

Guilherme Pina Cardim

Relatório de Sistemas Operacionais I

Presidente Prudente - SP, Brasil

30 de junho de 2010

Guilherme Pina Cardim

Relatório de Sistemas Operacionais I

Relatório das aulas ministradas na disciplina
de Sistemas Operacionais I na Faculdade de
Ciências e Tecnologia - FCT|Unesp

Professor: Maurício Araújo Dias

DEMEC - DEPARTAMENTO DE MATEMÁTICA, ESTATÍSTICA E COMPUTAÇÃO
FCT - FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNESP - UNIVERSIDADE ESTADUAL PAULISTA

Presidente Prudente - SP, Brasil

30 de junho de 2010

Resumo

Este trabalho foi desenvolvido a partir das anotações feitas durante as aulas da disciplina de Sistemas Operacionais I, decorridas durante o primeiro semestre letivo do ano de 2010 na Faculdade de Ciências e Tecnologia - FCT|Unesp.

Sumário

Lista de Figuras

1	INTRODUÇÃO	p. 8
1.1	O que é um Sistema Operacional?	p. 8
1.1.1	<i>Trabalhar como uma Máquina Virtual</i>	p. 8
1.1.2	<i>Gerenciar Recursos</i>	p. 8
2	ATUAÇÕES DOS SISTEMAS OPERACIONAIS	p. 10
3	INTRODUÇÃO A PROCESSOS E THREADS	p. 11
3.1	Processos	p. 11
3.2	Threads	p. 11
4	PROGRAMAÇÃO	p. 13
5	MEMÓRIAS	p. 14
5.1	ROM - Read Only Memory	p. 14
5.2	RAM - Random Access Memory	p. 15
6	CHAMADAS DE SISTEMAS	p. 16
7	INTERRUPÇÕES	p. 17
8	ARQUIVOS E DIRETÓRIOS	p. 18
8.1	Arquivos	p. 18
8.2	Diretório ou Pasta	p. 18

9 HISTÓRICO	p. 19
9.1 Primeira Geração - Ausência de SO	p. 19
9.2 Segunda Geração - Sistemas Batch ou em Lotes	p. 19
9.3 Terceira Geração - Multiprogramação	p. 20
9.4 Quarta Geração - Atuais	p. 20
10 ESTRUTURAS DOS SISTEMAS OPERACIONAIS	p. 21
10.1 Monolítico	p. 21
10.2 Camadas	p. 21
10.3 Máquinas Virtuais	p. 21
10.4 Modelo Cliente - Servidor	p. 22
10.5 Etc	p. 22
11 PROCESSOS	p. 23
11.1 Multiprogramação	p. 24
11.2 Tabela de Processos	p. 24
12 THREADS	p. 26
13 ESCALONAMENTO DE PROCESSOS	p. 27
13.1 Quando o escalonamento ocorre	p. 28
13.2 Objetivo do Escalonador	p. 28
13.2.1 Objetivos do escalonador para sistema Batch	p. 29
13.2.2 Objetivos do escalonador para sistemas interativos	p. 29
13.2.3 Objetivos do escalonador para sistemas de tempo real	p. 29
13.3 Algoritmos de escalonamento	p. 30
13.3.1 Algoritmo de escalonamento de sistemas Batch	p. 30
13.3.2 Algoritmo de escalonamento de sistemas interativos	p. 30

13.3.3	Algoritmo de escalonamento de sistemas de tempo real	p.31
13.4	Política X Mecanismo	p.31
14	Escalonamento de Threads	p.32
14.1	Modo Usuário	p.32
14.2	Modo Kernel	p.32
15	PROBLEMAS CLÁSSICOS DE COMUNICAÇÃO ENTRE PROCESSOS	p.33
15.1	Condição de Corrida e Região Crítica	p.33
15.2	O Problema dos Filósofos Glutões	p.34
15.3	O Problema dos Leitores e Escritores	p.34
15.4	Problema dos produtores e consumidores	p.35
15.5	Problema do Barbeiro Dorminhoco	p.36
15.6	Condições Para Solucionar Problemas	p.36
15.7	Soluções para Problemas de Comunicação	p.37
15.7.1	Exclusão Mútua	p.37
15.7.1.1	Exclusão mútua com espera ocupada	p.37
15.7.2	Bloqueio e Desbloqueio	p.37
15.7.3	Atomicidade	p.38
15.7.4	Mecanismos de Sincronização	p.38
16	DEADLOCK	p.40
16.1	O Algoritmo da Avestruz	p.41
16.2	Detecção e recuperação	p.41
16.2.1	Um Recurso de Cada Tipo	p.41
16.2.2	Vários Recurso de Cada Tipo	p.42
16.2.3	Quando fazer a detecção?	p.43

16.2.4	Recuperação de situações de deadlock	p. 43
16.3	Evitando Deadlocks	p. 44
16.3.1	Conceitos Iniciais	p. 44
16.3.2	Trajetórias de Recursos	p. 44
16.3.3	Estado Seguro e Inseguro	p. 45
16.3.4	Algoritmo do Banqueiro	p. 45
16.3.4.1	Um Único Tipo de Recurso	p. 45
16.3.4.2	Recursos de Diversos Tipos	p. 46
16.4	Prevenindo Deadlocks	p. 46
16.4.1	Ataque à Exclusão Mútua	p. 46
16.4.2	Ataque à Posse e à Espera	p. 47
16.4.3	Ataque à não-preempção	p. 47
16.4.4	Ataque ao problema da condição de espera circular	p. 47
17	Outros Aspectos	p. 48
17.1	Bloqueio em Duas Fases	p. 48
17.2	Preterição Indefinida	p. 48
	Referências	p. 49

Lista de Figuras

1	Estados de um processo	p. 23
2	Escalonamento em 3 níveis	p. 30
3	Trajetória de Recursos e Processos	p. 44

1 *INTRODUÇÃO*

1.1 O que é um Sistema Operacional?

Um Sistema Operacional, ou simplesmente SO, é um programa computacional que administra os recursos do computador, sejam eles recursos de *hardware* ou de *software*. Pode-se comparar um Sistema Operacional, com um gerente de uma loja, onde este administra e gerencia a loja, assim como o SO gerencia e cuida de todos os recursos do computador. Todo SO possui duas funções principais, a primeira é a de trabalhar como uma máquina virtual e a segunda é a gerenciar recursos.

1.1.1 *Trabalhar como uma Máquina Virtual*

A primeira função principal de todo SO é a de trabalhar como uma Máquina Virtual. O SO cria uma interface mais fácil para interação entre o usuário e a máquina, o que gera mais facilidade não só para os usuários mas também para os programadores.

Do ponto de vista de um cientista da computação é importante conhecer o funcionamento total da máquina e saber trabalhar em sua linguagem binária.

1.1.2 *Gerenciar Recursos*

A segunda função principal de todo SO é a de gerenciar recursos do computador, como o processador, as memórias, arquivos, periféricos etc. Essa função possibilita o uso adequado dos recursos de *hardware* e *software* pela execução de programas em dois modos:

- Modo usuário;

Nesse modo, o Sistema Operacional não permite o acesso a recursos significantes.

- Modo "Kernel" ou Supervisor.

É nesse modo que o Sistema Operacional permitirá ao programa a utilização de recursos mais delicados.

Um mesmo programa pode conter partes executadas em modo Kernel, enquanto que outras partes podem estar sendo executadas em modo usuário.

Imagine um processador que seja capaz de executar um único programa por vez, inicialmente o SO toma posse desse recurso, mas precisa liberá-lo para outros programas, e para que o processador recupere o recurso do processador, antes de liberá-lo ele determina um tempo em que o programa utilizará esse recurso, após o término desse tempo o próprio *hardware* faz a interrupção do programa.

2 ATUAÇÕES DOS SISTEMAS OPERACIONAIS

Dentre todas as tarefas que um Sistema Operacional é responsável, pode-se destacar:

1. Escalonamento de Processos e Threads utilizando preempção.

Preempção é a capacidade do SO retirar um recurso de um programa e recuperá-lo para si.

Um recurso não preemptível, é um recurso que não poderá ser recuperado até que ele seja liberado pelo programa que o está utilizando. Sendo assim, o SO atribui um recurso para um programa por um determinado tempo, para que esse recurso possa ser recuperado assim que o tempo se esgotar.

2. Gerência de Memória.

O Sistema Operacional organiza os programas e as informações utilizadas e manipuladas por eles dentro da memória, sobretudo, na memória RAM.

3. Organização de Sistemas de Arquivos.

Responsável pela organização dos arquivos em Disco Rígido, Memória Flash, CD's, DVD's etc.

4. Gerência de Entrada e Saída.

O SO é responsável para organizar os recurso de entrada e saída, como mouse, teclado, vídeo, impressora etc.

3 INTRODUÇÃO A PROCESSOS E THREADS

3.1 Processos

Ao se ouvir a palavra processos em computação, a primeira coisa a se pensar é um programa. Qualquer programa nasce de uma idéia. Um algoritmo é a sequência de passos para realizar a solução de um problema e um programa é a tradução do algoritmo para uma linguagem de programação.

Pode-se determinar como processo o programa e todos os recursos englobados por ele, assim como tudo que estiver ligado a esse *software* para encontrar o resultado esperado.

3.2 Threads

O conceito de Threads é a divisão dos processos para realizar tarefas em paralelo. Ou seja, se dividir um processo existente em tarefas independentes que podem ser executadas em paralelo, cada uma dessas tarefas receberá o nome de Thread.

Os Sistemas Operacionais atuais não executam os programas inteiramente de uma vez, ao invés disso ele divide os *softwares* em várias threads que são executadas em modo de concorrência. Essa troca de programas ou threads é chamada de escalonamento.

Esse escalonamento de processos é executado pelo fato dos programas tomarem recursos para si, e se eles fossem colocados inteiramente para execução esses recursos ficariam indisponíveis, e caso outra máquina que compartilhasse esse recurso necessitasse dele não conseguiria utilizá-lo até o programa acabar sua execução.

Para exemplificar esse caso pode-se pensar em uma empresa que possui uma impressora compartilhada com várias máquinas. Ao colocar um programa inteiramente em execução no processador, e se esse programa necessitar da impressora, ele prenderá esse re-

curso até o programa acabar de ser executado por completo, e sendo assim outros usuários em outras máquinas não conseguirão utilizar esse recurso durante esse período.

4 PROGRAMAÇÃO

Uma função engloba um conjunto de atividades para executar uma tarefa. Toda função que necessita de recursos sofisticados precisa efetuar uma chamada de sistema, um pedido, para que o SO execute essas tarefas em modo Kernel. Essas funções podem ser funções que necessitam de recursos de vídeo, que manipulem arquivos etc. O Kernel de um Sistema Operacional é um conjunto de programas que só são acessíveis através de chamadas de sistemas.

5 MEMÓRIAS

5.1 ROM - Read Only Memory

O usuário não consegue alterar esse tipo de memória, apenas ler as informações contidas nela. Essa memória é gravada pelo fabricante, é nesta memória que está a BIOS do computador, um pequeno Sistema Operacional utilizado para fazer a verificação inicial da máquina, sendo sua última tarefa chamar o Sistema Operacional do computador.

Linguagens de descrição de *hardware*: VHDL

V: Very-High Speed Integrated Circuit

H: Hardware

D: Description

L: Language

Dentro dessa linguagem existe a biblioteca *ieee.std_logic_1164.all* que contém a definição dos tipos de dados.

Todo circuito é uma entidade e toda entidade deve conter uma entrada **in** e uma saída **out**. A função que cria uma entidade é chamada de **entity**.

O tipo *std_logic* pode possuir quatro valores: 0, 1, X e Z. Um *std_logic_vector* é um vetor do tipo de dados *std_logic* (bits).

O valor X significa informação estranha, sem identificação, ou seja, um erro. Já o valor Z significa que a entidade está dormindo, ou seja, não está liberando informação.

4 downto 0 significa a utilização de 5 *bits* de forma decrescente. Pode-se utilizar também de forma crescente, porém deve-se utilizar apenas uma dessas formas em toda a programação.

O comando **architecture** é responsável pelo início da criação da arquitetura, do

comportamento da entidade.

architecture *nome_da_arquitetura* **of** *nome_da_entidade* **is**

Uma palavra de memória é constituída por um conjunto de bits: Ex: *10110*. Cada palavra de memória é endereçada e pode ser acessada.

5.2 RAM - Random Access Memory

A memória RAM, é uma memória de acesso randômico, ao contrário das antigas fitas magnéticas que possuíam um acesso sequencial.

Apesar da memória ROM não ser volátil, pode-se considerá-la como uma memória de acesso randômico, porém isso não é válido em provas de computação, uma vez que as memórias RAM e ROM são consideradas totalmente diferentes.

6 CHAMADAS DE SISTEMAS

As Chamadas de Sistemas são responsáveis por:

- Gerenciamento de Processos;
- Gerenciamento de Arquivos;
- Gerenciamento de Diretórios;
- Chamadas Diversas;
- API (*Application Programming Interface*).

7 *INTERRUPÇÕES*

Uma interrupção pode ser resumida como uma parada ou finalização. Uma interrupção pode ser qualquer evento externo que suspenda momentaneamente o processo que estava sendo executado. Como exemplo temos uma cozinheira fazendo um bolo que é interrompida pelo telefone tocando.

Quando uma interrupção acontece, o SO deve tratar o evento externo, esquecendo o processo que estava em execução até que a interrupção seja tratada adequadamente. Podemos utilizar o exemplo de uma impressora pedindo uma nova página de impressão para o SO, isso interromperá o processo em execução.

A interrupção mais comum acontece devido a um cronômetro, ou contador, próprio da máquina. Esse contador recebe um valor do SO que entrará em repouso para que outro programa utilize o recurso de processador, enquanto esse programa está em execução, o contador irá decrementar, e quando for igual a zero, o *hardware* interromperá automaticamente o programa em execução. Esse processo é chamado de Escalonamento de processos.

Toda vez que um processo for interrompido seu contexto deve ser armazenado em memória para que possa ser recuperado quando o processo for retomado. Esse contexto do processo envolve todo o conteúdo dentro dos registradores e instruções executadas. Uma instrução que estiver sendo executada deve ser finalizada antes que a interrupção aconteça, contudo a próxima instrução do processo deve aguardar até que a interrupção seja tratada com outras instruções.

Existem vários vetores de interrupção, um para cada tipo de tratamento. Esses vetores de interrupção são rotinas a serem seguidas para que a interrupção seja tratada adequadamente e suas principais atividades são salvar o contexto do processo e depois chamar o Sistema Operacional de volta. Um vetor de interrupção não faz o tratamento de um *dead-lock* apenas permite seu escalonamento, porém não é ele que executa o escalonamento, ele apenas permite que isso ocorra.

8 ARQUIVOS E DIRETÓRIOS

8.1 Arquivos

Arquivo é um documento armazenado em alguma memória, principalmente memórias secundárias, que são responsáveis por armazenar dados mesmo com a máquina desligada.

8.2 Diretório ou Pasta

Diretório é uma pasta de armazenamento de arquivos, isso serve para que os arquivos sejam melhores organizados. Além de arquivos, um diretório também pode armazenar outros diretórios.

9 HISTÓRICO

9.1 Primeira Geração - Ausência de SO

O computador, com a idéia dos computadores atuais, começou a ser inventado em meados da década de 40 e foram criados para ajudar em cálculos de precisão de lançamento de mísseis por norteamericanos na Segunda Guerra Mundial. Esses Computadores eram compostos por válvulas.

No final da década de 40 e início da década de 50 não havia Sistema Operacional instalados nessas máquinas. Nas universidades devia-se fazer a reserva de um horário para a utilização do computador. O programador devia chegar com seus programas já prontos em cartões perfurados para realizar os testes. A interrupção nesse caso era feita por um funcionário responsável pela reserva do computador que limitava o usuário a determinado tempo agendado, o que gerava um escalonamento do recurso.

Curiosidade: O termo *Bug do milênio* foi criado por causa dos insetos que eram atraídos pela luz produzida pelas válvulas que acabavam queimando-as.

9.2 Segunda Geração - Sistemas Batch ou em Lotes

Geração marcada pelo final da década de 50, década de 60, e começo da década de 70. Foi uma época marcada pelos transistores, resistores e diodos. As máquinas nessa época eram feitas com componentes da Eletrônica Analógica.

Pode-se dizer que o Sistema Batch era uma evolução da agenda e era caracterizado por máquinas separadas que traduziam os cartões perfurados para fitas magnéticas. Nas universidades, os interessados deixavam seus cartões perfurados com funcionários que traduziam esses cartões para fitas magnéticas, processavam o resultado no computador e depois traduziam de volta para os cartões. Nesse caso já observa-se um pouco de paralelismo, pois o programa era executados em partes, ou seja, em lotes.

9.3 Terceira Geração - Multiprogramação

As arquiteturas nessa geração são marcadas pela integração dos transistores que reduziram de tamanho e começou a ser gravado em silício. Isso marcou o início dos circuitos integrados, foi quando começou a surgir os *Personal Computer* (PC's) que obtiveram uma redução significativa em seus tamanhos.

Os sistemas foram marcados pela multiprogramação que proporcionava a execução de vários programas ao mesmo tempo dentro de uma máquina através do escalonamento.

9.4 Quarta Geração - Atuais

Na arquitetura não houve grandes avanços. Os computadores continuam a trabalhar com Circuitos Integrados, porém o nível de integração é altíssimo.

Nessa geração os Sistemas Operacionais ganharam interface gráfica que facilita a programação e manipulação dos computadores pelos usuários.

10 ESTRUTURAS DOS SISTEMAS OPERACIONAIS

10.1 Monolítico

Organizado de forma única, sendo instalado na máquina como um único bloco. Os Sistemas Operacionais que mais são instalados nos dias atuais utilizam dessa estrutura, como por exemplo temos o *Windows* e o *Linux*.

10.2 Camadas

O Sistema Operacional nessa caso é implementado em camadas que só se comunicam com suas camadas vizinhas, ou seja, cada camada só se comunica com sua camada inferior e sua camada superior. Caso a camada do topo necessite comunicar-se com a camada mais inferior, é necessário que as instruções sejam passadas de camada a camada até chegarem na camada desejada.

10.3 Máquinas Virtuais

A idéia dessa estrutura é a utilização de um único *hardware* com um único Sistema Operacional, porém esse SO cria vários Sistemas Operacionais trabalhando independentes em várias máquinas virtuais.

Vários Sistemas Operacionais virtuais estão funcionando com vários processadores virtuais que estão em cima de um único Sistema Operacional que está funcionando sobre um único processador físico.

10.4 Modelo Cliente - Servidor

Um único núcleo do Sistema Operacional está tratando diretamente com o *hardware* recebendo pedidos de sistemas. O restante está dividido em servidores (de processos, de arquivos, de gerência de memória, de E/S etc), sendo que cada um desses servidores pode ser distribuído em máquinas diferentes.

10.5 Etc

Existem muitas outras estruturas de Sistemas Operacionais, porém as mencionadas até o momento são as principais.

11 PROCESSOS

Pode-se definir um programa como um conjunto de ordens, ou instruções, passadas ao computador para que este execute um trabalho para o homem. Para executar esse programa, o computador necessita executar várias atividades, além de necessitar de recursos do computador. O conjunto de todas essas atividades junto com os recursos e o programa é chamado de processo.

Todo processo possui um estado de execução. Observe a ilustração da Figura 1 que representa esses estados e suas alterações.

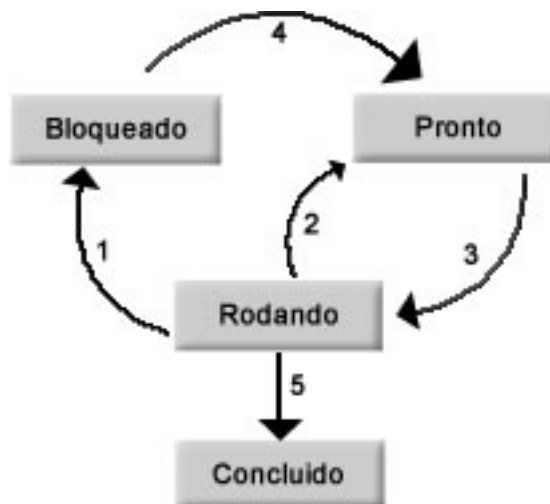


Figura 1: Estados de um processo

- Bloqueado

Acontece depois de uma interrupção feita por causa de um sinal de um periférico, pois o processo necessita de uma resposta do periférico para continuar sua execução, sendo assim, não há necessidade do processo ocupar o processador. Nesse estado, o processo está temporariamente interrompido e impedido de voltar a sua execução até que o periférico lhe retorne o resultado desejado.

- Pronto

O processo não necessita de nenhuma informação adicional a aquelas salvas em seu contexto para que possa entrar em execução, o que faz com que ele esteja apenas esperando pelo chamado do Sistema Operacional para entrar em execução no processador.

- Rodando

Nesse estado o processo está ocupando o processador, pois está em atividade.

- Concluído

Processo já finalizado.

11.1 Multiprogramação

O conceito de multiprogramação envolve a execução de vários processos em um único processador, incluindo a idéia de executar pequenas partes de cada processo por vez. A idéia da multiprogramação não é apenas gerar programação mais justas, mas também evitar com que processos de execução infinita travem o computador.

11.2 Tabela de Processos

Cada processo possui um conjunto de informações particulares armazenadas dentro da tabela de processos(TP). Essas informações são chamadas de contexto do processo.

O contexto do processo engloba o estado, a memória alocada, contador de programa¹, registradores, prioridade, contabilidade no uso de recursos, etc.

Esse contexto deve ser armazenado na tabela de processos para que ele possa ser recuperado posteriormente. Essa tabela é salva na memória RAM com sequências de bits.

A Tabela de Processos é acessada toda vez que o processador efetua um escalonamento de processos e decide qual o processo vai executar pelas informações contidas na tabela de processos e pela política de escalonamento do sistema operacional. Ela é acessada pelo vetor de interrupção.

O vetor de interrupções (VI) é uma tabela com vários ponteiros para as rotinas de tratamento de interrupções, sendo que cada rotina começa salvando o contexto do processo

¹Armazena a próxima instrução do programa a ser executada.

interrompido na TP.

12 THREADS

Uma thread é cada uma das atividades que compõem o processo. Uma thread pode ser o conceito do processo inteiro, ou pode haver várias threads dentro de um processo. Uma thread pode ser pedaços, ou partes, de um programas responsável por alguma atividade em particular. Para que um programa possa ser executado em threads separadas, tanto a linguagem como o sistema operacional devem ser capazes de executá-las.

Também é possível fazer escalonamento de threads, neste caso as informações são menores e são salvas nas tabelas thread. Esse escalonamento pode ser feito pelo programador ou pode acontecer através da ação do SO.

No caso do escalonamento feito pelo programador, este possui um maior poder de gerenciamento, enquanto que quando o escalonamento é feito pelo sistema operacional, este pode fazer o escalonamento não só das threads de um processo, mas também entre threads de processos diferentes, fazendo com que a multiprogramação seja ainda mais justa. Neste caso a tabela de thread é salva no kernel do SO que é responsável por garantir que um processo não interfira em outro processo.

13 ESCALONAMENTO DE PROCESSOS

O escalonamento de processos é o ato de escolher um entre vários processos em estado pronto para utilizar o recurso do processador, além de determinar quando a utilização do processador irá se iniciar e por quanto tempo ele ficará no estado rodando.

O mecanismo responsável por realizar o escalonamento de processos é chamado de escalonador e a lógica empregada por ele é implementada no algoritmo de escalonamento.

A primeira coisa a ser feita em um escalonamento é salvar o contexto do processo que está em execução até o momento. Esse contexto irá permitir retomar o processo no mesmo ponto onde ele foi interrompido. Contudo, o ato de salvar o contexto de um processo não é trivial e nem instantâneo. Quanto mais vezes o escalonamento acontecer em um determinado tempo, mais vezes contextos serão salvos e recuperados, o que pode acarretar em uma demora maior do que a desejada.

Por outro lado, se as threads do processo são grandes e o processo pequeno, pode acontecer menos escalonamentos, porém o tempo de resposta poderá ser muito grande. Sendo assim, o ideal é ter threads de tamanho médio.

Muitos estudos são realizados tentando encontrar o número de escalonamentos ideal que deve ocorrer no Sistema Operacional de forma que não atrapalhe significativamente o tempo de resposta dos processos.

Existem dois tipos de processos:

- A: Utiliza muito o processador, porém existe poucas chamadas de Entrada e Saída(E/S).
- B: Utiliza pouco o processador, contudo existe muitas chamadas de E/S.

Normalmente os sistemas operacionais priorizam processos do tipo B. Se o SO priorizasse processos do tipo A, a cpu ficaria muito tempo ocupada enquanto que os periféricos ficariam ociosos. O correto é que o SO não deixe nada "parado".

13.1 Quando o escalonamento ocorre

- Criação de um processo (Processo de Execução)

Para executar o processo, o escalonamento deve ocorrer fazendo com que o processo tome posse do processador.

- Bloqueio

Ao ocorrer um bloqueio de processo, o escalonamento deve ocorrer instantaneamente para que o processador não fique ocioso.

- Por tempo

O SO atribui um tempo de execução para cada processo. Quando esse tempo terminar o escalonamento ocorre colocando outro processo para utilizar o recurso do processador.

- Entrada e Saída

O periférico manda um sinal para o SO que chama o escalonamento para recuperar o processo referente ao sinal do periférico.

- Fim do processo

Quando um processo é finalizado o escalonamento age de modo a liberar o processador para outro processo.

13.2 Objetivo do Escalonador

1. É objetivo de todo Sistema Operacional **atribuir um tempo para todos os processos**. Todos programas podem e devem ser executados.
2. **Respeitar a política de escalonamento**. Alguns processos podem possuir prioridade em relação a outros. A política de prioridades é a maneira como o SO trata a prioridade dos processos determinando o funcionamento do escalonamento.
3. **Manter todo o sistema computacional o mais ocupado possível**. Quanto menos coisas estiverem paradas, mais o sistema estará produzindo.

13.2.1 Objetivos do escalonador para sistema Batch

Os sistemas Batch não trabalhava com multiprogramação. Neste caso a única diferença é que não ocorre escalonamento por tempo. Sendo assim o escalonamento ocorrerá por bloqueio, interrupção de E/S e pelo fim do processo. Os objetivos do escalonamento de sistema Batch são:

1. Maximizar o número de processos por hora
2. Minimizar o tempo de execução para cada processo
3. Manter a CPU ocupada o tempo todo.

13.2.2 Objetivos do escalonador para sistemas interativos

Esse tipo de sistemas é aquele que necessita de muitos recursos mandando várias chamadas de sistemas e comandos para o SO. Seus objetivos são:

1. Respostas rápidas
2. Atender as expectativas do usuário

Se o usuário possui expectativas de receber a resposta em determinado tempo, o SO deve tentar fazer isso da melhor forma possível. Determinados programas ou tarefas possuem um tempo esperado de resposta.

13.2.3 Objetivos do escalonador para sistemas de tempo real

Sistemas Operacionais desse tipo realizam processamentos sobre informações captadas em tempo real, como filmagens de segurança e temperaturas. Esse tipo de sistema deve estar atento as informações captadas para que essas não se percam. Os objetivos do escalonador nesse SO são:

1. Impedir perda de dados
2. Impedir a degradação da qualidade em sistemas multimídia.

13.3 Algoritmos de escalonamento

Um algoritmo de escalonamento é a maneira como o escalonador irá escolher qual processo utilizará o processador naquele momento.

13.3.1 Algoritmo de escalonamento de sistemas Batch

- O primeiro processo a chegar é o primeiro que irá utilizar o recurso, é uma fila de processos;
- Menor trabalho primeiro;
- Menor tempo restante;
- Escalonamento em três níveis
 - Escalonamento de admissão (1ª vez de execução do programa no sistema);
 - Escalonamento da CPU;
 - Escalonamento de memória;

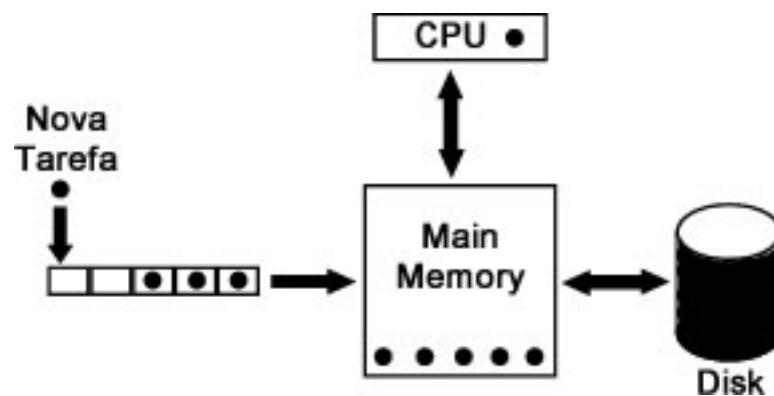


Figura 2: Escalonamento em 3 níveis

13.3.2 Algoritmo de escalonamento de sistemas interativos

- Round-Robim;
- Prioridade;
- Filas Múltiplas;
- Processo Mais Curtos;

- Escalonamento Garantido;
- Loteria;
- Fair-Share (Igualdade entre usuários - Intercâmbio entre usuários).

13.3.3 Algoritmo de escalonamento de sistemas de tempo real

- Hard (não pode perder informações de maneira alguma)
- Soft (De certa forma, pode haver pequenas perdas de dados)
- Periódico (Recebe informações por períodos determinados)
- Não-Periódicos (Período incerto, como no sismógrafo)
- Estático (As regras do escalonamento são definidas antecipadamente)
- Dinâmico (As regras do escalonamento se alteram conforme o necessário)

13.4 Política X Mecanismo

O mecanismo é o próprio algoritmo de escalonamento, enquanto que a política de escalonamento é como o algoritmo irá trabalhar. O mesmo mecanismo do algoritmo pode se adequar a várias políticas.

14 Escalonamento de Threads

O escalonamento de Threads ocorre de duas maneiras distintas, uma no modo kernel do sistema operacional e a outra no modo usuário.

14.1 Modo Usuário

Neste caso a tabela de contexto de threads é salva dentro do processo, o qual é o responsável por realizar o escalonamento das threads. Enquanto o próprio programa é o responsável pelo escalonamento das threads o escalonar fica responsável por realizar o escalonamento apenas dos processos.

- Possível: A1, A2, A3, A1, A2, A3.
- Impossível: A1, B1, A2, B2, A3, B3.
- Vantagem: Ao escalonar threads o contexto do processo não precisa ser recuperado.
- Desvantagem: Pouco flexível.

14.2 Modo Kernel

Nesse tipo de escalonamento o sistema operacional será responsável por escalonar os processos e as threads.

- Possível: A1, A2, A3, A1, A2, A3.
- Possível: A1, B1, A2, B2, A3, B3.
- Vantagem: Muito flexível.
- Desvantagem: Em todo escalonamento de thread o contexto do processo deve ser recuperado.

15 PROBLEMAS CLÁSSICOS DE COMUNICAÇÃO ENTRE PROCESSOS

Ao falar em comunicação, surge logo a idéia de troca de mensagens. A troca de mensagens é um meio de comunicação de processos, contudo a comunicação de processos não é feita somente por troca de mensagens.

Comunicação entre processos tem por idéia os processos competindo por recursos. Esta relacionada com processos conversando entre si para que não haja conflitos na busca e alocação de recursos.

15.1 Condição de Corrida e Região Crítica

Imagine a situação onde carros possam ser comparados com processos, e estão em uma condição de corrida em um cruzamento de vias, ou seja, os carros estão disputando o mesmo recurso, que neste caso seria parte da rodovia. Uma vez que os carros, ou os processos estão em condição de corrida, esse cruzamento se torna uma área perigosa, ou seja, uma região crítica, pois nesta área pode ocorrer um conflito, ou uma colisão entre os carros (processos).

Os processos tendem a competir por recursos, isso é o que chamamos de condição de corrida dentro do sistema da computação. A região crítica é um pedaço do código do programa, e uma vez que o programa está contido dentro do processo, é parte do processo, em que o programa vai requisitar ou estar requisitando o uso de um determinado recurso.

Quando dois processos estão na mesma condição de corrida eles irão causar uma colisão caso não for utilizados métodos para evitar colisão entre processos.

Na nossa analogia com os carros em vias, podemos corrigir esse problema com a utilização de semáforos, que servirá para sincronizar eventos, ou utilizar sinalizações de

forma a dar preferência a certos processos, ou carros.

A idéia é fazer com que dois ou mais processos não entre na mesma região crítica ao mesmo tempo, ou seja, apenas um processo pode entrar na região crítica por vez.

15.2 O Problema dos Filósofos Glutões

O problema dos filósofos Glutões (Comilões) funciona da seguinte forma: Existem 5 pratos em uma mesa redonda com macarronada, sendo que foi posto muito molho de forma de que cada um dos 5 filósofos necessite de dois garfos para conseguir pegar o macarrão sem que este escorregue. Contudo só há um garfo para cada prato colocado a sua lateral.

Problema: Irá faltar garfos para que todos os filósofos comam ao mesmo tempo. Dessa forma, quem irá comer primeiro?

Pensando na solução: Apenas dois filósofos irão comer ao mesmo tempo, sendo que eles nunca estarão um ao lado do outro. Como todos estão com fome a tendência é que todos tomem posse de seu garfo rapidamente. Contudo, se isso ocorrer nenhum deles conseguirá comer a macarronada e um *deadlock* acontecerá, pois todos estarão esperando que o outro largue o garfo.

Solução: quando acontecer um *Deadlock*, todos devolvem os garfos a mesa, porém isso ainda não resolverá o problema. Isso é uma condição dentro do sistema chamado *Starvation*, onde o sistema sempre está trabalhando mas não produz nada útil. Como é o caso de pegar garfo e devolver garfo continuamente. A idéia da solução é que quando devolverem o garfo a mesa, cada filósofo deve esperar um tempo diferente para poder pegar o garfo novamente. Trazendo esse problema para um sistema de computação, os pratos representarão os processos e os garfos os recursos. Os filósofos estão correndo para pegar os garfos, ou seja, isso é uma condição de corrida e todos eles estão entrando ou tentando entrar em suas regiões críticas.

15.3 O Problema dos Leitores e Escritores

Condição: vários leitores tentando ler um livro ao mesmo tempo em que vários escritores tentando escrever naquele livro. Se dois ou mais escritores acessarem a mesma base de texto, cada um pode escrever um registro de assuntos diferentes na mesma linha,

assim a informação vai ficar bem confusa caso isso ocorra.

Solução: Cada escritor vai ter que escrever em seu tempo, uma vez que vários escritores escrevendo ao mesmo tempo é uma condição de corrida do sistema. Não temos problemas quando várias pessoas estão lendo a mesma base de dados, pois os leitores não irão retirar ou adicionar informações. O escritor está fazendo o processo de escrever e de apagar informações, dessa forma um escritor não pode apagar enquanto estiver leitores fazendo leitura, pois isso tornaria a base de dados inconsistente.

Se priorizarmos os leitores, o escritor pode nunca escrever, pois pode chegar leitores a todo momento. Uma das possíveis solução seria fazer uma fila, onde quem chegar primeiro utiliza a base de dados primeiro. Quando um escritor estiver escrevendo, ele tem que sinalizar que esta escrevendo, para que outros escritores não entrem na região crítica. Isso pode ser feito por uma sinalização de 0 e 1.

15.4 Problema dos produtores e consumidores

O braço é a base de dados e as mãos é o processo. Uma das tarefas do sistema operacional é evitar que o processo tenha acesso ao código de outro processo. Dessa forma a mão, que no caso é um processo, não pode ser apagada, então cada processo vai receber uma área de memória. Um processo não pode entrar na área de memória do outro processo, então ele não pode escrever nem apagar nada de outro processo.

Tudo o que for compartilhado tem que estar em uma área de memória que pode ser vista por ambos os processos. Se o sistema operacional não tomar conta disso, o processo A poderia interferir no processo B. Essa é uma das tarefas do gerente de memórias.

Condição: consumidor pode ir apagando até o ponto que não houver mais informação na base de dados e o produtor pode ir adicionando informações até que a base de dados fique cheia. Depois que acabar o espaço para gravação na base de dados ou acabar os dados da mesma, o processo para pois não tem como continuar sua execução. Se o consumidor parou de trabalhar significa que a base de dados está cheia, ao contrário do produtor que irá parar quando a base de dados estiver cheia.

15.5 Problema do Barbeiro Dorminhoco

Condição: um barbeiro disposto a cortar o cabelo dos clientes possui em sua barbearia 5 cadeiras para os clientes. Quando a barbearia está sem clientes, o barbeiro aproveita para dormir e descansar em uma das cadeiras, porém quando um cliente chega ele acorda para fazer o serviço desejado. Isso é uma máquina de estado finito, que é a mudança dos estados dos processos. Quando o barbeiro está cortando o cabelo, outro cliente vai esperar nas cadeiras, mas todo mundo corta o cabelo com ele. Quando as cinco cadeiras já possuírem clientes e o sétimo cliente chegar, este ficará bloqueado até que o barbeiro acabe seu trabalho no cliente atual de forma a liberar uma cadeira em sua barbearia.

15.6 Condições Para Solucionar Problemas

- Dois ou mais processos não podem entrar em suas regiões críticas simultaneamente; Apenas um processo poderá tomar um recurso para si em um determinado instante de tempo, ou seja, apenas um processo poderá entrar em sua região crítica por vez.
- Não considerar velocidade relativa dos processos e número de processadores; Um programa deve funcionar em qualquer arquitetura.
- Qualquer processo fora de sua região crítica não pode bloquear outro processo; Se um processo tomou para si um recurso, ele deve avisar o SO que está usando aquele recurso, e quando o SO perceber que existe outro processo precisando utilizar o mesmo recurso, o SO deve bloquear o segundo processo evitando qualquer problema de *deadlock*.
- Um processo não pode esperar para infinitamente para entrar em sua região crítica; O SO deve determinar um tempo para um processo utilizar um determinado recurso. Após o tempo expirar o SO deve disponibilizar o recurso para outro processo.

Obs: A conversa entre dois processos normalmente ocorre com o intermédio do SO, porem isso não é obrigatório. Uma outra maneira de dois processos se comunicarem é através de uma variável global compartilhada entre os processos.

15.7 Soluções para Problemas de Comunicação

Existem quatro soluções para problemas de comunicação. São elas:

- Exclusão mútua;
- Bloqueio e Desbloqueio;
- Atomicidade;
- Mecanismos de Sincronização.

O SO normalmente trabalha com as 4 soluções ao mesmo tempo.

15.7.1 Exclusão Mútua

Um processo tem a capacidade de excluir outro processo no sentido de impossibilitar esse a utilizar o mesmo recurso do qual já se faz uso. Ou seja, quando um processo toma um recurso para si ele exclui a possibilidade de outros processos tomarem o mesmo recurso.

Se os recursos trabalharem de forma sincronizada, dificilmente haverá colisões. Para sincronizar o movimento dos recursos deve-se utilizar sinalizações, conhecidas como mecanismos de sincronização que estão sempre baseados no uso de variáveis.

15.7.1.1 Exclusão mútua com espera ocupada

Imagine que você possui um carro e deseja sair da garagem e andar em uma rua de São Paulo muito congestionada. Uma vez que é impossível retirar o carro da garagem nessas condições você decide pegar o carro e ficar andando pra frente e pra trás dentro da garagem. Isso é algo que funciona, mas não é algo bom para o sistema, pois o desempenho do sistema diminui consideravelmente pelo fato do SO fazer o processo executar tarefas que não resultam em nada, ou seja, o processo entrará em um *loop* de processamento sem resultar em nada de útil para o processo.

15.7.2 Bloqueio e Desbloqueio

O processo A toma o recurso para si, contudo o processo B também deseja o mesmo recurso. Sendo assim o processo B não pode utilizar o recurso até que ele seja liberado,

logo o SO bloqueia o processo B que ficará aguardando o recurso ser liberado pelo processo A.

Para bloquear ou retomar um processo, o SO faz uso das primitivas (Chamadas de Sistemas) *sleep*, *wakeup*, *down* e *up*.

- Primitivas *Sleep*: Bloqueia um processo;
- Primitivas *Wake up*: Colocar o processo como pronto ou rodando;
- Primitivas *down* e *up*: Mudança de estado (0 e 1) em uma variável global.

Quando um processo tenta utilizar um recurso e não consegue, ele mesmo pode enviar um primitiva *sleep* para o SO, com o intuito que o SO o bloqueie até o recurso ser liberado.

15.7.3 Atomicidade

Em química a palavra Átomo significa uma partícula que não pode ser dividida. Em computação Atomicidade são ações indivisíveis. Um processo que entrar em uma execução de atomicidade, mesmo que receba um sinal de interrupção, o SO deve permitir ao processo a execução sem pausa dessa sequência de tarefas atômicas.

Exemplo:

- Verificar o valor da variável de sinalização (detectar se o recurso esta disponível);
- Modificar o valor da variável de sinalização;
- Chamada da primitiva *sleep*.

15.7.4 Mecanismos de Sincronização

- Mútex;

Funciona como um semáforo de trânsito com dois estados (0 e 1). Um processo que vai utilizar um recurso seta a variável para 0, quando outro processo tentar utilizar o mesmo recurso ele verifica essa variável e percebe que não será possível, pois ele já está em uso. Quando o processo liberar o recurso ele deve setar a variável para 1 liberando o recursos para outros processos.

- Semáforos;

Em SOs, semáforos são equivalentes a semáforos graduais de tempo indicando quantos processos estão utilizando um determinado recurso. Em programação podemos fazer isso utilizando uma variável contadora.

- Contador de eventos (Sem exclusão mútua);

Utilização de vários semáforos possibilitando que dois processos distintos utilizem o mesmo recurso sem a exclusão mútua.

- Monitores;

O uso de monitores depende de uma linguagem de programação que permita a implementação desse recurso. Uma vez que um processo entra dentro de uma *procedure*, nenhum outro processo pode entrar nessa mesma *procedure*.

- Troca de Mensagens;

- *Send*(destino, mensagem);

- *Receive*(destino, mensagem);

O processo A quer receber determinadas informações do processo B, sendo assim ele manda uma mensagem para o processo B dizendo a quantidade de informações que ele deseja receber ("manda o engradado de cerveja vazio"). Dessa forma, o processo B saberá a quantidade de informações que poderá enviar para o processo A ("engradado cheio").

- Barreiras.

Existe uma variação de acordo com o tempo. O SO pode bloquear todos os processos até que todos cheguem a uma determinada barreira para a partir disto liberar todos os processos para seguirem seu processamento.

O SO faz com que todos os processos atinjam um determinado nível, para só assim liberar todos novamente para um novo processo(nível)

16 DEADLOCK

Definição formal: Um conjunto de processos está em uma situação de *deadlock*, se cada processo do conjunto estiver esperando por um evento que somente outro processo pertencente ao conjunto poderá fazer acontecer.

Em uma situação de *deadlock* todos os processos estarão esperando pela liberação de recursos que outros processos detém, sendo que esses por sua vez também estão esperando por outros recursos, criando assim uma situação cíclica onde os processos ficarão bloqueados eternamente.

Deadlock normalmente envolve recursos, sendo eles *hardware* ou *software*. Sendo que é necessário pelo menos dois recursos, assim como dois processos também são necessários. Contudo um *deadlock* também poderá ocorrer com a espera de eventos. Como exemplo podemos imaginar um processo que esteja esperando que um termômetro ambiente atinja 60°, ele estará esperando a ocorrência de um evento, o que pode levar o processo a ficar bloqueado eternamente, levando futuramente a um possível *deadlock*.

Existem quatro condições obrigatórias para a ocorrência de *deadlocks*, ou seja, são quatro condições que necessitam acontecer simultaneamente para que o *deadlock* ocorra.

- Condição de exclusão mútua: cada recurso está alocado a um único processo;
- Condição de posse e de espera: processos detendo recursos podem solicitar novos recursos;
- Condição de não-preempção: recursos não podem ser tomados a força;
- Condição de espera circular: cadeia circular de processos esperando recurso.

Estratégias para tratar *deadlock*:

- Algoritmo da Avestruz: Ignorar completamente o problema;

- Detecção e recuperação da situação de *deadlock*: Remediar o deadlock;
O deadlock ocorre, só depois o SO o trata.
- Evitar *deadlock*: Alocação cuidadosa de recursos;
O SO faz o máximo possível para que o deadlock não ocorra, contudo ele ainda existirá com um número menor de ocorrências.
- Prevenção de *deadlock*: Negar uma das quatro condições para ocorrência de deadlock.
Fazer com que o *deadlock* não ocorra nunca. É possível, mas o custo em desempenho é altíssimo.

16.1 O Algoritmo da Avestruz

A primeira forma de se tratar o deadlock é não fazer absolutamente nada em relação a esse problema. A maior parte dos SOs fazem isso, simplesmente ignoram o *deadlock*. Nesse algoritmo o SO não faz nada em relação ao problema do *deadlock*, o SO simplesmente para de funcionar travando todo o sistema.

16.2 Detecção e recuperação

Características:

- Perda de desempenho do SO;
- O SO permite a ocorrência dos deadlocks;
- O SO detecta as ocorrências;
- Normaliza a situação.

16.2.1 Um Recurso de Cada Tipo

O SO trabalha com um único recurso de cada tipo. Para o computador não é fácil detectar os *deadlock*. Embora seja fácil para um ser humano observar um *deadlock* a partir de um grafo, o algoritmo de detecção não trabalha com grafos, ele irá verificar as

informações presentes na tabela de processos, pelas quais o algoritmo consegue fazer uma interpretação da realidade do sistema e determinar a ocorrência de um *deadlock*.

O algoritmo parte de algum nó N aleatório, e procura um *deadlock* acompanhando o sentido da seta. Se algum dos nós nessa busca se repetir, significa que existe um *deadlock*. Dessa forma, o algoritmo pode avisar o SO que irá tratar o *deadlock* encontrado.

16.2.2 Vários Recurso de Cada Tipo

O SO trabalha com vários recursos de cada tipo (Vários hds, várias impressoras, vários scanners, ...). Tudo que existe em questão de recursos está no vetor E, e ainda mais, esse vetor E separa os recursos pelo tipo a que pertencem. A matriz C representa os recursos alocados para processos. A quantidade de linhas dessa matriz determina a quantidade de processos ativos no SO.

O vetor A representa os recursos que estão disponíveis no sistema. Para obtê-lo basta somar a quantidade de recursos alocados por tipo e subtrair pelo seu tipo no vetor E (vetor de recursos existentes). A matriz R representa a matriz das requisições sendo que cada linha representa novamente cada processo ativo no SO, enquanto que cada coluna representa os recursos por tipo.

O sistema vai detectar na matriz R se existe algum processo que possa ser atendido em relação a suas requisições. Caso encontre, ele avisa o SO, mais especificamente o escalonador de processo, para que este coloque o recurso em estado rodando para que ele possa tomar para si aqueles recursos que estão disponíveis e ele está requisitando.

Se houver algum recurso que possa ser atendido totalmente quanto as suas requisições, o algoritmo avisa o escalonador para coloca-lo para rodar para que ele possa ser concluído, caso não haja processos que possam ser atendidos totalmente, o algoritmo detecta a ocorrência então de um *deadlock*, avisando o SO sobre essa ocorrência e esse poderá tomar então uma providência.

Representação matemática para o algoritmo de detecção de *deadlocks*:

$$\sum_{i=1}^m C_{ij} + A_j = E_j$$

Tudo que existe no sistema ou já está alocado para algum processo ou ainda está disponível para alocação. Os sistemas são dinâmicos, ou seja, a todo tempo podemos ter recursos novos, ou recursos sendo desativados. No mesmo momento que o algoritmo esteja

realizando a detecção, novos recursos podem surgir ou desaparecer, porém não farão parte da análise, essa alteração só fará parte da próxima detecção.

16.2.3 Quando fazer a detecção?

- Toda vez que um processo requisita um novo recurso. É possível e pode ser executado.

Ponto positivo: Ocorreu deadlock, imediatamente o SO poderá tratar, porque ele irá verificar em "cima do lance".

Ponto negativo: Grande perda de desempenho.

- Pode-se definir um tempo para que o algoritmo entre em funcionamento.
- Pode-se chamar o algoritmo quando existe perda de desempenho do sistema, pois pode haver um *deadlock* ocorrendo.

16.2.4 Recuperação de situações de deadlock

- Recuperação através da preempção;

O Sistema deve estar bem preparado para que isto seja possível, pois é um método de força bruta.

- Recuperação através de volta ao passado;

A tabela de processos vai ser gravada em disco (HD) várias vezes no decorrer do sistema, sendo que cada gravação não sobrescreve a outra. Caso ocorra um *deadlock*, o sistema restaura a tabela de processos a partir dessas gravações feitas no disco. O SO irá voltando e modificando o caminho até que não apareça o *deadlock*.

- Recuperação através da eliminação de processos. (Mais usado e mais fácil de implementar)

É o método mais utilizado por possuir uma implementação mais fácil. A idéia é matar um processo, acabar com ele instantaneamente. O SO pode eliminar um processo que esteja envolvido no *deadlock* através de alguma política de prioridade ou até mesmo de forma aleatória. O SO também pode escolher um processo que esteja fora do *deadlock*, pois este pode possuir vários recursos de interesse para os processos que estão em *deadlock*.

16.3 Evitando Deadlocks

16.3.1 Conceitos Iniciais

A primeira coisa a ser feita pelo SO é perguntar se é seguro alocar o processador para determinado processo. Mas como ele decide isso? A questão principal por trás disso é que os algoritmos devem trabalhar com informações pré-definidas. Os processos devem saber de antemão quais os recursos que irão utilizar durante sua execução.

16.3.2 Trajetórias de Recursos

Imagine um plano cartesiano, onde o eixo X representa o processo A, possuindo todas as instruções referentes ao processo A, e o eixo Y representa o processo B, apresentando todas as instruções referentes ao processo B. Dessa forma, cada processo possui muitas instruções e são independentes um do outro. As linhas tracejadas representam a execução das instruções de acordo com o tempo. Cada ponto representa um escalonamento, onde o escalonador de processos faz a troca de processos em execução. Depois da instrução I_1 , o processo A conseguiu alocar a impressora para seu processamento. No ponto s está tudo ok até então, porém no ponto t , o processo B poderá requisitar o plotter, se o plotter for liberado para o processo B, e a impressora já estiver alocada para o processo A, ocorrerá o deadlock no ponto y , onde cada processo está aguardando a liberação de um novo recurso sendo que estes estão alocados um ao outro.

Todas as áreas tracejadas são impossíveis de serem atingidas, pois se um processo detém um recurso, outro processo não conseguirá obtê-lo, representando assim momentos onde um recurso está alocado a mais de um processo o que é impossível de ocorrer por causa da exclusão mútua.

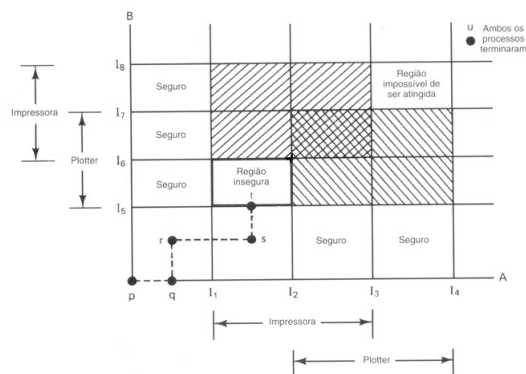


Figura 3: Trajetória de Recursos e Processos

16.3.3 Estado Seguro e Inseguro

Estado seguro é o estado onde o processo pode ser executado tranquilamente, pois não existe nenhum risco de ocorrência de *deadlock*.

Estado Inseguro é o estado onde o SO não consegue garantir a execução dos processos, uma vez que existe a possibilidade de ocorrer um *deadlock*. Quando o sistema entrar em um estado inseguro, o *deadlock* só não ocorrerá se por algum motivo desconhecido um processo finalize ou libere recursos por motivo indeterminado.

A única diferença entre estado seguro e estado inseguro é o que os caracteriza, onde um processo em estado seguro, o SO consegue determinar seu funcionamento por completo, ao contrário do estado inseguro, onde o SO não consegue dar essa garantia.

Só é possível evitar o *deadlock* se o SO souber tudo que irá acontecer durante todo o percurso de execução de um processo, contudo, se um processo é iterativo com o usuário é praticamente impossível de se identificar todos os recursos que serão utilizados durante a execução desse processo.

16.3.4 Algoritmo do Banqueiro

O algoritmo do Banqueiro é utilizado para tentar evitar o *deadlock*.

16.3.4.1 Um Único Tipo de Recurso

O que se espera do algoritmo do banqueiro neste caso é que este consiga manter o sistema sempre no estado seguro fazendo com que o SO garanta assim a execução por completo do processo.

O algoritmo começa a distribuir os recursos aos poucos, pois sabe de antemão a quantidade que cada processo irá necessitar. Contudo, o algoritmo deve liberar os recursos de forma a garantir que sempre haja pelo menos um processo que poderá ser atendido com a quantidade de recursos disponíveis naquele momento e que os outros processos possam ser atendidos futuramente. Cada vez que um processo libera recurso, ele irá procurar pelo processo que pode ser atendido naquele momento.

16.3.4.2 Recursos de Diversos Tipos

Nesse caso temos vários recursos de diversos tipos. Deve-se possuir uma tabela D capaz de indicar quantos recursos e quais recursos estão alocados para cada processo, sendo que as colunas representarão os tipos de recursos enquanto que as linhas dessa tabela representarão os processos em execução. Uma segunda tabela R deve ser utilizada com a mesma estrutura mas armazenando a quantidade de requisições feita por cada processo para cada tipo de recurso.

Além dessas duas tabelas deve-se possuir o vetor E indicando a quantidade de recursos total presentes no sistema, o vetor P representando o total de recursos que já foram alocados independentes do processo e o vetor A indicando a quantidade de recursos ainda disponíveis no sistema.

O algoritmo irá procurar na tabela R qual linha, ou seja, qual processo poderá ser satisfeito em comparação com o vetor A. Dizendo de outra forma, o algoritmo irá verificar qual linha da tabela é menor ou igual ao vetor A para todos os recursos, garantindo sempre que o sistema irá satisfazer não só um processo, mas todos os processos ao longo do tempo.

16.4 Prevenindo Deadlocks

Para prevenir o *deadlock* deve-se utilizar uma vacina para as causas desse problema. Sendo assim, deve-se procurar a vacina adequada para cada situação.

Sabe-se que para ocorrer um *deadlock* quatro condições são necessárias simultaneamente, dessa forma, para prevenir *deadlocks* deve-se combater uma das quatro condições necessárias.

16.4.1 Ataque à Exclusão Mútua

Cada recurso está alocado a um único processo. Através da exclusão mútua, estaremos garantindo que dois processos não entrem em sua região crítica ao mesmo tempo. Dessa forma temos que tomar cuidado com uma vacina para esse caso.

A vacina deve agir de forma que nenhum recurso fique de posse de apenas um processo, contudo um recurso disponível a vários processos pode causar uma confusão, ou seja, pode causar problemas. Um meio termo para esse problema seria a utilização do SPOOL.

SPOOL: Fila de trabalhos. Todos os processos possuem a idéia de possuir o recurso,

fazendo com que ele não precise esperar a liberação de recurso. Porém suas requisições ficarão em fila. Isso garante a exclusão mútua, porém possibilita aos processos a percepção que a exclusão mútua está sendo ignorada pelo sistema.

A vacina spool não funciona para todos os recursos, pois existem recursos que infelizmente não podem entrar em fila de spool.

16.4.2 Ataque à Posse e à Espera

O objetivo dessa vacina é impedir a posse de recursos por um processo enquanto este está esperando por outros recursos.

O processo deve informar ao SO todos os recursos que irá utilizar, contudo isso é praticamente impossível, pois normalmente, e principalmente processos que interagem com usuários, não sabem todos os recursos que utilizará durante sua execução. Se isso fosse realmente viável, poderíamos esquecer a prevenção de *deadlocks* e utilizar o algoritmo do banqueiro, pois era esse o grande problema desse algoritmo.

Pode-se denominar essa vacina como o algoritmo do tudo ou nada, pois o processo possuiria todos os recursos que necessitará durante sua execução, ou não possuirá nenhum.

Enquanto os recursos não estão disponíveis em sua totalidade, o processo ficará inativo. Desse modo pode ocorrer que determinado processo nunca entre em execução.

16.4.3 Ataque à não-preempção

A vacina tem por idéia tomar os recursos alocados a processos bloqueados a força. Isso pode causar alguns problemas aos processos, contudo é a maneira de se combater esse causa de *deadlock* evitando assim esse problema ao sistema.

16.4.4 Ataque ao problema da condição de espera circular

A idéia da vacina é rotular todos os recursos com números e fazer com que cada processo tome posse de um recurso por vez de forma crescente. Sendo assim, quando um processo já possuir recursos, ele não poderá tomar posse de recursos com numeração inferior ao recurso que ele já detém. Caso ele necessite alocar um recurso de numeração inferior, ele deverá liberar o recurso que detém, tentar alocar o recurso inferior e depois tentar alocar o recurso que ele já tinha novamente. Dessa forma o algoritmo evita a criação de círculos de processos e recursos prevenindo a ocorrência de *deadlocks*.

17 *Outros Aspectos*

17.1 Bloqueio em Duas Fases

Ex: Base de dados distribuída.

1ª fase: bloqueio dos registros que serão utilizados;

2ª fase: atualização dos registros.

O processo irá tentar alocar todas as bases para si, e só depois atualizar os registros.

Limitação: só funciona para programas que podem parar em qualquer momento da primeira fase, pois nem todos os processos podem ficar tentando alocar recursos.

17.2 Preterição Indefinida

Imagine que o sistema dê maior prioridade para processos que irão utilizar os recursos por um tempo menor, dessa forma toda vez que um processo possua um arquivo menor do que outro da fila, ele passe a sua frente.

Problema: Pode ocorrer de um processo nunca utilizar o recurso, uma vez que este irá demorar muito com o recurso.

A preterição indefinida também é um tipo de *starvation*. Imagine o caso de processos querendo imprimir e que um processo que queira imprimir um arquivo gigantesco fique na espera infinitamente, uma vez que outros processos detendo arquivos menores passe a sua frente sempre.

Referências

Tanenbaum, A. S. Sistemas Operacionais Modernos. 2ª ed, Prentice Hall.