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: III

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.*

Barreira a dois (revisão)

- 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 *(revisão)*

- 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?!

Barreira N em MP (1)

- *Esboço de uma solução em memória partilhada:*
  - **`int jaChegaram= 0`**;
  - *Mutex* **`exm= 1`** *para a variável* **`jaChegaram`**
  - *Semáforo* **`block= 0`** *para bloquear o processo invocador quando ainda não chegaram todos à barreira.*

Barreira N em MP (2)


```
void Barreira( int N ) {
int i;

 P(exm);
    jaChegaram++;
    if (jaChegaram < N)
      // Bloqueamo-nos
    else {
      // É o último a chegar:
          - Acordar todos
         jaChegaram= 0;
    }
 V(exm);
}
```

Barreira N em MP (3)


```
void Barreira( int N ) {
int i;

 P(exm);
    jaChegaram++;
    if (jaChegaram < N)
      {  V(exm); P(block); P(exm);  }
    else {
      // É o último a chegar:
          - Acordar todos
         jaChegaram= 0;
    }
 V(exm);
}
```

Solução igual à da reserva de recursos (slide 28)

Barreira N em MP (4)

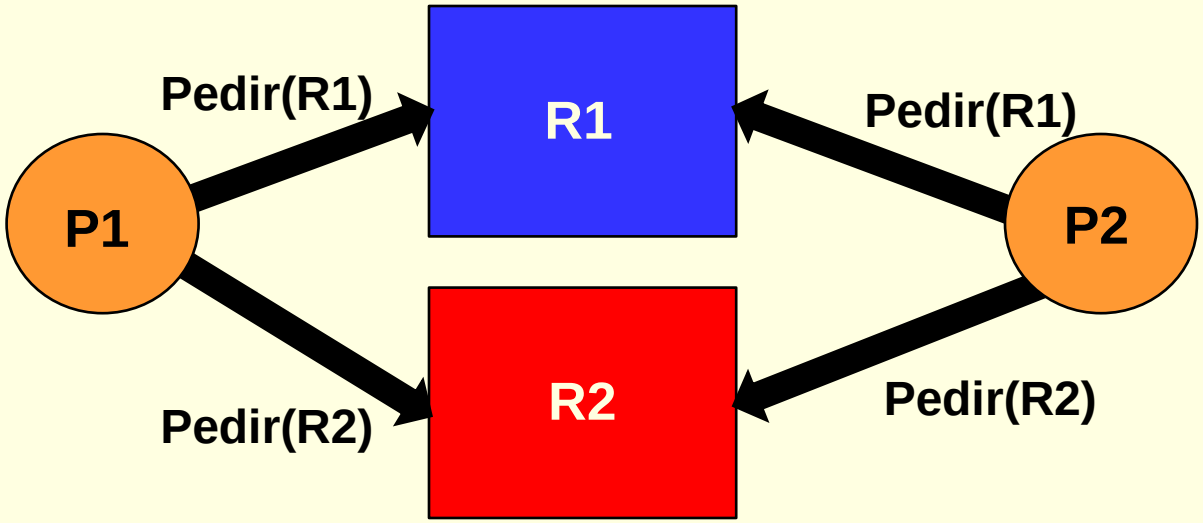

```
void Barreira( int N ) {
int i;

 P(exm);
    jaChegaram++;
    if (jaChegaram < N)
      {  V(exm); P(block); P(exm);  }
    else while ( jaChegaram > 0 ) {
      V(block); jaChegaram--;
    }
 V(exm);
}
```

Solução igual à da reserva de recursos (slide 28)

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?).*



Reserva de 2 recursos em MP (1)


```
semR1= 1; semR2= 1;
```

Uma execução correcta:

```
// Processo 1
   P(semR1);
   P(semR2);
    Acede a
    R1 e R2
   V(semR1);
   V(semR2);
```

```
// Processo 2
      ...
      ...
      ...
      ...
   P(semR2);
   P(semR1);
    Acede a
    R1 e R2
   V(semR2);
   V(semR1);
```

Reserva de 2 recursos em MP (2)

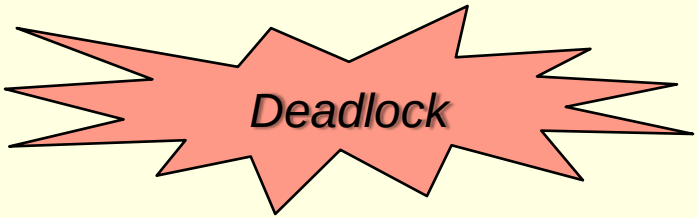

```
semR1= 1; semR2= 1;

// Processo 1                          // Processo 2

   P(semR1);                              P(semR2);
   P(semR2);                              P(semR1);
    Acede a                                Acede a
    R1 e R2                                R1 e R2
   V(semR1);                              V(semR2);
   V(semR2);                              V(semR1);
```

Uma execução com DEADLOCK:

```
   P(semR1);
     ...                                  P(semR2);
   P(semR2);                                ...
                                          P(semR1);
```



Reserva de 2 recursos em MP (3)

- Qual dos cenários descritos anteriormente (1) ou (2) ocorre?
  - Qualquer um é possível, depende da progressão relativa dos 2 processos
- Qual é o problema?
  - A reserva de cada recurso deveria ser uma acção atómica, mas chamamos P() duas vezes…
- Solução 1: vectores de semáforos
  - Oferecidos na API de semáforos do UNIX System V: uma operação (de inicialização, ou de tipo P ou V) pode ser aplicada atomicamente sobre todos os semáforos do vector (pode falhar ☺)

Reserva de 2 recursos em MP (4)

- Solução 2: acesso pela mesma ordem em todos os processos
  - O acesso aos recursos deve ser pedido pelos processos sempre pela mesma ordem (a libertação deve ser por ordem inversa).

```
semR1= 1; semR2= 1;

// Processo 1

   P(semR1);
   P(semR2);
    Acede a
    R1 e R2
   V(semR2);
   V(semR1);
```

```
// Processo 2

   P(semR1);
   P(semR2);
    Acede a
    R1 e R2
   V(semR2);
   V(semR1);
```

Reserva de 2 recursos em MP (5)

- Solução 3: detecção de deadlocks
  - Se as duas soluções anteriores não puderem ser aplicadas, há a possibilidade de deixar que deadlocks possam ocorrer e, nesse caso, detectá-los (preventivamente ou à posteriori) e intervir
  - Pode detectar-se a formação / existência de deadlock por construção do grafo de esperas (X espera por Y que espera por Z e W, que …); a existência de um ciclo indica um deadlock.
  - Detectado um deadlock pode intervir-se de uma de duas formas: impedindo que este se forme (preventivo) ou então sinalizando (eventualmente terminando) o processo que o provocou.

Nota: esta técnica não é usada em APIs de semáforos, mas sim em SGBDs, etc.

Conclusões sobre Semáforos (1)

- Mecanismo "universal": resolve quase todos os problemas de sincronização
- Permite resolver os principais padrões de interacção entre processos:
  - regiões críticas (exclusão mútua)
  - produtores / consumidores, via buffers
  - simples troca de sinais de sincronização
  - controlo da activação de processos
  - leitores / escritores de bases de dados
  - clientes / servidores, via filas de pedidos
  - reserva / libertação de recursos

Conclusões sobre Semáforos (2)

- É ainda um mecanismo que:
  - Não controla a ordem de execução dos processos, sob o controlo de um Sistema de Operação
  - Só é aplicável quando há Memória Partilhada, isto é, não é aplicável entre sistemas ligados por redes de computadores
  - Só admite 3 operações: inicializar/P/V
  - Exige uma operação explícita P() para um processo aguardar um sinal de sincronização de outro
  - A operação P() bloqueia se S não for positivo
  - O valor do semáforo não pode ser testado (Nota: o Unix System V permite isso, mas não se deve utilizar! - Dijkstra dixit)