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: II

Resumo de aulas anteriores…

- *A API conhecida como IPC (Unix System V) para comunicação entre processos define, entre outros, mecanismos de comunicação como pipes e memória partilhada, bem como semáforos e vectores de semáforos.*
- *A norma POSIX (IEEE 1003.1c) define uma API para gestão de processos leves – criação, controle (prioridade, suspensão, terminar). Define também primitivas de sincronização: mutexes e semáforos generalizados, entre outras.*
- *Os problemas da programação concorrente são desafios complexos, e conceptualmente equivalentes quaisquer que sejam os mecanismos de controle de fluxo (processos vs. threads) e de comunicação (semáforos, mutexes, pipes, etc.) usados.*

Sincronização simples

- Problema: Um “processo” (ou thread) aguarda que outro assinale a ocorrência de um evento.
- Exemplo: B aguarda que A assinale que mudou o valor de uma variável
- Solução: semáforo inicializado a 0
  - B: …
    …
    P(s);
    …

B bloqueia enquanto V(s), que assinala o evento, não acontecer

A: …
…
// evento
V(s);
…

Sincronização (barreira) a dois

- Problema: Dois “processos” (ou threads) aguardam mutuamente que “o outro” assinale a ocorrência de um evento.
- Exemplo: B aguarda que A assinale que lhe fez um pedido (i.e., mudou o valor de uma variável) e A aguarda que B lhe dê a resposta (i.e., mude o valor de uma outra variável)
- Solução: dois semáforos, s_1 e s_2 inicializados a 0
  - B: …
    // evento
    V(s_2)
    P(s_1);
    …

B assinala o seu evento, e bloqueia enquanto o de A não acontecer

A assinala o seu evento, e bloqueia enquanto o de B não acontecer

A: …
// evento
V(s_1)
P(s_2);
…

Barreira a três

- Problema: Três “processos” (ou threads) aguardam que “cada um dos outros” assinale que “chegou à barreira”.
- Solução: generalizando a anterior, chegamos a quatro!!! semáforos, s_1, s_2, s_3 e s_4 inicializados a 0

A: …	B: …	C: …
$P(s_1)$	$V(s_1)$	$P(s_3)$
$V(s_2)$	$P(s_2)$	$V(s_4)$
------- A e B aqui -------		
	$V(s_3)$	
	$P(s_4)$	
	------- B e C aqui -------	
$P(s_1)$		$V(s_1)$
------- A, B e C aqui -------		

Barreira de Sincronização a N

- Problema: Um conjunto de N “processos” (ou threads) aguardam até que todos cheguem a um dado ponto da sua execução - a barreira.
- Exemplo: Um processo P lança N outros, passando a cada um conjunto de dados; cada um dos N calcula o valor máximo do seu conjunto e passa-o a P; quando todos terminam, P calcula o valor máximo final.
- Solução: N+??? semáforos inicializados a 0 ?
  - Parece ser possível, mas intratável ☺ Será que não conseguimos encontrar um algoritmo melhor?!

Mais tarde voltaremos a este caso… ☺

Cliente / Servidor (1)

- *Problema: Um conjunto de "processos" (ou threads), ditos clientes, colocam "pedidos" a que outro(s), o(s) servidor(s), que os processam (podendo ou não "responder" aos clientes).*
- *Exemplo: Um spooler de impressão recebe múltiplos pedidos de aplicações para imprimir numa dada impressora, e efectua ele mesmo a impressão, tratando sequencialmente os pedidos, e libertando rapidamente o cliente.*
- *Solução:*
  - *Uma "fila" de pedidos em memória partilhada*
  - *Um semáforo contador,* **`pedidos`**, *inicializado a 0*
  - *Um semáforo de exclusão mútua,* **`exmut`**

Cliente / Servidor (2)


```
Cliente:
    ...
    P(exmut);
    inserePedido();
    V(exmut);
    V(pedidos);
    ...
   // recebe, ou não
   // a resposta …
```

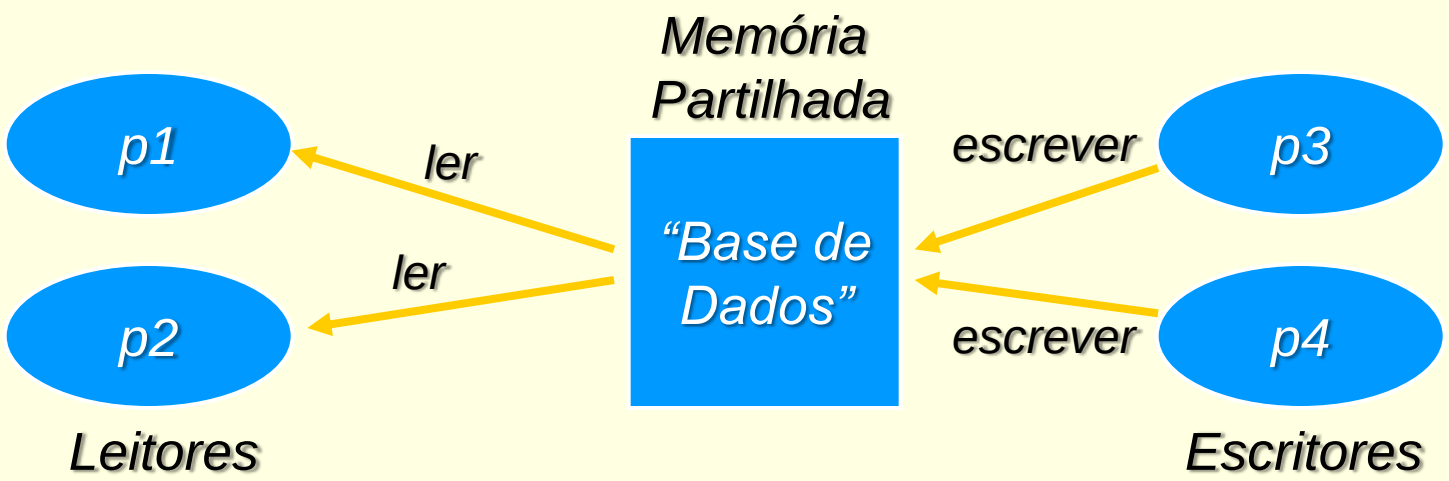
```
Servidor:
    ...
    while (!fim) {
        P(pedidos);
        P(exmut);
        retiraPedido();
        V(exmut);
    }
```

Nota: Que pode acontecer se, no servidor, executar primeiro **`P(exmut)`** *e só depois* **`P(pedidos)`***?*

Leitores / Escritores (1)

- *Problema: Um conjunto de "processos" (ou threads), partilham uma "base de dados" em memória; os leitores apenas consultam os dados, enquanto os escritores os modificam.*
- *Solução: uma boa solução maximiza a concorrência, i.e.,*
  - *Um escritor opera em exclusão mútua, quer seja com leitores quer com escritores.*
  - *Os leitores podem aceder sem restrições*
- *Mas… uma boa solução também*
  - *Tem de garantir que um leitor ou escritor não espera indefinidamente (o que é designado por "starvation") para aceder à "BD"…*

Leitores / Escritores (2)




```
Leitor:
    Pede-para-Ler();    (1)
    Ler();
    Deixa-de-Ler();     (2)
```

```
Escritor:
    Pede-para-Escrever();  (3)
    Escrever();
    Deixa-de-Escrever();   (4)
```

Leitores / Escritores: uma solução (1)

- *Escritores em exclusão mútua → semáforo* **`Wex`**
  - *Ninguém, leitor ou escritor, pode entrar: os escritores tentam obter o semáforo* **`Wexmut`***; para os leitores, ver abaixo*
- *O primeiro leitor obtém o semáforo* **`Wex`**,
  - *Os escritores deixam de conseguir entrar.*
  - *Os outros leitores podem aceder sem restrições, pois não tentam obter o semáforo.*
- *O último leitor liberta o semáforo* **`Wex`**,
  - *Os escritores passam a poder entrar.*

Leitores / Escritores: uma solução (2)


```
R= 0;    // contador de leitores em memória partilhada
Wex      // semáforo E.M. (inicializado a 1)
```

Leitor:

```
R++;
if (R==1)
  P(Wex);
```
(1)

```
Ler();
```

```
R--;
if (R==0)
  V(Wex);
```
(2)

Escritor:

```
P(Wex);
```
(3)

```
Escrever();
```

```
V(Wex);
```
(4)

Leitores / Escritores: uma solução (3)

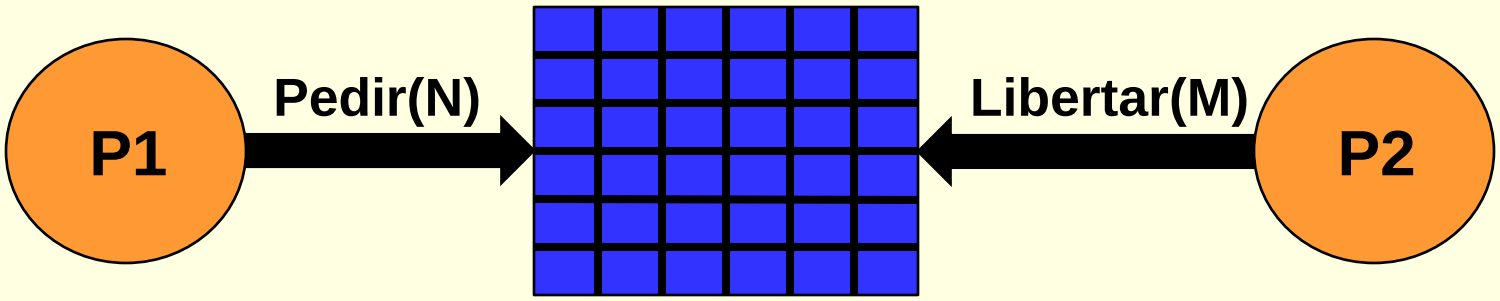

```
R= 0;        // contador de leitores em memória partilhada
Wex, exmut // semáforos; R tem de ser acedido em excl. mútua!!

Leitor:
    P(exmut);
    R++;
    if (R==1) P(Wex);   (1)
    V(exmut);
    Ler();
    P(exmut);
    R--;
    if (R==0) V(Wex);   (2)
    V(exmut);

Escritor:
    P(Wex);             (3)
    Escrever();
    V(Wex);             (4)
```

Leitores / Escritores: para pensar…

- *Problemas da solução encontrada:*
  - *Enquanto há leitores a ler, novos podem entrar imediatamente*
    - *Os leitores passam à frente dos escritores*
    - *Os escritores não conseguem entrar ☹*
- *Proposta de solução:*
  - *Quando chega um escritor este impede que mais leitores possam entrar…*
  - *Tente realizar esta proposta… ou pense noutra ☺*

Reserva de Recursos (1)

- *Problema: Um conjunto de "processos" (ou threads), têm acesso a um reservatório de recursos; um processo reserva (N unidades de um) recurso, usa-o, e finalmente liberta-o.*
- *Exemplo: um reservatório (pool) de memória com uma capacidade de B bytes é usado por processos distintos que lá vão, cada um, buscar um buffer de capacidade N (variável), e, quando não mais precisam destes "devolvem" a capacidade ao reservatório.*



Reserva de Recursos (2)

- *Esboço de uma solução:*
  - *Quando um processo precisa de reservar N unidades do recurso invoca* **`Reservar(N)`**;
  - *Que comportamentos para* **`Reservar(N)`**?
    - *Se não há N unidades disponíveis, devolver erro (o processo pode tentar de novo mais tarde)*

    versus
    - *Se não há N unidades disponíveis, bloquear o processo até que estas estejam disponíveis*
  - *Quando o processo não precisa mais,* **`Libertar(N)`**.

Reserva de Recursos (3)

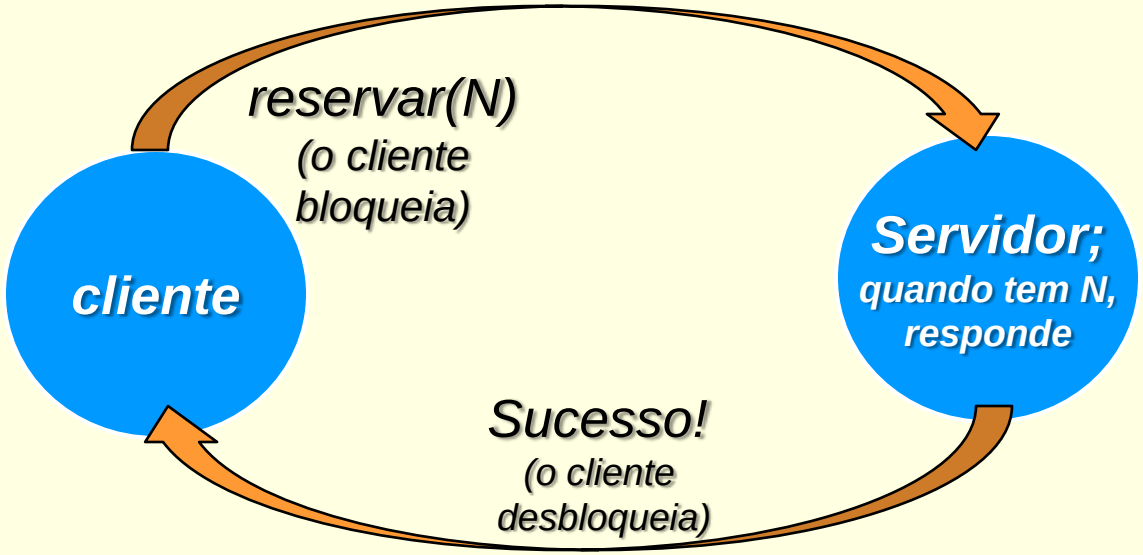
- *Reserva "de uma só vez":*
  - *Não se pode deixar um processo reservar unidades "em tranches". Exemplo: há 3 unidades disponíveis*
    - *P1 quer 2, e tenta:* **`Reservar(1); Reservar(1)`**
    - *P2 quer 3, e tenta:* **`Reservar(1); Reservar(1); Reservar(1)`**
  - *O resultado pode ser que P1 consegue 1 unidade e P2 consegue 2; resultado: nenhum consegue avançar, e temos um deadlock (interblocagem).*
  - *Cada processo deve obter todos os recursos de forma atómica (ou esperar…); i.e.,* **`Reservar(N)`** *é atómica.*

Reserva de Recursos (4)

- *Como realizar* **`Reservar()`**/**`Libertar()`** *?*
  - *Usando o paradigma cliente/servidor:*
    - *Os processos (clientes) tentam* **`Reservar()`***; os pedidos são submetidos a um servidor, que os trata de forma sequencial. Enquanto o cliente não recebe a resposta, permanece bloqueado.*
  - *Os processos interagem directamente uns com os outros, sem intermediários; as operações* **`Reservar()`** *e* **`Libertar()`** *acedem a estruturas de dados partilhadas e decidem o rumo a seguir…*

Reserva de Recursos (5)

- **`Reservar()`/`Libertar()`** *em cliente/servidor*



  - *TPC: realize as operações* **`Reservar()`** *e* **`Libertar()`** *…*

Reserva de Recursos em MP (1)

- *Esboço de uma solução em memória partilhada:*
  - **`int disponível= MAX`**;
  - *Mutex* **`mtx`** *para a variável* **`disponível`**
  - *Semáforo* **`block`** *para bloquear o processo invocador quando não há recursos suficientes.*

Reserva de Recursos em MP (2)

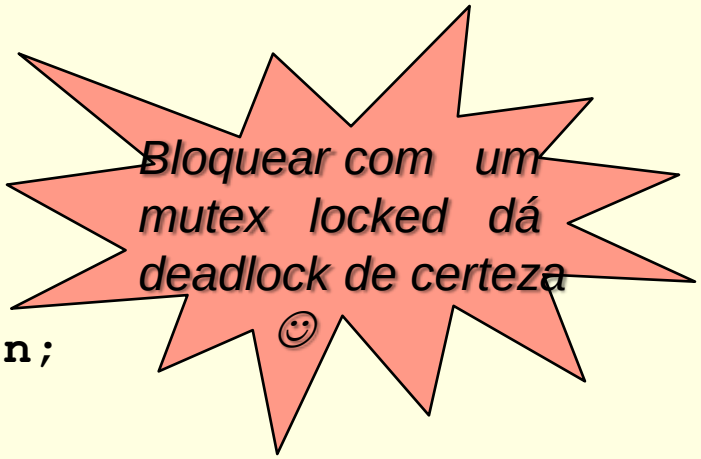

```
Reservar(n) {
  if (n > disponível) {
    ...
  } else
    disponível= disponível – n;
}

Libertar(n) {
  disponível= disponível + n;
  ...
}
```

Reserva de Recursos em MP (3)


```
Reservar(n) {
   P(exm);
     if (n > disponível) {
        ...
     } else
       disponível= disponível - n;
   V(exm);
}

Libertar(n) {
   P(exm);
     disponível= disponível + n;
   V(exm);
}
```

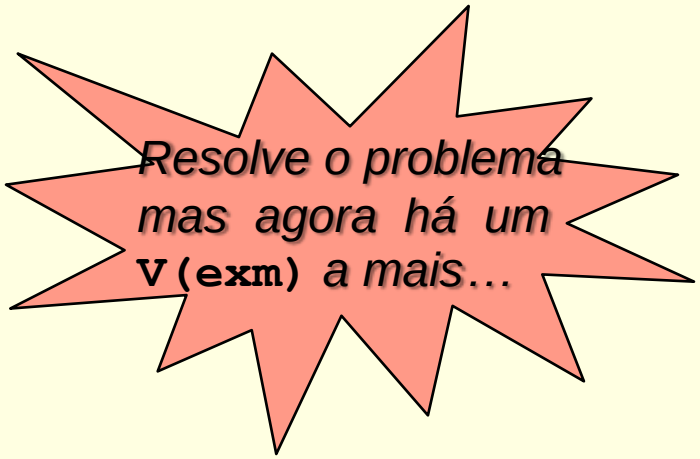
Reserva de Recursos em MP (4)


```
Reservar(n) {
  P(exm);
    if (n > disponível) {
      P(block);
    } else
      disponível= disponível – n;
  V(exm);
}

Libertar(n) {
  P(exm);
    disponível= disponível + n;
    V(block);
  V(exm);
}
```



Reserva de Recursos em MP (5)


```
Reservar(n) {
  P(exm);
  if (n > disponível) {
    V(exm); P(block);
  } else
    disponível= disponível – n;
  V(exm);
}
```



Nota: pode haver vários processos bloqueados; mas o **`V(block)`** *em* **`Libertar(n)`** *só liberta um, e esse pode não ser satisfeito com o número de unidades que foram libertadas…*

Reserva de Recursos em MP (6R)


```
int bloqueados= 0; // em MP


Reservar(n) {
  P(exm);
  if (n > disponível) {
    bloqueados++;
    V(exm); P(block);
  } else
    disponível= disponível - n;
  V(exm);
}
```

Reserva de Recursos em MP (6L)


```
Libertar(n) {
  P(exm);
    disponível= disponível + n;
    while (bloqueados > 0) {
      bloqueados--;
      V(block);
    }
  V(exm);
}
```

Nota: agora **`Libertar()`** *liberta todos os processos bloqueados; mas da forma como construímos* **`Reservar()`** *quando um processo é desbloqueado assumimos que a sua necessidade de unidades foi satisfeita, o que pode não ser verdade…*

Reserva de Recursos em MP (6R)


```
Reservar(n) {
  P(exm);
    while (n > disponível) {
      bloqueados++;
      V(exm); P(block);
    }
    disponível= disponível – n;
  V(exm);
}
```

Nota: agora quando o processo acorda "dentro" de **`Reservar()`** *testa de novo no* **`while`** *se a sua necessidade de unidades foi satisfeita, mas… Oops! Estamos a aceder a* **`disponível`** *mas não estamos numa RC!!!*

Reserva de Recursos em MP (fim)


```
Reservar(n) {
  P(exm);
    while (n > disponível) {
      bloqueados++;
      V(exm); P(block); P(exm);
    }
    disponível= disponível – n;
  V(exm);
}

Libertar(n) {
  P(exm);
    disponível= disponível + n;
    while (bloqueados > 0) { bloqueados--; V(block); }
  V(exm);
}
```

Equilibramos os P() e V() e temos a RC!!!

Para pensar: Reserva de 2 Recursos distintos

- *Problema: Dois "processos" (ou threads), precisam de usar dois recursos; um dos processo reserva um recurso primeiro e depois o outro, enquanto o outro processo também reserva cada um dos recursos (pela mesma ordem? Pode fazê-lo por ordem inversa?).*