

Sistemas Operacionais

Notas de Aula

Francisco Rapchan
rapchan@writeme.com
<http://www.inf.ufes.br/~rapchan>

Capítulo 2 – Processos

2.1 Introdução

Os sistemas operacionais multitarefa dão ao usuário a ilusão do paralelismo ou o “pseudoparalelismo” por causa da forma como alocam o processador um pouco para cada processo.

Em alguns sistemas operacionais, processos são chamados de tarefas (*tasks*).

Modelo do Processo

Um processo é: um programa em execução, incluindo os valores correntes de todos os registradores do hardware, e das variáveis manipuladas por ele no curso de sua execução.

O tempo de execução de um dado processo **não é constante**.

A diferença entre um **processo e um programa** é sutil, porém crucial.

Analogia do Eng. Cozinheiro (Tanenbaum pag21):

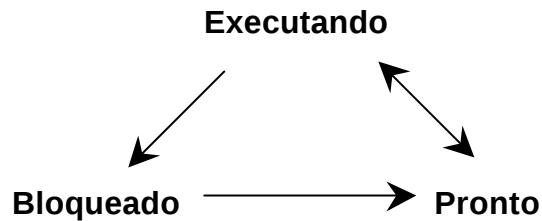
- Receita é o programa
- Cientista é o processador
- Processo é a atividade resultante da preparação do bolo: leitura da receita, busca dos ingredientes...
- Cuidar da filha picada por abelha é um processo de maior prioridade
- Deve salvar o “status” do processo anterior antes de mudar.

Processo possui: um programa, uma entrada, uma saída e um estado.

Um único processador pode ser compartilhado entre vários processos através de um algoritmo de **escalonamento**.

No UNIX, processos são criados pela chamada de sistema **FORK**. Este *system call* cria um processo idêntico ao original.

Há **3 estados de processo**: Rodando, Bloqueado e Pronto.



Há **4 transições** possíveis [Ver Tanenbaum figura 2.2 pag22] :

1. *Rodando para Bloqueado: para aguardar entrada de dados*
2. *Rodando para Pronto: quando o escalonador escolhe um novo processo para ocupar o processador.*
3. *Pronto para Rodando: quando o escalonador entrega o processador para o processo*
4. *Bloqueado para Pronto: quando a entrada de dados for concluída*

[Ver Tanenbaum figura 2.3 pag23.]

Implementação de Processos

A **tabela de processos** contém informações sobre: estado do processo, memória alocada, valor do contador de programa (PC) e do apontador de pilha (SP), arquivos abertos, prioridade, etc.

No UNIX a tabela de processos é chamada de **PCB – Process Control Block**.

O chaveamento entre processos se dá através de uma interrupção de hardware. O **vetor de interrupções** mantém uma relação entre o número de uma interrupção e o endereço na memória da rotina que trata desta interrupção.

[Ver Tanenbaum figura 2.5 pag24 sobre o que acontece em uma interrupção]

O UNIX prioriza a execução de processos que fazem muita entrada/saída (**I/O Bound**) em detrimento daqueles que consomem muito processador (**CPU Bound**).

2.2 Comunicação entre processos (IPC)

IPC – *InterProcess Communication*

Exemplo do diretório de spool: Quando um processo deseja imprimir um arquivo, ele entra com o nome do arquivo em um diretório especial de spool. Há duas variáveis compartilhadas controlando o spool: *out* e *in*. O problema começa com as variáveis compartilhadas - a lei de Murphy pode acontecer ☹...

Condições de corrida: situações onde dois ou mais processos estão acessando dados compartilhados, e o resultado final do processamento depende da ordem de acesso a estes dados.

A forma de evitar condições de corrida é encontrar uma forma de implementar a **exclusão mútua** de execução.

A condição de corrida do exemplo do spool acontece porque um processo começou a usar um dado compartilhado antes que o outro processo terminasse de usar o mesmo dado. Chamamos esta parte do programa que pode levar à ocorrência de condições de corrida de **região crítica**.

Quatro condições para termos multitarefa e evitarmos condições de corrida:

1. *Dois ou mais processo não podem estar simultaneamente dentro de suas regiões críticas correspondentes.*
2. *Nenhuma consideração pode ser feita a respeito da velocidade relativa dos processos, ou a respeito do número de processadores disponíveis no sistema.*
3. *Nenhum processo que esteja rodando fora de sua região crítica pode bloquear a execução de outro processo.*
4. *Nenhum processo pode ser obrigado a esperar indefinidamente para entrar em sua região crítica.*

Exclusão mútua com espera ocupada (*busy wait*)

Há várias soluções para a obtenção da exclusão mútua. O livro começa apresentando 3 soluções que não funcionam muito bem:

- **Inibição das interrupções.** Além de não devermos permitir que processos do modo usuário inibam as interrupções do sistema esta solução não funciona em sistemas com mais de um processador.
- **Variáveis de travamento.** Esta “solução” só posterga o problema.
- **Estrita alternância.**

A estrita alternância é uma solução classificada de **exclusão mútua com espera ocupada**. Nestas soluções o processo é colocado para executar um loop até que seja permitido entrar na região crítica.

O livro apresenta 2 soluções corretas baseadas em espera ocupada:

- **Solução de Peterson.**
- **Instrução TSL.**

Solução de Peterson

Criada em 1981, é baseada em duas rotinas:

- **enter_region** que deve ser chamada antes de entrar na região crítica
- **leave_region** que deve ser chamada após sair da região crítica.

```
#include "prototypes.h"
#define FALSE    0
#define TRUE     1
#define N        2          /* number of processes */

int turn;                  /* whose turn is it? */
int interested[N];         /* all values initially FALSE */

void enter_region (int process) /* process: who is entering (0 or 1) */
{
    int other;              /* number of other processes */

    other = 1 - process;    /* the other process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;         /* set flag */
    while (turn == process && interested[other] == TRUE) /* null */ ;
}

void leave_region (int process) /* process: who is leaving (0 or 1) */
{
    interested[process] = FALSE; /* indicate departure from critical
section */
}
```

Instrução TEST-AND-SET (TSL)

Muitos microprocessadores projetados para suportar múltiplos processadores possuem uma instrução básica tipo **TSL**.

Esta instrução executa a seguinte **operação atômica**: transfere o conteúdo de um endereço de memória para um registrador, e depois armazena um valor não nulo em tal endereço.

```
enter_region:
    tsl  reg, flag          ; copy flag to reg and set flag to 1
    cmp  reg, #0            ; was flag zero?
    jnz  enter_region       ; if it was non-zero, lock was set; loop
    ret                    ; return; critical region entered

leave_region:
    mov  flag, #0           ; store zero in flag
    ret                    ; return
```

O TSL funciona inclusive para sistema com mais de um processador porque o barramento de acesso à memória é bloqueado (lock) durante a instrução.

Nem todos os processadores possuem a instrução *Test_and_Set*, porém esta instrução pode ser implementada por *software* da seguinte forma:

```
enter_region:
    CLI                ; Desabilita interrupções
    MOV    registro, flag ; copia flag para registro
    MOV    flag,1        ; Faz flag = 1
    STI                ; Habilita interrupções
    CMP    registro,0     ; Registro = 0 ?
    JNZ    enter_region   ; Se não for zero, reg.Crítica ocupada
    RET

leave_region:
    MOV    flag,0
    RET
```

Bloqueio e Desbloqueio de Processos

Tanto a solução de Peterson quanto a baseada na instrução TSL além de usar a espera ocupada, podem gerar uma situação conhecido como o **problema da prioridade invertida**.

Como o **loop de espera ocupada deixa o processo como Pronto para rodar**, se um processo A de alta prioridade (sempre que estiver pronto ele é colocado para rodar) entrar em uma espera pela região crítica que esteja sendo ocupada um processo B de baixa prioridade, B nunca terá a oportunidade de sair da região crítica e A vai permanecer eternamente no loop de espera ocupada.

A solução é criar formas de bloquear o processo para que ele não fique ocupando o processador em um estado de pronto. Uma forma é criar duas primitivas:

- **SLEEP ()** : é um system call que bloqueia o processo que a chamou.
- **WAKEUP (processo)**: é um system call que “acorda” um processo.

O problema do Produtor-Consumidor

Neste problema um processo (**produtor**) coloca informação em um buffer limitado ao mesmo tempo em que outro retira informação do buffer (**consumidor**).

[Ver Tanenbaum fig2.10 pag29]

Este programa não funciona adequadamente porque ocorre uma condição de corrida por causa do acesso a *count* ser irrestrito.

Exercício

Faça uma tabela com os eventos do Produtor e do Consumidor a partir da situação dada a partir do último parágrafo da página 29 até o fim do primeiro parágrafo da página 30.

Produtor	Consumidor
	Lê o valor de count e é chaveado (count == 0).
Incrementa o valor de count	
Como count==1, então Wakeup (consumidor)	
	Baseado no valor lido de count, Sleep()
“enche o buffer”	
Sleep()	

A essência deste problema é que um sinal enviado para acordar um processo que não está dormindo é perdido.

Passaremos a analisar uma série de soluções para este problema, tais como:

- Semáforos
- Contadores de eventos
- Monitores
- Troca de mensagens

Na realidade estaremos estudando estes conceitos e aplicando-os na solução dos problemas consumidor – fornecedor entretanto eles se aplicam a uma gama muito maior de problemas.

(Aula 4)

Semáforos

Em 1965 E. W. Dijkstra sugeriu o uso de um tipo de variável inteira para contar o número de sinais armazenados. Chamou este tipo de **Semáforo**.

Criou duas operações atômicas para serem aplicadas sobre o semáforo:

- **Down:** Se o valor do semáforo for maior do que Q , decrementa o valor e continua. Se o valor do semáforo for Q , o processo que executou o down é posto para dormir.
- **Up:** Incrementa o valor do semáforo. Se houver um ou mais processos dormindo em um semáforo, um processo é acordado (segundo algum critério).

Na figura 2.11 é apresentada a solução do problema produtor – consumidor usando semáforos.

[Ver Tanenbaum fig2.11 pag31]

Observe que são necessárias três variáveis:

- **mutex:** exclusão mútua de execução. Controla o acesso à região crítica. Semáforos aplicados ao problema da exclusão mútua são chamados **binários** ou **mutex** (*MUTual EXclusion*), por assumirem apenas os valores 0 e 1.
- **empty:** assegura que o produtor pare de rodar se o buffer estiver cheio
- **full:** assegura que o consumidor pare de rodar quando o buffer estiver vazio.

A seguir é mostrado uma implementação de Semáforo em linguagem Pascal usando inibição de interrupções.

```

TYPE Semáforo = Record
    Valor : Integer;
    Fila-Espera:(* Lista de pendências*);
end;
PROCEDURE Down (Var S:Semsforo) ;
Begin
    Desabilita_Interrupcao;
    IF (S.Valor=0)
        Then Coloca_processo_na_fila_de_espera
        Else S.Valor := S.Valor - 1;
    Habilita_Interrupcao;
end;
PROCEDURE UP (Vav S:Semsforo);
Begin
    Desabilita_Interrupcao;
    IF (Tem_processo_esperando)
        Then tira_processo_da_fila_de_espera
        Else S.Valor := S.Valor + 1;
    Habilita_Inteerrupcao;
end;

```

Exercício

Faça uma tabela com as variáveis mutex, empty e full. Suponha que $N = 2$ e que houve a seguinte sequência de eventos:

1. Produtor gera itens até encher o buffer e ir dormir
2. Consumidor consome todos os itens
3. Produtor gera um item, depois dá down(&empty) e down(&mutex) e vai para Pronto
4. Consumidor consome todos os itens
5. Produtor volta a Rodando.

Passo	Mutex	Empty	Full	Itens
1	1	2	0	{}
	0	1		{1}
	1		1	
	0	0		{1,2}
	1		2	
2	0		1	{1}
	1	1		
	0		0	{}
	1	2		

Passo	Mutex	Empty	Full	Itens
3	0	1		{3}
	1		1	
	0	0		
4	Dorme		0	
5				{3,4}
	1		1	

Exercício

Descrever o que acontece caso os dois DOWNS no código do consumidor tenham sua ordem trocada.

A variável mutex será decrementada antes de empty. Se o buffer estiver completo, o produtor será bloqueado com mutex igual a 0. Consequentemente, a próxima vez que o consumidor tentar acesso ao buffer, ele será bloqueado ocorrendo então deadlock.

Contadores de Evento

Outra solução é criar um tipo de variável chamado **contador de evento** onde são possíveis 3 operações:

- **Read (E)**: Retorna o valor corrente de E.
- **Advance (E)**: Incrementa E de uma unidade atômica.
- **Await (E, v)**: Espera até que E tenha um valor maior ou igual a v.

Monitores

Monitores foram inventados por Hoare e P Brinch-Hansen (1975) oferecem abstrações separadas para exclusão mútua e para sincronização. Monitores são uma abstração de alto nível combinando 3 características:

- Dados compartilhados
- Operações sobre esses dados
- Sincronização e escalonamento

Um monitor é um conjunto de procedimentos, variáveis e estruturas de dados, todas agrupadas juntas em um módulo especial. São parecidos com classes, mas com sincronização embutida no mecanismo.

Sua grande característica é que somente um processo pode estar ativo dentro do monitor em um dado instante de tempo.

Os monitores são uma construção fornecida pela linguagem e o compilador é que gera o código necessário para sua gerência. Normalmente os compiladores geram o código referente aos monitores baseados em semáforos binários.

[Ver Tanenbaum fig2.13 pag33 – A estrutura monitor]

Monitores introduziram uma estrutura especial chamada variável de condição sobre a qual as seguintes operações são definidas:

- **Wait (condição)**: libera o lock do monitor e coloca o processo para dormir. Quando o processo acorda, ele tenta readquirir o lock do monitor imediatamente.
- **Signal (condição)**: acorda um dos processo que estão esperando pela variável de condição. O processo que executa o signal sai do monitor imediatamente. Se mais de um processo estiver esperando, o escalonador determina quem será acordado.

Embora Wait e Signal sejam semelhantes a Sleep e Wakeup, eles não estão sujeitos a condições de corrida porque a exclusão mútua dos procedimentos do monitor é automática.

[Ver Tanenbaum fig2.14 pag34 – Solução do problema Produtor x Consumidor em uma ling. Like-Pascal]

Existem poucas linguagens que implementam monitores (Concurrent Euclid).

Conclusão: Os semáforos são soluções de muito baixo nível e os monitores precisam de uma linguagem específica para serem utilizados. Nenhum deles permite a troca de informações entre máquinas diferentes.

Troca de Mensagem

Troca de mensagem é um mecanismo de comunicação e sincronização entre processos, implementado pelo sistema operacional através de duas rotinas:

- **SEND** é responsável por enviar uma mensagem para um processo receptor.
Em C teríamos: `send (destino, &mensagem);`
- **RECEIVE** por receber uma mensagem de um processo transmissor
Em C teríamos: `receive (fonte, &mensagem);`

Os procedimentos SEND e RECEIVE, mesmo que não tendo suas execuções mutuamente exclusivas, permitem tanto a comunicação entre processos como a sincronização entre eles.

- **A comunicação** ocorre porque um processo, ao receber uma mensagem, obtém dados enviados de outro processo.
- **A sincronização** é obtida porque uma mensagem somente pode ser lida após ter sido enviada, restringindo assim, a ordem na qual os dois eventos devem ocorrer.

Para garantir que uma mensagem não se perca, o processo receptor, ao recebê-la, deve enviar ao processo transmissor uma mensagem de recebimento (**acknowledgement** - ACK).

A comunicação entre processos pode ser feita de duas formas:

- **Endereçamento direto:** basta que o processo que deseja enviar ou receber uma mensagem enderece explicitamente o nome do processo receptor ou transmissor.
- **Endereçamento indireto:** utiliza uma área compartilhada, onde as mensagens podem ser colocadas pelo processo transmissor e retiradas pelo receptor. Esse tipo de *Buffer* é conhecido como **mailbox**. Vários processos podem estar associados a *mailbox*, e os parâmetros dos procedimentos SEND e RECEIVE passam a ser nomes de *mailboxes* e não mais de processos.

Basicamente, existem **duas formas de comunicação** entre processos através de troca de mensagens:

- **A comunicação é síncrona** quando um processo envia uma mensagem e fica esperando até que o processo receptor leia a mensagem, ou quando um processo tenta receber uma mensagem e permanece esperando até que o processo transmissor envie alguma mensagem. Este tipo de comunicação dispensa a necessidade de *buffers*, porém a execução dos processos fica limitada ao tempo de processamento das mensagens.
- **Na comunicação assíncrona**, nem o receptor permanece aguardando envio de uma mensagem, nem o transmissor o seu recebimento. Nesse caso, além da necessidade de *buffers* para armazenar as mensagens, deve haver outros mecanismos de sincronização que permitam ao processo identificar se uma mensagem já foi enviada ou recebida.

A grande vantagem da comunicação assíncrona é o maior paralelismo na execução dos processos, comparado a comunicação síncrona, onde a necessidade de espera dos processos reduz o grau de concorrência.

[Ver Tanenbaum fig2.15 pag36 – Produtor x Consumidor com troca de mensagens]

Equivalência entre as primitivas

Semáforos para implementar Monitores e Troca de Menagens

Usando semáforos para implementar monitores

(Tanenbaum, Fig. 2-16)

- Associar um semáforo binário, mutex, inicialmente igual a 1, a cada monitor
- Associar um semáforo binário, inicialmente igual a 0, a cada variável de condição

Usando semáforos para implementar troca de mensagens

- associar um semáforo, inicialmente igual a 0, a cada processo; o processo bloqueia nesse semáforo quando um send ou
- receive tem que esperar para completar uma área de buffer compartilhada é utilizada para implementar os mailboxes ; cada mailbox contém um array de mensagens (ver Fig. 2-17)
- buffer compartilhado é protegido por um semáforo binário (mutex)

Funcionamento:

receive em um mailbox com pelo menos uma posição ocupada:

- retira a mensagem, atualiza os contadores e apontadores (operações protegidas por mutex)
- se houver processo esperando na fila de destino do mailbox (mailbox estava cheio), remove o processo da fila e executa um up no seu semáforo
- receptor sai da região crítica, permitindo o remetente recém-acordado a continuar

receive em um mailbox vazio:

- processo que está tentando receber é inserido na fila de recebimento do mailbox
- executa um up em mutex
- executa um down no seu semáforo (vai dormir)
- mais tarde, quando acordado, executa um down em mutex

send em um mailbox com pelo menos uma posição vazia:

- insere a mensagem, atualiza os contadores e apontadores (operações protegidas por mutex)
- se houver processo esperando na fila de recebimento do mailbox (mailbox estava vazio), remove o processo da fila e executa um up no seu semáforo
- remetente sai da região crítica, permitindo o destinatário recém-acordado a continuar

send em um mailbox cheio:

- processo que está tentando enviar é inserido na fila de destino do mailbox
- executa um up em mutex
- executa um down no seu semáforo (vai dormir)
- mais tarde, quando acordado, executa um down em mutex

Monitores para implementar Semáforos e Troca de Mensagens

(Ver no livro)

Troca de Mensagens para implementar Semáforos e Monitores

(Ver no livro)

2.3 Problemas clássicos de comunicação entre processos

- Filósofos Glutões
- Leitores e Escritores
- Barbeiro Dorminhoco

2.4 Escalonamento de Processos

Critérios que devem estar presentes em um bom algoritmo de escalonamento:

1. **Justiça:** (fairness) todos os processos devem ter chances iguais de uso do processador.
2. **Eficiência:** manter o processador ocupado 100% do tempo.
3. **Tempo de resposta:** minimizar o tempo de resposta para usuários interativos.
4. **Turnaround:** minimizar o tempo que os usuários batch devem esperar pela saída.
5. **Throughput:** maximizar o número de jobs processados na unidade de tempo.

Kleinroch (1975) mostrou que qualquer algoritmo de escalonamento que favoreça determinada classe de jobs prejudica uma outra.

A cada **quantum** (ou time-slice) de tempo, é gerada uma interrupção que faz com que o escalonador seja chamado para decidir qual processo será o próximo a rodar.

Os escalonadores podem fazer multitarefa **preeptiva** ou **não preemptiva**.

2.4.1 Escalonamento Round Robin (circular)+

- É um dos mais antigos e mais fáceis de implementar.
- Assume que todos os processos têm a mesma prioridade.
- O escalonador mantém uma **fila de processos prontos** para rodar. Após um processo ser executado durante um quantum de tempo, ele é posto no final da fila.
- A **troca de contexto** entre o processo e o escalonador e novamente entre o escalonador e o processo que receberá o quantum obviamente demanda algum tempo.
- A relação entre este tempo e o quantum do processo define a porcentagem do **overhead** gasto em tarefas administrativas do sistema.

2.4.2 Escalonamento com prioridade

- Assume que cada processo tem uma prioridade associada.
- Normalmente há uma fila de processos associada a cada prioridade.
- Dentro de uma fila em uma dada prioridade pode-se seguir o round-robin.
- Pode-se seguir alguma estratégia para impedir que processos com menor prioridade fiquem eternamente esperando para rodar. Exemplo: dar oportunidade para 4 processos de prioridade 4, 3 de prioridade 3, 2 de prioridade 2 e 1 de prioridade 1...
- Em um sistema I/O bound, pode-se associar a cada processo uma prioridade $1/f$, sendo f a fração do quantum que lê usou em sua última rodada antes de ir para bloqueado.

2.4.3 Escalonamento por filas múltiplas

Uma variação do escalonamento com prioridade é associar a cada fila de prioridade, um tamanho diferente do quantum de tempo dos processos. Assim, processos com maior prioridade teriam um quantum maior enquanto processos com menor prioridade teriam um quantum menor.

Para sistemas com trocas de contexto muito lentas ou que necessitem fazer muito swapping, há outra variação. A idéia consiste em alterar o quantum de tempo a cada vez que o processo é posto para rodar. Na primeira vez ele recebe 1 quantum, na segunda 2 na terceira 4, na quinta 8 e assim por diante.

[Ler terceiro parágrafo pag47, item 2.4.3 “Pode ocorrer que tenhamos um processo...”]

2.4.4 Menor Job Primeiro (Shortest-Job First - SJF)

- Esta técnica visa **diminuir o tempo de turnaround** (espera média), processando os jobs menores primeiro.
- SJF é tipicamente aplicada a sistema batch.
- Deve-se ter um conhecimento prévio do tempo aproximado de execução do job.

[Ver exemplo Fig2.26 pag47]

2.4.5 Escalonamento garantido

- É uma forma completamente diferente de tratar a questão do escalonamento.
- A idéia é **garantir uma certa performance média** a cada usuário. Exemplo: $1/(\text{Numero de usuários})$ da capacidade do processador.
- Isto é feito mantendo-se um histórico de utilização da CPU para cada processo.
- Se um processo utilizou menos que sua parte, sua prioridade é aumentada, se utilizou mais, sua prioridade é diminuída.

2.4.6 Política versus Mecanismo

Separar o mecanismo de escalonamento da política de escalonamento permite aos processos que indiquem ao escalonador parâmetros sobre seus processos filhos.

Por exemplo, suponha que o kernel use um algoritmo de escalonamento com prioridades, mas possua um chamada de sistema através da qual o processo possa informar as prioridades de cada um de seus filhos.

2.4.7 Escalonamento em dois níveis

Em situações onde há um grande número de processos, alguns podem ter que ficar esperando em disco enquanto os que estão na memória principal são executados.

A idéia é fazer dois escalonadores de níveis diferentes:

- um escalonador de baixo nível fazendo escalonamento entre os processos que estão na memória principal, passando-os de pronto para rodando e vice versa usando algum dos algoritmos já vistos (round-robin, prioridade...).
- um escalonador de alto nível movendo os processos entre a memória principal e o disco.

[Ver Critérios dos escalonador de alto nível pag49]