Laboratório de Pesquisa em Redes e Multimídia

Sincronização de Processos

(Aula 10)

Universidade Federal do Espírito Santo
Departamento de Informática

Concorrência

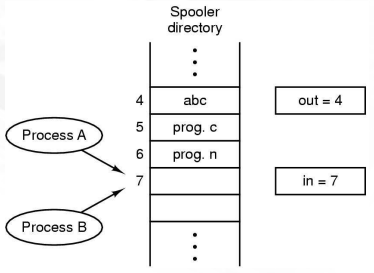
- Dificuldades:
  - Compartilhamento de recursos globais.
  - Gerência de alocação de recursos.
  - Localização de erros de programação (depuração de programas).
- Ação necessária:
  - Proteger os dados compartilhados (variáveis, arquivos e outros recursos globais).
  - Promover o acesso ordenado (controle de acesso) aos recursos compartilhados ⇒ *sincronização de processos*.

Exemplo 1

```
void echo();
{
   chin = getchar();
   chout = chin;
   putchar(chout);
}
```

- P1 invoca *echo()* e é interrompido imediatamente após a conclusão da função *input()*. Suponha que *x* tenha sido o caractere digitado, que agora está armazenado na variável *in*.
- P2 é despachado e também invoca *echo()*. Suponha que *y* seja digitado (*in* recebe *y*), sendo então exibido no dispositivo de saída.
- P1 retoma a posse do processador. O caractere exibido não é o que foi digitado (*x*), pois ele foi sobreposto por *y* na execução do processo P2. Conclusão: o caractere *y* é exibido duas vezes.
- Essência do problema: o compartilhamento da variável global *in*.

Exemplo 2 (1)



- Diretório de *spooler* com *n* entradas, cada uma capaz de armazenar um nome de arquivo.
- Servidor de impressão verifica se existem arquivos a serem impressos. Caso afirmativo, ele os imprime e remove os nomes do diretório.
- Variáveis compartilhadas: *out*, que aponta para o próximo arquivo a ser impresso; e *in*, que aponta para a próxima entrada livre no diretório.

Exemplo 2 (2)

- P_A e P_B decidem colocar um arquivo no *spool* de impressão quase ao mesmo tempo.
- P_A lê *in*, armazena o seu valor (7) na variável local *next-free-slot* e é interrompido.
- P_B é escalonado, lê *in* e coloca o nome do seu arquivo no *slot* 7, atualizando *in* para 8.
- P_A retorna e escreve o nome do seu arquivo na entrada 7 (valor de *next-free-slot*), apagando o nome colocado por P_B. A variável *next-free-slot* passa a valer 8.
- O servidor não notará nada de errado (o diretório está consistente) e P_B nunca realizará qualquer saída.

Exemplo 3

- *Requisito*: manter sempre o relacionamento de igualdade entre os itens de dados *a* e *b*.

```
P1:    a := a + 1;
       b := b + 1;

P2:    b := 2 * b;
       a := 2 * a;
```

- A execução concorrente abaixo não mantém esse requisito:

```
a := a + 1;
b := 2 * b;
b := b + 1;
a := 2 * a;
```


Condições de Corrida

- *Condições de corrida* são situações onde dois ou mais processos acessam dados compartilhados e o resultado final depende da ordem em que os processos são executados (o que, em última instância, é ditado pelo mecanismo de escalonamento do S.O.). Torna a depuração difícil.
- Condições de corrida são evitadas através da introdução de mecanismos de *exclusão mútua*. A exclusão mútua garante que somente um processo estará usando os dados compartilhados num dado momento.
- A parte do programa (trecho de código) que em que os dados compartilhados são acessados é denominada de ***região (ou seção) crítica***.

Interação de Processos (1)

- Processos sem conhecimento uns dos outros:
  - Não são especificamente projetados para trabalharem juntos.
  - Os resultados de um processo são independentes das ações dos outros.
  - Tipo de relacionamento: *Competição*
    - O S.O. precisa resolver a disputa por recursos ⇒ existe uma interação indireta.
  - Potenciais problemas:
    - Acesso não controlado à R.C. (controle de exclusão mútua).
    - *Deadlock*: (P_1,R_2) e (P_2,R_1),ambos precisam de (R_1,R_2).
    - *Starvation*: alternância entre P_1 e P_3 em detrimento de P_2.

Interação de Processos (2)

- Cooperação via compartilhamento
  - Processos podem usar e alterar dados compartilhados mas estão cientes de que outros processos podem acessar esses dados.
  - Preocupação com a integridade dos dados compartilhados (coerência dos dados), além dos problemas de controle de exclusão mútua, *deadlock* e *starvation*.
  - Os resultados de um processo podem depender de informações obtidas por outros.

Interação de Processos (3)

- Cooperação por comunicação
  - Processos participam de um esforço comum na realização de uma tarefa.
  - A comunicação provê uma maneira de sincronizar ou coordenar as várias atividades.
  - Primitivas para o envio e recebimento de mensagens são providas como parte de uma linguagem de programação ou pelo *kernel* do sistema operacional.

Interação de Processos (4)

- Cooperação por comunicação (cont.)
  - Como nada é compartilhado pelos processos não existem condições de corrida. Assim, existência de um mecanismo de exclusão mútua não é um requisito de controle.
  - Pode ocorrer *deadlock*:
    - Cada processo esperando por uma mensagem de um outro processo.
  - Pode ocorrer *starvation*:
    - Dois processos enviando repetitivamente mensagens um para o outro e um terceiro bloqueado, esperando por uma mensagem de um deles.

Abordagens para Exclusão Mútua

- Requisitos para uma boa solução:
  - A apenas um processo é permitido estar dentro de sua R.C. num dado instante.
  - Nenhum processo que executa fora de sua região crítica pode bloquear outro processo (ex: processo pára fora da sua R.C.).
  - Nenhuma suposição pode ser feita sobre as velocidades relativas dos processos ou sobre o número de CPUs no sistema.
  - Nenhum processo pode ter que esperar eternamente para entrar em sua R.C. ou lá ficar eternamente.

Tipos de Soluções

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup, Semáforos, Monitores

Inibição de Interrupções

- Usa um par de instruções do tipo DI / EI.
  - DI = *disable interrupt* EI = *enable interrupt*
- O processo desativa todas as interrupções imediatamente antes de entrar na sua R.C., reativando-as imediatamente depois de sair dela.
- Com as interrupções desativadas, nenhum processo que está na sua R.C. pode ser interrompido, o que garante o acesso exclusivo aos dados compartilhados.

Inibição de Interrupções (2)

```
while (true)
{
   /*disable interrupts*/;
   /*critical section*/;
   /*enable interrupts*/;
   /*remainder*/;
}
```


Problemas da Solução DI/EI

- É desaconselhável dar aos processos de usuário o poder de desabilitar interrupções.
- Não funciona com vários processadores.
- Inibir interrupções por um longo período de tempo pode ter conseqüências danosas. Por exemplo, perde-se a sincronização com os dispositivos periféricos.
  - OBS: inibir interrupções pelo tempo de algumas poucas instruções pode ser conveniente para o *kernel* (p.ex., para atualizar uma estrutura de controle).

Tipos de Soluções

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup, Semáforos, Monitores

Soluções com *Busy Wait*

- *Busy wait* = espera ativa ou espera ocupada.
- Basicamente o que essas soluções fazem é:
  - Quando um processo quer entrar na sua R.C. ele verifica se a entrada é permitida. Se não for, ele espera em um laço (improdutivo) até que o acesso seja liberado.
    - Ex: While (vez == OUTRO) do {nothing};
  - Conseqüência: desperdício de tempo de CPU.
- Problema da inversão de prioridade:
  - Processo *LowPriority* está na sua R.C. e é interrompido. Processo *HighPriority* é selecionado mas entra em espera ativa. Nesta situação, o processo *LowPriority* nunca vai ter a chance de sair da sua R.C.

1a. Tentativa - Variável de Bloqueio

- Variável de bloqueio, compartilhada, indica se a R.C. está ou não em uso.
  - *turn = 0* ⇒ R.C. livre *turn = 1* ⇒ R.C. em uso
- Tentativa para *n* processos:

```
turn =0; /*variável global*/

Process Pi:
...
while (turn == 1) {nothing};
turn = 1;
< critical section >
turn = 0;
...
```


Problemas da 1a. Tentativa

- A proposta não é correta pois os processos podem concluir "simultaneamente" que a R.C. está livre, isto é, os dois processos podem testar o valor de *turn* antes que essa variável seja feita igual a *true* por um deles.

2a. Tentativa – Alternância Estrita

- Variável global indica de quem é a vez na hora de entrar na R.C.
- Tentativa para 2 processos:

```
/*Process 0*/
...
while (turn !=0)
  /*do nothing*/
/*critical section*/
turn = 1;
...
```

```
/*Process 1*/
...
while (turn !=1)
  /*do nothing*/
/*critical section*/
turn = 0;
...
```


Problemas da 2a. Tentativa

- O algoritmo garante a exclusão mútua, mas obriga a alternância na execução das R.C.
- Não é possível a um mesmo processo entrar duas vezes consecutivamente na sua R.C.
  - Logo, a "velocidade" de entrada na R.C. é ditada pelo processo mais lento.
- Se um processo falhar ou terminar, o outro não poderá mais entrar na sua R.C., ficando bloqueado permanentemente.

3a. Tentativa

- O problema da tentativa anterior é que ela guarda a *identificação* do processo que quer entrar na R.C. Entretanto, o que se precisa, de fato, é de informação de *estado* dos processos (i.e., se eles *querem* entrar na R.C.)
- Cada processo deve então ter a sua própria "chave de intenção". Assim, se falhar, ainda será possível a um outro entrar na sua R.C.
- A solução se baseia no uso de uma variável *array* para indicar a intenção de entrada na R.C.

3a. Tentativa

- Antes de entrar na sua R.C, o processo examina a variável *array*. Se ninguém mais tiver manifestado interesse, o processo indica a sua intenção de ingresso ligando o bit correspondente na variável *array* e prossegue em direção a sua R.C.

```
enum boolean (false=0; true=1);
boolean flag[2]={0,0};
```

```
/* Process 0*/:
...
while (flag[1]); /*do nothing*/
flag[0] = true;
< critical section >
flag[0] = false;
...
```

```
/*Process 1*/:
...
while (flag[0]); /*do nothing*/;
flag[1] = true;
< critical section >
flag[1] = false;
...
```

Problemas da 3a. Tentativa

- Agora, se um processo falha fora da sua R.C. não haverá nenhum problema, nenhum processo ficará eternamente bloqueado devido a isso. Entretanto, se o processo falhar dentro da R.C., o problema ocorre.
- Não assegura exclusão mútua, pois cada processo pode chegar à conclusão de que o outro não quer entrar e, assim, entrarem simultaneamente nas R.C.
  - Isso acontece porque existe a possibilidade de cada processo testar se o outro não quer entrar (comando *while*) *antes* de um deles marcar a sua intenção de entrar.

4a. Tentativa

- A idéia agora é que cada processo marque a sua intenção de entrar *antes* de testar a intenção do outro, o que elimina o problema anterior.
- É o mesmo algoritmo anterior, porém com uma troca de linha.

```
/* Process 0*/:                    /*Process 1*/:
...                                ...
flag[0] = true;                    flag[1] = true;
while (flag[1]); /*do nothing*/    while (flag[0]); /*do nothing*/;
< critical section >               < critical section >
flag[0] = false;                   flag[1] = false;
...                                ...
```


Problemas da 4a. Tentativa

- Garante a exclusão mútua mas se um processo falha dentro da sua R.C. (ou mesmo após *setar* o seu *flag*) o outro processo ficará eternamente bloqueado.
- Uma falha fora da R.C. não ocasiona nenhum problema para os outros processos.
- Problemão:
  - Todos os processos ligam os seus *flags* para *true* (marcando o seu desejo de entrar na sua R.C.). Nesta situação todos os processos ficarão presos no *while* em um *loop* eterno (situação de *deadlock*).

5a. Tentativa

- Na tentativa anterior o processo assinalava a sua intenção de entrar na R.C. sem saber da intenção do outro, não havendo oportunidade dele mudar de idéia depois (i.e., mudar o seu estado para "*false*").
- A 5a. tentativa corrige este problema:
  - Após testar no *loop*, se o outro processo também quer entrar na sua R.C, em caso afirmativo, o processo com a posse da UCP declina da sua intenção, dando a vez ao parceiro.

5a. Tentativa (cont.)

```
/* Process 0*/:
...
flag[0] = true;
while (flag[1])
{
  flag[0]= false;
  /*delay*/
  flag[0] = true;
}
< critical section >
flag[0] = false;
...
```

```
/*Process 1*/:
...
flag[1] = true;
while (flag[0])
{
  flag[1]= false;
  /*delay*/
  flag[1] = true;
}
< critical section >
flag[1] = false;
...
```


5a. Tentativa (cont.)

- Esta solução é quase correta. Entretanto, existe um pequeno problema: a possibilidade dos processos ficarem cedendo a vez um para o outro "indefinidamente" (problema da "mútua cortesia")
  - Livelock
- Na verdade, essa é uma situação muito difícil de se sustentar durante um longo tempo na prática, devido às velocidades relativas dos processos. Entretanto, ela é uma possibilidade teórica, o que invalida a proposta como solução geral do problema.

5a. Tentativa – Exemplo

P_0 seta *flag[0]* para *true*.
P_1 seta *flag[1]* para *true*.
P_0 testa *flag[1]*.
P_1 testa *flag[0]*.
P_0 seta *flag[0]* para *false*.
P_1 seta *flag[1]* para *false*.
P_0 seta *flag[0]* para *true*.
P_1 seta *flag[1]* para *true*.

Solução de Dekker

- Trata-se da primeira solução correta para o problema da exclusão mútua de dois processos (proposta na década de 60).
- O algoritmo combina as idéias de variável de bloqueio e *array* de intenção.
- É similar ao algoritmo anterior mas usa uma variável adicional (*vez/turn*) para realizar o desempate, no caso dos dois processos entrarem no *loop* de mútua cortesia.