

## CAPÍTULO II

# PROCESSOS

### 1. INTRODUÇÃO

No método tradicional de S.O., considera-se todo o software como organizado em um número de processos sequenciais ou, mais simplesmente, processos.

### 2. DEFINIÇÃO

- Define-se um processo como um programa em execução, sendo que para sua especificação completa, deve-se incluir tanto o programa propriamente dito como os valores de variáveis, registradores, contador de programa (Program Counter, PC), e outros dados necessários à definição completa de seu estado.

### 3. MODELO DE UM PROCESSO

Cada processo trabalha como se possuísse para si uma UCP (unidade central de processamento, o processador principal do computador) própria, chamada UCP virtual. Na realidade, na grande maioria dos casos uma única UCP é compartilhada entre todos os processos. Isto é, existe apenas uma UCP real, mas tantas UCP virtuais quantos forem os processos. Alguns sistemas de multiprocessamento<sup>1</sup> apresentam diversas UCP reais, entretanto mesmo nestes casos ocorre a necessidade de compartilhamento de várias ou todas UCP pelos diversos processos.

Os processos, em uma mesma UCP real executam um por vez, mas essa execução é realizada de forma a criar a ilusão de que os mesmos estão executando em paralelo, conforme a Figura 2.1. apresentada abaixo, onde é mostrada a execução na UCP real, a aparência criada ao usuário, e a distribuição de tempo da UCP real entre os diversos processos.

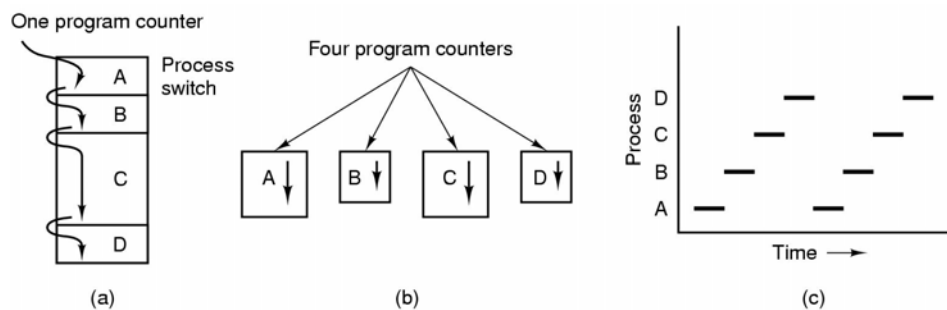


Figura 2.1 - (a) Multiprogramação de quatro processos. (b) Modelo conceitual de quatro processos independentes. (c) Execução real de 4 processos – o que pode ser comparado com uma monitora de creche dando de comer a quatro crianças.

Um fator derivado do método como a implementação das UCP virtuais é realizada, e que apresenta fundamental importância no desenvolvimento de sistemas

<sup>1</sup> corresponde a diversos processadores, dentro de um mesmo sistema de computação, executando programas diversos ou cooperando na execução de um mesmo programa.

operacionais, é o fato de que a taxa de execução de cada processo não é uniforme nem reproduzível. Com isto, quer-se dizer que não é possível assumir nada com relação à taxa com que cada processo será executado, nem em relação a outros processos, nem em relação a diversas execuções de um mesmo processo. Uma das implicações disto é que os programas não podem ser feitos levando em consideração as temporizações de execução das instruções (p.ex.: não se pode fazer um programa que execute certo número de repetições de um "loop" e com isso espere conseguir uma demora fixa de tempo, como um segundo). Isto ocorre porque não se sabe o momento em que a UCP será chaveada para outro processo, fazendo com que o atual tenha sua continuação retardada.

#### **4. DIFERENÇA ENTRE PROCESSO E PROGRAMA**

A diferença entre processo e processo e programa é sutil, mas crítica. Então para fixar melhor a diferença entre um programa e um processo, uma analogia pode ajudar. Suponha um padeiro não muito experiente. Para fazer um pão ele se utiliza de alguns ingredientes (farinha, sal, água, bromato, etc.) e segue a receita de um livro de receitas. Neste caso poder-se-ia estabelecer a seguinte comparação: O padeiro é a UCP, os ingredientes são as entradas e a receita é o programa (desde que o padeiro entenda tudo o que está escrito na receita, isto é, que a mesma esteja em uma linguagem apropriada). O processo neste caso corresponde ao ato de o padeiro estar executando o pão da receita com as entradas disponíveis. Suponha agora que, enquanto o padeiro está amassando o pão (um dos passos indicados pela receita) ele é picado por uma abelha. Como não é muito forte, ele começa a chorar e julga extremamente importante cuidar da picada de abelha antes de acabar o pão. Para isto ele dispõe de novas entradas (os medicamentos) e de um novo programa (o livro de primeiros socorros). Terminado o curativo da picada o padeiro volta a amassar o pão no mesmo ponto em que parou. Aqui tem-se uma interrupção (a picada de abelha) fazendo a UCP (padeiro) chavear do processo de preparação do pão para o processo de curativo da picada. Outro fato importante aqui é que o processo de preparação do pão deve ser guardado com todo o seu estado corrente (isto é, em que ponto da receita ele está e qual a disposição de todos os componentes da massa e seu estado atual) para que possa ser reassumido no mesmo ponto em que foi interrompido.

#### **5. HIERARQUIA DE PROCESSOS**

Em todos os S.O. de multiprogramação deve haver alguma forma de criar e terminar processos conforme o necessário à execução dos pedidos de usuários. Por isso, um processo tem a possibilidade de gerar outros processos, que por sua vez podem gerar outros, e assim sucessivamente, gerando uma hierarquia de processos. Na geração de um novo processo, o processo gerado recebe o nome de filho, e o que pediu a geração recebe o nome de pai. Um pai pode apresentar diversos filhos, mas cada filho só pode ter um pai.

#### **6. ESTADOS DO PROCESSO**

Durante a sua existência, os processos podem se apresentar, do ponto de vista do sistema, em diferentes estados. Serão apresentados os três mais importantes e os fatos que levam os processos a mudar de um estado à outro. O primeiro estado a considerar consiste naquele em que um processo está efetivamente executando, isto é, está

rodando. Durante a sua existência, um processo pode necessitar interagir com outros processos. Por exemplo, um processo pode gerar uma saída que será utilizada por outro.

Outro estado a considerar é quando o processo tem todas as condições de executar, mas não pode pois a UCP foi alocada para a execução de um outro processo. Neste caso o processo está pronto.

Na Figura 2.2 se apresenta os estados de um processo e as transições entre cada um de eles.

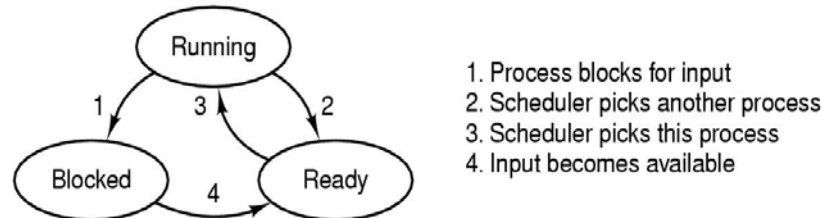


Figura 2.2 – Estados de um processo.

Onde: as transições entre os processo estão numeradas de acordo com a anotação:

1. bloqueamento por falta de entrada
2. escalonador selecionou um outro processo
3. escalonador selecionou este processo
4. entrada ficou disponível

Em alguns sistemas, o processo deve requisitar o bloqueamento, quando notar que não dispõe de entradas. Em outros sistemas, o bloqueamento é realizado automaticamente pelo próprio sistema, sem que o usuário precise se ocupar disso durante a programação. O escalonador (scheduler) citado acima é uma parte do S.O. responsável pelo chaveamento da UCP entre os diversos processos, tendo como principal objetivo a conciliação da necessidade de eficiência do sistema como um todo e de justiça para com os processos individuais.

## 7. IMPLEMENTAÇÃO DE PROCESSOS

Para a implementação de processos, além dos programas à serem executados, devem ser guardadas algumas informações, necessárias ao escalonador, para permitir que um processo seja reassumido exatamente no ponto em que foi interrompido. Esses dados são armazenados na chamada tabela de processos, que consiste em um vetor de estruturas comum a entrada por processo. Esta estrutura contém dados como: contador de programa (PC), ponteiro de pilha (SP), estado dos arquivos abertos, registradores, além de diversas outras informações.

# COMUNICAÇÃO ENTRE PROCESSOS

## 1. CONDIÇÕES DE DISPUTA

Quando existe compartilhamento entre processos de uma memória, na qual cada um pode escrever ou ler, podem ocorrer as chamadas condições de disputa. Exemplifica-se isto através de um *spooler* de impressora (que corresponde a um programa que permite

diversos usuários utilizarem uma mesma impressora, simplesmente fazendo com que cada um envie o nome de um arquivo a ser impresso, e imprimindo os arquivos na ordem em que foram solicitados pelos diversos usuários).

Considera-se que o programa funciona lendo os nomes dos arquivos a imprimir de um diretório de *spool*. Cada programa que deseja a impressão de um arquivo envia o nome do mesmo para o diretório de *spool*.

Considera-se o diretório de *spool* como organizado em diversas unidades numeradas, sendo que cada unidade pode conter o nome de um arquivo, veja na Figura 2.3.

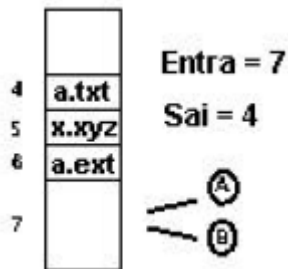


Figura 2.3 Condição de disputa

Tem-se também duas variáveis, **entra** e **sai** que são utilizadas respectivamente para indicar qual a próxima unidade de diretório vazia (e que portanto pode ser ocupada por um novo nome de arquivo) e qual a próxima unidade a ser impressa.

Supondo que, no caso indicado na Figura 2.3, onde **entra=7**, um processo **A** resolve requisitar a impressão de um arquivo chamado **a.prg**. Para isto, ele precisa ler o valor da variável **entra**, colocar o nome do arquivo na unidade correspondente e incrementar o valor de **entra** de um. Supondo então que o processo **A** leu **entra=7**, após o que foi interrompido pelo escalonador, que determinou o início da execução do processo **B**. Este processo (**B**), resolve então imprimir o arquivo **b.prg**. Para isto lê o valor de **entra**, onde encontra 7, coloca **b.prg** na unidade 7 do diretório de *spool*, e incrementa **entra** para 8. Após isto o escalonador determina a volta da execução do processo **A**, que retorna exatamente no ponto onde estava.

Então, para o processo **A**, o valor de **entra** continua sendo 7 e, portanto, ele colocar **a.prg** na unidade 7 do diretório de *spool* (exatamente onde tinha sido colocado **b.prg**) e incrementa o valor de **entra** que leu (isto é , 7) gerando como resultado 8, que será armazenado em **entra**.

O resultado de tudo isto é que o pedido de impressão de **b.prg** desapareceu, e portanto o processo que o requisitou não será servido.

Isto é o que se chama de uma condição de disputa.

## 2. SEÇÕES CRÍTICAS

Claramente as condições de disputa, como a apresentada acima, devem ser evitadas, e os S.O. devem ser construídos de forma a evitar disputas entre os diversos processos. Note que o problema de disputa ocorreu, no programa de *spool* da impressora, devido ao fato de existirem dois processos acessando "simultaneamente" os dados compartilhados entre os mesmos. Isto nos indica que as condições de disputa podem ser evitadas, proibindo que mais de um processo leia e escreva simultaneamente em uma área de dados compartilhada. Isto é o que se chama de **exclusão mútua**.

Durante a maior parte do tempo um processo executa computações internas, que não requerem acesso a dados de outros processos. Os trechos de programa em que os processos estão executando computações sobre dados compartilhados com outros processos são chamados de **seções críticas**. Para evitar disputas basta então garantir que não haverá dois processos simultaneamente em suas seções críticas.

Além disto, para haver uma cooperação eficiente e correta entre os processos, deve-se satisfazer as seguintes condições:

1. não podem haver dois processos simultaneamente em suas seções críticas;
2. não são feitas suposições sobre a velocidade relativa dos processos e sobre o número de UCPs;
3. nenhum processo parado fora de sua região crítica pode bloquear outros processos;
4. nenhum processo deve esperar um tempo arbitrariamente longo para entrar em sua região crítica.

### **3. EXCLUSÃO MÚTUA COM ESPERA OCUPADA**

#### **3.1. Desabilitando interrupções**

A forma mais simples de garantir a exclusão mútua é fazer com que cada processo desabilite interrupções ao entrar na região crítica, e as reabilite imediatamente antes de sair.

Isto impede que a UCP seja chaveada para outro processo, pois o chaveamento é realizado através de uma interrupção periódica vinda de um relógio que ativa o escalonador.

Esta solução apresenta os seguintes problemas:

1. os usuários devem ter o direito de desabilitar interrupções, o que significa que se algum se esquecer de reabilitá-las, o S.O. não poderá mais executar;
2. se o computador possuir várias UCP o método não funciona, pois somente serão desabilitadas as interrupções da UCP que estiver rodando o programa.

Daí conclui-se que a desabilitação de interrupções, necessária a algumas tarefas do *kernel*, deve ser restrita ao mesmo.

#### **3.2. Variáveis de Comporta**

Pode-se pensar numa solução em que se estabeleça uma variável auxiliar, denominada variável de comporta (lock variable), que quando em 0 indica que a região crítica está livre, e quando em 1 indica que a mesma está ocupada.

Desta forma, é possível fazer com que cada processo, antes de entrar, teste o valor da comporta: se for 0 (aberta), coloca em 1 e prossegue o processamento, colocando em 0 quando terminar, e se for 1 (fechada) aguarda até se tornar 0.

O grande problema com esta solução é que a disputa apenas se transferiu da região crítica para a variável de comporta (pense no que ocorre se um processo for interrompido imediatamente depois de ler o valor da variável e antes de alterá-lo...).

#### **3.3. Alternância Estrita**

Esta é uma solução que obriga que a região crítica seja dada a um dos processos por vez, em uma estrita alternância. O algoritmo apresentado abaixo representa um exemplo disto, com a utilização de uma variável **vez**, que indica de qual processo é a vez de entrar na região crítica:

```
procedure proc_0;  
begin  
    while (TRUE)
```

```

do begin
    while (vez ≠ 0) (*espera*) do;
    secao_critica;
    vez := 1;
    secao_normal;
end
end;
procedure proc_1;
begin
    while (TRUE)
    do begin
        while (vez ≠ 1) (*espera*) do;
        secao_critica;
        vez := 0;
        secao_normal;
    end
end;
end;

```

O teste contínuo de uma variável na espera de um certo valor é chamado de espera ocupada, e representa, como é evidente, um enorme desperdício de UCP.

O problema com a solução de alternância estrita é que requer que os dois processos se alternem precisamente, o que significa que o número de acessos de cada processo deve ser exatamente igual ao do outro. Além disto, existe uma violação da regra 3 apresentada acima, pois se um dos processos pára fora de sua seção crítica, o outro não poderá prosseguir normalmente, pois após entregar a seção crítica para o outro processo não pode mais pedi-la novamente.

#### 4. SLEEP e WAKEUP

As soluções com espera ocupada têm o grave inconveniente de desperdiçar tempo de UCP nos *loops* de espera para a entrada na região crítica. Além disto apresenta um outro problema quando se trata de processos com prioridades diferente. Suponha dois processos: um, chamado H, de alta prioridade e outro, chamado L de baixa prioridade. Se o processo L estiver executando em sua região crítica quando o processo H é selecionado para execução, então, se o processo H tenta entrar em sua região crítica não pode, pois o processo L está dentro da mesma e, portanto fica em um loop de espera. Mas, como H tem alta prioridade, o processo L não poder executar até que H termine, pois apresenta prioridade mais baixa.

Tem-se neste caso uma situação chamada de **deadlock** (impasse), onde nenhum dos processos pode prosseguir pois está aguardando alguma condição que somente pode ser atingida pela execução do outro.

Para isto, define-se duas rotinas SLEEP e WAKEUP, que realizam a espera através do bloqueamento do processo, ao invés do desperdício do tempo de UCP. SLEEP: faz com que o processo que está executando seja transferido do estado de rodando para o de bloqueado. WAKEUP: pega um processo em estado bloqueado e o transfere para o estado pronto, colocando-o disponível para execução quando o escalonador julgar adequado.

Para exemplificar a utilização de SLEEP e WAKEUP, observe o problema do produtor e do consumidor. Neste problema clássico existem dois processos: um chamado **produtor** que coloca dados em um buffer, e outro chamado **consumidor** que retira dados do buffer. O buffer apresenta uma capacidade finita de reter dados, de forma que surgem problemas em duas situações:

- i) quando o produtor deseja colocar mais dados em um buffer cheio;
- ii) quando o consumidor deseja retirar dados de um buffer vazio.

Em ambos os casos, faz-se com que o processo que não pode acessar o buffer no momento execute um SLEEP e seja bloqueado, somente sendo acordado quando o outro processo alterar a situação do buffer e executar WAKEUP.

No programa abaixo tem-se uma tentativa de resolver o problema do produtor-consumidor com a utilização de SLEEP e WAKEUP. A variável **cont** é compartilhada entre os dois processos:

```
cont:=0; (* inicialmente buffer vazio *)
```

```
procedure produtor;
begin
  while (TRUE)
  do begin
    produz_item(item); (* produz um item *)
    if (cont=N) then SLEEP; (* se buffer cheio, dorme *)
    entra_item(item); (* coloca item no buffer *)
    cont:= cont+1; (* incrementa n. de itens *)
    if (cont=1)
    then WAKEUP(consumidor); (*se buffer estava vazio, end acorda
                               consumidor *).
  end;
end;

procedure consumidor;
begin
  while (TRUE)
  do begin
    if (cont=0) then SLEEP; (* se buffer vazio, dorme *)
    remove_item(item); (* lê item do buffer *)
    cont:= cont-1; (* decrementa n. de itens *)
    if (cont=N-1)
    then WAKEUP(produtor); (* se buffer estava cheio, acorda produtor *)
    consome_item(item); (* utiliza o item lido *)
  end;
end;
```

Infelizmente este programa apresenta um problema na utilização da variável cont. Veja nos seguintes passos:

1. consumidor lê cont=0
2. escalonador interrompe consumidor e roda produtor

3. produtor coloca item e incrementa cont para 1
  4. como descobre que cont=1, chama WAKEUP para o consumidor
  5. consumidor volta a executar, mas como já estava rodando o WAKEUP é perdido
  6. para ele cont=0 (foi o valor lido) e portanto, dorme
  7. em algum momento o produtor lota o buffer e também dorme
- Tem-se portanto uma condição em que tanto o produtor como o consumidor estão dormindo, não podendo ser acordados pelos outros, o que caracteriza novamente um **deadlock**.

Para este caso simples, o problema pode ser resolvido com a inclusão de um bit de espera de WAKEUP. Assim, quando um WAKEUP for enviado para um processo já acordado, o bit será marcado, sendo que a próxima vez que o processo realizar SLEEP, se o bit estiver marcado então o processamento prossegue como se o WAKEUP tivesse sido automático. Entretanto, esta solução não resolve o problema geral. Pode-se, por exemplo, construir casos com três processos onde um bit de espera não é suficiente. O aumento para três ou mais bits poderia resolver esses casos, porém o problema geral ainda permaneceria.

## 5. SEMÁFOROS

Para resolver este problema, DIJKSTRA (1965) propôs a utilização de uma variável inteira que conta o número de WAKEUP realizados. A essa variável se deu o nome de **semáforo**.

Para trabalhar com os semáforos Dijkstra propôs duas primitivas:

P(v) : generalização de SLEEP, que pode ser definida como:

```

if (v>0)
then v:= v-1
else SLEEP;
```

V(v) : generalização de WAKEUP, que pode ser definida como:

```

if (v=0)
then WAKEUP
else v:= v+1;
```

Veja que, nas representações acima, são consideradas todas as ações de P(v) e V(v) como indivisíveis, isto é, o processamento é realizado sem interrupções.

A solução do produtor-consumidor com semáforos é apresentada abaixo, onde utilizamos três variáveis semáforo:

mutex: semáforo binário que garante a exclusão mútua dentro da região crítica  
vazio: que indica o número de posições vazias no buffer  
cheio : que indica o número de posições cheias no buffer.

```

amutex:= 1;  (* primeiro o produtor *)
vazio:= N;    (* N locais vazios *)
cheio:= 0;    (* 0 locais cheios *)
procedure produtor;
begin
  while (TRUE)
  do begin
    produz_item(item);
```



```

        P(vazio);      (* um vazio a menos *)
        P(mutex);      (* testa exclusao mutua *)
        entra_item(item); (* poe item no buffer *)
        V(mutex);      (* sai da exclusao mutua *)
        V(cheio);       (* um a mais cheio *)
    end
end;

procedure consumidor;
begin
    while (TRUE)
    do begin
        P(cheio);      (* um cheio a menos *)
        P(mutex);      (* testa exclusao mutua *)
        remove_item(item); (* le item do buffer *)
        V(mutex);      (* sai da exclusao mutua *)
        V(vazio);       (* um vazio a mais *)
        consome_item(item);
    end
end;

```

A melhor forma de ocultar uma interrupção é através da utilização de semáforos. Por exemplo, associamos um semáforo chamado disco à interrupção de disco, com o valor inicialmente de 0. Assim, quando um processo necessita de dados do disco executa um P(disco), ficando bloqueado. Ao surgir a interrupção de disco esta executa V(disco), desbloqueando o processo requisitante.

DIJKSTRA, E. W.; 1965. Cooperating sequential processes. Technical Report EWD-123, Technical University, Eindhoven, Holanda. Reproduzido em *Geniys* 1968, 43-112.

## 6. MONITORES

Os semáforos resolvem o problema de acesso às regiões críticas, entretanto apresentam grande dificuldade de utilização. Por exemplo, no código para o produtor-consumidor, se revertermos acidentalmente o P(vazio) com o P(mutex) no produtor, ocorrerá deadlock quando o buffer estiver cheio, pois o produtor já terá pedido a exclusão mútua (com a execução de P(mutex)) quando percebe que o buffer está cheio (com P(vazio)) e então para.

Mas então o consumidor não pode mais retirar itens do buffer pois não pode mais entrar na região crítica. Isto mostra que a programação com semáforos deve ser muito cuidadosa.

Para evitar essas dificuldades, Hoare e Brinch Hansen propuseram uma primitiva de sincronização de alto nível conhecida como monitor.

Em um monitor se agrupam conjuntamente rotinas, variáveis e estruturas de dados. Os monitores apresentam as seguintes características:

- os processos chamam rotinas no monitor quando desejam, mas não têm acesso a suas variáveis e estruturas de dados internas;

- apenas um processo pode estar ativo no monitor em cada instante;
- o monitor é reconhecido pelo compilador e, portanto, tratado de uma forma especial;
- as primeiras instruções de cada rotina do monitor realizam um teste para verificar se existe outro processo ativo no monitor no mesmo instante. Se houver, o processo chamante é suspenso, e se não houver o processamento prossegue;
- o compilador é o responsável pela implementação da exclusão mútua entre as rotinas do monitor (em geral através de um semáforo binário);
- quem escreve as rotinas do monitor não precisa saber como a exclusão mútua será implementada, bastando colocar todas as seções críticas em rotinas de monitor.

Para possibilitar o bloqueamento de processos, quando estes não podem prosseguir, introduziram-se as variáveis de condição, além das operações WAIT e SIGNAL. Quando uma rotina de monitor não pode continuar executa um WAIT em alguma variável de condição (por exemplo: cheio, quando o produtor encontra o buffer cheio).

Um outro processo então pode acordá-lo executando um signal na mesma variável de condição). Para evitar que dois processos estejam ativos no monitor simultaneamente, devemos ter uma regra que diga o que deve ocorrer após SIGNAL. Hoare propôs que o processo recém acordado fosse executado.

Brinch Hansen propôs que o processo que executa um SIGNAL deve deixar imediatamente o monitor (isto é, o signal deve ser a última operação realizada por uma rotina de monitor. Esta última é mais simples de implementar e conceitualmente mais fácil: se um SIGNAL é executado em uma variável de condição em que diversos processos estão esperando, apenas um deles (determinado pelo escalonador) é revivido.

O problema do produtor-consumidor com a utilização de monitor é apresentado pelo pseudocódigo abaixo, onde utilizamos um dialeto de PASCAL que supomos reconhecer os comandos de monitor.

**monitor** ProdutorConsumidor;

var cheio,vazio: boolean;

cont: integer

procedure coloca;

begin

if cont = N

then wait(cheio);

entra\_item;

cont := cont+1;

if count = 1 then signal(vazio);

end;

procedure retira;

begin

if cont=0

then wait(vazio);

remove\_item;

cont:=cont-1;

if cont=N-1

```

        then signal(cheio);
    end;
    cont:=0
end monitor;

procedure produtor;
begin
    while true
    do begin
        produz_item;
        ProdutorConsumidor.Coloca;
    end
end;

procedure consumidor;
begin
    while true
    do begin
        ProdutorConsumidor.retira;
        consome_item;
    end
end;

```

As operações WAIT e SIGNAL são muito parecidas com as de SLEEP e WAKEUP, que tinham problemas graves de disputa. No caso atual a disputa não ocorre, pois no monitor não podem haver duas rotinas simultaneamente.

Os monitores possibilitam uma grande facilidade para a programação paralela. Entretanto, possui alguns inconvenientes:

- a. são uma construção de alto nível, que deve ser reconhecida pelo compilador, isto é , a linguagem que os implementa deve reconhecer os comandos de monitor empregados (muito poucas linguagens de fato os implementam);
- b. eles se baseiam (tanto quanto os semáforos) na consideração da existência de uma memória comum compartilhada por todos os processadores. Portanto, são inaplicáveis para sistemas distribuídos, onde cada processador conta com sua própria e independente memória.

## 7. PASSAGEM DE MENSAGENS

Para possibilitar o trabalho com sistemas distribuídos e também aplicável à sistemas de memória compartilhada temos o conceito de passagem de mensagens. Este método usa as seguintes primitivas:

SEND : envia uma mensagem para um destino dado:

**SEND(destino,&mensagem);**

RECEIVE : recebe uma mensagem de uma origem especificada:

**RECEIVE(origem,&mensagem);**

Se nenhuma mensagem estiver disponível no momento de executar RECEIVE, o receptor pode bloquear até que uma chegue.

Se pensamos na conexão de processadores por uma rede, pode ocorrer de uma mensagem ser perdida. Para se resguardar disto o processo que envia e o que recebe podem concordar em que, tão logo uma mensagem seja captada pelo receptor, este envie uma outra mensagem acusando a recepção. Assim, se o processo que envia não receber a recepção dentro de certo tempo, pode concluir que a mensagem foi perdida e tenta a transmissão novamente. Estas mensagens de volta são mensagens especiais chamadas de mensagens de reconhecimento.

Suponha agora que a mensagem original é recebida corretamente, mas o reconhecimento é perdido. Neste caso, o enviador irá retransmitir a mensagem, e o receptor irá recebê-la duas vezes. Portanto, deve ser possível a distinção entre uma mensagem antiga retransmitida e uma nova mensagem, o que é normalmente realizado pela colocação de números consecutivos em cada mensagem original.

Sistemas de mensagens também precisam saber exatamente quais os nomes dos diversos processos, de forma a identificar corretamente a origem e o destino de uma mensagem.

Normalmente isto é conseguido com uma nomeação do tipo `processo@máquina`. Se o número de máquinas for muito grande e não houver uma autoridade central, então pode ocorrer que duas máquinas se dêem o mesmo nome. Os problemas de conflito podem ser grandemente diminuídos pelo agrupamento de máquinas em domínios, sendo o endereço formado por **processo@máquina.domínio**. Outro fator importante a considerar é o de autenticação, de forma a garantir que os dados estão sendo realmente recebidos do servidor requisitado, e não de um impostor, e também para um servidor saber com certeza qual o cliente que requisitou. Em geral a “criptação” das mensagens com alguma chave é útil.

Utilizando a passagem de mensagens e sem memória compartilhada, podemos resolver o problema do produtor-consumidor como apresentado no programa abaixo:

```
procedure produtor;
begin
    while (TRUE)
    do begin
        produz_item(item);
        RECEIVE(consumidor,m);  (* aguarda mensagem vazia *)
        faz_mensagem(m,item);   (* constroi mensagem *)
        SEND(consumidor,m);     (* envia item ao consumidor *)
    end
end;

procedure consumidor;
begin
    for i:=0 to N-1 do SEND(produtor,m);    (* envia N vazios *)
    while (TRUE)
    do begin
        RECEIVE(produtor,m);    (* pega mensagem *)
        extrai_item(m,item);     (* pega item da mensagem *)
        consome_item(item);
        SEND(produtor,m);       (* envia resposta vazia *)
    end
end;
```

end  
end;

Nesta solução assumimos que todas as mensagens têm o mesmo tamanho e que mensagens enviadas, mas ainda não recebidas, são automaticamente armazenadas pelo sistema operacional.

No início são enviados N vazios para permitir ao produtor encher todo o buffer antes de bloquear. Caso o buffer esteja cheio, o produtor bloqueia no RECEIVE e aguarda a chegada de uma nova mensagem vazia.

A passagem de mensagens pode ser feita de diversas formas, das quais citaremos três:

1. assinalando-se um endereço único para cada processo e fazendo as mensagens serem endereçadas aos processos;
2. com a utilização de mailbox, uma nova estrutura de dados que representa um local onde diversas mensagens podem ser bufferizadas. Neste caso, os endereços em SEND e RECEIVE são as mailbox e não os processos. Se um processo tenta enviar para uma mailbox cheia ou receber de uma mailbox vazia ele é automaticamente suspenso até que a mailbox apresente as condições para a operação;
3. eliminando toda a bufferização, fazendo com que, se o processo que envia executa SEND antes do de recepção, então, ele é bloqueado até que este último esteja pronto (e viceversa). Este mecanismo é conhecido como rendezvous.

## ESCALONAMENTO DE PROCESSOS

Chama-se de escalonamento (scheduling) de processos o ato de realizar o chaveamento dos processos ativos, de acordo com regras bem estabelecidas, de forma a que todos os processos tenham a sua chance de utilizar a UCP. O escalonador (scheduler) é, portanto, a parte do S.O. encarregada de decidir, entre todos os processos prontos para executar, qual o que será rodado em cada instante. Chama-se de algoritmo de escalonamento o método utilizado pelo escalonador para decidir, a cada instante, qual o próximo processo a ser executado.

Podemos estabelecer os seguintes critérios para um algoritmo de escalonamento:

- a. justiça: cada processo deve conseguir sua parte justa do tempo de UCP;
- b. eficiência: garantir uma ocupação de 100% do tempo da UCP;
- c. tempo de resposta: minimizar o tempo de resposta a comandos de usuários interativos;
- d. minimização do tempo que os usuários de lotes (batch) devem aguardar até conseguir a saída desses pedidos;
- e. maximizar o número de serviços (jobs) processados por hora.

Como pode ser facilmente notado, vários critérios acima entram em contradição entre si, de forma que devemos prejudicar alguns para conseguir melhoras em outros. O escalonador deve, portanto, realizar algum compromisso entre os diversos critérios. Para dificultar que os objetivos sejam alcançados, ainda temos o fato de que cada processo é único e imprevisível.

Com o objetivo de prevenir que um processo rode durante um tempo excessivo, a maioria dos computadores dispõe de um relógio, que é um dispositivo de temporização que interrompe a UCP a cada intervalo dado de tempo. A cada interrupção, o S.O. interrompe

a execução do processo atual e roda o escalonador para decidir qual o próximo processo a ser executado. Esta estratégia de interromper um processo em execução é chamada de escalonamento preemptivo (preemptive scheduling), e contrasta com a estratégia de execução até o término.

Veja agora algumas formas de implementar o escalonamento.

### 1. ESCALONAMENTO "ROUND-ROBIN"

Neste método, a cada processo se dá um intervalo de tempo, chamado de quantum, no qual se permite que ele execute. Se o processo ainda está rodando ao fim de seu quantum, a UCP é tomada deste processo e o escalonador seleciona um novo para rodar. A UCP também pode ser entregue a outro processo caso o processo bloqueie ou termine antes do fim do quantum. Neste caso, a UCP é entregue à outro processo no momento do término ou do bloqueio.

Para a implementação deste método, o escalonador mantém uma lista de processos executáveis (isto é, processos prontos). Quando um quantum termina sem o processo acabar, o mesmo é colocado no fim dessa lista. O escalonador seleciona, então, o primeiro processo dessa lista para execução. Veja a Figura 1 para melhor esclarecimento.

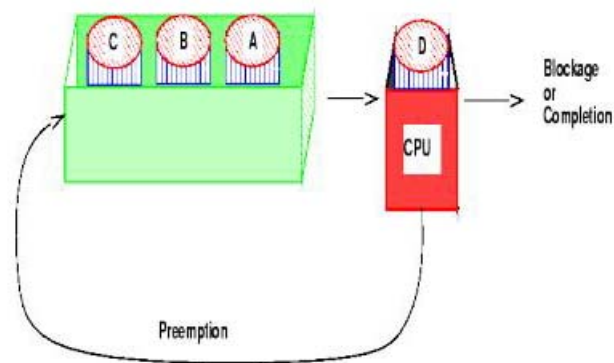


Figura 1. Esquema de funcionamento do escalonamento *Round-Robin*

O valor do quantum deve ser cuidadosamente escolhido. Isso ocorre porque existe o tempo de chaveamento (tempo para que o processo em execução seja interrompido; tenha seu estado atual salvo; o escalonador decida qual o próximo processo e o estado deste novo processo seja restaurado). Assim, se o valor do quantum for muito pequeno, teremos uma grande proporção do tempo de execução da UCP gasta com o processo de chaveamento, que não é de interesse para os usuários. Por exemplo, se todo o chaveamento de processos leva 5ms, e o quantum for de 20ms, teremos uma utilização de 20% do tempo total para o chaveamento.

Por outro lado, se o valor do quantum for muito grande, isto pode acarretar um tempo de resposta muito grande para usuários interativos. Veja por exemplo se, para um chaveamento de 5ms escolhemos um quantum de 500ms. Neste caso o tempo gasto com o chaveamento é menos que 1% do tempo total. Entretanto, se temos 10 processos executando, o tempo de resposta para um processo pode ser de até 5s. Devemos pois

escolher um valor de compromisso, nem muito grande nem muito pequeno, de acordo com o tempo de chaveamento.

## 2. ESCALONAMENTO COM PRIORIDADE

O esquema de *round-robin* trata igualmente todos os processos dos sistemas, sejam eles altamente urgentes e importantes, sejam eles de pouca importância ou sem urgência. Para poder oferecer um tratamento distinto à processos diversos, devemos introduzir o conceito de prioridade.

No instante da criação de um processo, associamos ao mesmo uma prioridade. Quando o escalonador tiver que escolher, entre os processos prontos, qual o que será executado em seguida, ele escolhe o de mais alta prioridade. Como esse esquema pode fazer com que um processo seja executado indefinidamente, não deixando espaço a outros processos, o escalonador pode, a cada vez que o processo é escalado para execução, decrementar sua prioridade. No momento em que sua prioridade fica abaixo de um outro processo pronto ele é interrompido e o outro processo (agora com prioridade mais alta) é executado. Tal como se observa na Figura 2.

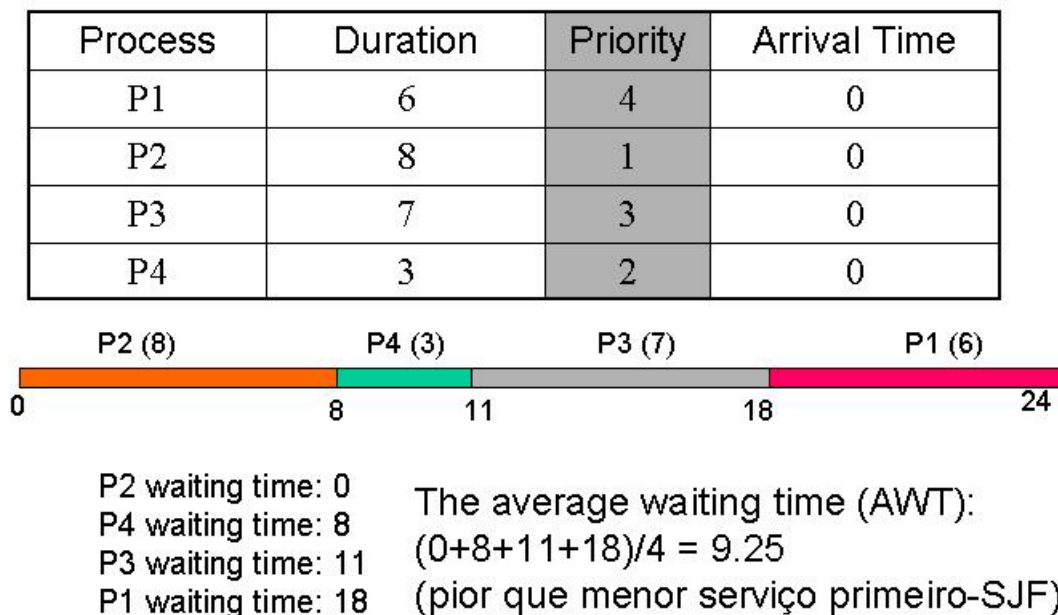


Figura 2. Exemplo do funcionamento do escalonamento com Prioridade

As prioridades podem ser associadas a processos de forma estática ou dinâmica. Dizemos que uma prioridade é estática quando ela é associada no momento da criação do processo (mesmo que depois ela seja decrementada para impedir o processo de monopolizar a UCP).

Uma prioridade é dinâmica quando o escalonador é que decide seu valor, baseado em estatísticas sobre a execução deste processo em quanta anteriores.

Como exemplo de prioridades estáticas, podemos imaginar que em um sistema implantado em um quartel, os processos iniciados por um general devem ter prioridade sobre os processos iniciados por um coronel, e assim por diante, por toda a hierarquia.

Como exemplo de prioridade dinâmica, podemos citar um sistema que, quando percebe que um dado processo efetua muitas operações de entrada/saída, aumenta a sua prioridade. Isto é feito pois as operações de entrada/saída são realizadas independentemente da UCP, de forma que, se um processo executa muitas dessas operações, é conveniente deixar que ele requisiute uma nova operação o mais rápido possível. Lembre-se que, ao requisitar uma operação de entrada/saída o processo faz uma chamada para o S.O. e fica bloqueado, aguardando o fim da operação. Assim, processos com grande utilização de E/S não significam muita carga para a UCP, pois enquanto suas transferências estão se executando a UCP pode estar alocada a outro processo que executa cálculos. Para implementar essa forma de prioridade, o escalonador calcula a fração de tempo que o processo utilizou do seu quantum na última chamada. Se o processo utilizou apenas uma pequena fração (e não terminou) é provavelmente porque está esperando dados de um dispositivo de E/S. Desta forma, podemos utilizar para prioridade, um valor proporcional ao inverso da fração de quantum ocupado.

Muitas vezes é útil agrupar os processos em classes de prioridades, fazem um escalonamento em dois níveis:

- i. escalonamento de prioridade entre as diversas classes, executando todos os processos das classes de prioridades mais altas antes das classes mais baixas;
- ii. escalonamento round-robin dentro de cada classe, fazendo com que todos os processos de uma mesma classe revezem-se como explicado para o escalonamento round-robin.

### **3. FILAS MÚLTIPLAS**

O fato importante aqui é que o computador forçava um chaveamento de processos extremamente lento, pois apenas um processo podia estar presente na memória em cada instante, fazendo com que os processos devessem ser lidos do disco e escritos no disco a cada chaveamento.

A utilização prática demonstrou que era necessário que os processos com muita utilização de UCP (também chamados de processos limitados pela UCP, "CPU-bound") ganhassem um quantum grande de uma só vez, de forma a reduzir o número de chaveamentos executados nesse processo. Por outro lado, como já vimos, não podemos atribuir quanta grandes para todos os processos, pois isso implicaria em um tempo de resposta muito grande.

A solução encontrada foi o estabelecimento de classes de prioridade. Os processos de classes mais altas são escolhidos para execução mais freqüentemente que os de classes mais baixas. Por outro lado, os processos de classes mais baixas recebem um número maior de quanta para processamento. Assim, os processos da classe mais alta existente recebem 1 quantum, os processos da classe imediatamente abaixo recebem 2 quanta, os processos da classe seguinte, 4 quanta, e assim sucessivamente, nas potências de 2.

Adicionalmente a isso, cada vez que um processo utiliza todos os quantum que recebeu ele é abaixado de uma classe (fazendo com que seja escolhido menos freqüentemente, mas execute durante mais tempo).

Como exemplo, num processo que necessita de 100 quanta para executar, teremos execuções de 1, 2, 4, 8, 16, 32, 37. Isto é, ele é selecionado para execução 7 vezes (7 chaveamentos para este processo), sendo que no método de round-robin tradicional ele seria selecionado 100 vezes até acabar (100 chaveamentos).



Por outro lado, este método, como definido até aqui, tem o grave inconveniente de que, se um processo começa com uma grande quantidade de cálculos, mas em seguida se torna interativo (isto é, exige comunicação com o usuário), teríamos um tempo de resposta muito ruim. Para eliminar este problema, foi implementado o seguinte método: cada vez que um <RET> é teclado no terminal de um processo, este processo é transferido para a classe de prioridade mais alta, na esperança de que o mesmo irá se tornar interativo. Esta solução funcionou bem até que um usuário descobriu que, durante a execução de um processo grande, bastava sentar no terminal e teclar aleatoriamente <RET> para ter o tempo de execução melhorado. Em seguida ele contou sua "descoberta" a todos. Vê-se então que a implementação prática de uma estratégia eficiente é muito mais difícil na prática do que em princípio.

#### **4. MENOR SERVIÇO (JOB) PRIMEIRO**

As estratégias apresentadas anteriormente são úteis para sistemas de time-sharing. A estratégia de menor serviço primeiro (shortest job first) é útil fundamentalmente para sistemas de lotes (batch). Nestes sistemas, muitas vezes os usuários já têm uma boa estimativa dos tempos de execução do programa (em geral porque o programa é rodado constantemente).

A estratégia consiste em escolher para execução, entre todos os jobs disponíveis, aquele de menor tempo de execução.

Para ver como esta estratégia consegue melhorar o tempo médio de resposta do sistema, consideremos o seguinte caso: existem quatro jobs disponíveis (A, B, C e D), sendo um de 8 minutos (o A) e três de 4 minutos (B, C e D). Se o job de 8 minutos for escolhido antes, teremos a seguinte configuração:

tempo de resposta de A: 8 minutos  
tempo de resposta de B:  $8 + 4 = 12$  minutos  
tempo de resposta de C:  $8 + 4 + 4 = 16$  minutos  
tempo de resposta de D:  $8 + 4 + 4 + 4 = 20$  minutos  
tempo de resposta médio: 14 minutos.

Se escolhemos os três jobs de 4 minutos antes do de 8, teremos:

tempo de resposta de B: 4 minutos  
tempo de resposta de C:  $4 + 4 = 8$  minutos  
tempo de resposta de D:  $4 + 4 + 4 = 12$  minutos  
tempo de resposta de A:  $4 + 4 + 4 + 8 = 20$  minutos  
tempo de resposta médio: 11 minutos.

Esta estratégia garante o mínimo tempo de resposta médio possível, desde que se conheçam todos os serviços simultaneamente. Se os serviços não são todos conhecidos simultaneamente o tempo de resposta médio pode não ser o mínimo, mas ainda é uma boa aproximação.

## 5. ESCALONAMENTO DIRIGIDO A POLÍTICA

Outra forma de cuidar do escalonamento é fazer uma promessa ao usuário e procurar cumpri-la. Um exemplo de uma promessa simples e fácil de cumprir é a de que, se existem  $n$  usuários ativos, então, cada um receberá aproximadamente  $1/n$  do tempo de UCP. Para cumprir essa promessa podemos usar o seguinte método:

- i. para cada usuário, é mantido um valor do tempo de UCP que ele já utilizou desde que entrou, que será chamado tempo utilizado;
- ii. compute quanto tempo já se passou desde que o usuário iniciou sua seção e divide-se pelo número de usuários, gerando o que será chamado de tempo destinado ao processo;
- iii. compute, para cada usuário, a razão entre o tempo utilizado e o tempo destinado;
- iv. escolhe para execução aquele usuário que apresenta a menor razão.

## 6. ESCALONAMENTO EM DOIS NÍVEIS

Até aqui, consideramos que todos os processo executáveis estão na memória. Este geralmente não é o caso, pois dificilmente a memória principal comportará todos os dados necessários. Precisaremos, pois, manter parte dos processos em disco.

O problema que surge é que o tempo para ativar um processo que está em disco é muito maior que o tempo necessário para ativar um processo que está na memória (uma a duas ordens de grandeza maior).

A melhor solução para isto é a utilização de um escalonador em dois níveis. O funcionamento será então da seguinte forma:

- i. um subconjunto dos processos executáveis é mantido na memória;
- ii. um outro subconjunto é mantido no disco;
- iii. um escalonador (chamado de baixo nível) é utilizado para realizar o chaveamento (por qualquer dos métodos já descritos, ou outros semelhantes) apenas entre os processos que estão na memória;
- iv. um outro escalonador (chamado de alto nível) é utilizado para trocar periodicamente o conjunto de processo que estão na memória (substituindo-os por alguns que estavam no disco).

Para realizar a escolha de qual conjunto de processos deve ser colocado na memória principal o escalonador de alto nível deve responder perguntas tais como:

- Quanto tempo se passou desde que o processo foi posto ou retirado ?
- Quanto tempo de UCP o processo teve recentemente ?
- Qual o tamanho do processo ?
- Qual a prioridade do processo ?

# GERENCIAMENTO DE MEMÓRIA

Trataremos agora da parte do S.O. conhecida como gerenciador de memória, que é o responsável por cuidar de quais partes da memória estão em uso, quais estão livres, alocar memória a processos quando eles precisam, desalocar quando eles não mais necessitarem e gerenciar a troca (swapping) dos processos entre memória principal e disco, quando a memória principal não é suficientemente grande para manter todos os processos.

## 1. GERENCIAMENTO DE MEMÓRIA SEM TROCA OU PAGINAÇÃO

Troca e paginação são métodos utilizados de movimentação da memória para o disco e vice-versa durante a execução dos processos. Entretanto, antes de estudar esses métodos, estudaremos o caso mais simples em que não é realizada nem troca nem paginação.

## 2. MONOPROGRAMAÇÃO SEM TROCA OU PAGINAÇÃO

Neste caso, temos um único processo executando por vez, de forma que o mesmo pode utilizar toda a memória disponível, com exceção da parte reservada ao sistema operacional, que permanece constantemente em local pré-determinado da memória. O S.O. carrega um programa do disco para a memória, executa-o e, em seguida, aguarda comandos do usuário para carregar um novo programa, que se sobrepõe ao anterior.

## 3. MULTIPROGRAMAÇÃO

Apesar de largamente utilizada em microcomputadores, a monoprogramação praticamente não é mais utilizada em sistemas grandes. Algumas razões para isso são as seguintes: - Muitas aplicações são mais facilmente programáveis quando as dividimos em dois ou mais processos.

- Os grandes computadores em geral oferecem serviços interativos simultaneamente para diversos usuários. Neste caso, é impossível de se trabalhar com um único processo na memória por vez, pois isso representaria grande sobrecarga devido à constante necessidade de chavear de um processo para outro (o que, com apenas um processo por vez na memória representaria constantemente estar lendo e escrevendo no disco).
- É necessário que diversos processos estejam "simultaneamente" em execução, devido ao fato de que muitos deles estão constantemente realizando operações de E/S, o que implica em grandes esperas, nas quais, por questão de eficiência, a UCP deve ser entregue a outro processo.

### 3.1. MULTIPROGRAMAÇÃO COM PARTIÇÕES FIXAS

Já que percebemos a importância da existência de diversos processos na memória, devemos agora analisar de que forma este objetivo pode ser conseguido. A forma mais simples é dividir a memória existente em  $n$  partições fixas, possivelmente diferentes. Essas partições poderiam ser criadas, por exemplo, pelo operador, ao inicializar o sistema.

Uma forma de tratar com essas partições seria então a seguinte:

- i. cria-se uma fila para cada partição existente;
- ii. cada vez que um processo é iniciado, ele é colocado na fila da menor partição capaz de o executar;

- iii. os processos em cada partição são escolhidos de acordo com alguma forma de política (p.ex.: o primeiro a chegar é atendido antes).

Este método pode ser simbolizado como na Figura 3 :

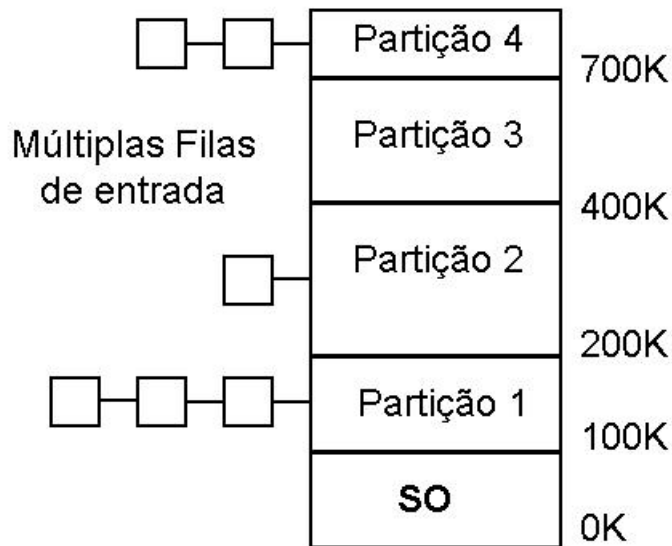


Figura 3. Esquema de filas Múltiplas

Uma desvantagem óbvia desse esquema é a de que pode ocorrer que uma partição grande esteja sem utilização, enquanto que diversos processos estão aguardando para utilizar uma partição menor (p.ex.: na figura acima temos três processos aguardando pela partição 1, enquanto que a partição 3 está desocupada).

Podemos resolver esse problema da seguinte forma:

- estabelecemos apenas uma fila para todas as partições;
- quando uma partição fica livre, um novo processo que caiba na partição livre é escolhido e colocado na mesma, conforme indicado na Figura 4:

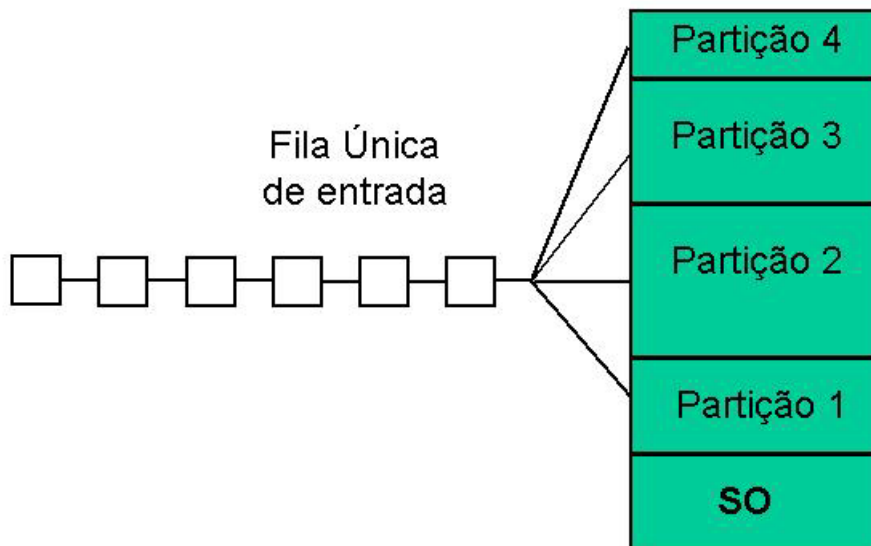


Figura 4. Esquema de filas Única

O problema que surge aqui é a forma de escolha implícita no item (ii). Se a partição livre for entregue para o primeiro processo da fila, pode ocorrer de que uma partição grande seja entregue a um processo pequeno, o que não é desejável (pois implica desperdício de memória). Por outro lado, se percorremos a fila procurando o maior processo aguardando que caiba na partição, isto representará servir melhor os processos grandes em detrimento dos pequenos.

### **3.2. RELOCAÇÃO E PROTEÇÃO**

Relocação e proteção são dois problemas introduzidos pela multiprogramação, e que precisam ser resolvidos. A necessidade da relocação pode ser vista nas figuras anteriores, onde fica claro que processos diferentes executam em posições de memória diferentes e, portanto, com endereços diferentes.

Suponha uma rotina em um programa que, ao término da ligação dos módulos deste, é colocada na posição 100 em relação ao início do programa. É claro que se esse programa for executado na partição 1, todas as chamadas dessa rotina devem ser enviadas para a posição de memória  $100k+100$ . Se o programa for executado na partição 2, as chamadas para essa mesma rotina devem ser enviadas para a posição  $200k+100$ .

Uma possível solução é modificar as instruções conforme o programa é carregado na memória. Desta forma, quando o S.O. carrega o programa, adiciona a todas as instruções que se referenciem a endereços, o valor do ponto inicial de carga do programa (i.e., o início da partição escolhida para o mesmo). Esta solução exige que o ligador (linker) coloque no início do código binário do programa uma tabela (ou uma lista) que apresente as indicações das posições no programa que devem ser modificadas no carregamento.

Esta solução entretanto não resolve o problema da proteção, pois nada impede que um programa errado ou malicioso leia ou altere posições de memória de outros usuários (dado que as referências são sempre as posições absolutas de memória).

Uma solução para isto adotada no IBM 360 foi a de dividir a memória em unidades de 2K bytes e associar um código de proteção de 4 bits a cada uma dessas regiões. Durante a execução de um processo, o PSW contém um código de 4 bits, que é testado com todos os acessos à memória realizados pelo processo, e gera uma interrupção se tentar acessar uma região de código diferente.

Uma solução alternativa tanto para o problema da relocação como para o da proteção é a utilização de registradores de base e limite. Sempre que um processo é carregado na memória, o S.O. ajusta o valor do registrador de base de acordo com a disponibilidade de memória. Este registrador é utilizado de forma que, sempre que um acesso é realizado na memória pelo processo, o valor do registrador de base é automaticamente somado, o que faz com que o código do programa em si não precise ser modificado durante o carregamento. O registrador de limite é utilizado para determinar o espaço de memória que o processo pode executar. Todo acesso realizado pelo processo à memória é testado com o valor do registrador limite, para verificar se esse acesso está se realizando dentro do espaço reservado ao processo (caso em que ele é válido) ou fora do seu espaço (acesso inválido). Uma vantagem adicional do método dos registradores base e limite é o de permitir que um programa seja movido na memória, mesmo após já estar em execução. No método anterior isto não era possível sem realizar todo o processo de alteração dos endereços novamente.

### 3.3. TROCA (swapping)

Num sistema em batch, desde que se mantenha a UCP ocupada o máximo de tempo possível, não existe razão para se complicar o método de gerenciamento da memória. Entretanto, num sistema de time-sharing, onde muitas vezes existe menos memória do que o necessário para manter todos os processos de usuário, então é preciso que uma parte dos processos sejam temporariamente mantidos em disco. Para executar processos que estão no disco, eles devem ser enviados para a memória, o que significa retirar algum que lá estava.

Este processo é denominado troca (swapping).

### 3.4. MULTIPROGRAMAÇÃO COM PARTIÇÕES VARIÁVEIS

A utilização de partições fixas para um sistema com swapping é ineficiente, pois implicaria em que grande parte do tempo as partições estariam sendo utilizadas por processos muito pequenos para as mesmas.

A solução para isso é tratarmos com partições variáveis, isto é, partições que variam conforme as necessidades dos processos, tanto em tamanho, como em localização, como em número de partições. Isto melhora a utilização de memória mas também complica a alocação e desalocação de memória.

É possível combinar todos os buracos formados na memória em um único, o que é conhecido como compactação de memória, mas isto é raramente utilizado, devido à grande utilização de UCP requerida.

Um ponto importante a considerar é quanta memória deve ser alocada a um processo, quando o mesmo é iniciado. Se os processos necessitam de uma certa quantidade pré-fixada e invariante de memória, então basta alocar a quantidade necessária para cada processo ativo e o problema está resolvido. Entretanto, muitas vezes os processos necessitam de mais memória durante o processamento (alocação dinâmica de memória). Neste caso, quando um processo necessita de mais memória, podem ocorrer dois casos:

- a. existe um buraco (região não ocupada) de memória próximo ao processo: neste caso, basta alocar memória desse buraco;
- b. o processo está cercado por outros processos, ou o buraco que existe não é suficiente, neste caso, podemos tomar alguma das seguintes ações:
  - mover o processo para um buraco de memória maior (grande o suficiente para as novas exigências do processo);
  - se não houver tal espaço, alguns processos devem ser retirados da memória, para deixar espaço para esse processo;
  - se não houver espaço no disco para outros processos, o processo que pediu mais espaço na memória deve ser morto (**kill**).

Quando se espera que diversos processos cresçam durante a execução, pode ser uma boa idéia reservar espaço extra para esses processos quando os mesmos são criados, para eliminar a sobrecarga de lidar com movimentação ou troca de processos.

Trataremos agora de duas das formas principais de cuidar da utilização de memória: mapas de bits, listas e sistemas de desabrochamento.