Comunicação Interprocessos

Condições de disputa: O armazenamento compartilhado pode estar na memória principal (possivelmente em uma estrutura de dados do núcleo) ou em um arquivo compartilhado; o local da memória compartilhada não altera a natureza da comunicação ou dos problemas que surgem. Para ilustrar a questão da disputa é interessa imaginar a situação na qual se deseja imprimir um arquivo, onde um processo entra com o nome do arquivo em um ***diretório de spool*** especial. Um outro processo, o ***daemon de impressão***, verifica periodicamente se há algum arquivo para ser impresso e, se houver, os imprime e então remove seus nomes do diretório.

Situações nas quais dois ou mais processos estão lendo ou escrevendo algum dado compartilhado e cujo resultado final depende das informações de quem e quando executa precisamente – são chamadas de ***condições de disputa*** (race conditions).

Regiões críticas: A parte do programa em que há acesso à memória compartilhada é chamada ***região crítica*** (*critical region)* ou ***seção crítica*** *(critical section)*.

Para lidar com esta situação é preciso encontrar algum modo de impedir que mais de um processo leia e escreva ao mesmo tempo na memória compartilhada. Em outras palavras, precisamos de ***exclusão mútua (mutual exclusion)***, isto é, algum modo de assegurar que outros processos sejam impedidos de usar uma variável ou um arquivo compartilhado que já estiver em uso por um processo. Para atingir tal objetivo existem quatro exigências básicas para que processos paralelos cooperem bem:

1. Nunca dois processos podem estar simultaneamente em suas regiões críticas.
2. Nada pode ser afirmado sobre a velocidade ou sobre o número de CPUs.
3. Nenhum processo executando fora de sua região crítica pode bloquear outros processos.
4. Nenhum processo deve esperar eternamente para entrar em sua região crítica.

Exclusão mútua

Desabilitando interrupções: A solução mais simples é aquela em que cada processo desabilita todas as interrupções logo depois de entrar em sua região crítica e reabilita-as imediatamente antes de sair dela. Com as interrupções desabilitadas, não pode ocorrer qualquer interrupção de relógio. Com as interrupções desligadas, a CPU não será mais alternada para outro processo.

De modo geral, essa abordagem não interessa porque não é prudente dar aos processos dos usuários o poder de desligar interrupções. Além disto é perigoso que a interrupção fique desligada.

Desabilitar interrupções é uma técnica bastante útil dentro do próprio Sistema Operacional (SO), mas inadequada como um mecanismo geral de exclusão mútua para processos de usuário.

Variáveis de impedimento (lock variables): Pode-se tentar utilizar uma variável de controle para entrar na região crítica, assim se um determinado valor for encontrada um processo poderá ou não acessar a zona crítica. Contudo esta solução não resolve o problema da disputa, pois dois ou mais processos podem tentar acessar a mesma variável *lock.*

Alternância obrigatória: Esta solução requer que os dois processos alternem obrigatoriamente a entrada em suas regiões críticas. Para tornar mais claro veja os dois algoritmos abaixo:

```
while(TRUE){                          while(TRUE){
        while(turn != 0)                      while(turn != 1)
            critical_region();                    critical_region();
        turn=1;                               turn=0;
        noncritical_region();                 noncritical_region();
}       Processo 0                    }       Processo 1
```

Inicialmente, o processo 0 inspeciona a variável *turn,* encontra lá o valor 0 e entra em sua região não crítica. O processo 1 também encontra lá o valor 0 e então fica em um laço fechado estando continuamente para ver quando a variável *turn* se torna 1. Testar continuamente uma variável até que algum valor apareça é chamado de ***espera ocioso (busy waiting)***. A espera ociosa deveria em geral ser evitada, já que gasta tempo de CPU. *Somente quando há uma expectativa razoável de que a espera seja breve é que ela é usada*. Uma variável de impedimento que usa a espera ociosa é chamada ***spin lock***.

Imagine agora a situação em que de repente, o *processo 0* termina sua região não crítica e volta ao início de seu laço. Infelizmente, a ele não será permitido entrar em sua região crítica agora, pois a variável *turn* está em 1 e o *processo 1* está ocupado com sua região não crítica. Ele fica suspenso em um laço *while* até que o *processo 1* coloque a variável *turn* em 0. Em outras palavras, alternar a vez não é uma boa ideia quando um dos processos for muito mais lento que o outro.

Essa situação viola a condição 3 estabelecida anteriormente: o processo 0 esta sendo bloqueado por um processo que não está em sua região crítica. Embora esse algoritmo evite todas as disputas, ele não é um candidato realmente sério para uma solução, pois viola a condição 3. Portanto não é uma boa solução.

Solução de Peterson: Antes de usar as variáveis compartilhadas (ou seja, antes de entrar em sua região crítica), cada processo chama *enter_region* com seu próprio número de processo, 0 ou 1, como parâmetro. Essa chamada fará com que ele fique esperando, se for necessário, até que seja seguro entrar. Depois que terminou de usar as variáveis compartilhadas, o processo chama *leave_region* para indicar o término e permitir que outro processo entre.

Veja o algoritmo mais abaixo. Inicialmente, nenhum processo está em sua região crítica. Então o processo 0 chama *enter_region,* que manifesta seu interesse escrevendo em seu elemento do arranjo *interested* e põe a variável *turn* em 0. Como o processo 1 não está interessado, *enter_region* retorna imediatamente. Se o processo 1 chama *enter_region* agora, ele ficará suspenso ali até que *interested[0]* vá para *FALSE*, um evento que ocorre somente quando o processo 0 chama *leave_region* para indicar seu término e permitir que outro processo entre, se assim desejar.

```
#define FALSE 0
#define TRUE   1
#define N            2
int turn;
int interested[N];
void enter_region(int process){                          void leave_region(int process){
    int other;                                               interested[process] = FALSE;
    other = 1 – process;                                 }
    interested[process] = TRUE;
    turn = process;
    while(turn == process && interested[other] == TRUE);
}
```

Instrução TSL: Para usar a instrução TSL, partiremos de uma variável de impedimento compartilhada, *lock*, para coordenar o acesso à memória compartilhada. Quando a variável *lock* for 0, qualquer processo poderá torná-la 1 usando a instrução TSL e então ler ou escrever na memória compartilhada.

```
enter_region:
        TSL REGISTER, LOCK
        CMP REGISTER, #0
        JNE enter_region
        RET
leave_region:
        MOVE LOCK, #0
        RET
```

Dormir e acordar

O problema do produtor-consumidor: O problema origina-se quando o produtor quer colocar um novo item no *buffer*, mas ele já está cheio. A solução é pôr o produtor para dormir e só despertá-lo quando o consumidor remover um ou mais itens. Da mesma maneira, se o consumidor quiser remover um item do *buffer* e perceber que o mesmo está vazio, ele dormirá até que o produtor ponha algo no *buffer* e o desperte. Esse método parece bastante simples, mas acarreta em diversas condições de disputa. Para manter o controle do número de itens no *buffer*, precisaremos de uma variável, *count*. Se o número máximo de itens que o *buffer* pode conter for N, o código do produtor verificará primeiro se o valor da variável *count* é N. Se for, o produtor dormirá; do contrário, o produtor adicionará um item e incrementará a variável *count*. A condição de disputa pode ocorrer pelo fato de a variável *count* ter acesso irrestrito. Seria possível ocorrer a seguinte situação: o *buffer* está vazio e o consumidor acabou de ler a variável *count* para verificar se seu valor é 0.A essência do problema consiste na perca do envio de um sinal de acordar para um processo que (ainda) não está dormindo. Se ele não fosse perdido, tudo funcionaria. Uma solução rápida é modificar as regras, adicionando ao contexto um ***bit de espera pelo sinal de acordar***.

```
#define N 100
int count=0
void producer(void){                          void consumer(void){
  int item;                                     int item;
  while(TRUE){                                  while(TRUE){
     item = produce_item();                  if(count == 0) sleep();
     if(count == N) sleep();                 item = remove_item();
     insert_item(item);                            count = count – 1;
     count = count + 1;                      if(count == N – 1) wakeup(producer);
     if(count == 1) wakeup(consumer);        consume_item(item);
  }                                             }
}                                             }
```

Semáforos: Dijkstra sugeriu usar uma variável inteira para contar o número de sinais de acordar salvos para uso futuro. Semáforo, é uma variável que poderia conter o valor 0 – indicando que nenhum sinal de acordar foi salvo – ou algum valor positivo se um ou mais sinais de acordar estivessem pendentes.

A operação *down* sobre um semáforo verifica se seu valor é maior que 0. Se for, o decrescerá de um (gasta um sinal) e prosseguirá. Se o valor for 0, o processo será posto para dormir, sem terminar o *down*, pelo menos por enquanto. Verificar o valor, alterá-lo e possivelmente ir dormir são tarefas executadas todas como uma única ***ação atômica*** e indivisível. Garante-se, uma vez iniciada uma operação de semáforo, nenhum outro processo pode ter acesso ao semáforo até que a operação tenha terminado ou sido bloqueada. A operação *up* incrementa o valor de um dado semáforo. Portanto, depois de um *up* em um semáforo com processos dormindo nele, o semáforo permanecerá 0, mas haverá um processo a menos dormindo nele. A operação de incrementar o semáforo e acordar um processo é também indivisível.

Resolvendo o problema produtor-consumidor usando semáforos: O modo normal é baseado na implementação de um *up* e *down* como chamadas ao sistema, com o SO desabilitando todas as interrupções por um breve momento enquanto estiver testando o semáforo, atualizando-o e pondo o processo para dormir, se necessário. Como todas essas ações requerem somente algumas instruções, elas não resultam em danos ao desabilitar as interrupções. Se múltiplas CPUs estiverem sendo usadas, cada semáforo deverá deverá ser protegido por uma variável de impedimento, com o uso da instrução TSL para assegurar que somente uma CPU por vez verificará o semáforo. O uso do TSL para impedir que várias CPUs tenham acesso simultâneo ao semáforo é muito diferente da espera ocioso provocada pelo produtor ou pelo consumidor, aguardando que o outro esvazie ou preencha o *buffer*. Essa solução usa três semáforos: um chamado *full* para contar o número de lugares que estão preenchido, um chamado *empty* para contar o número de lugares que estão vazios e um chamado *mutex* para assegurar que o produtor e o consumidor não tenham acesso ao *buffer* ao mesmo tempo.

Em sistemas baseado no uso de semáforos, o modo natural de ocultar interrupções é ter um semáforo inicialmente em 0, associado a cada dispositivo de I/O. Logo depois de iniciar um dispositivo de I/O, o processo de gerenciamento faz um *down* sobre o semáforo associado, bloqueando o processo imediatamente. Quando a interrupção chega, o tratamento de interrupção faz um *up* sobre o semáforo associado, que torna o processo em questão pronto para executar novamente. O semáforo *mutex* é usado para exclusão mútua. Ele é destinado a garantir que somente um processo por vez esteja lendo ou escrevendo no *buffer* e em variáveis associadas.

```
#define N 100
typedef int semaphore;
semaphore mutex = 1, empty = N, full=0;
void producer(void){                          void consumer(void){
        int item;                                     int item;
        while(TRUE){                                  while(TRUE){
          item=produce_item();                          down(&full);
          down(&empty);                                 down(&mutex);
          down(&mutex);                                 item = remove_item();
          insert_item(item);                            up(&mutex);
          up(&mutex);                                    up(&empty);
          up(&full);                                    consume_item(item);
        }                                             }
}                                             }
```

Mutexes: Mutexes são adequados apenas para gerenciar a exclusão mútua de algum recurso ou parte de código compartilhada. Um mutex (*mutual exclusion*) é uma variável que pode estar em um dos dois estados seguintes: desimpedido ou impedido. Quando um *thread* (ou processo) precisa ter acesso a uma região crítica, ele chama *mutex_lock*. Se o mutex estiver desimpedido, a chamada prosseguirá e o *thread* que chamou *mutex_lock* ficará livre para entrar na região crítica.

Se múltiplos *threads* estiverem bloqueados sobre o mutex, um deles será escolhido aleatoriamente e liberado para adquirir o impedimento. O código do *mutex_lock* é similar ao código do *enter_region*, mas com uma diferença fundamental. Quando falha ao entrar na região crítica, o *enter_region* continua testando repetidamente a variável de impedimento (espera ociosa). Ao final, o tempo de CPU se esgota e algum outro processo é escalonado para executar. Cedo ou tarde o processo que detém o impedimento é executado e o libera.

Com *threads*, a situação é diferente porque não há relógio que pare os *threads* que estiverem executando há muito tempo. Consequentemente, um *thread* que tentar obter a variável de impedimento pela espera ociosa ficará em um laço infinito e nunca conseguirá obter essa variável, pois ele nunca permitirá que qualquer outro *thread* execute e libere a variável de impedimento.

Monitores: Unidade básica de sincronização de alto nível, ele é uma coleção de procedimentos, variáveis e estruturas de dados, tudo isso agrupado em um tipo especial de módulo ou pacote. Os processo podem chamar os procedimentos em um monitor quando quiserem, mas não ter acesso direto às estruturas internas de dados do monitor a partir de procedimentos declarados fora do monitor. Os monitores apresentam uma propriedade importante que os torna uteis para realizar a exclusão mútua: somente um processo pode estar ativo em um monitor em um dado momento. Em geral, quando um processo chama um procedimento do monitor, algumas das primeiras instruções do procedimento verificarão se qualquer outro processo está atualmente ativo dentro do monitor. Se estiver, o processo que chamou será suspenso até que o outro processo deixe o monitor. Se nenhum outro processo estiver usando o monitor, o processo que chamou poderá entrar.

Variáveis condicionais, tem duas operações sobre elas: *wait* e *signal*. Quando um procedimento do monitor descobre que não pode prosseguir, emite um *wait* sobre alguma variável condicional. Essa ação resulta no bloqueio do processo que está chamando. Ela também permite que outros processo anteriormente proibido de entrar no monitor agora entre. Esse outro processo pode acordar seu parceiro adormecido a partir da emissão de um *signal* para a variável condicional que seu parceiro está esperando. Basicamente um *signal* só poderá aparecer como último comando de um procedimento do monitor.

Troca de mensagens (message passing): *send* e *receive*, são semelhantes a semáforos mas diferente dos monitores, são chamadas do sistema e não construções de linguagem. Este mecanismo possui alguns tópicos novos, são eles:

- Processos comunicantes podem estar em máquinas diferentes conectadas por uma rede.
- A autenticação
- Desempenho

Barreiras: Dirigido aos grupos de processos. Algumas aplicações são divididas em fases e têm como regra que nenhum processo pode avançar para a próxima fase até que todos os processos estejam prontos a fazê-lo. Isso pode ser conseguido por meio da colocação de uma barreira no final de cada fase. Quando alcança a barreira, um processo permanece bloqueado até que todos os processos alcancem a barreira.