TÓPICO 2

Gerência de Processos

Visão Geral

- O conceito de processo é um dos conceitos mais importantes para qualquer sistema operacional
- ***Um processo é um programa em execução***.
- Além do código executável, nós temos uma pilha de execução, um apontador para esta pilha, um contador de programa, valores dos registradores da máquina, outras informações.

Visão Geral

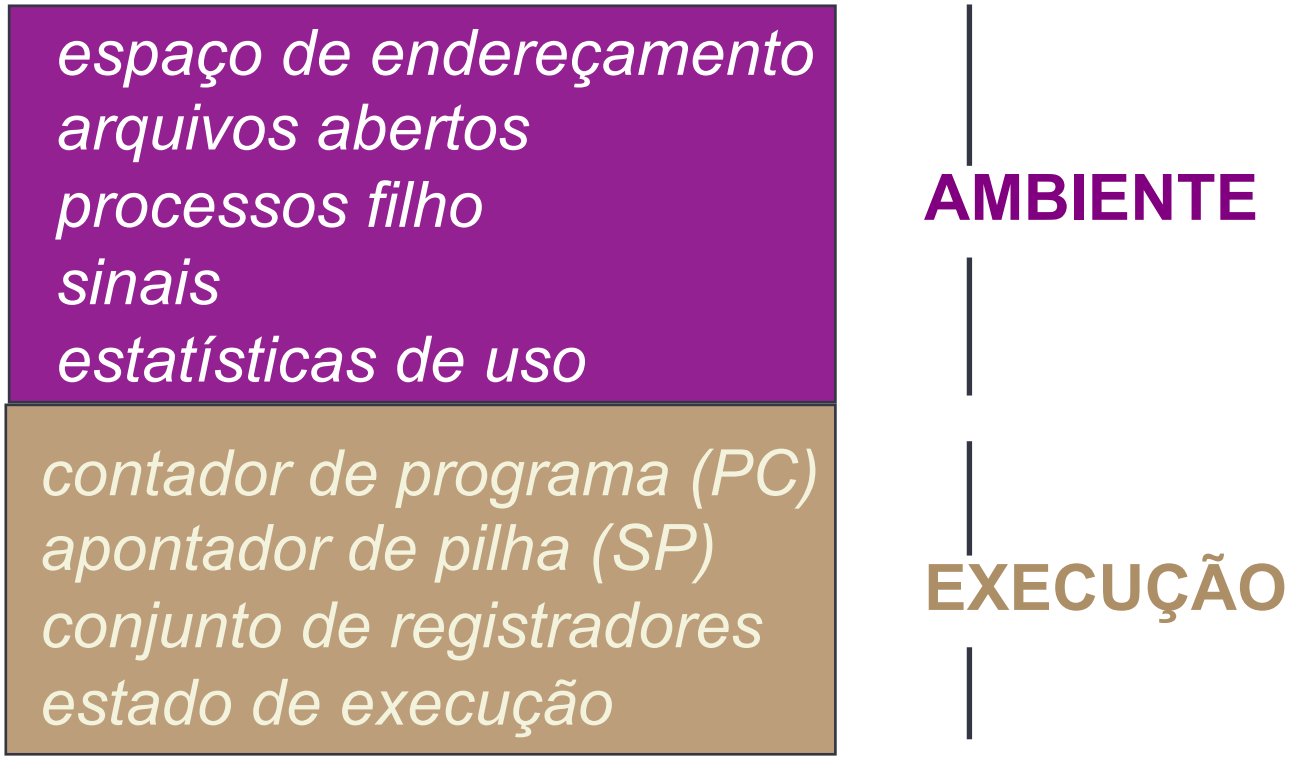
- Cada processo tem um identificador único, conhecido como pid (process id).
- As informações sobre um processo estão na tabela de processos, que é acessada pelo pid.
- Durante a execução, o processo compartilha o processador com outros processos em execução (escalonamento de processador).
- Um processo interage com outros processos através de mecanismos de comunicação.

Visão geral

- Diferença entre um processo e um programa (por analogia):
  - programa: receita de bolo (passivo)
  - processo: ato de fazer o bolo (ativo)
- A noção de processo envolve sempre uma noção de atividade.

Modelo de Processo

- Informações que dizem respeito a um processo:



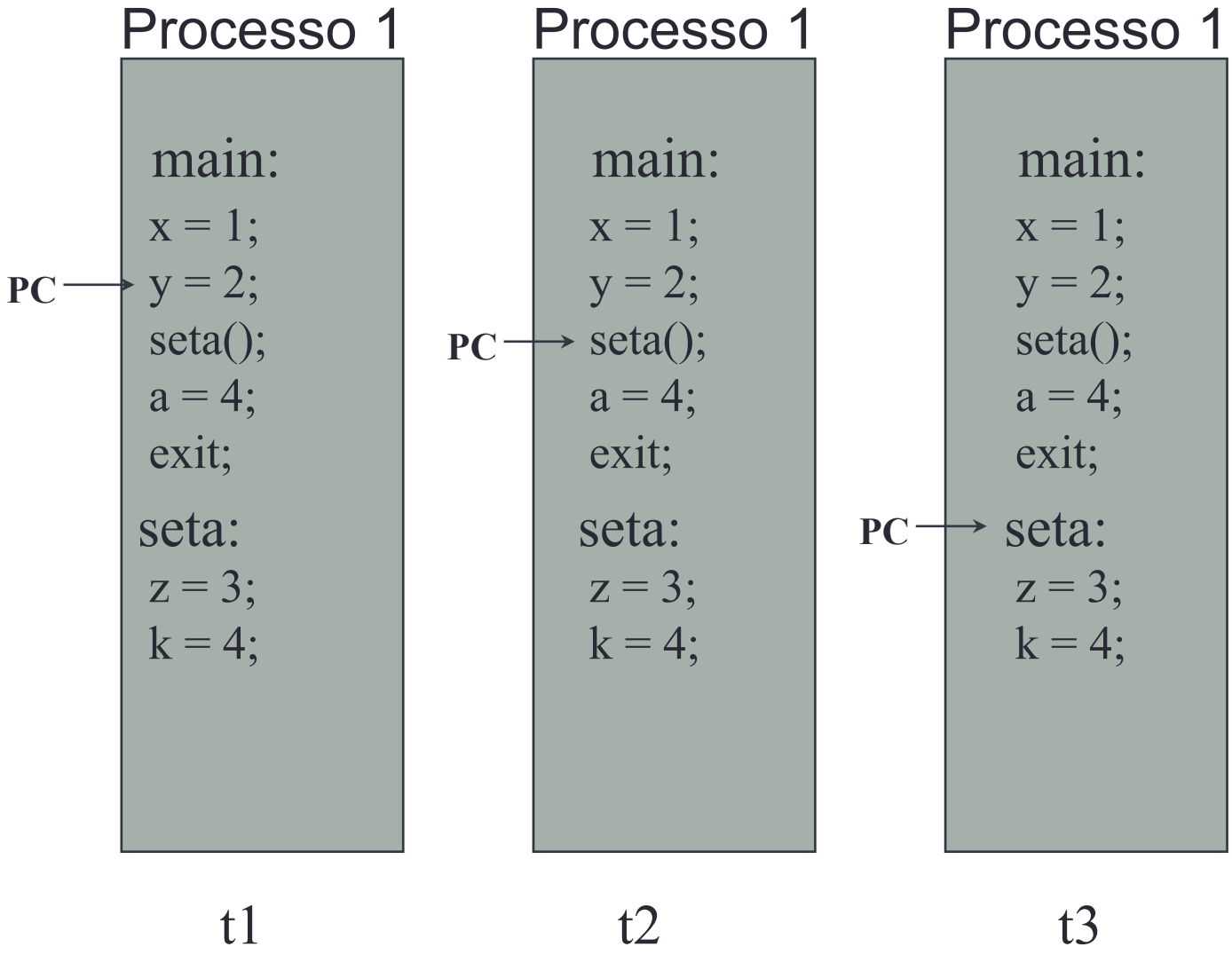
Modelo de Processo

- Classificação dos modelos de processos quanto ao custo de troca de contexto e de manutenção
  - “heavyweight” (processo tradicional)
  - “lightweight” (threads)

Modelo de Processo Tradicional Heavyweight

- O processo é composto tanto pelo ambiente como pela execução: Processo tradicional: ambiente + execução.
- Cada processo possui um único fluxo de controle (contador de programa) e roda de forma independente dos demais.

Modelo de Processo Tradicional Heavyweight

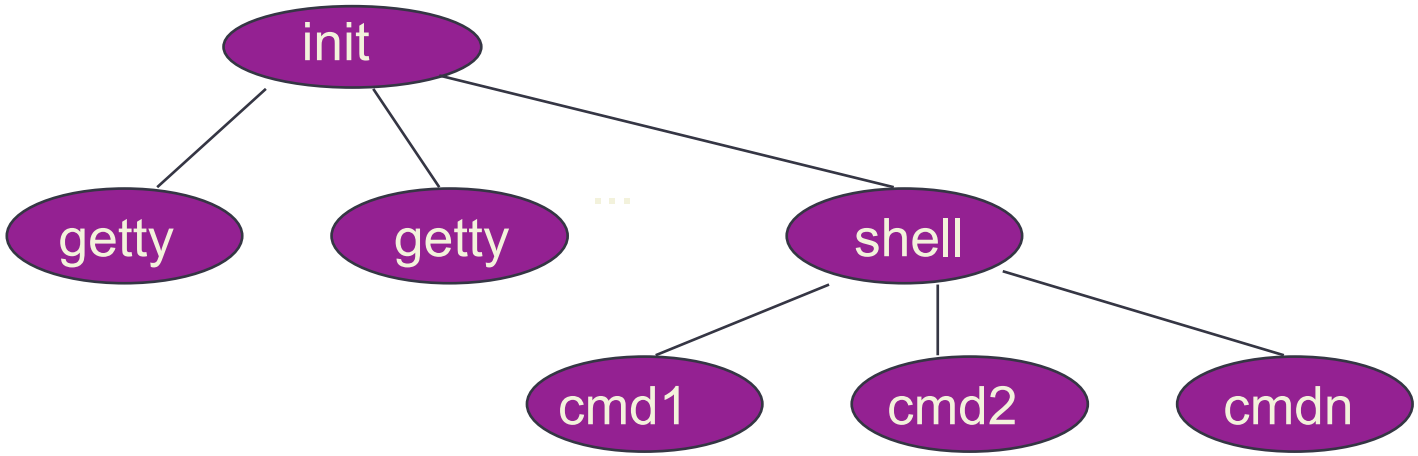


Modelo de Processo Tradicional

- Como, em um dado instante, pode haver vários processos ativos ao mesmo tempo, o processador é chaveado entre os diversos processos.
- Por esta razão, fica praticamente impossível prever o tempo de execução de um processo, pois este dependerá da carga do sistema.

Modelo de Processo Tradicional

- Todo sistema operacional deve possuir mecanismos que permitam a criação de processos. Geralmente, um processo somente é criado por outro processo, o que nos leva a uma hierarquia em árvore.



Modelo de Processo Tradicional

- Um processo é criado a partir de:
  - um programa (criação tradicional) (MS-DOS):
    - create_process(“teste.exe”);
  - outro processo (clonagem) (Unix):
    - fork();

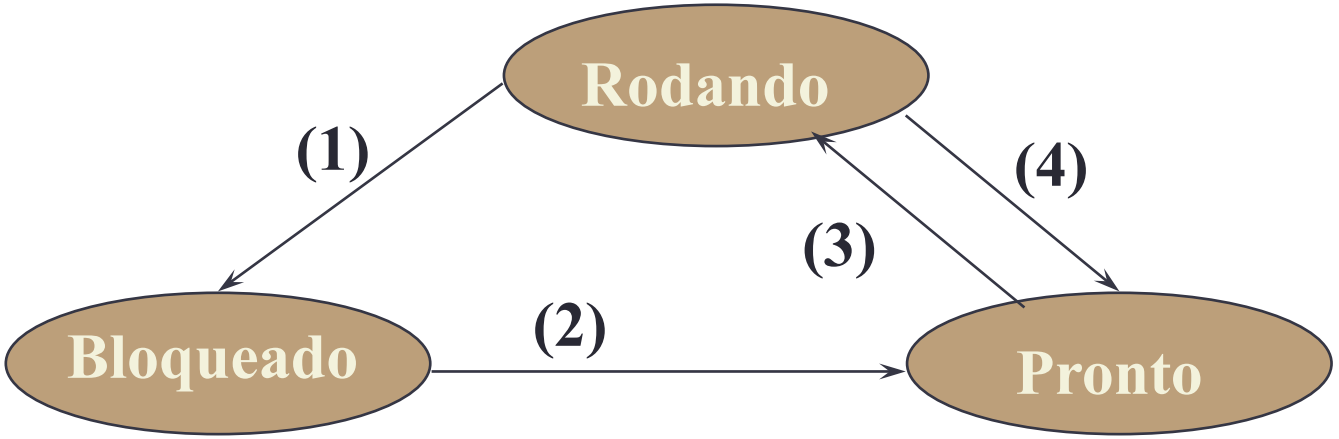
Estados do Processo

- Apesar dos processos serem relativamente auto-suficientes, muitas vezes eles necessitam de acessar outros recursos (discos, terminais) ou mesmo se comunicar com outros processos.
- Quando um processo está ocioso esperando que um evento aconteça, nós dizemos que ele está bloqueado. Em algumas situações, o processo pode ser bloqueado a revelia pelo sistema operacional.

Estados do Processo

- Os estados básicos de um processo são:
  - rodando (running),
  - bloqueado (blocked) e
  - pronto (ready).
- Em um sistema monoprocessado, só temos um único processo rodando a cada instante.

Estados do Processo



(1) O processo bloqueia-se à espera de um evento

(2) O evento esperado pelo processo ocorreu. Ele pode agora se executar. Passa então ao estado "pronto"

(3) O processo é escolhido para execução

(4) O tempo de posse do processador pelo processo esgotou-se. O sistema operacional retira o processador do processo

Implementação de Processos

- Todas as informações sobre um processo são mantidas na tabela de processos.
- A tabela de processos contem campos que dizem respeito à gerência do processo, à gerência da memória e à gerência de arquivos.
- A tabela de processos possui uma entrada por processo e os campos nela contidos variam de sistema operacional para sistema operacional.

Tabela de Processos

- Dados referentes ao processo na tabela de processos:
  - identificador do processo (pid), valor dos registradores, valor do contador de programa (PC), valor da palavra de estado (PSW), valor do apontador de pilha (SP), estado do processo, instante do início do processo, tempo de processador utilizado, etc.

Tabela de Processos

- Dados referentes à memória na tabela de processos:
  - endereço do segmento de texto, dados e pilha, estado da saída, informaões sobre proteção, etc
- Dados referentes à gerência de arquivos na tabela de processos:
  - diretório raiz, diretório de trabalho, descritores de arquivos abertos, parâmetros de chamadas em andamento, etc

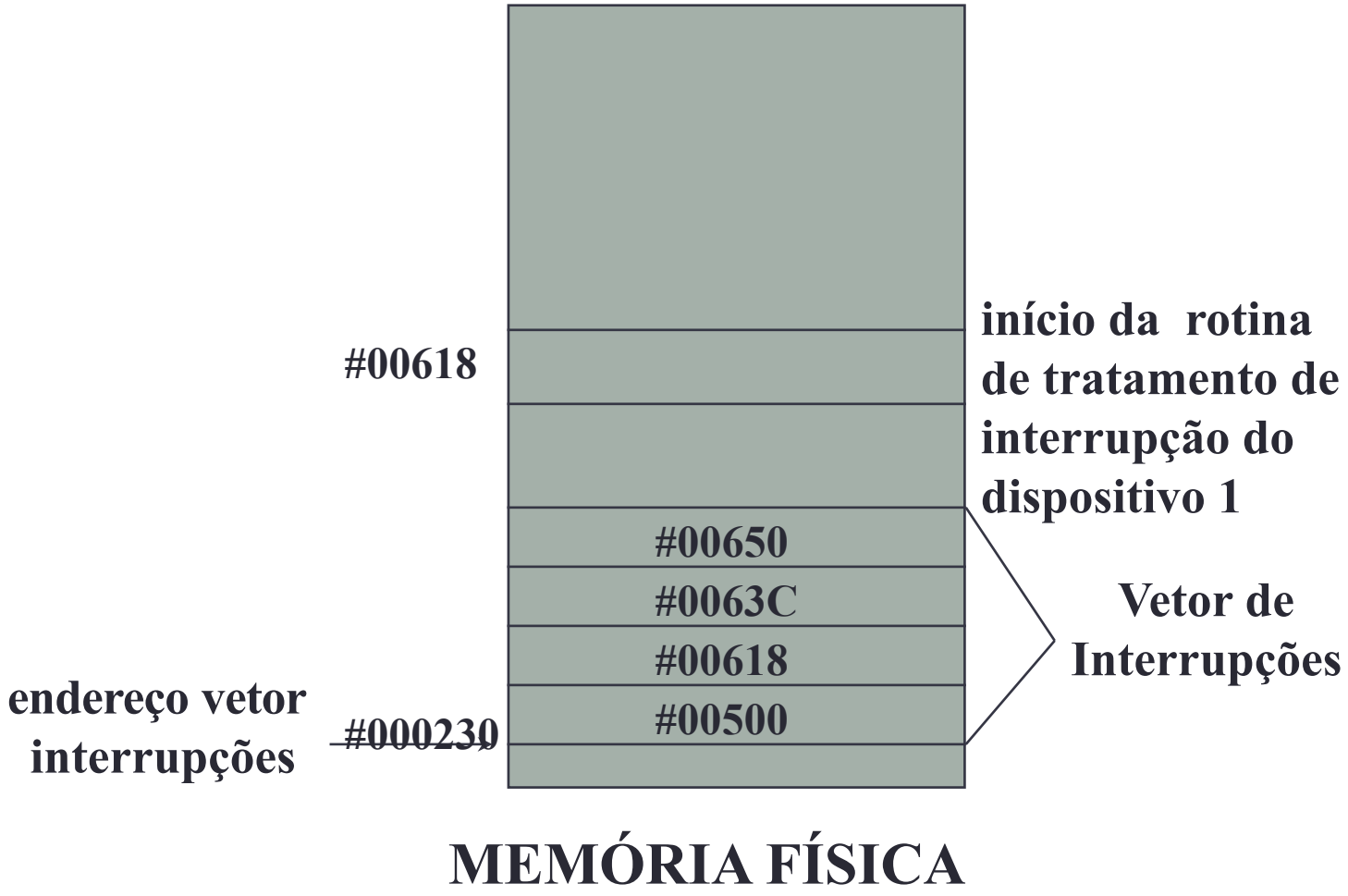
Implementando a Multiprogramação

- O processo p1 solicita uma operação de E/S. Ele é bloqueado pelo sistema operacional à espera da conclusão da operação. As informações de p1 são atualizadas na sua tabela de processos.
- O sistema operacional escolhe um dos processos na fila dos prontos (e.g. processo p2) e o coloca para executar.

Implementando a Multiprogramação

- O resultado da operação solicitada por p1 chega.
- O hardware interrompe p2, salvando o seu estado de execução (registradores, descritores, etc) na pilha.
- O hardware acessa um endereço de memória física específico que contem o vetor de interrupções. O vetor de interrupções contem o endereço da rotina de tratamento de interrupções geradas por cada classe de dispositivo (disco, floppy, terminal, clock, etc).

Vetor de Interrupções

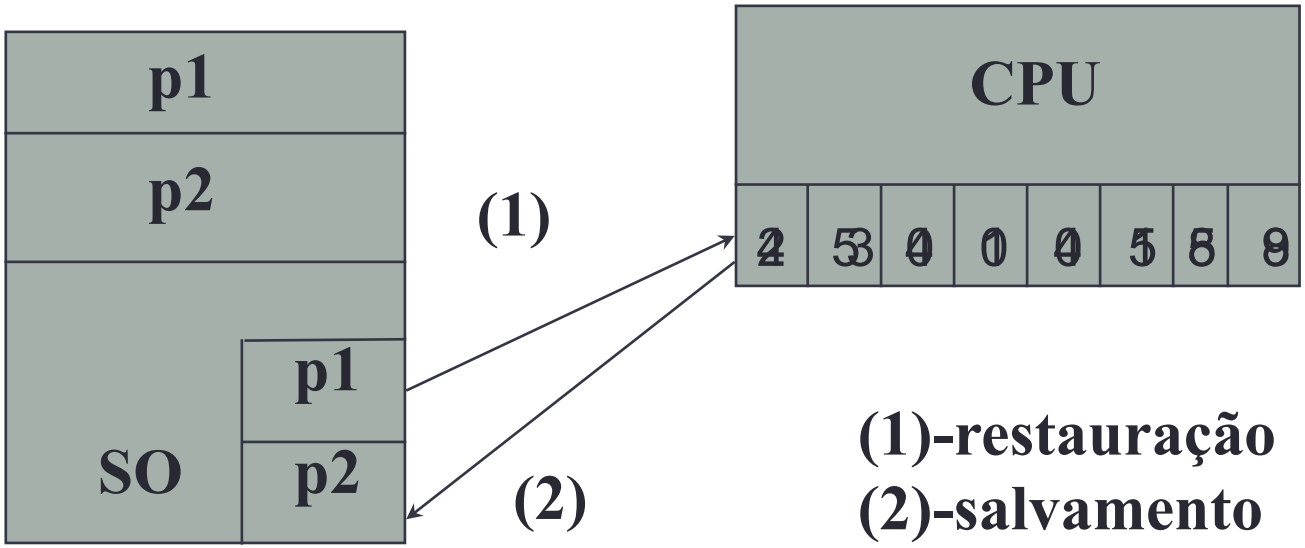


Implementando a Multiprogramação

- A rotina de tratamento de interrupção é executada pelo SO. Os registradores que foram empilhados pelo hardware são salvos na tabela de processos do processo p2. A interrupção é tratada. O processo p1, que solicitou o serviço, é colocado na fila de prontos.
- O SO acessa a entrada da tabela de processos do processo escolhido e carrega o conteúdo da tabela nos registradores de máquina (restauração).
- O processo escolhido reinicia a execução.

Implementando a Multiprogramação

- Troca de contexto: a operação de salvamento dos registradores de um processo e posterior restauração de registradores de outro processo é chamada de troca de contexto. A troca de contexto permite a troca de processador entre processos.

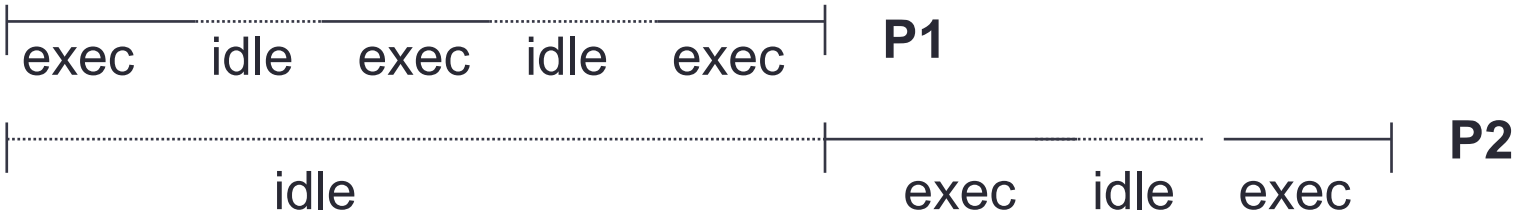


Escalonamento de Processos

- O algoritmo de escalonamento de processos é o responsável pela determinação de que processo (dentre os prontos) vai rodar e por quanto tempo. O algoritmo de escalonamento define, assim, a política de utilização do processador pelos processos.
- Quando um processo solicita operações blocantes (E/S, por exemplo), sua execução fica suspensa até que o evento solicitado ocorra.

Escalonamento de Processos

- Execução de 2 processos sem concorrência:



- Execução de 2 processos com concorrência:

- Executando vários processos concorrentemente, obtemos uma melhor utilização da CPU

Critérios do Escalonamento

- Ao se projetar um escalonador, devemos observar vários critérios que devem estar presentes em um bom algoritmo de escalonamento:
  - Justiça (fairness): garantir que todos os processos do sistema terão chances justas de uso do processador (chances iguais é muito forte!!)
  - Eficiência: quando houver trabalho a fazer, o processador deve estar ocupado
  - Minimizar o tempo de resposta: minimizar o tempo de resposta dos usuários interativos.

Critérios de Escalonamento

- Minimizar o turnaround: O tempo de turnaround é o tempo que vai desde o lançamento do processo até o seu término. É a soma dos seguintes componentes: tempo de espera por memória, tempo de espera pelo processador, tempo de espera por I/O e tempo de utilização da CPU. Este critério visa minimizar a espera de resultados. Mais utilizado em processamento batch.
- Minimizar o waiting time: Este critério visa minimizar o tempo de espera pela CPU.
- Maximizar o throughput: maximizar o número de jobs executados em uma unidade de tempo

Critérios de Escalonamento

- Infelizmente, a maioria destes critérios é contraditório:
  - minimizar o turnaround x minimizar o tempo de resposta: para que os usuários interativos obtenham um tempo de resposta pequeno, geralmente os usuários batch são penalizados com um tempo de execução maior e vice-versa.
  - Um algoritmo que maximiza o throughput geralmente não é justo com os processos de execução demorada.

Classificação dos Escalonadores Quanto à Preempção

- *Preempção*: suspensão temporária da execução de um processo
- Os escalonadores podem ser:
  - Preemptivos
  - Não-Preemptivos

Escalonadores Não-Preemptivos

- *Escalonador não-preemptivo:*Quando um processo obtem o processador, ele roda até o fim ou até que ele peça uma operação que ocasione o seu bloqueio.
- Nenhuma entidade externa “tira a CPU à força” do processo

Escalonadores Preemptivos

- Cada processo possui um tempo máximo de permanência de posse do processador. Quando este tempo se esgota, o SO retira o processador deste processo e permite que outro processo se execute.
- Como controlar o tempo de execução do processo?
- Todo processador moderno possui um clock que gera interrupções em uma frequência determinada. Cada uma destas interrupções é chamada *clock tick*.

Escalonadores Preemptivos

- O SO mantem um contador que é decrementado a cada clock tick.
- Se o contador chegar a 0, o tempo de permanência do processo acabou.
- O valor inicial deste contador corresponde ao tempo máximo de permanência do processo com a CPU e denomina-se *time slice*.

Escalonadores Preemptivos x Escalonadores Não-Preemptivos

- Os escalonadores não-preemptivos são de projeto extremamente simples, porém permitem que um processo detenha a CPU por um tempo arbitrário.
- Neste caso, um processo pode obter o monopólio do processador, impedindo os outros processos de rodarem.
- Isso viola vários dos critérios de um bom escalonador (justiça, tempo mínimo de resposta, etc).

Escalonadores Preemptivos x Escalonadores Não-Preemptivos

- Os escalonadores preemptivos asseguram um uso mais balanceado da CPU e são utilizados na maioria dos SO modernos.
- Porém, o projeto de tais escalonadores, além de ser complexo, introduz complicações na programação de processos concorrentes.
- Como agora os processos podem ser interrompidos em um tempo arbitrario, eles devem proteger suas estruturas de dados contra a interferência de outros processos (regiões críticas).

Algoritmos de Escalonamento

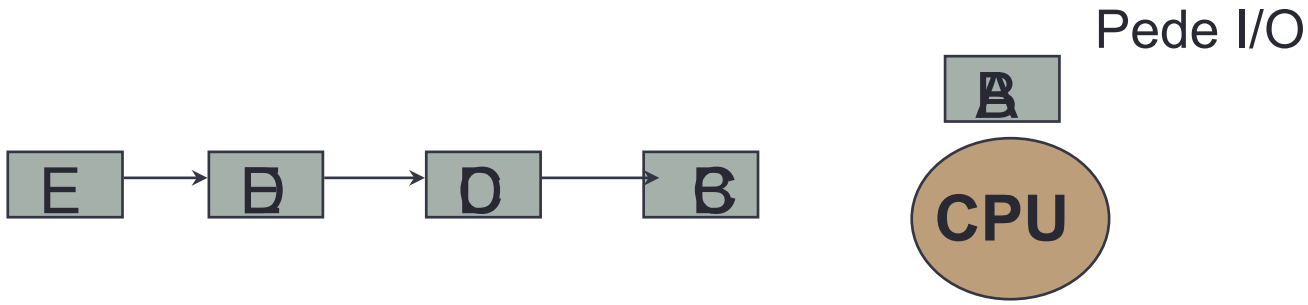
- O problema a ser resolvido pelos algoritmos de escalonamento é o seguinte: dado um conjunto de processos que devem ser executados, como dividir a utilização do processador entre estes processos?
- Algoritmos de escalonamento:
  - First Come First Served
  - Round-Robin
  - Prioridades
  - Shortest Job First

First Come First Served (FCFS)

- O processo obtem a CPU de acordo com a ordem de chegada das solicitações. O processo que pede a CPU primeiro obtem a CPU em primeiro lugar.
- O escalonamento FCFS é não-preemptivo. Assim, um processo CPU/bound pode fazer com que vários processos esperem por um tempo indeterminado.

First Come First Served (FCFS)

- Implementação: Os processos que solicitam a CPU são colocados em uma fila de prontos, que é gerenciada segundo a política FIFO.



Round-Robin

- Cada processo tem o direito de usar o processador por um intervalo de tempo pré-definido. Este intervalo de tempo é denominado *quantum*. Quando o quantum se esgota, o processador é dado a outro processo.

➔ Algoritmo justo

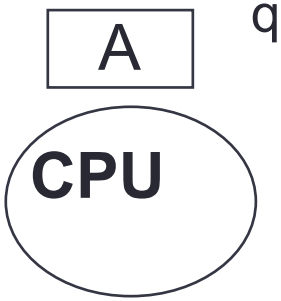
Round-Robin

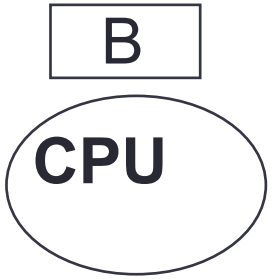
- Funcionamento:

t1

t2

Esgotou quantum





Round-Robin

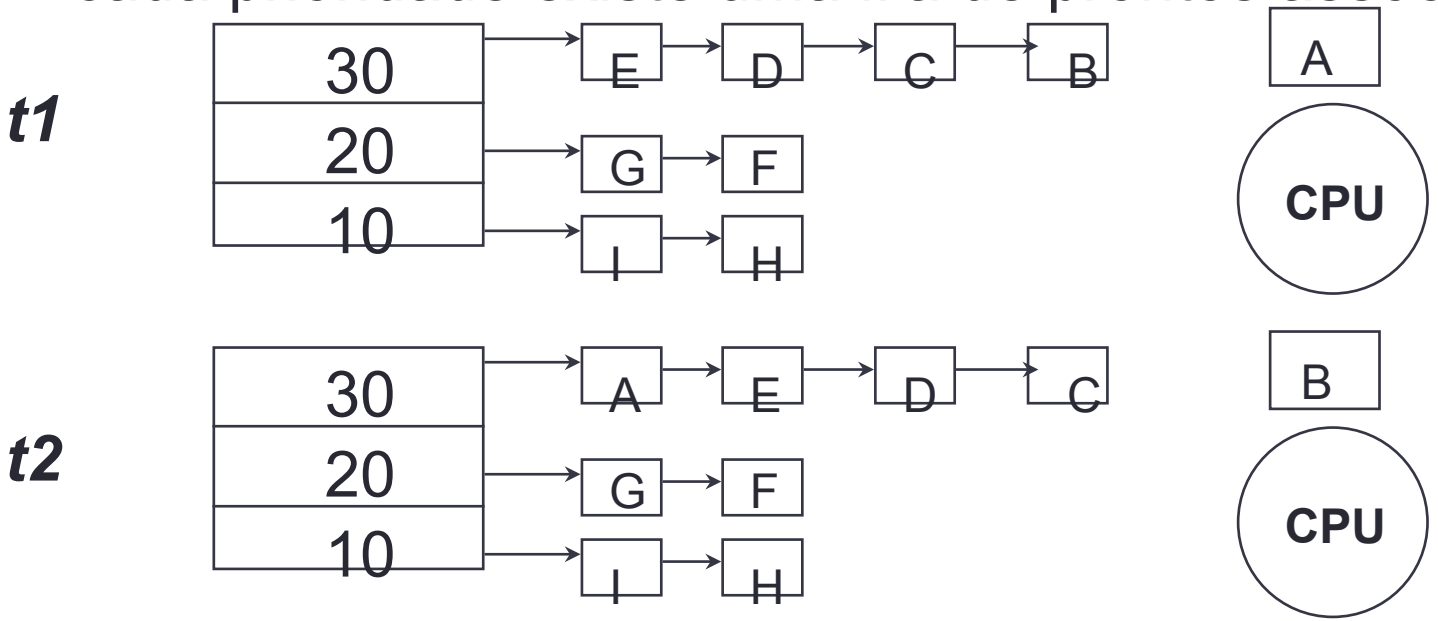
- Um dos maiores problemas do algoritmo de escalonamento round-robin diz respeito à determinação do valor a ser atribuído ao quantum. Para a determinação deste valor, devemos levar em consideração o tempo médio da troca de contexto e o tempo de resposta desejado.
  - normalmente, o quantum fica em torno de 100 ms.

Escalonamento com Prioridades

- Baseia-se no fato de que alguns processos são prioritários e, assim, devem ser executados antes dos outros.
- A cada processo é atribuída uma prioridade. Processos com prioridade maior rodam primeiro.

Escalonamento com Prioridades

- Como atribuir as prioridades?
  - *de forma estática:* os processos são divididos em classes e a cada classe é atribuída uma prioridade. A cada prioridade existe uma fila de prontos associada



Escalonamento com Prioridades

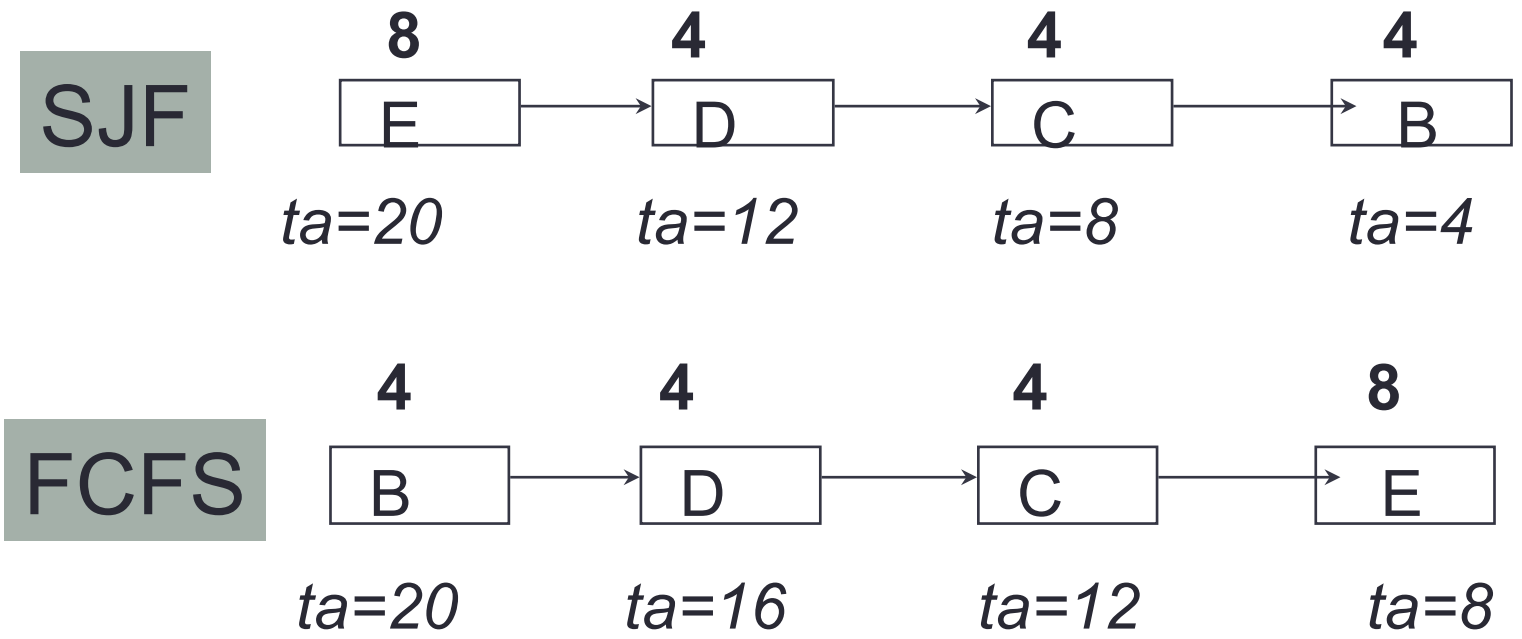
- *de forma dinâmica:* o sistema analisa o comportamento dos processos e atribui prioridades favorecendo um certo tipo de comportamento.
  - Exemplo: processos I/O bound devem prossuir prioridade alta. Prioridade dinâmica: *1/f*, onde *f* é a fração do quantum de tempo usada na última rodada do processo.

Shortest Job First

- Algoritmo projetado para sistemas batch
- Objetivo: reduzir o tempo de turnaround
- Requer que o tempo total de execução do job seja conhecido antes do início da execução
- Algoritmo: Dado um conjunto de jobs prontos, execute os jobs com menor tempo de execução antes.

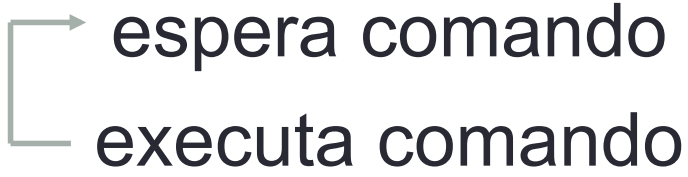
Shortest Job First

- Funcionamento:



Shortest Job First

- Aplicação para sistemas interativos:
- Processo interativo:



- Se considerarmos cada “executa comando” como um job, podemos aplicar o SJF para processos interativos.

Shortest Job First

- Problema: Como determinar o tempo de execução do comando?
  - Usar a técnica conhecida como aging,que estima um valor baseando-se e lembrando-se dos valores passados.
  - Quando um valor fica muito antigo, ele praticamente não influencia mais na estimativa.
- Exemplo: T0: tempo médio de execução de um comando, T1: tempo medido na última rodada
  - Tempo estimado = T
  - T = aT1 + (1-a) T0
  - O que acontece se a=1/2?

Escalonamento em Dois Níveis

- Um caso típico de escalonamento em dois níveis é o algoritmo que considera tanto os processos que estão em memória como os processos que estão em disco.
  - Primeiro nível: manipula os processos que estão carregados em memória.
    - Pode ser usada uma das 4 classes de algoritmos descritas anteriormente
  - Segundo nível: examina periodicamente o tempo de execução dos processos e os tira ou os carrega em memória (operações de swap in/ swap out).
- Quando o modelo de processos inclui threads, podemos ter também um algoritmo de 2 níveis. O primeiro nível determina que processo irá rodar e o segundo nível determina qual thread do processo selecionado irá executar.

Comunicação entre Processos

- Ao longo de sua execução, um processo necessita frequentemente interagir com outros processos.
- A interação entre os processos pode ser de 2 maneiras:
  - competição
  - cooperação

Competição entre Processos

- Neste caso, os processos entram em conflito pela utilização de recursos.
- O processo que utiliza um recurso deve sempre deixá-lo em estado consistente, pois o mesmo poderá ser utilizado por vários outros processos que se desconhecem mutuamente.
- As relações de competição afetam todos os processos que executam em um mesmo computador.
- São ditas *relações mínimas entre os processos*.

Cooperação entre Processos

- Neste caso, os processos que interagem entre si possuem conhecimento da existência de outros processos.
- Tipos de Cooperação:
  - Por Compartilhamento de variáveis
  - Por Troca de mensagens

Cooperação entre Processos

- *Cooperação por compartilhamento:* os processos não se conhecem explicitamente, interagindo entre si através de variáveis compartilhadas.
- *Cooperação por troca de mensagens*: Neste caso, os processos enviam explicitamente valores a outros processos. As primitivas de comunicação existentes são **enviar** e **receber**. A comunicação só se completa quando o par de primitivas **enviar-receber** é executado.
- A cooperação entre processos é descrita frequentemente na literatura como IPC (inter-process communication).

Condições de Corrida

- Como o sistema operacional determina, através da política de escalonamento, o processo que vai rodar e por quanto tempo, não sabemos a priori em que ordem dois processos ativos irão se executar.
- Condições de corrida: Acontecem frequentemente quando dois ou mais processos acessam concorrentemente as mesmas posições de memória. Se o valor final contido nestas posições depende da ordem na qual os processos foram executados, estamos diante de uma condição de corrida (race condition).
- A existência de condições de corrida em um sistema pode levar a resultados inesperados, devido ao não-determinismo inerente a estas condições.

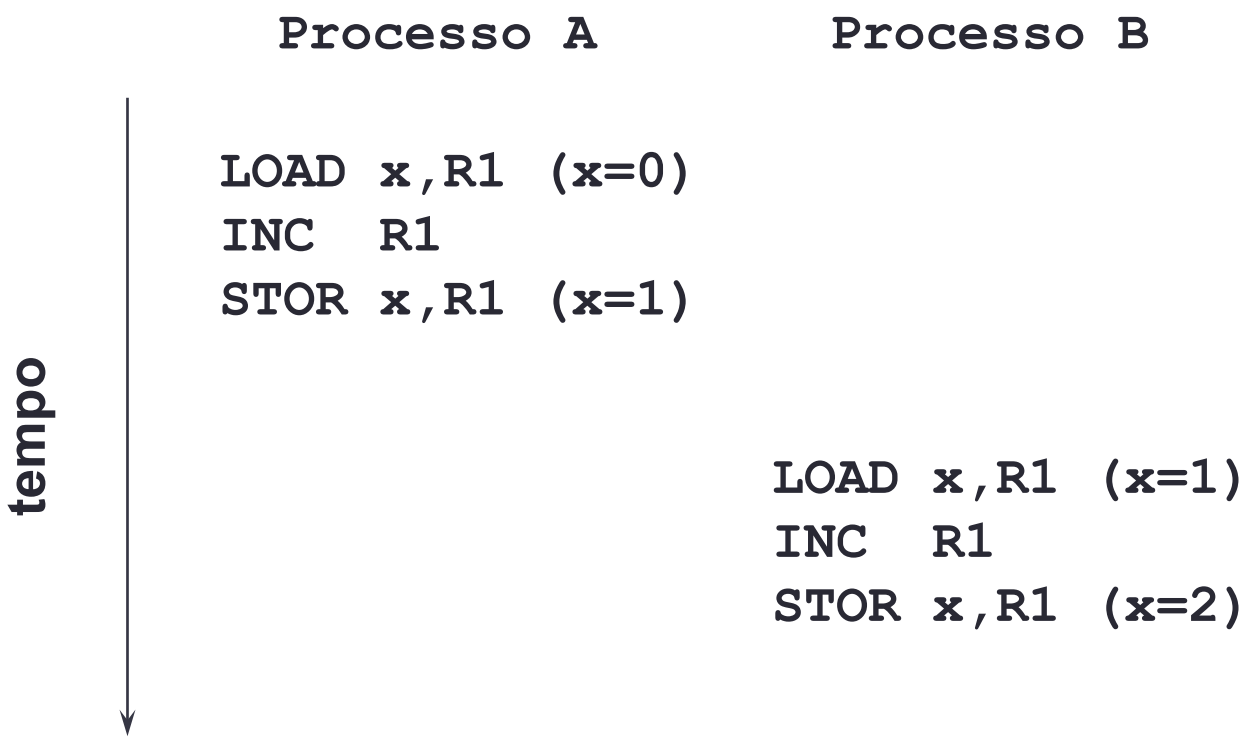
Condições de Corrida

- Primeiro exemplo de condições de corrida:
  - Considere dois processos A e B com o seguinte código:

Processo A	Processo B
`x=x+1;`	`x=x+1;`

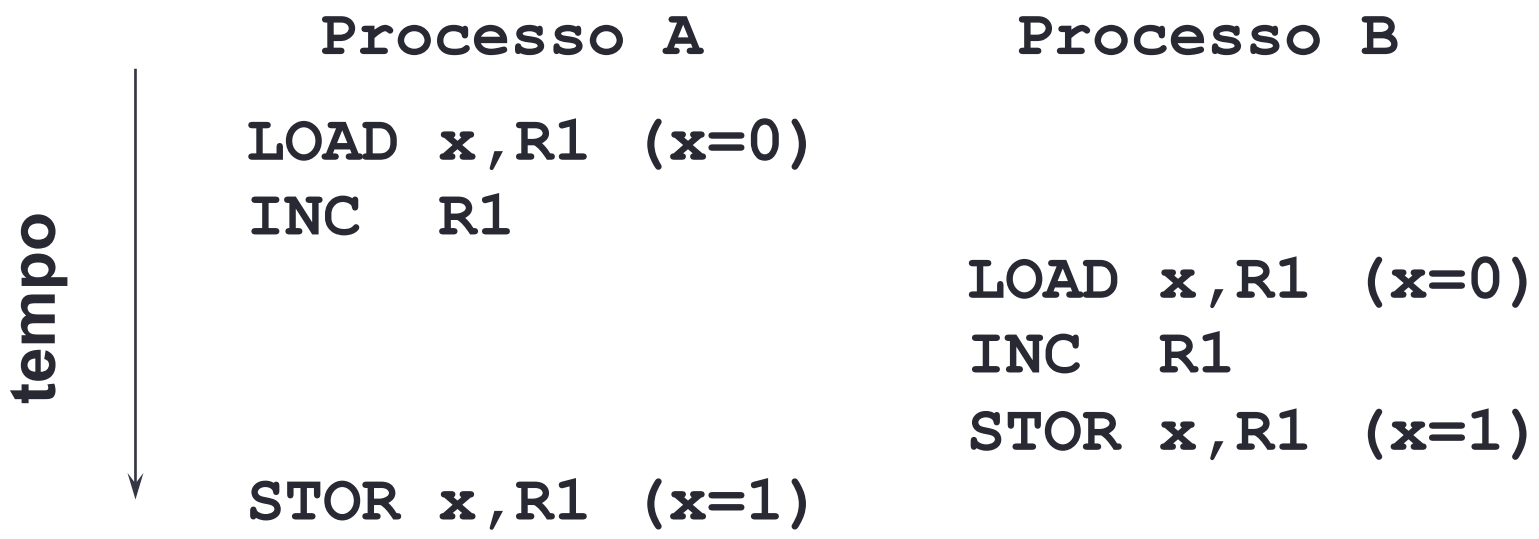
  - Considere que, inicialmente, o valor de x=0. Que valores de x podemos obter quando os processos A e B se executam simultaneamente? (Suponha um escalonador preemptivo).

Caso 1: Processo A -> Processo B



Valor final: x=2

Caso 2:
Processo A -> Processo B -> Processo A



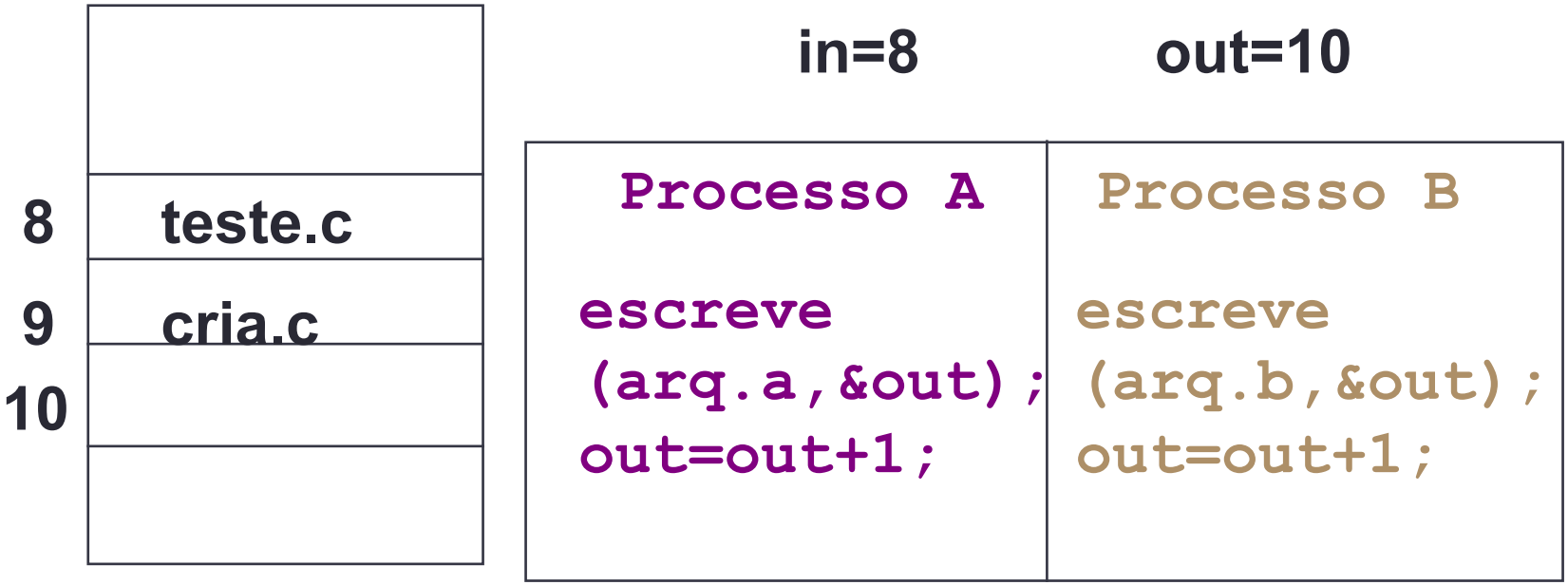
Valor final: x=1

- Esta situação pode ocorrer? Se sim, em que condições?
- Voce considera que o programador admitia, ao fazer o seu programa, que a variavel x pudesse assumir o valor 1 ao final da execução?

Condições de Corrida

- Segundo exemplo de condições de corrida:
  - Para que um arquivo seja impresso, deve-se colocar o seu nome em uma fila em memória. Esta fila é controlada por duas variáveis: *out*, que indica a posição onde o nome do arquivo deve ser escrito e *in*, que indica o arquivo que deve ser impresso no momento.
  - O processo que deseja imprimir arquivos recupera a variável *out,* coloca o nome do arquivo na posição indicada por *out* e incrementa *out* de uma posição.

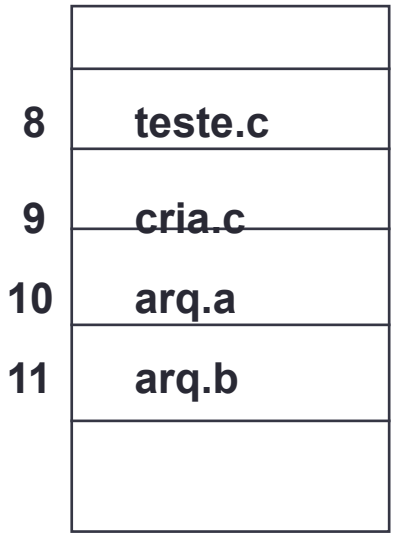
Condições de Corrida
Segundo Exemplo



processo A: escreve arquivo arq.a na fila de impressão
processo B: escreve arquivo arq.b na fila de impressão

processo spool: imprime

Caso 1: Processo A -> Processo B



tempo

Processo A

```
LOAD out,R1 out
(=10)
(coloca arq.a na
posição 10)
INC  R1
STOR out,R1
(out=11)
```

Processo B

```
LOAD out,R1
(out=11)
(coloca arq.b na
posição 11)
INC  R1
STOR out,R1
(out=12)
```

Valores Finais

in=8 out=12

Caso 2:
Processo A -> Processo B -> Processo A

Processo A

```
LOAD out,R1 (out=10)
(coloca arq.a na
posição 10)
```

```
INC  R1
STOR out,R1
(out=11)
```

Processo B

```
LOAD out,R1 (out=10)
(coloca arq.b na
posição 10)
INC  R1
STOR out,R1 (out=11)
```

Valores Finais

in=8 out=11

Condições de Corrida

- As condições de corrida levam geralmente a resultados inesperados e, por isso, devem ser evitadas.
- O que nós precisamos é de um meio de garantir que uma porção de código seja executada somente por um processo de cada vez. Fazemos, assim, uma serialização da execução.

Condições de Corrida

- Esta exclusividade na execução de partes de código é garantida por mecanismos que implementam a exclusão mútua.
- A parte do código protegida pelos mecanismos de exclusão mútua é denominada seção crítica.
- No exemplo 1, a seção crítica é simplesmente a instrução x=x+1. E no exemplo 2?

Exclusão Mútua

- Os mecanismos de exclusão mútua devem garantir que jamais dois ou mais processos estejam dentro da mesma seção crítica simultaneamente.
- A implementação da exclusão mútua pode ser feita de duas maneiras:
  - Espera Ocupada
  - Bloqueio de Processos

Implementação com Espera Ocupada

- Espera ocupada (busy waiting): O processo espera a permissão de entrada na seção crítica em um loop de teste de permissão:

```
while (vez != minha) { }
```

  - A espera ocupada desperdiça CPU!!
  - Só é interessante utilizá-la quando se sabe que a espera será pequena.
  - Em arquiteturas multicore, a espera ocupada é interessante.

Implementação com Bloqueio de Processos

- Bloqueio de processos: O processo que espera a permissão de entrada na seção crítica executa uma primitiva que causa o seu bloqueio até que a seção crítica seja liberada.

```
if (vez != minha)
    sleep_until_it_is_my_turn();
```

  - O bloqueio ocasiona troca de contexto entre processos -> esperas longas

Exclusão Mútua
Requisitos

- Uma boa solução para o problema da exclusão mútua deve garantir os seguintes requisitos:
  1. exclusão mútua: Se o processo P1 está executando na seção crítica, nenhum outro processo pode estar executando nesta seção crítica simultaneamente.
  2. progresso: Se não há processo executando na seção crítica x e existem processos querendo entrar nela, somente os processos que desejam executar a seção crítica devem participar na decisão de que processo pode acessá-la.
  2. espera limitada (bounded waiting): deve existir um limite no número de vezes que outros processos acessam a seção crítica quando 1 processo está esperando para entrar nela.

Técnicas de Implementação da Exclusão Mútua

- Inibir interrupções
- Com espera ocupada:
  - Estrita Alternância
  - Algoritmo de Dekker
  - Algoritmo de Peterson
  - Utilizar hardware adicional
- Com bloqueio de processos:
  - Semáforos
  - Locks
  - Contadores de eventos
  - Monitores
  - Variáveis de Condição

Inibindo Interrupções

- O processo que deseja acessar a seção crítica simplesmente desabilita as interrupções, acessa a seção crítica e habilita novamante as interrupções ao final da execução da seção crítica.
- Desabilitando as interrupções, o processo não pode ser interrompido pelo escalonador pois este retira a CPU dos processos através de interrupções de tempo.

Inibindo Interrupções

```
desabilita_interrupções();
/*executa a seção critica */
x=x+1;
habilita_interrupções();
```

- Características do algoritmo : exclusão mútua, progresso e espera limitada
- Problemas: por ser uma função crítica, a inibição de interrupções geralmente só pode ser executada em modo kernel. O usuário comum não tem acesso a esse tipo de instrução.

Estrita Alternância

- Neste caso, cada processo tem a sua vez de entrar na seção crítica. A vez de entrar na seção crítica é controlada pela variável vez.

```
while (TRUE)
{
  while (vez != 0) { }
  regiao_critica();
  vez = 1;
  regiao_não_critica();
}
```

P1

```
while (TRUE)
{
  while (vez != 1) { }
  regiao_critica();
  vez = 0;
  regiao_não_critica();
}
```

P2

Estrita Alternância

- Características do algoritmo : exclusão mútua.
- O algoritmo permite somente a entrada alternada de dois processos na seção crítica.
- Se os dois processos possuem velocidades diferentes de execução do loop, a execução é não otimizada

Algoritmo de Dekker

- Primeiro algoritmo que permite que dois processos acessem corretamente uma seção crítica sem a alternância estrita. Além da variável `vez`, serve-se de um vetor que indica a intenção de um processo de entrar na seção crítica.
- Os processos são Pi e Pj, onde j=1-i.

Algoritmo de Dekker

```
while (TRUE)
{
  flag[i] = TRUE;
  while (flag[j] != FALSE)
  {
    if (vez == j)
    {
       flag[i] = FALSE;
       while (vez == j){ }
       flag[i] = TRUE;
    }
  }
  secao_critica();
  turn = j;
  flag[i] = FALSE;
}
```

Pi = processo
j = 1 - i

Variáveis globais:flag, vez

Variáveis locais: i,j

Inicialização:
vez=0;
flag[0..1] = FALSE

Algoritmo de Peterson

- Em 1981, Peterson melhorou o algoritmo de Dekker, reduzindo o número de loops e valendo-se de condição de corrida para decidir o acesso à seção crítica.
- O controle é feito em duas rotinas: enter_region e leave_region.
- Os processos continuam sendo Pi e Pj, onde j=1-i.

Algoritmo de Peterson (1/2)

```
int vez;
int interessados[2];
enter_region(int processo)
{
    int outro;
    outro = 1 - processo;
    interessado[processo] = TRUE;
    vez = processo;
    while (vez==processo && interessado[outro] == TRUE) ;
}
leave_region(int processo)
{
    interessado[processo] = FALSE;
}
```

Algoritmo de Peterson (2/2)

```
...
enter_region(p);
  /* seção crítica */
leave_region(p);
...
```

Características do algoritmo:
– preserva a exclusão mútua;
– permite o progresso;
– satisfaz a espera limitada;
– Este algoritmo resolve o problema da seção crítica somente para 2 processos !!

Soluções com Hardware Adicional

- Na maioria das máquinas comerciais, existem algumas instruções de hardware que auxiliam a implementação da exclusão mútua:
- Instruções Test&Set: testam e setam um valor de maneira indivisível.

```
test&set(l): temp=l; l=1;
             return(temp);
reset(l):    l=0;
```

Soluções com Hardware Adicional

- Implementação da Exclusão Mútua com Hardware Adicional:

```
while (test&set(v) != 0){ }
/* seção crítica */
reset(v);
```

Espera Ocupada X Bloqueio de Processos

- Até agora, foram apresentadas soluções que utilizam a espera ocupada para implementar a exclusão mútua.
- No entanto, a exclusão mútua geralmente é implementada com o bloqueio dos processos que esperam a permissão de entrada na seção crítica.

Espera Ocupada X Bloqueio de Processos

- Argumentos em favor do bloqueio de processos:
  - Evitar o disperdício de CPU
  - Evitar o problema da prioridade invertida:
    - Se o algoritmo de escalonamento utilizado é um algoritmo de escalonamento por prioridades estáticas, e se os processos que acessam a seção crítica possuem classes de prioridades diferentes, podemos chegar à seguinte situação:

Espera Ocupada X Bloqueio de Processos

- processo P1 (prioridade baixa) entra na seção crítica
- processo P2 (prioridade alta) entra no loop de teste de acesso à seção crítica
- como a execução do loop não causa o bloqueio de P2, ele só sai pelo quantum mas nesse caso, um processo com a mesma prioridade que a de P2 ou maior será escolhido
- P1 não é executado nunca mais

Implementação da Exclusão Mútua com Bloqueio de Processos

- Geralmente são usadas primitivas usadas para implementar o bloqueio de processos à espera da seção crítica.
- Sempre temos pares de primitivas:
  - Uma primitiva para bloquear o processo quando a seção crítica já estiver ocupada e
  - Outra primitiva para desbloquear os processos à espera da permissão de acesso à seção crítica.

Semáforos

- Os semáforos foram definidos por Dijkstra em 1965.
- Um semáforo é uma variável inteira que admite somente valores não negativos.
- Duas operações indivisíveis podem ser executadas sobre um semáforo:
  - P(sem) ou Down(sem)
  - V(sem) ou Up(sem)

Semáforos

- *Down(sem) ou P(sem):*Decrementa o valor do semáforo se este for maior que 0. Se o valor do semáforo for 0, o processo que executou a operação é bloqueado.

```
P(sem): if (sem > 0)
            sem = sem -1;
        else
            bloqueia(sem);
```

Semáforos

- *Up(sem) ou V(sem)*: Incrmenta o valor do semáforo e acorda os processos que estiverem bloqueados no semáforo.

```
V(sem): if (processos_espera())
          acorda_processo();
        sem = sem +1;
```

Semáforos

- Delimitando uma seção crítica com semáforos:
- Admita que o valor inicial do semáforo *sem* é 1.

```
P(sem);
/* seção crítica */
V(sem);
```

Semáforos

- *Semáforos binários*: só podem assumir os valores 0 ou 1.
- *Semáforos generalizados*: podem assumir também valores negativos
- *Semáforos com busy waiting:* Os semáforos também podem ser implementados de maneira mais simples, com as operações abaixo. Este tipo de implementação não é muito utilizado atualmente.

```
P(sem): while (sem <= 0){ }
        sem = sem - 1;

V(sem): sem = sem + 1;
```

Locks

- Os locks são estruturas de dados compatilhadas que garantem a exclusão mútua por software.
- Em geral, o lock é condedido a um processo de cada vez.
- Operações sobre locks:
  - acquire(l) ou lock(l) : obtem a propriedade do lock l;
  - release(l) ou unlock(l): libera o lock l;

Locks

- As primitivas de acesso aos locks são chamadas de sistema e podem ser implementadas tanto como espera ocupada (spin locks) ou com o bloqueio do processo (locks mutex).
- Delimitando uma seção crítica com locks:

```
acquire(x);
/* secao critica */
release(x);
```

Contadores de Eventos

- Este mecanismo é usado para dar ordem na execução dos processos.
- Três operações são executadas em um contador de eventos:
  - Read(ev): lê o valor corrente de ev;
  - Advance(ev): incrementa ev de 1,
  - Await(ev, v): espera até que ev >= v.
- Os contadores de eventos são monotônicos crescentes.

Monitores

- Até agora, ao programador é dado um conjunto de primitivas e ele deve garantir que estas primitivas são utilizadas corretamente (ex: todo P(s) deve ter um V(s) associado). A programação da concorrência é dita, neste caso, programação de baixo nível.
- Em 1974, Hoare propôs um mecanismo de alto nível para a sincronização de processos, chamado monitor.

Monitores

- Monitor: conjunto de procedures e dados agrupados por módulo. Cada módulo é um monitor. Os processos só podem chamar os procedimentos, nunca acessando diretamente os dados.
- Exemplo de monitor:

```
monitor nome
/* declaracao de variaveis */
procedure p0;
....
end p0;
end monitor;
```

Monitores

- Característica importante do monitor: não pode haver dois processos ativos dentro do monitor simultaneamente.
- O monitor normalmente é uma construção de linguagem de programação.
- Assim, o compilador desta linguagem, ao reconhecer que é uma chamada a monitor e não uma chamada a procedure comum, deve garantir que o processo só executa esta procedure se não houver mais ninguem executando.

Monitores

- Quem se preocupa em garantir a exclusão mútua é o compilador e não o programador.
- A única tarefa do programador é identificar as seções críticas e colocá-las dentro do monitor.
- Para implementar a exclusão mútua dentro de um monitor, o compilador usa geralmente estruturas de baixo nível, como por exemplo, semáforos.
- Problema: o conceito de monitor deve estar contido na linguagem de programação e a maioria das linguagens atuais não suporta este conceito.

Variáveis de Condição

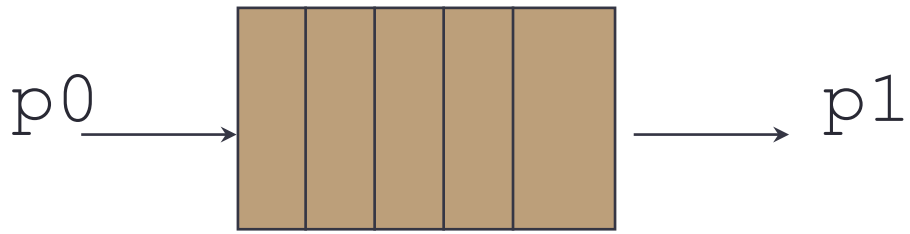
- Variáveis de condição são utilizadas para ordenar a execução de diferentes processos. Elas permitem que um processo se bloqueie esperando uma ação de outro processo.
- Existem duas operações possíveis sobre as variáveis de condição:
  - wait(var): bloqueia o processo em var
  - signal(var): acorda o(s) processo(s) bloqueados em var.

Variáveis de Condição no pacote pthreads

- As variáveis de condição são oferecidas no pacote pthreads (POSIX threads) para sistemas Unix.
- Funcionamento das variáveis de condições no pacote pthreads:
  - Cada variável de condição possui um lock associado.
  - O processo obtem o lock (mutex_lock).
  - O processo libera o lock e espera na variável de condição (cond_wait)
  - Quando a condição desejada ocorre, o processo sinaliza (cond_signal)
  - O processo em wait é acordado e quando ele volta a executar está de posse do lock.

Exemplo de Utilização dos Mecanismos de Exclusão Mútua

- Problema: Produtor-Consumidor
- Neste problema, dois processos p0 e p1 compartilham um buffer de tamanho fixo. O processo p0 escreve dados no buffer e o processo p1 retira dados do buffer.



Exemplo de Utilização dos Mecanismos de Exclusão Mútua

- Como o buffer tem tamanho fixo, devemos ter uma variável que controle o número de mensagens no buffer, para que o processo produtor não escreva no buffer cheio e o processo consumidor não retire dados do buffer vazio.
- O número de mensagens no buffer é uma variável compartilhada e o acesso a ela pode levar a condições de corrida

Produtor/Consumidor com Monitores

```
monitor produtor-consumidor
condicao cheio, vazio;
int cont;
cont = 0;
procedure insere
{
    if (cont == N) wait(cheio);
    insere_item();
    cont++;
    if (cont == 1) signal(vazio);
}
procedure remove
{
     if (cont == 0) wait(vazio);
     remove_item();
     cont--;
     if (cont == N-1) signal(cheio);
}
end monitor;
```

Produtor/Consumidor com Monitores

```
produtor()
{
    while(TRUE)
    {
      produz_item(&item);
      produtor-consumidor.insere();
    }
}
consumidor()
{
    while(TRUE)
    {
      produtor-consumidor.remove();
      consome_item(item);
    }
}
```

Produtor/Consumidor com Semáforos

```
semaphore mutex = 1; /* controla a seção crítica */
semaphore vazio = N; /* vazias */
semaphore cheio = 0; /* ocupadas */
```

```
void produtor()
{
  int item;
  while (TRUE)
  {
    produz_item(&item);
    P(&vazio);
    P(&mutex);
    insere_item(&item);
    V(&mutex);
    V(&cheio);
  }
}
```

```
void consumidor()
{
  int item;
  while (TRUE)
  {
    P(&cheio);
    P(&mutex);
    remove_item(&item);
    V(&mutex);
    V(&vazio);
    consome_item(item);
  }
}
```

Comunicação Por Troca de Mensagens

- Até agora admitimos que os processos cooperantes possuem um meio de compartilhar variáveis.
- Em sistemas onde a memória física é única ou compartilhada entre vários processadores, os mecanismos descritos até então podem ser utilizados.
- Em ambientes distribuídos, onde não existe uma memória física compartilhada, devemos utilizar outro tipo de mecanismo. Foram, então, propostos sistemas baseados em troca de mensagens.

Comunicação Por Troca de Mensagens

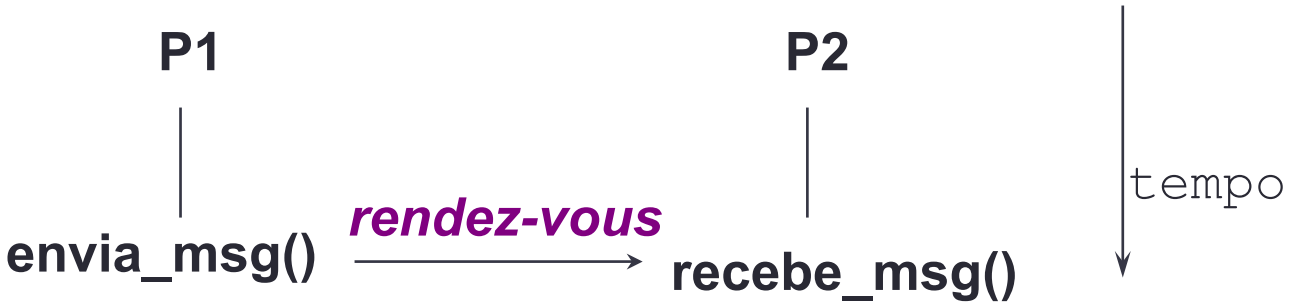
- A troca de mensagens pode ser utilizada tanto em sistemas com memória compartilhada como em sistema com memória distribuída.
- A troca de mensagens é explícita. Um processo envia uma mensagem e outro processo a recebe. Só neste momento, a comunicação é estabelecida.
- Primitivas básicas de troca de mensagens:
  - send: envia uma mensagem
  - receive: recebe uma mensagem

Comunicação Por Troca de Mensagens

- Quando projetamos a comunicação por troca de mensagens devemos nos preocupar com os seguintes aspectos no projeto das primitivas:
  - tipo das primitivas: bloqueadas/não-bloqueadas
  - tipo da comunicação: 1 a 1, n a 1, n a n

Primitivas Bloqueadas x Não Bloqueadas

- A comunicação por troca de mensagem é composta por duas partes: uma que quer enviar mensagens e outra que deseja recebê-las. O momento onde a troca de mensagens realmente ocorre é denominado rendez-vous.

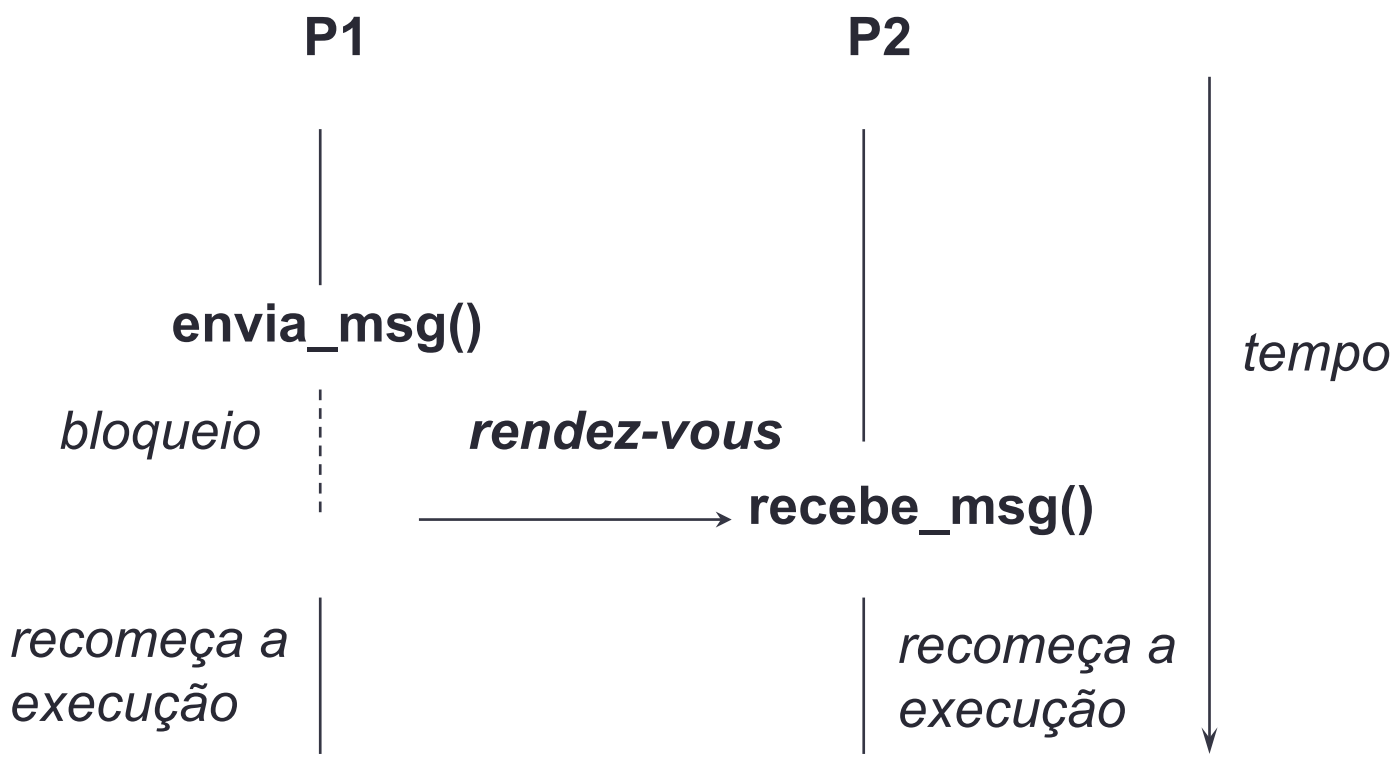


Primitivas Bloqueadas x Não Bloqueadas

- No projeto das primitivas de comunicação, deve-se determinar o comportamento no caso da outra parte não estar em ponto de rendez-vous quando executamos a nossa primitiva.
- Podemos tomar basicamente duas atitudes:
  - esperamos que a outra parte chegue ao ponto de rendez-vous, bloqueando o processo que executou a primeira primitiva até que isso aconteça (primitivas bloqueadas)
  - permitimos que o processo que executou a primeira primitiva continue a sua execução, transmitindo a mensagem somente quando a primitiva correspondente for executada (primitivas não-bloqueadas).

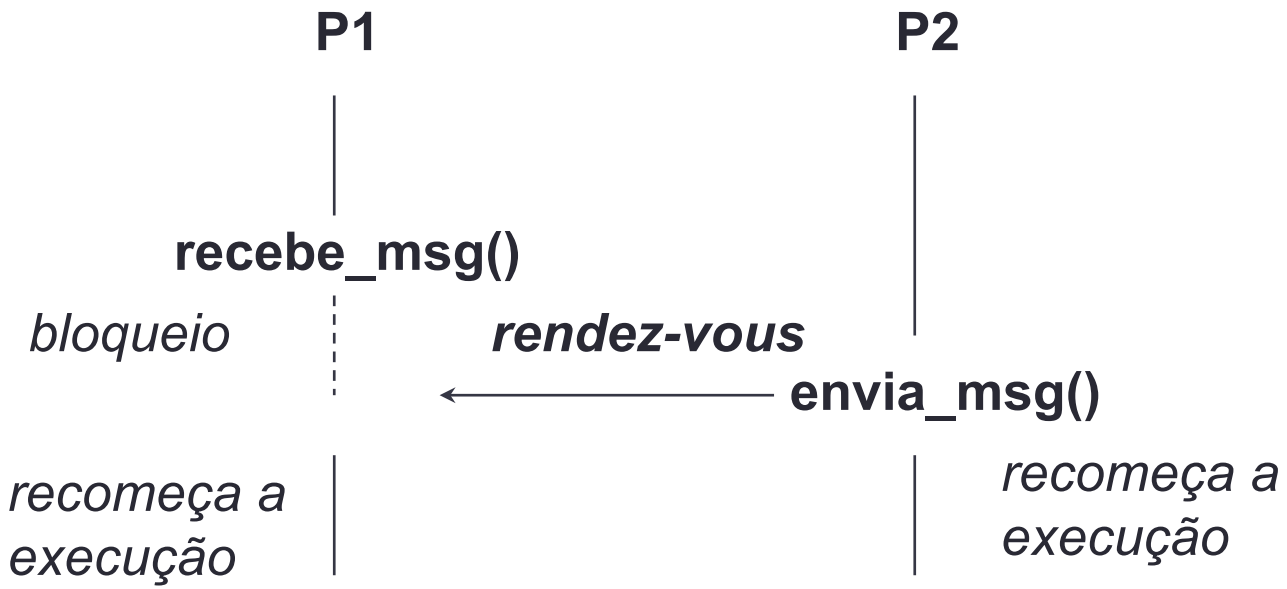
Primitivas Bloqueadas

- Comportamento (send antes do receive)



Primitivas Bloqueadas

- Comportamento (receive antes do send)

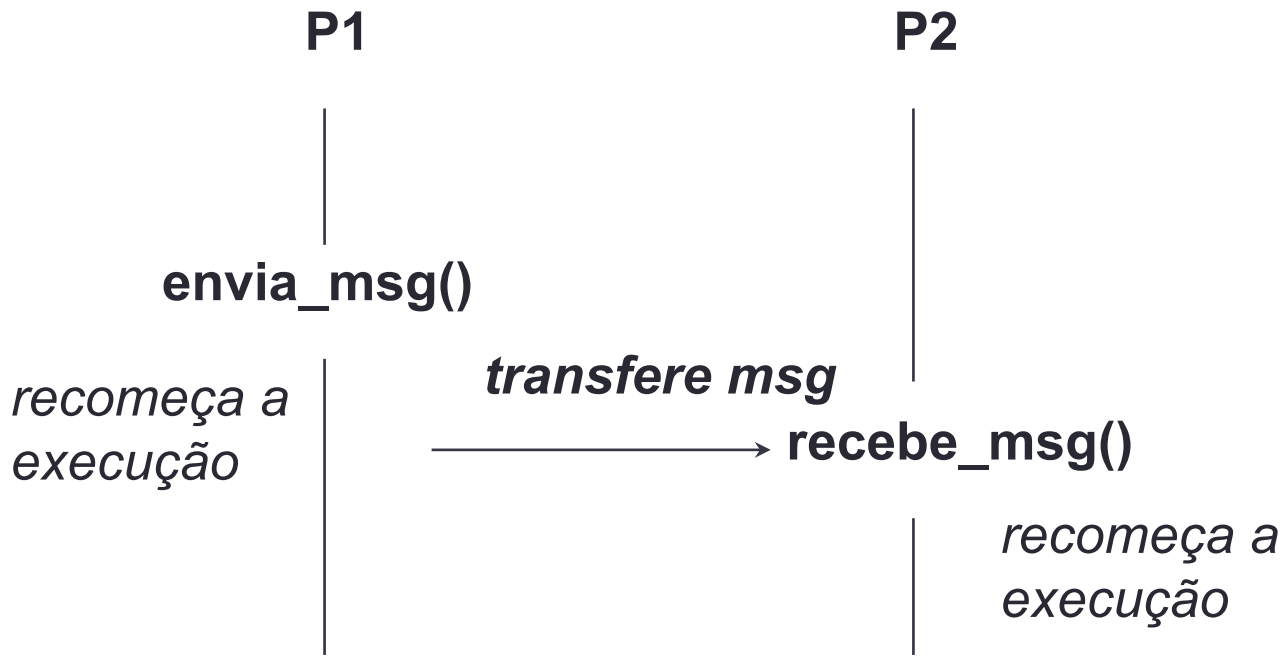


Primitivas Bloqueadas

- A comunicação que utiliza primitivas bloqueadas é dita comunicação síncrona porque, no momento da troca de mensagens, cada processo envolvido na comunicação sabe exatamente em que estado da execução se encontra o outro processo (ele está executando a primitiva de comunicação). Assim, além de comunicar dados, os processos trocam implicitamente informações sobre o seu estado de execução

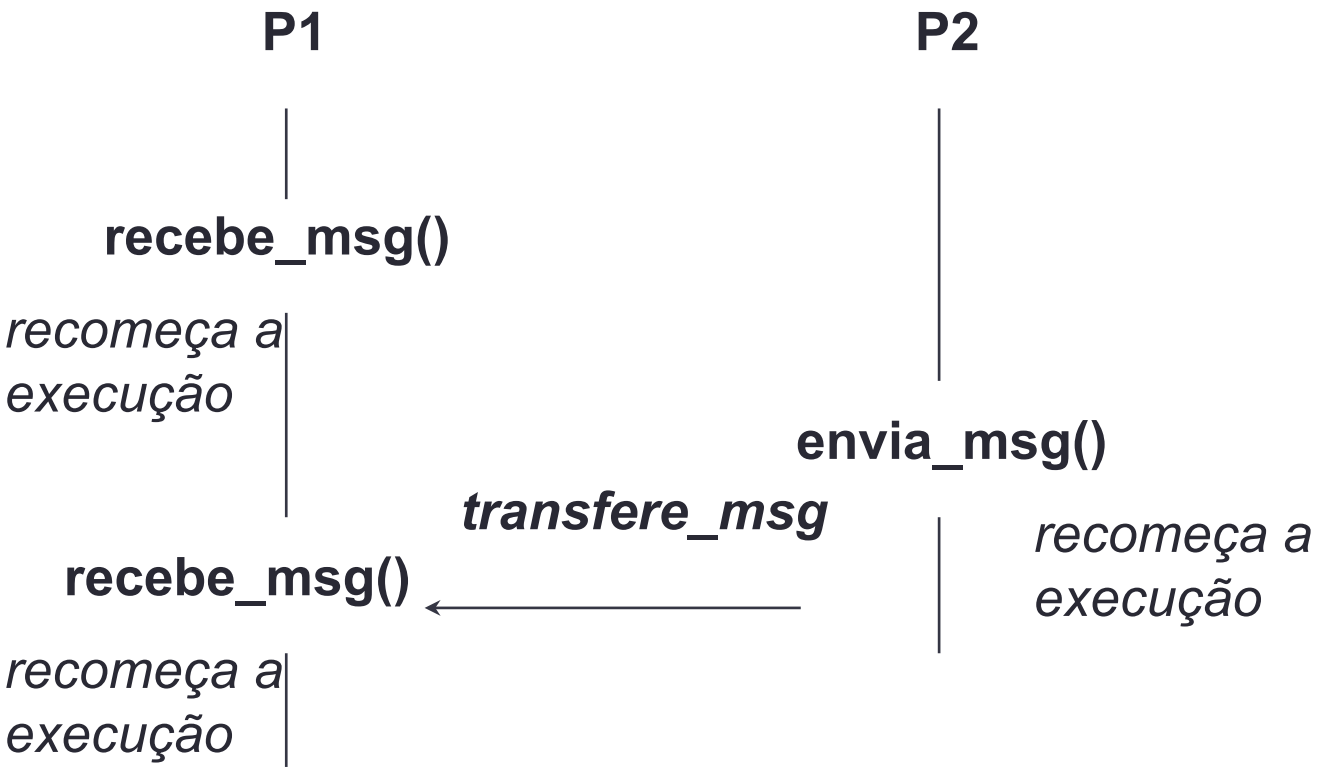
Primitivas Não Bloqueadas

- Comportamento (send antes do receive)



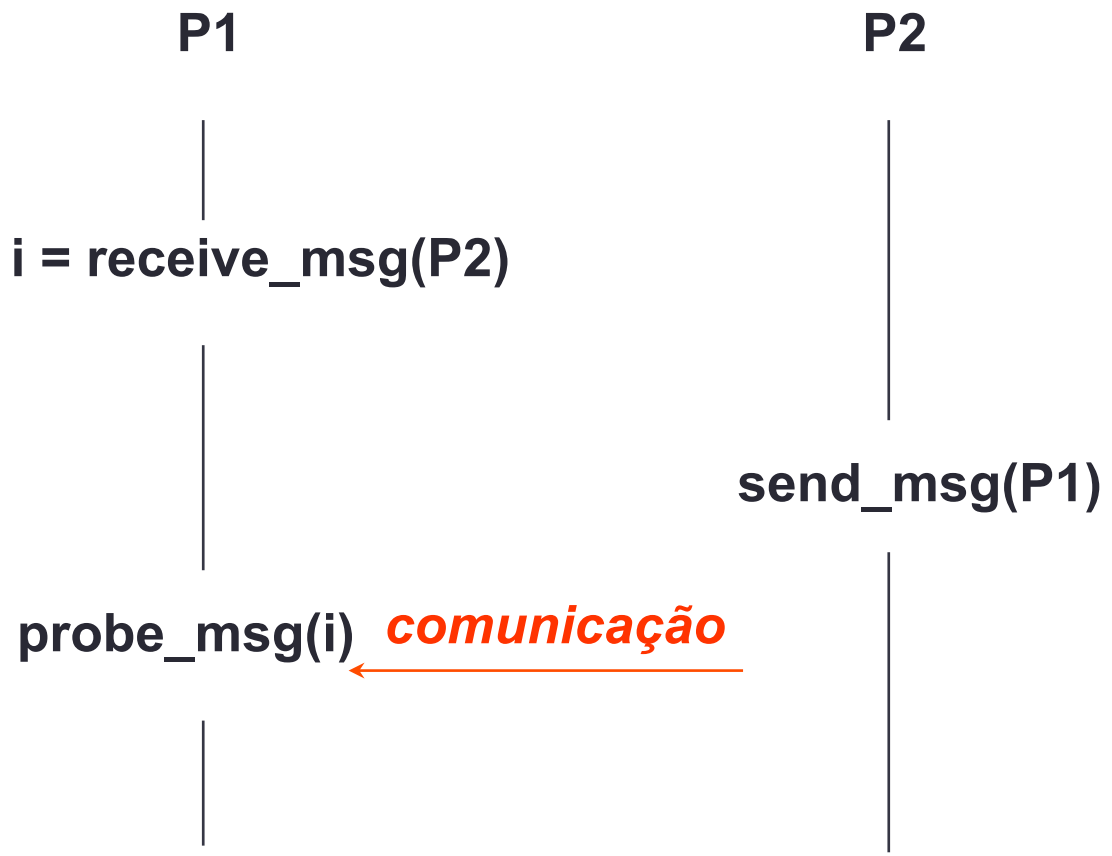
Primitivas Não Bloqueadas

- Comportamento (receive antes do send)
  - Comportamento 1: receive retorna NOMSG



Primitivas Não Bloqueadas

- Comportamento (receive antes do send)
  - Comportamento 2: receive-probe

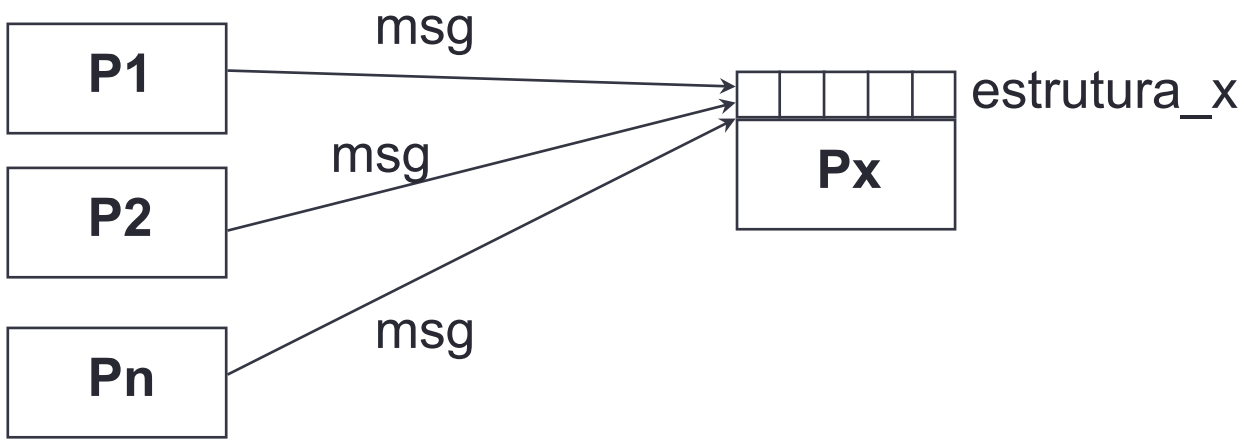


Processos Envolvidos na Comunicação

- **1 a 1**: O processo *x* envia uma mensagem ao processo *y*. O processo *y* recebe somente a mensagem do processo *x*.
  - Exemplo:
    - send(x, &msg);
    - receive(y, &msg);

Processos Envolvidos na Comunicação

- **n a 1**: O processo *x* envia uma mensagem ao processo *y*. O processo *y* recebe mensagem de qualquer processo.
- Exemplo:
  - send(estrutra_x, &msg);
  - receive(estrutura_x, &msg);
- Mecanismos que implementam a comunicação n a 1: portas, caixas-postais restritas.

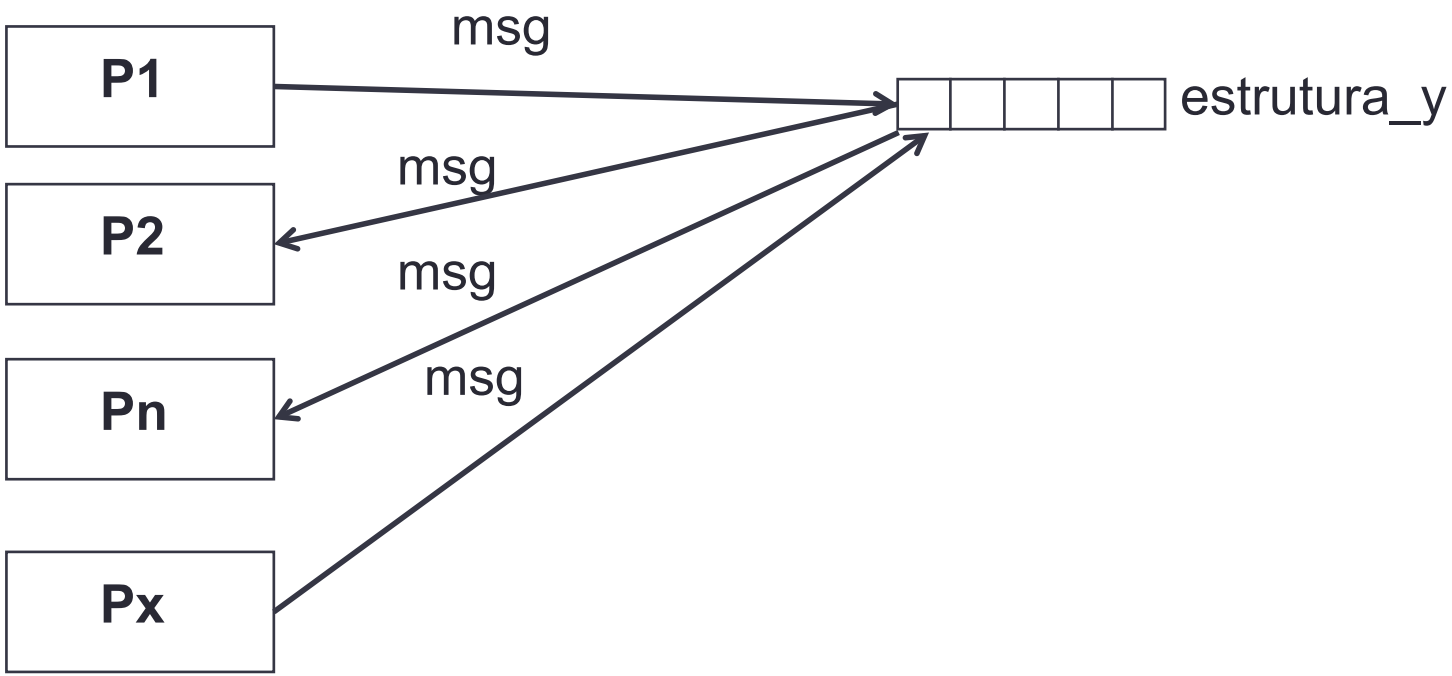


Processos envolvidos na Comunicação

- **n a n**: O processo *x* envia uma mensagem a qualquer processo. O processo *y* recebe mensagem de qualquer processo.
- Exemplo:
  - estrutura_y = aloca_caixa_postal();
  - send(estrutra_y, &msg);
  - receive(estrutura_y, &msg);
- Mecanismos que implementam a comunicação n a n: caixas-postais genéricas.

Processos envolvidos na Comunicação

- Comunicação n a n: Exemplo:



Produtor/Consumidor com Troca de Mensagens

- Neste exemplo, vamos utilizar primitivas síncronas 1 a 1.

```
produtor()
{
   while(TRUE)
   {
      produz_item(&item);
      send(p2, &item);
   }
}
```

```
consumidor()
{
   while(TRUE)
   {
      receive(p1, &item);
      consome_item(item);
   }
}
```