7 SINCRONIZAÇÃO DE PROCESSOS

Quando se trabalha com processos cooperativos, eles podem afetar uns aos outros. Eles compartilham recursos, principalmente posições de memória e arquivos. O acesso a dados compartilhados pode gerar informações inconsistentes. Dessa maneira, é altamente recomendado que haja um mecanismo para ordenar a execução dos processos.

7.1 FUNDAMENTOS

A sincronização dos processos é a parte do estudo dos sistemas operacionais que se preocupa em ordenar as execução dos processos. Isto é necessário porque em muitos casos a ordem de execução dos processos gera diferentes resultados. Muitos desses resultados podem ser danosos para os processos como, por exemplo, perda momentânea de recursos e travamentos.

Imagine uma situação onde há um espaço de memória chamado C compartilhado por dois processos A e B. Neste espaço pertence a uma variável que é incrementada ou decrementada. O processo A tem a função de incrementar o valor contido nesse espaço de memória, enquanto o processo B tem a função de decrementar esta posição de memória.

Fazendo uma analogia com uma indústria de móveis, o processo A é a linha de produção dos móveis que aloca os móveis no estoque, C é o local de estoque e B é o setor de entrega de encomendas que retira os móveis do estoque.

Tanto o setor de produção como o de entregas trabalham simultaneamente. De forma geral, eles são independentes, porém há um elo de ligação entre eles: o estoque. Se não houver mais espaço no estoque, não há como a linha de produção continuar. Já se não houver produtos no estoque, o setor de entregas também pára.

Voltando esta situação para o ambiente computacional, em ambientes monoprocessadores existe uma série de limitações e cuidados a serem tomados. Primeiro, os dois processos não rodam de forma simultânea, mas em pseudo-paralelismo como o uso de um escalonador com característica round-robin.

Ao invés de ser um alento, se analisarmos a estrutura de um computador é um fator de complicação. Vamos especificar um pouco melhor os processos A e B.

1. A usa o registrador AX para incrementar C;
2. B usa o registrador BX para decrementar C;
3. Em linguagem de máquina, a atualização dos registradores ocorre antes da atualização da memória;

4. Vamos supor que em C está o valor 15;
5. Supondo que A atualize AX e, no exato momento em que iria atualizar a memória, A é colocado na fila de processos PRONTOS e B começa a ser executado;
6. B decrementa duas vezes BX e atualiza C, ou seja, C passa a ser 13;
7. Quando A voltar a executar, AX atualizará o valor como 16 e não como 14, gerando um dado inconsistente e falso.

Colocando como uma situação real, onde C é o contador de arquivos a serem impressos. A quantidade de arquivos é limitada em 16. Repare que não será possível naquele momento adicionar, apesar de na realidade haver espaço para mais dois arquivos.

Situações onde a ordem de execução de processos ou *threads* interfere no resultado final são chamadas de *race condition*. Para evitar as *race conditions* é preciso assegurar que somente um dos processos ou *threads* tenha acesso à área compartilhada.

7.2 SEÇÃO CRÍTICA

A primeira atitude a ser tomada é identificar a parte do programa que acessa o recurso compartilhado. Ela é declarada como uma seção crítica, ou seja, ela pode levar a uma race condition. Conseqüentemente, ela deve ser tratada. Uma solução de race condition deve atender a três aspectos:

Exclusão mútua – somente pode estar em execução uma única seção crítica por vez do início até o final;

Progresso – se não houver um processo em sua região critica, mas houver um conjunto de processos ou threads que desejam entrar em suas respectivas regiões críticas, um desses processos ou threads deve ser escolhido para acessar a sua região crítica. Esta decisão não deve demorar;

Espera limitada – nenhum processo pode esperar eternamente para entrar em sua seção crítica;

7.2.1 EXCLUSÃO MÚTUA COM ESPERA OCUPADA

Nesta seção são apresentados mecanismos de garantir a exclusão mútua, onde apenas um processo está na sua Região Crítica.

INIBIÇÃO DE INTERRUPÇÕES

Como primeira solução para implementar a exclusão mútua é inibir as interrupções. Dessa forma, um processo não será suspenso enquanto estiver na sua Região Crítica.

Entretanto, inibir as interrupções não uma opção muito atrativa, uma vez que os processos dos usuários podem não reativar as interrupções.

VARIÁVEIS DE TRAVAMENTO

Uma outra solução é utilizar uma variável de travamento. Quando um processo irá entrar em sua Região Crítica, ele deve verificar o *status* de uma variável, se o seu valor for 0, significa que pode entrar em sua região crítica, caso contrário não pode.

Parece uma boa opção, entretanto, supondo que o processo A leia a variável com valor 0, antes que o seu valor seja modificado ocorre uma interrupção chamando um outro processo B. Então o processo B seta a variável com valor 1 e entra na sua Região Crítica. Porém, quando o processo A voltar a ser executado, para ele o valor da variável é 0 e também entrará em sua região crítica.

ESTRITA ALTERNÂNCIA

A Estrita Alternância refere-se a atribuir a cada processo um valor que o indentificará na lista de processos que é usado em uma variável de estado. Entretanto, esta solução é aplicável a apenas dois processos que desejam entrar em suas Regiões Críticas.

Um processo A é identifica por 0 e um processo B por 1. O A é executado e entra em sua Região Crítica. Quando sair de sua Região Crítica a variável de estado é setada em 1, indicando que é a vez do processo B. Supondo que B não entre na sua Região Crítica, A também não pode entrar na sua. Isto ocorre porque B não executou sua Região Crítica e mudou a variável de estado para 0 novamente.

Um outro ponto negativo, é que um processo que deseja entrar na sua Região Crítica fica constantemente lendo a variável de estado.

INSTRUÇÃO TSL

A instrução TSL é uma chamada de sistema que bloqueia o acesso à memória até o término da execução da instrução.

7.3 BLOQUEIO E DESBLOQUEIO: PRIMITIVAS SLEEP/WAKEUP

As soluções por espera ocupada possuem a série deficiência de consumirem tempo do processador. Outro problema é a prioridade invertida, cuja descrição é dada a seguir:

- Dois processos L e H, de baixa e alta prioridade;

- H está bloqueado e L está na sua região crítica;
- Acaba o *time slot* de L e ele é suspenso;
- H está em estado *Ready*;
- Como H tem maior prioridade de L, então H é executado antes;
- Entretanto, H deseja executar sua região crítica. Como L está na sua região crítica, H entra em *loop* para verificar se pode entrar em região crítica até esgotar o seu *time slot* de tempo
- No entanto, tanto H como L estão em estado *Ready* e sempre H vai ser escolhido para execução;
- Dessa maneira, L nunca será executado e deixará sua região crítica e H ficará em *loop* eterno.

Conseqüência Grave : Todo o sistema computacinal estará bloqueado;

Uma outra solução é o bloqueio e desbloqueio de processos. Esta metodologia propõe o seguinte algoritmo básico:

- Dois processos A e B;
- A entra em sua região crítica;
- A é suspenso pois, terminou seu *time slot*;
- B deseja entrar sua região crítica. Ao invés de entrar em loop, onde ficará verificando se o acesso a sua região crítica está livre até esgotar seu time slot, ele é bloqueado (*Blocked*) e o processador liberado;
- A sai de sua região crítica e desbloqueia B, o qual entra em estado *Ready* e aguarda atentimento para entrar na região crítica;

O mecanismo de bloqueio e desbloqueio utiliza duas primitivas:

1. SLEEP – bloqueia o processo que o chamou;
2. WAKEUP – descbloqueia um processo;

PROBLEMA DO PRODUTOR E CONSUMIDOR

Este problema é também conhecido como problema do *buffer* limitado.

Assume-se a existência de dois processos que compartilham um determinado *buffer,* chamado *count*: um chamado Produtor que cria entidades e aloca no *buffer*; e um outro processo chamado Consumidor que retira entidades do *buffer*.

Problema:

1) - O *buffer* está cheio e o Produtor deseja colocar mais entidades no *buffer*.

2) – O *buffer* está vazio e o Produtor deseja retirar uma entidade do *buffer*.

Solução:

1)- Bloquear o Produtor e convocar o Consumidor.

2)- Bloquear o Consumidor e convocar o Produtor.

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

void consumer(void)
{
        int item;
        while (TRUE)
        {
                if (count = = 0) sleep();
                remove_item(&item);
                count = count -1;
                if (count = = N-1) wakeup(producer);
                consume_item(item);
        }
}
```

Entretanto, esta proposta deixa o acesso a *count*, que é o contador do *buffer*, irrestrito. Uma situação de *race condition* é:

- *Buffer* vazio e o Consumidor ler *count* = 0;
- Neste exato momento, acaba o *time slot* do Consumidor;
- Produtor é convocado e começa a encher o *buffer*;
- Supondo que o *buffer* fique cheio, o Produtor se bloqueia e chama o Consumidor;

- Entretanto, o Consumidor não está bloqueado (*blocked*), mas interrompido (*Ready*);
- A mensagem *Wakeup* é perdida;
- Quando o Consumidor retorna, ele continua a assumir *count* = 0 e bloqueia-se;
- Ambos estão bloqueados esternamente.

Este problema é chamado de Problema de Mensagem Perdida.

7.4 SEMÁFOROS

Uma solução para o Problema da Mensagem Perdida é a adição de contadores de mensagens. Este contador é chamado Semáforo. Um semáforo não pode assumir valor menor que 0. Sua implementação emprega duas primitivas:

- DOWN (correspondente de SLEEP) – que verifica se o valor de um semáforo é maior que 0. Caso seja, seu valor é decrementado e continua a execução do processo. Se for 0, o processo é bloqueado;
- UP (correpondente de WAKEUP) – incrementa o valor do semáforo. Se houver algum ou alguns processos bloqueados, impedidos de chamar a primitiva DOWN, um deles é escolhido para desbloqueio. Então após um UP sobre um semáforo com processos dormindo, o semáforo continua em 0, mas com um processo a menos associado ao semáforo. Nenhum processo é bloqueado ao executar um UP;

As operações de semáforos são atômicas, ou seja não podem ser interrompidas. Sua implementação pode ser realizada através de variáveis de travamento, inibindo as interrupções durante sua execução.

O semáforo *Mutex* é binário (assume 0 ou 1) e é responsável por garantir a exclusão mútua. Enquanto *full* e *empty* controlam o estado do *buffer*;

Solução para o Produtor Consumidor com Semáforos

```
#define N 100
semaphore mutex = 1; - região crítica; semáforo binário
semaphore empty = N; posições vazias
semaphore full = 0; posições ocupadas

void producer(void)
{
        int item;
        while (TRUE)
        {
                produce_item(&item);
```

```
            down(&empty);
            down(&mutex);
            enter_item(item);
            up(&mutex);
            up(&full);
        }
}

void consumer(void)
{
        int item;
        while (TRUE)
        {
            down(&full);
            down(&mutex);
            remove_item(&item);
            enter_item(item);
            up(&mutex);
            up(&empty);
            consume_item(item);
        }
}
```

7.5 CONTADORES DE EVENTOS

Um Contador de Eventos é uma variável implementada para evitar a *race condition* sem exclusão mútua. Em seu conceito é possível apenas três operações sobre um **contador de evento**:

- *Ready*(E) – retorna o valor corrente de E;
- *Advance*(E) – incrementa E;
- *Await*(E,v) – espera até que E tenha um valor maior ou igual a v.

```
#define N 100
event_counter in = 0; conta os itens inseridos no buffer
event_count out = 0; conta os itens retirados do buffer

void producer(void)
{
        int item, sequence = 0
```

```
        while (TRUE)
        {
                produce_item(&item);
                sequence = sequence+1;
                await(out, sequence-N);
                enter_item(item);
                advance(&in);
        }
}

void consumer(void)
{
        int item;
        while (TRUE)
        {
                sequence = sequence+1;
                await(in, sequence);
                remove_item(&item);
                advance(&out);
                consume_item(item);

        }
}
```

7.6 MONITORES

A utilização de semáforos, se for implementada de forma correta, é bastante atrativa. Entretanto, supondo que durante a implementação do problema do Produtor-Consumidor, o programador troque a posição dos *Down(empty)* com *Down(mutex)* no procedimento do Consumidor. Dessa maneira, quando o Consumidor é chamado, *mutex* é decrementado antes de *empty*. Se o *buffer* estiver cheio, o Produtor será bloqueado com *mutex* igual a 0. Consequentemente, quando o Consumidor for executado pela próxima vez, verificará que *mutex* vale 0 e

também ficará bloqueado. Esta situação é chamada ***deadlock***. Ou seja, um único erro na utilização de semáforos e teremos todas as condições de corrida.

Estes problemas ocorrem por causa do contato direto do usuário final com essas primitivas. Estão foi proposto o encapsulamento dessas primitivas em uma estrutura que foi chamada de monitor. As características de um monitor são:

- É implementado em linguagem de alto nível;
- É formado por um conjunto de procedimentos, variáveis e estruturas de dados;
- Nenhum procedimento tem acesso direto às variáveis e estruturas de dados, apesar de poderem chamar os procedimentos;
- Somente um processo, pode estar ativo dentro de um monitor;
- A exclusão mútua pode ser implementada através de semáforos. Abaixo está uma implementação de um monitor para o problema do Produtor-Consumidor.

```
Monitor produtor_consumidor
Condition full, empty;
Integer count;

Procedure enter;
Begin
        If count = N then wait(full);
        Enter_item;
        Count := count+1;
        If count = 1 them signal(empty);
End;

Procedure remove;
Begin
        If count = 0 then wait(empty);
        Remove_item;
        Count := count-1;
        If count = 1 them signal(full);
End;

Procedure produtor;
Begin
        While true do
        Begin
        Produce_item;
        Produtor_Consumidor.enter;
```

```
End;
End;

Procedure consumidor;
Begin
        While true do
        Begin
        Produtor_Consumidor.remove;
        consume_item;
End;
End;
```

Entretanto, a solução das deficiências do semáforo utilizando monitores está longe do ideal. Primeiro, que as atuais linguagens de programação de alto nível não fornecem suporte para este tipo de implementação. Outro fator negativo é que esta proposta não possui suporte para sistemas distribuídos, sendo utilizada apenas para sistemas centralizados, pois nos sistemas distribuídos assume-se que cada processo está em um processador e, a princípio, com sua própria memória. Não existem comunicação entre os processadores.

7.7 TROCA DE MENSAGENS

O primeiro passo para implementar a exclusão mútua em sistemas descentralizados é permitir a comunicação entre os processos. O método por troca de mensagens utiliza duas primitivas:

- Send – envia uma mensagem;
- Receive – recebe uma mensagem;

7.7.1 ASPECTOS DE PROJETO

Em um sistema baseado em troca de mensagens entre processos existem novos tipos de problemas a serem resolvidos na implementação.

- Uma mensagem pode perder-se. Como forma de prevenção o receptor deve comunicar o transmissor de que recebeu a mensagem através de um pacote de reconhecimento.
- O pacote de reconhecimento também pode ser perdido e o transmissor vai enviar uma cópia da mensagem. Para diferenciar uma nova mensagem de uma cópia é recomendado que as mensagens sejam numeradas. Dessa maneira, uma cópia portará o mesmo número da original.

Uma outra questão refere-se a identificação dos processos. Uma possível identificação é processo@máquina. Caso o número de processos seja grande, as máquinas podem ser separadas em domínios. Nesta situação, um processo pode ser identificado como processo@maquina.domínio.

Outra questão refere-se à segurança. Uma entidade mal intencionada pode coletar as mensagens para fins danosos ao sistema.

Produtor Consumidor com troca de mensagens

```
#include <prototypes.h>
#define N 100
#define MSIZE 4

void producer(void)
{
        int item;
        message m;
        while (TRUE)
        {
                produce_item(&item);
                receive(consumer,&m);
                build_message (&m, item);
                send(consumer, &m)
        }
}

void consumer(void)
{
        int item;
        message m;
        for (i= 0; i < N; i++) send(producer, &m);
        while (TRUE)
        {
                receive(producer, &m);
                extract_item(&m, &item);
                send(producer, &m);
                consumer_item(item);
        }
}
```