Alessandro e Corbet, Jonathan. Linux Devices Drivers. O´Reilly&Associates In., CA (2001).

[Shaw 1974] Shaw, Alan C. The Logical Design of Operating Systems. Prentice-Hall, Inc. New Jersey(1974).

[Silberschatz 2001] Silberschatz, A.; Galvin, P. B.; Gagne, G. Sistemas Operacionais – Conceitos e Aplicações. 7a. Tiragem Elsevier Editora Ltda. Rio de Janeiro, RJ (2001).

[Silberschatz 2002] Silberschatz, A.; Galvin, P. B.; Gagne, G. Operating system concepts. 6.ed. John Wiley & Sons, Inc. (2002).

[Stallings 1998] Stallings, W. Operating Systems – Internals and Design Principles. 3.ed. Englewood Cliffs, NJ : Prentice-Hall (1998).

[Tanembaun 1992] Tanembaun, A. S. Modern Operating Systems. Englewood Cliffs, N.J. (1992).

[Tanembaun e Woodhul 1997] Tanembaun, A. S. e Woodhull, A. S. Operating Systems: Design and Implementation. Second Edition, Englewood Cliffs, N.J. (1997).

[Tay e Ananda 1990] Tay, B. H. e Ananda, A. L. A Survey of Remote Procedure Call. Operating Systems Review. Volume 24, Número 3 (JulY de 1990), Páginas 68 – 79.