Aula 09 – Comunicação Interprocessos

Norton Trevisan Roman
Clodoaldo Aparecido de Moraes Lima

25 de setembro de 2014

Espera Ocupada – Instrução TSL

- Test and Set Lock:
  - Instrução TSL RX, LOCK
    - Lê o conteúdo de lock (endereço de memória) e armazena no registrador RX; na sequência armazena um valor diferente de zero em lock
    - Operação indivisível → nenhum outro processador pode acessar a memória até que a instrução seja terminada – bloqueia o barramento de memória
  - Lock é compartilhada
    - Se lock=0, qualquer processo poderá torná-la 1, via TSL → região crítica liberada
    - Se lock≠0, então região crítica ocupada

Espera Ocupada – Instrução TSL

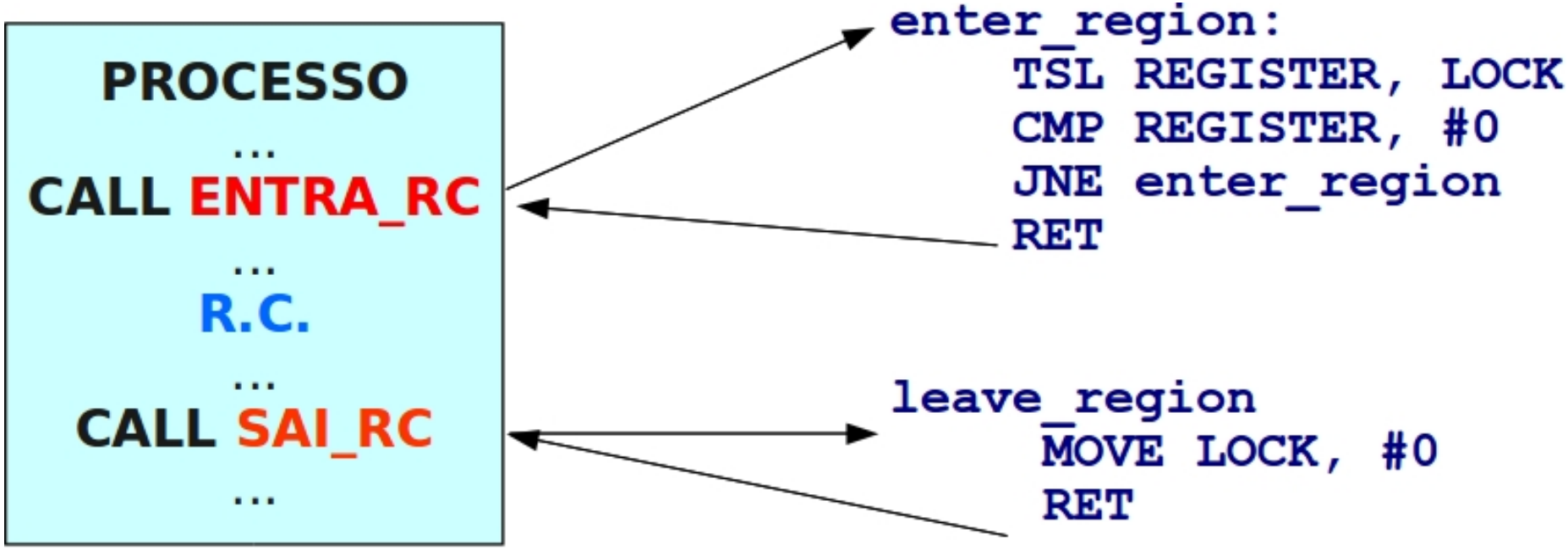
- Como impedir que dois processos entrem simultaneamente em suas regiões críticas?

```
enter_region:
    TSL REGISTER, LOCK  | Copia lock para reg. e lock=1
    CMP REGISTER, #0    | lock valia zero?
    JNE enter_region    | Se não, continua no laço
    RET                 | Retorna para o processo chamador
                        | pois entrou na região crítica

leave_region
     MOVE LOCK, #0      | lock=0
     RET                | Retorna para o processo chamador
```

Espera Ocupada – Instrução TSL

- Uso por processos:



Espera Ocupada – Instrução TSL

- Alternativa (Intel x86):
  - XCHG (Exchange)
    - Troca os conteúdos de duas posições atomicamente

```
enter_region:
        MOVE REGISTER,#1        | insira 1 no registrador
        XCHG REGISTER,LOCK      | substitua os conteúdos do registrador e a variação de lock
        CMP REGISTER,#0         | lock valia zero?
        JNE enter_region        | se fosse diferente de zero, lock estaria ligado, portanto continue no laço de repetição
        RET                     | retorna a quem chamou; entrou na região crítica

leave_region:
        MOVE LOCK,#0            | coloque 0 em lock
        RET                     | retorna a quem chamou
```

Espera Ocupada

- Em suma:
  - Quando um processo deseja entrar na sua região crítica, ele verifica se a entrada é permitida. Se não for, o processo ficará em um laço de espera, até entrar.
- Desvantagens:
  - Desperdiça tempo de CPU
  - Pode provocar "bloqueio perpétuo" (deadlock) em sistemas com prioridades

Deadlock

- Um conjunto de processos está em deadlock se cada processo no conjunto estiver esperando por um evento que somente outro processo no conjunto pode causar
  - Ocorrem tanto em recursos de hardware quanto software
- Muitas vezes, um processo espera por um recurso que outro processo nesse conjunto detém
  - Deadlock de recurso
- Há algoritmos (limitados) para preveni-los
  - Não cobertos nesse curso

Deadlock

Deadlock

Exemplo:

- Suponha 2 processos, um de (A)lta e outro de (B)aixa prioridades
  - O escalonador sempre roda A quando este fica pronto
- Quando B estiver na região crítica e A ficar pronto, ele é escalonado, tendo que esperar pelo B, para entrar em sua região crítica
  - A espera de A pode ser longa, pois B tem baixa prioridade e pode demorar a receber o processador novamente
  - (Problema da Inversão de Prioridade)

Deadlock

Exemplo:

- Note que isso piora se for usada espera ocupada
  - O processo A está sempre rodando (ou na fila de prontos, caso um de maior prioridade rode), pois está sempre executando uma operação legítima (o laço)
  - Potencialmente B nunca será escalonado
  - Tudo isso é evitado se, ao não conseguir entrar na região crítica, A for bloqueado
    - For colocado, pelo S.O., na lista de processos bloqueados
    - Ainda assim, pode esperar demais, para alguém que é prioritário...

Soluções de Exclusão Mútua

- Espera Ocupada (Busy Waiting)
- Primitivas Sleep/Wakeup
- Semáforos
- Monitores
- Troca de Mensagem

Primitivas Sleep/Wakeup

- Para solucionar esse problema de espera, um par de primitivas – Sleep e Wakeup – é utilizado:
  - São chamadas ao sistema para bloqueio e desbloqueio de processos
    - Implementadas via traps
  - Sleep bloqueia o processo que a chamou
    - Suspende sua execução até que outro processo o "acorde" (via wakeup)
    - Não há espera ocupada
  - A primitiva Wakeup é uma chamada de sistema que "acorda" um determinado processo

Primitivas Sleep/Wakeup

- Possível implementação:
  - Sleep bloqueia quem a chama
  - Wakeup tem um parâmetro: o processo a ser acordado
- Alternativamente:
  - Ambas possuem um único parâmetro: um endereço de memória usado para ligação entre um sleep e seu wakeup correspondente

Primitivas Sleep/Wakeup – Exemplo

- Problema do Produtor/Consumidor (bounded buffer):
  - Dois processos compartilham um buffer de tamanho fixo. O processo produtor coloca dados no buffer e o processo consumidor retira dados do buffer
  - Problema: O produtor deseja colocar dados e o buffer está cheio; enquanto que o consumidor deseja retirar dados quando o buffer está vazio
  - Solução: colocar os processos para "dormir"
    - O consumidor acorda o produtor quando remover um ou mais itens
    - O produtor acorda o consumidor quando colocar algo no buffer

Primitivas Sleep/Wakeup – Exemplo

- Problema do Produtor/Consumidor:
  - Situação bastante comum em Sistemas Operacionais
  - Ex: Servidor de impressão:
    - Processos usuários produzem "impressões"
    - Impressões são organizadas em uma fila a partir da qual um processo (consumidor) os lê e envia para a impressora

Primitivas Sleep/Wakeup – Exemplo

- Produtor/Consumidor – Implementação:
  - Buffer:
    - Uma variável count controla a quantidade de dados presente no buffer
  - Produtor:
    - Antes de colocar dados no buffer, o processo produtor checa o valor dessa variável
    - Se a variável está com valor máximo, o processo produtor é colocado para dormir
    - Caso contrário, o produtor coloca dados no buffer e incrementa count

Primitivas Sleep/Wakeup – Exemplo

- Problema do Produtor/Consumidor – Implementação:
  - Consumidor:
    - Antes de retirar dados no buffer, o processo consumidor checa o valor da variável count para saber se ela está com 0 (zero)
    - Se está, o processo vai "dormir"
    - Senão ele retira os dados do buffer e decrementa o contador
  - Tanto o produtor quanto o consumidor sempre testam para ver se o outro deve ser acordado, acordando-o se for o caso

Primitivas Sleep/Wakeup – Exemplo

- Produtor/Consumidor – Sincronização:
  - Para os casos extremos de ocupação do buffer (cheio/vazio), deverão funcionar as seguintes regras de sincronização:
    - Se o produtor tentar depositar uma mensagem no buffer cheio, ele será suspenso (estado de bloqueado) até que o consumidor retire pelo menos uma mensagem do buffer
    - Se o consumidor tenta retirar uma mensagem do buffer vazio, ele será suspenso até que o produtor deposite pelo menos uma mensagem no buffer

Primitivas Sleep/Wakeup – Exemplo

- Problema do Produtor/Consumidor – Código:

```
#define N 100                                   /* número de lugares no buffer */
int count = 0;                                  /* número de itens no buffer */

void producer(void)
{
        int item;

        while (TRUE) {                          /* repita para sempre */
                item = produce_item( );         /* gera o próximo item */
                if (count == N) sleep( );       /* se o buffer estiver cheio, vá dormir */
                insert_item(item);              /* ponha um item no buffer */
                count = count + 1;              /* incremente o contador de itens no buffer */
                if (count == 1) wakeup(consumer);   /* o buffer estava vazio? */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* repita para sempre */
                if (count == 0) sleep( );       /* se o buffer estiver cheio, vá dormir */
                item = remove_item( );          /* retire o item do buffer */
                count = count – 1;              /* decresça de um o contador de itens no buffer */
                if (count == N – 1) wakeup(producer);   /* o buffer estava cheio? */
                consume_item(item);             /* imprima o item */
        }
}
```

Primitivas Sleep/Wakeup – Exemplo

- Produtor/Consumidor – Problemas:
  - Acesso à variável count é irrestrito
    - O buffer está vazio e o consumidor acabou de testar a variável count (valor 0)
    - O escalonador (por meio de uma interrupção) decide que o processo produtor será executado
    - Então o processo produtor insere um item no buffer e incrementa a variável count (valor 1)
    - Imaginando que o processo consumidor está dormindo (pois devia haver 0 em count antes), o processo produtor chama wakeup para acordá-lo

Primitivas Sleep/Wakeup – Exemplo

- Produtor/Consumidor – Problemas:
  - Acesso à variável count é irrestrito
    - No entanto, o processo consumidor não está dormindo (status $\neq$ bloqueado), e não recebe o sinal de wakeup – o sinal faria o SO retirá-lo da fila de bloqueados, colocando-o na de prontos
    - Assim que o processo consumidor é executado novamente, continua de onde parou (executando a chamada a sleep)
    - Nesse instante, o consumidor é colocado para dormir (posto na lista de bloqueados)

Primitivas Sleep/Wakeup – Exemplo

- Produtor/Consumidor – Problemas:
  - Acesso à variável count é irrestrito
    - Mais cedo ou mais tarde, o produtor irá encher o buffer, indo dormir
    - Ambos os processos dormem para sempre...

  - A essência do problema está no fato de, por não estar dormindo (ou seja, na fila de processos bloqueados), o consumidor ter perdido o sinal de wakeup
    - O problema ocorre entre a checagem da variável e a chamada a sleep

Primitivas Sleep/Wakeup – Exemplo

- Produtor/Consumidor – Solução:
  - Cria-se um "bit de espera de wakeup"
    - Quando um Wakeup é mandado a um processo ainda acordado, este bit é feito 1
    - Depois, quando o processo tenta ir dormir, se o bit de espera de Wakeup estiver ligado, este bit será desligado, e o processo será mantido acordado
  - No entanto, no caso de vários pares de processos, vários bits devem ser criados, sobrecarregando o sistema!!!!
    - Deve haver uma solução melhor... Semáforos

Soluções de Exclusão Mútua

- Espera Ocupada (Busy Waiting)
- Primitivas Sleep/Wakeup
- Semáforos
- Monitores
- Troca de Mensagem

Semáforo

- Variável inteira utilizada para controlar o acesso a recursos compartilhados
  - Sincronizar o uso de recursos em grande quantidade
  - Nasceu (Dijkstra. 1965) como proposta para contar o número de wakeups armazenados para uso futuro
    - Conta o número de recursos ainda disponíveis no sistema
  - Semáforo=0 $\rightarrow$ não há recurso livre
    - Nenhum wakeup está armazenado
  - Semáforo>0 $\rightarrow$ recurso livre
    - Um ou mais wakeups estão pendentes

Semáforo

- Dijkstra propôs 2 operações sobre semáforos:
  - Generalizações de sleep e wakeup
  - Down(semáforo) ou Wait(semáforo):
    - Verifica se o valor do semáforo é maior que 0
    - Se for, semáforo=semáforo – 1, então continua
    - Se for 0, o processo que executou o down é colocado para dormir, sem completar o down
    - Executada sempre que um processo deseja usar um recurso compartilhado

Semáforo

- Dijkstra propôs 2 operações sobre semáforos:
  - Up(semáforo) ou Signal(semáforo):
    - semáforo=semáforo + 1
    - Se há processos dormindo nesse semáforo, escolhe um deles e o desbloqueia (permite que complete o down)
    - Nesse caso, o valor do semáforo permanece o mesmo (zero)
    - Executada sempre que um processo liberar o recurso

Semáforo

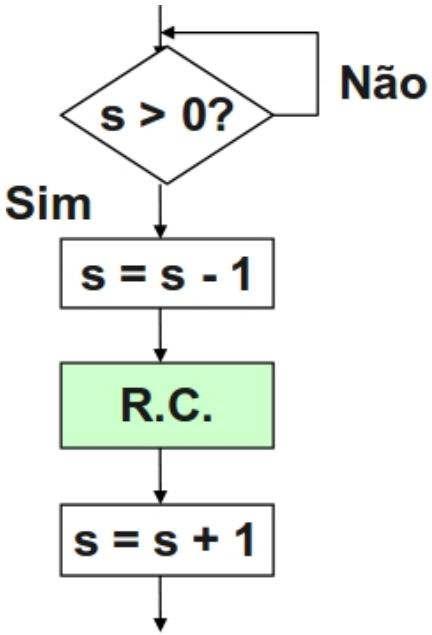
- Operações sobre semáforos são atômicas
  - Garante-se que uma vez que uma operação de semáforo iniciou, nenhum outro processo pode acessar o semáforo até que a operação seja completada ou bloqueada
    - Verificar o valor, alterá-lo e possivelmente dormir são executadas como uma ação única, não bloqueável
  - Geralmente implementadas como chamadas ao sistema
  - O S.O. desabilita todas as interrupções enquanto está testando o semáforo, atualizando-o, e colocando o processo para dormir
    - Se houver múltiplas CPUs, cada semáforo é protegido por uma variável Lock, com a instrução TSL

Tipos de Semáforo – Geral

- Semáforo geral (counting semaphore):
  - Usados para controlar acessos a um determinado recurso com um número finito de instâncias
    - Pode tomar qualquer valor inteiro não negativo
  - Funcionamento:
    - O semáforo é inicializado com o número de recursos disponíveis
    - Cada processo que deseja usar um recurso executa um down()
    - Quando um processo libera um recurso, executa um up()
    - Quando o contador do semáforo vai a 0, todos os recursos estão sendo usados – o processo vai dormir

Semáforo – Implementações

- 1ª Implementação – Espera ocupada
  - Não é a melhor, apesar de ser fiel à definição original.



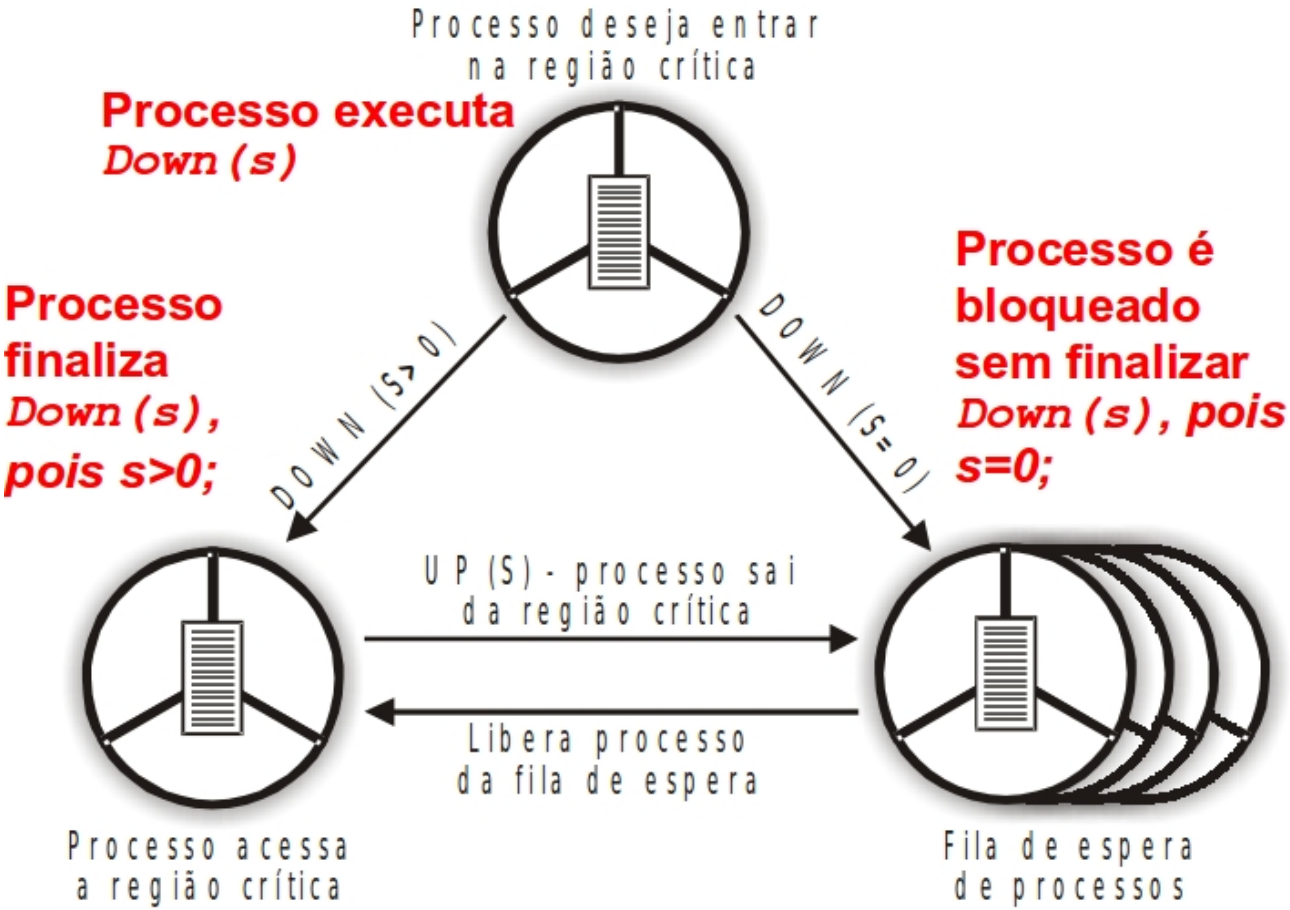
Down(s): Espera até que s >0 e então decrementa s;

Up(s): Incrementa s;

Semáforo – Implementações

- 2^a Implementação – Associando uma fila Q_i de processos bloqueados a cada semáforo s_i
  - Quando se utiliza este tipo de implementação (que é muito comum), as primitivas down e up apresentam o seguinte significado:
    - down(s_i): Se $s_i > 0$, então decrementa s_i (o processo continua); senão, bloqueia o processo, adicionando-o a Q_i
    - up(s_i): Se a fila Q_i está vazia então incrementa s_i, senão acorda processo de Q_i, removendo-o da fila (esse processo prosseguirá no down, entrando na região crítica e decrementando novamente s_i)

Tipos de Semáforo – Geral



Semáforo – Utilidade

- Útil para sincronização:
  - Suponha dois processos, P_1 e P_2, com comandos C_1 e C_2
  - Suponha que queremos que C_2 seja executado somente após C_1
  - Podemos criar uma variável sincr, inicialmente 0, fazendo:

```
C1;
up(sincr);
```

Processo 1

```
down(sincr);
c2;
```

Processo 2

Semáforo – Utilidade

- Problema produtor/consumidor:
  - Resolve o problema de perda de sinais enviados
- Solução utiliza três semáforos:
  - Full:
    - Conta o número de slots no buffer que estão ocupados
    - Iniciado com 0
    - Resolve sincronização: assegura que o consumidor pare de executar num buffer vazio

Semáforo – Utilidade

- Solução utiliza três semáforos:
  - Empty:
    - Conta o número de slots no buffer que estão vazios
    - Iniciado com o número total de slots no buffer
    - Resolve sincronização: assegura que o produtor pare de executar num buffer cheio
  - Mutex:
    - Permite a exclusão mútua
    - Garante que os processos produtor e consumidor não acessem o buffer ao mesmo tempo
    - Iniciado com 1

Semáforo – Utilidade

- Problema produtor/consumidor:

```
#define N 100                                   /* número de lugares no buffer */
typedef int semaphore;                          /* semáforos são um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso à região crítica */
semaphore empty = N;                            /* conta os lugares vazios no buffer */
semaphore full = 0;                             /* conta os lugares preenchidos no buffer */

void producer(void)
{
        int item;

        while (TRUE) {                          /* TRUE é a constante 1 */
                item = produce_item( );         /* gera algo para pôr no buffer */
                down(&empty);                   /* decresce o contador empty */
                down(&mutex);                   /* entra na região crítica */
                insert_item(item);              /* põe novo item no buffer */
                up(&mutex);                     /* sai da região crítica */
                up(&full);                      /* incrementa o contador de lugares preenchidos */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* laço infinito */
                down(&full);                    /* decresce o contador full */
                down(&mutex);                   /* entra na região crítica */
                item = remove_item( );          /* pega item do buffer */
                up(&mutex);                     /* sai da região crítica */
                up(&empty);                     /* incrementa o contador de lugares vazios */
                consume_item(item);             /* faz algo com o item */
        }
}
```

Semáforo – Problema

- Produtor/ consumidor
  - Erro de programação pode gerar um deadlock
  - Ex: inverter ordem dos downs no produtor

	empty	*mutex*	*full*
	0	1	N
P: down(mutex)	0	0	N
C: down(full)	0	0	N-1
P: down(empty)			
C: down(mutex)			

```
void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                down(&mutex);
                down(&empty);
                insert_item(item);
                up(&mutex);
                up(&full);
        }
}
void consumer(void)
{
        int item;

        while (TRUE) {
                down(&full);
                down(&mutex);
                item = remove_item( );
                up(&mutex);
                up(&empty);
                consume_item(item);
        }
}
```

Semáforo – Problema

- Produtor/ consumidor
  - Se o código for trocado no processo produtor:

```
..                          ..
down(&empty);               down(&mutex);
down(&mutex);               down(&empty);
..                          ..
```

  - E o buffer estiver cheio, o produtor será bloqueado após fazer mutex = 0
    - Na próxima vez em que o consumidor tentar acessar o buffer, ele tenta executar um down sobre o mutex, ficando também bloqueado

Semáforo – Problema

- Suponha que existam 2 processos, P_0 e P_1, com semáforos S e Q inicialmente valendo 1
- Suponha a seguinte sequência de operações:

P_0	P_1
down(S)	down(Q)
down(Q)	down(S)
.	.
.	.
.	.
up(S)	up(Q)
up(Q)	up(S)

Semáforo – Problema

- P_0 executa down(S) e P_1 executa down(Q)
  - Quando P_0 executar down(Q), deverá esperar até que P_1 execute up(Q)
  - Quando P_1 executar down(S), deverá esperar até que P_0 execute up(S)
  - Ambos ficam presos $\rightarrow$ deadlock

P_0	P_1
down(S)	down(Q)
down(Q)	down(S)
.	.
.	.
.	.
up(S)	up(Q)
up(Q)	up(S)

Tipos de Semáforo – Mutex

- Um semáforo usado para implementar exclusão mútua é chamados de mutex (mutual exclusion semaphor), binário ou booleano.
  - Só pode tomar os valores 0 e 1
  - O recurso compartilhado é a própria região crítica
  - Mutex é então uma variável com apenas 2 estados: bloqueada (0) e desbloqueada (1)
- Funcionamento:
  - Quando uma thread ou processo precisa acessar uma região crítica, trava o mutex (down)

Tipos de Semáforo – Mutex

- Funcionamento:
  - se o mutex estiver desbloqueado (região crítica disponível), a chamada prossegue $\rightarrow$ pode-se entrar na região crítica
  - Se o mutex estiver bloqueado, a thread que fez o down fica bloqueada até que a thread na região crítica termine
  - Quando termina, destrava o mutex (up)
  - Se mais de uma thread estiver bloqueada no mutex, uma será escolhida para rodar
    - Cabe ao escalonador escolher quem acessa em seguida

Tipos de Semáforo – Mutex

- Em sua implementação, pode ser útil inverter o sentido do mutex, para uso da TSL
  - 1 → bloqueado e 0 → livre

```
mutex_lock:
    TSL REGISTER, MUTEX     | copia MUTEX para o registrador, fazendo
                            | MUTEX=1
    CMP REGISTER, #0        | o mutex estava livre?
    JZE ok                  | se sim, está desbloqueado - retorna
    CALL thread_yield       | mutex ocupado, escalone outra thread (não há
                            | espera ocupada)
    JMP mutex_lock          | tente novamente
ok: RET                     | retorna a quem chamou; entrou na região crítica

mutex_unlock:
    MOVE MUTEX, #0          | libera o mutex
    RET                     | retorna a quem chamou
```

Pode ser todo implementado em espaço de usuário – sem chamadas ao kernel

Tipos de Semáforo – Mutex

- Com, XCHG, contudo, isso não é necessário
  - 1 → livre e 0 → bloqueado

```
mutex_lock:
    MOVE REGISTER,#0        | armazena 0 no registrador
    XCHG REGISTER,LOCK      | troca oconteúdo de REGISTER e LOCK
    CMP REGISTER,#1         | o mutex estava livre?
    JZE ok
    CALL thread_yield       | mutex ocupado, escalone outra thread (não há
                            | espera ocupada)
    JMP mutex_lock          | tente novamente
ok: RET                     | retorna a quem chamou; entrou na região crítica

mutex_unlock:
    MOVE MUTEX, #1          | libera o mutex
    RET                     | retorna a quem chamou
```

Pode ser todo implementado em espaço de usuário – sem chamadas ao kernel

Comunicação de Processos: Observações

- Threads operam em um espaço de endereçamento comum
  - Os espaços de endereçamento de processos, contudo, são disjuntos
- Como variáveis como semáforos e a solução de Peterson podem ser implementadas?
  - Solução 1:
    - Algumas estruturas de dados (como os semáforos) podem ficar no kernel, acessíveis via chamadas ao sistema

Comunicação de Processos: Observações

- Como variáveis como semáforos e a solução de Peterson podem ser implementadas?
  - Solução 2:
    - Alguns S.O.s (Unix, Linux, Windows) permitem que processos compartilhem uma parte de seu espaço com outros
    - Veremos mais adiante
  - No pior caso, em que nada mais é possível, pode-se usar um arquivo compartilhado

Semáforos – Curiosidades

- Semáforos podem ser usados para ocultar interrupções:
  - Associa-se um semáforo (inicialmente em 0) a cada dispositivo de E/S
  - Ao inicializar um dispositivo, o gerenciador faz um down sobre o semáforo associado
    - Bloqueiao processo
  - Quando a interrupção chega, seu tratador faz um up no semáforo do dispositivo, deixando o processo pronto para execução