Fundamentos de Sistemas de Operação

Unix Windows NT Netware MacOS DOS/VS Vax/VMS
Linux Solaris HP/UX AIX Mach
Chorus

Programação Concorrente:

Conceitos, Problemas & Soluções: I

Resumo da aula anterior…

- *O processo, conceito oferecido pelo SO,*
  - *"É um contentor" que tem um CPU, uma memória e periféricos idênticos aos da "máquina real". Cada programa corre nesta máquina virtual como se mais nada existisse…*
  - *Mas um processo tem de comunicar - por exemplo, com outros processos; e pode fazê-lo usando memória partilhada.*
  - *A comunicação entre processos usando memória partilhada é complementar à comunicação por troca de mensagens:*
    - *Os processos comunicam lendo e escrevendo em variáveis partilhadas;*
    - *Tais operações, ao contrário do que acontece com a comunicação por troca de mensagens, não requerem cópia de dados entre os espaço endereçamento dos processos, mas colocam questões de sincronização entre os processos, além de outras…*

O problema...

Programa A

```
int deposita(int v)
{ int t;
  return(saldo + v);
}

int main() {
  ...
  saldo= deposita(100);
}
```

Programa B

```
int saldo= 1000;

int levanta(int v)
{ int t;
  return(saldo - v);
}

int main() {
  ...
  saldo= levanta(100);
}
```

- *Suponha que os programas (main) executam uma única vez as rotinas deposita (em A) e levanta (em B)…*
  - *Quanto fica em saldo? R: 900, ou 1000 ou 1100 !!!*

Procurando as razões…


```
int deposita(int v)
{ int t;
  return(saldo + v);
}
```

Código assembly equivalente a `return(saldo + v)`

```
  mov eax, [saldo]
  mov ebx, [ebp+...]
  add eax, ebx
  ret
```

- *Os registos dos "processador virtual" do Processo A são independentes dos do Processo B; apenas a posição* **`saldo`** *é comum; por isso,*

Duas execuções incorrectas…

Processo A

```
mov eax, [saldo] ; saldo= 1000
mov ebx, [ebp+…] ; ebp= 100

add eax, abx     ; eax= 1100
ret
```

Processo B

```
mov eax, [saldo] ; saldo= 1000
mov ebx, [ebp+…] ; ebp= 100

sub eax, abx     ; eax= 900
ret
```

- *Suponha que* **`depositar()`** *em A e* **`levantar()`** *em B se executam "quase simultaneamente" de tal forma que ambos observam o saldo inicial a 1000;*
  - *Então, se* **`depositar()`** *em A retorna primeiro, o valor 1100 é em seguida "esmagado" pelo 900 do* **`levantar()`** *em B,*
  - *ou vice-versa…*

Duas execuções correctas…

Processo A

```
mov eax, [saldo] ; saldo= 1000
mov ebx, [ebp+…] ; ebp= 100

add eax, abx     ; eax= 1100
ret
```

Processo B

```
.
.
.
.
mov eax, [saldo] ; saldo= 1100
mov ebx, [ebp+…] ; ebp= 100

sub eax, abx     ; eax= 1000
ret
```

- *Neste exemplo* **`depositar()`** *em A é executado antes de* **`levantar()`** *em B; o resultado está correcto; o mesmo acontece se* **`levantar()`** *em B for executado antes de* **`depositar()`** *em A.*

As causas…

- *A execução das instruções dos processos A e B é entrelaçada (interleaved) de uma forma não determinística, porque o SO distribui, de forma não determinística, o(s) CPU(s) pelos processos; as causas do não-determinismo são variadas:*
  - *O número de processos no sistema varia ao longo do tempo;*
  - *A comutação de processos (sai um processo do CPU para "dar a vez a outro") pode ocorrer não só ao ritmo do relógio de fatias de tempo (time-slice) - o que é previsível - mas também como resposta a interrupções desencadeadas por periféricos - estas imprevisíveis.*

Um pouco ☺ de formalismo (1)

- *Sejam p e q duas acções, ou eventos; a notação*
  - **`p ; q`** *indica que a acção* **`q`** *só se inicia após* **`p`** *ter acabado; as duas acções executam-se em sequência, são sequenciais…*
  - **`p | q`** *indica que nem a acção* **`q`** *só se inicia após* **`p`** *ter acabado, nem o contrário, i.e., a acção* **`p`** *também não se inicia após* **`q`** *ter acabado! As duas acções são concorrentes.*
- *Exemplo:*
  - *Um exemplo para* **`p`** *ou* **`q`** *pode ser uma das instruções assembly listadas no slide 6.*

Um pouco ☺ de formalismo (2)

- *Sejam P e Q dois processos concorrentes tais que* **P={p_1 ; p_2}** *e* **Q={q_1 ; q_2}**. *Os seguintes traços de execução são possíveis:*
  - **p_1 ; p_2 ; q_1 ; q_2** *(equivalente a* **P ; Q** *)*
  - **q_1 ; q_2 ; p_1 ; p_2** *(equivalente a* **Q ; P** *)*
  - **p_1 ; q_1 ; p_2 ; q_2**
  - **p_1 ; q_1 ; q_2 ; p_2**
  - **q_1 ; p_1 ; q_2 ; p_2**
  - **q_1 ; p_1 ; p_2 ; q_2**

Um pouco ☺ de formalismo (3)

- *Um programa (ou aplicação) concorrente especifica múltiplos fluxos (processos) concorrentes.*
  - *Estará correcto se e só se todos os traços possíveis produzem um resultado correcto!*
- *Se os processos são independentes, a correcção é garantida pelo SO para todos os traços de execução (o SO impede-os de interferirem uns com os outros)*
- *Se os processos são cooperantes, temos de arranjar soluções que impeçam a execução de sequências (traços) que conduzem a resultados incorrectos.*

– Como?

Regiões Críticas e Exclusão Mútua

- *São regiões (ou secções) críticas aquelas em que o código de um conjunto de processos acede a um recurso partilhado, e nas quais a execução entrelaçada das instruções pode conduzir a resultados incorrectos.*
- *Exemplo:*

```
int deposita(int v)
{ int t;
```

Região Crítica

```
  return(saldo + v);
}
```

```
int levanta(int v)
{ int t;
  return(saldo - v);
}
```

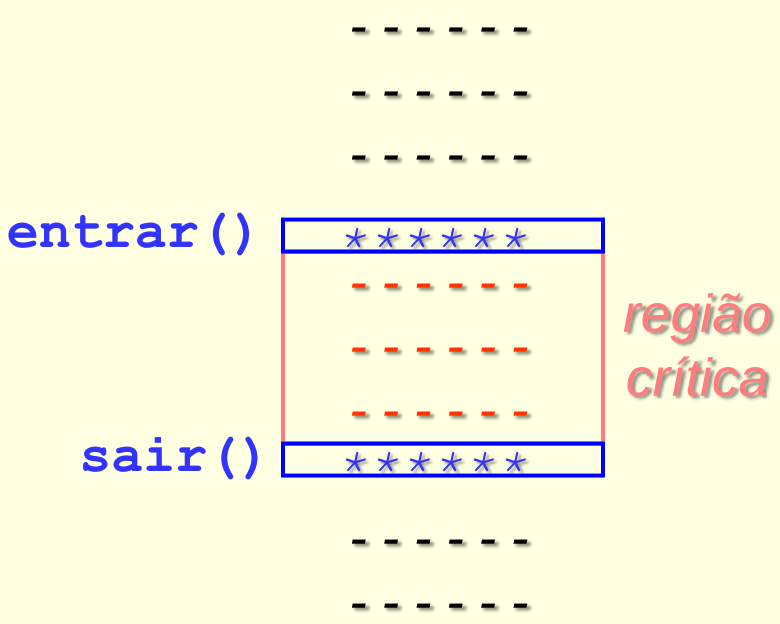
Exclusão Mútua (1)

- *Se a execução entrelaçada das instruções numa Região Crítica pode conduzir a resultados incorrectos, então para os evitar é preciso:*
  - *Impor uma dada ordenação,*

  ou
  - *Impedir alguns dos entrelaçamentos*
- *Exclusão mútua é uma forma de impor uma ordenação, fazendo com que quando um processo executa (as instruções) a "sua" RC, todos os outros sejam impedidos de executar as "suas" RCs.*

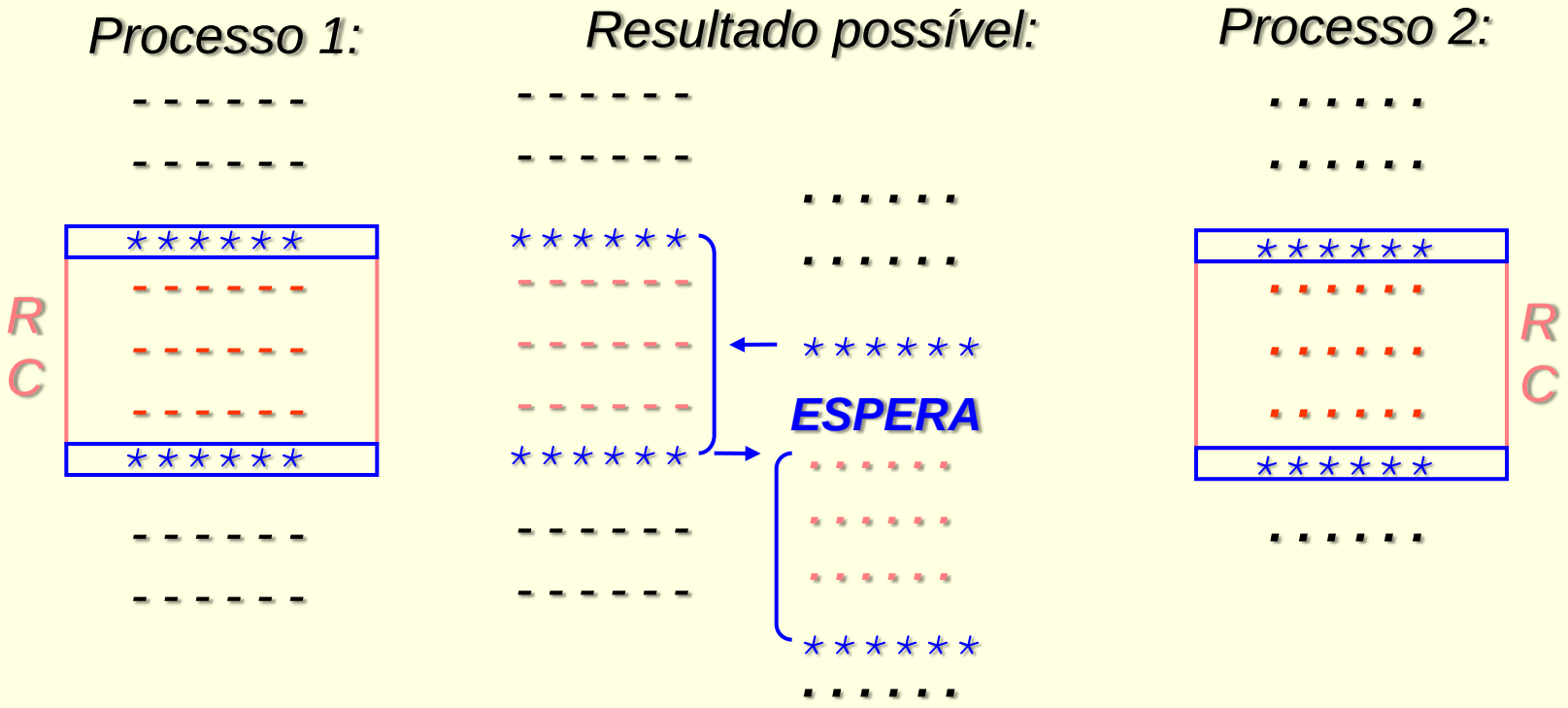
Exclusão Mútua (2)

- *Para definir uma RC usamos duas funções, a que chamaremos por agora* **`entrar()`** *e* **`sair()`**



Exclusão Mútua (3)

- *O objectivo é que cada RC seja atómica*



Exclusão Mútua (4)

- *As acções* **`entrar()`** *e* **`sair()`** *têm de ser "indivisíveis" ou "atómicas", i.e., quando estão a ser executadas o SO não pode interromper o processo que as está a executar e passar o controle a outro que também vai executar uma delas…*
- *E como garantir isso?*
  - *Uma solução é essas funções serem chamadas de sistema! Se é o SO que "passa o controle" de uns processos para outros, nada melhor ☺ que criar uma função do SO para garantir a atomicidade destas*
  - *Mais tarde estudaremos outras soluções, que não necessitam do SO…*

Exclusão Mútua (5)

- *Resumindo, as classes de soluções para suporte de RC são:*

  1. *Código que se executa integralmente em modo utilizador;*

  versus

  2. *Chamadas ao sistema.*

  A. *Soluções que usam espera activa*

  versus

  B. *Soluções que bloqueiam os processos*

- *Contudo apenas as combinações 1A, 2A, e 2B são possíveis.*

Semáforos Binários

- *Tipo abstracto de dados, proposto por E. W. Dijkstra*
  - *Criado com valor inicial 1*
  - *Manipulado por duas operações, P (de "proberen", testar) e V (de "verhogen", incrementar)*
  - **`P(s)`***: bloqueia o processo invocador até que o valor do semáforo* **`s`** *seja 1; quando tal acontecer, coloca-o a 0 e sai da função, continuando a execução.*
  - **`V(s)`***: coloca o valor do semáforo* **`s`** *a 1; quando tal acontecer, se houver processo(s) bloqueado(s) em* **`P(s)`**, *um deles consegue progredir, continuando a execução, enquanto os outros continuam à espera da sua oportunidade ☺*

Semáforos Generalizados

- *Tipo abstracto de dados, proposto por E. W. Dijkstra*
  - *"Contentor" para valores inteiros não-negativos*
  - *Definida uma operação de inicialização e duas de manipulação, P (de "proberen", testar) e V (de "verhogen", incrementar)*
  - **`P(s)`***: bloqueia o processo invocador até que o valor do semáforo* **`s`** *seja positivo (não-nulo); quando tal acontecer, decrementa-o e sai da função, continuando a execução.*
  - **`V(s)`***: incrementa o valor do semáforo* **`s`***; quando tal acontecer, se houver processo(s) bloqueado(s) em* **`P(s)`***, um deles consegue progredir, continuando a execução, enquanto os outros continuam à espera da sua oportunidade ☺*

Aplicações de Semáforos (1)

- *Os semáforos são uma solução generalista que pode ser usada para resolver muitíssimos (?todos?) os problemas de sincronização de processos concorrentes; exemplos:*
  - *Semáforo para resolver problemas de Exclusão Mútua*
    - *Inicializado a 1, "oscila" entre 0 e 1*
  - *Semáforo "contador"*
    - *"Oscila" entre 0 e N (podendo ser inicializado a N e decrescer/crescer entre N e 0 ou, pelo contrário, ser inicializado a 0 e crescer/decrescer entre 0 e N)*
  - *Outros (que não abordamos agora ☺)*

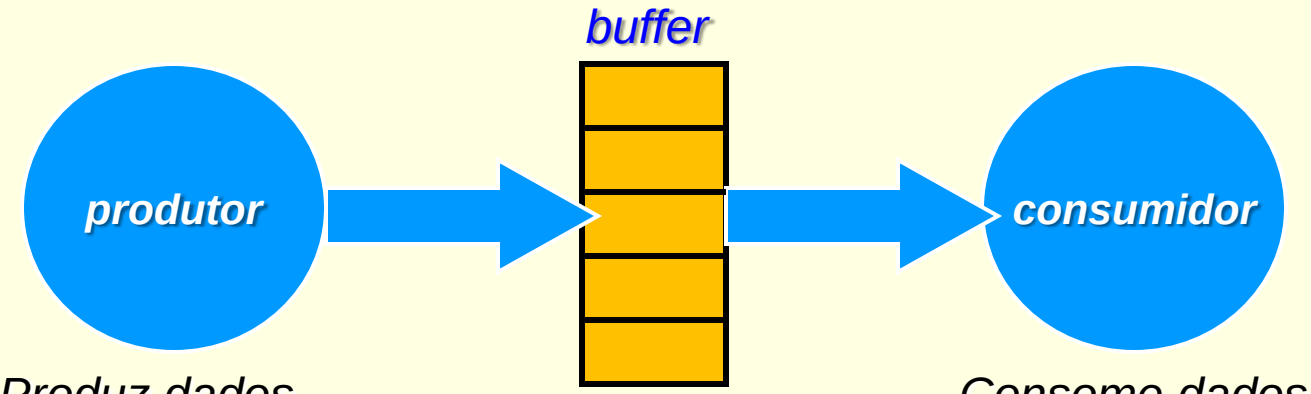
Aplicações de Semáforos (2)

- *Exclusão mútua / Região Crítica*
  - *Seja* **s** *um semáforo inicializado a 1*
  - *O seguinte fragmento realiza uma RC, ou zona de Exclusão Mútua:*

```
P(s)
// Código a executar em EM
V(s)
```

Aplicações de Semáforos (3)

- *Produtor/Consumidor com Buffer de capacidade N*



Produz dados
Coloca-os no buffer

Consome dados
Retira-os do buffer

SINCRONIZAÇÃO

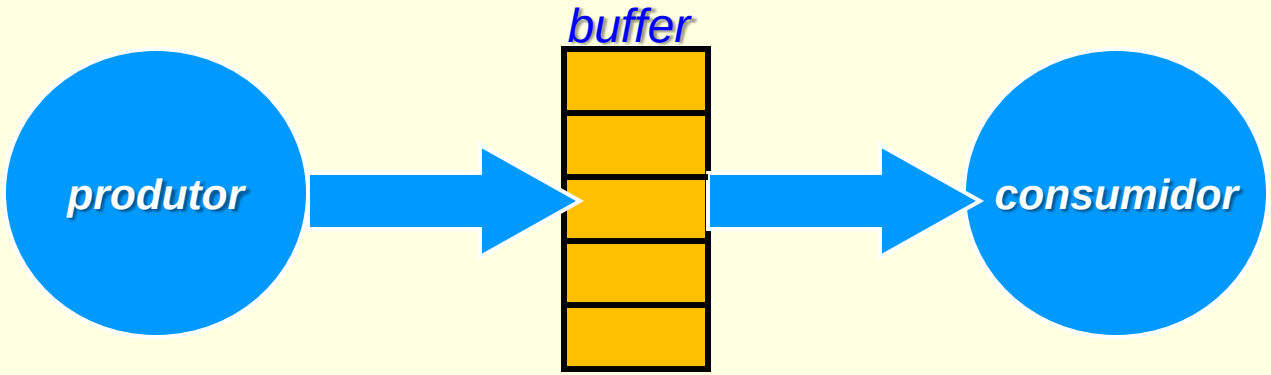
Não pode colocar se buffer cheio! Bloquear!

Não pode retirar se buffer vazio! Bloquear!

Acesso ao buffer em Exclusão Mútua!

Aplicações de Semáforos (4)

- *Produtor/Consumidor com Buffer de capacidade N*

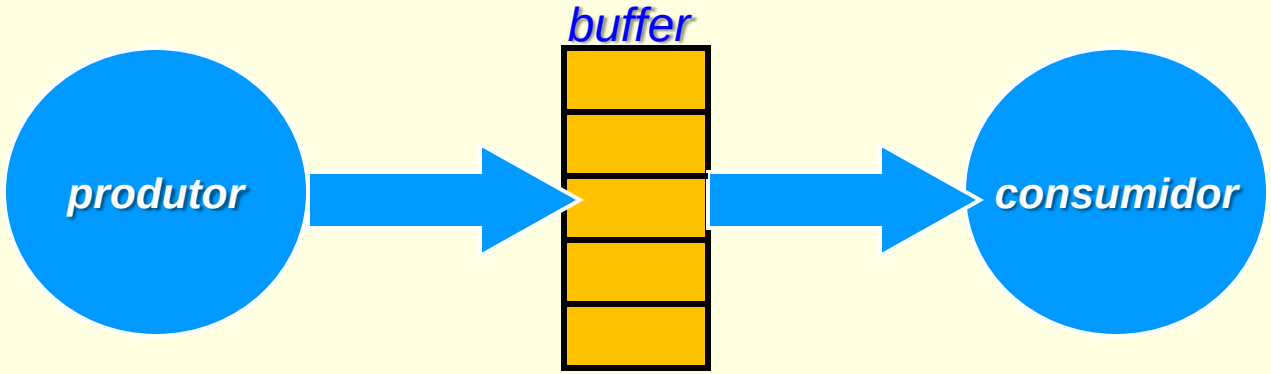



```
Sem slotsCheios= 0;                    Sem slotsVazios= N;

    Produzir();                            P(slotsCheios);
    P(slotsVazios);                RC:     TirarDoBuffer();
RC: PorNoBuffer();                         Consumir();
    V(slotsCheios);                        V(slotsVazios);
```

Aplicações de Semáforos (5)

- *Produtor/Consumidor: exclusão mútua no acesso ao buffer*




```
Sem exmut= 1;
```

```
...
P(exmut);
  PorNoBuffer();
V(exmut);
...
```

```
...
P(exmut);
  TirarDoBuffer();
V(exmut);
...
```

APIs de Semáforos

- *Em sistemas Unix estão disponíveis duas APIs*
  - *Semáforos System V (ou Unix IPC)*
    - *Vectores de semáforos generalizados*
    - *Operações de inicialização e atómicas do tipo P/V sobre vectores de semáforos*
  - *Semáforos POSIX*
    - *Semáforos generalizados "à la Dijkstra"*

Semáforos POSIX (1)

- *Declaração*
  - **`sem_t semEM;`**
- *Inicialização*
  - **`int sem_init(sem_t *s; int pshared, int v);`**

  Inicializa um semáforo **`s`** *com o valor* **`v`**. *O argumento* **`pshared`** *indica se o semáforo é local (partilhado apenas entre threads do processo), ou global (partilhado com outros processos); semáforos globais têm de ser declarados de forma partilhada: por* **`fork()`** *ou em "shared memory".*

Semáforos POSIX (2)

- *P*
  - **`int sem_wait(sem_t *s);`**

  A operação **`sem_wait`** *respeita integralmente a semântica definida por Dijkstra para* **`P`**.

- *V*
  - **`int sem_post(sem_t *s);`**

  A operação **`sem_post`** *respeita integralmente a semântica definida por Dijkstra para* **`V`**.

Semáforos POSIX (3)

- *Remoção*
  - **`int sem_destroy(sem_t *s);`**

  A operação **`sem_destroy`** *torna o semáforo inacessível a outras operações que não (um novo)* **`sem_init`**.

- *Notas finais*
  - *A norma POSIX define outras operações para além das propostas por Dijkstra; por exemplo,* **`sem_getvalue()`** *e* **`sem_trywait()`**. *Tais "extensões" da proposta de Dijkstra não serão aqui abordadas.*
  - *A API apresentada designa-se semáforos POSIX "sem nome" (unnamed); existe uma outra, semáforos POSIX "com nome", que não estudamos aqui.*

Para pensar…

- *Produtor/Consumidor com Buffer de capacidade N*
  - *Realize em C uma aplicação produtor/consumidor, com*
    - *Um buffer (vector de caracteres com dimensão 32) em memória partilhada;*
    - *Semáforos POSIX criados e inicializados no produtor, que depois faz* **`fork()`** *do consumidor...*
    - *O produtor gera as sequências de letras maiúsculas e minúsculas, transferindo uma letra de cada vez para o buffer; o consumidor recebe-as e imprime-as no ecrã.*