



# MAC422 Sistemas Operacionais

Prof. Marcel P. Jackowski

Aula #10

Spin locks e Semáforos

# Intel x86 “bts”

- “Bit Set and Test”
- Copia bit para o Carry Flag (CF) e liga o bit correspondente.

```
/* dados compartilhados */      entrada:  
int lock = 0; /* trava livre */      stc  
/* entrada */                  /* copia bit 0 de lock para CF */  
while(BTS(lock,0));              /* liga bit 0 de lock */  
                                bts $lock,0  
                                jc entrada  
                                secao_critica:  
                                /* *** seção crítica *** */  
                                saida:  
                                lea eax,$lock  
                                mov [eax],0h
```

# Intel x86 “xchg”

- Troca atômica entre valores de um registrador e um valor em memória
- Mais flexível pois qualquer valor pode ser atribuído à “lock”

```
/* dados compartilhados */
int lock = 0; /* trava livre */

/* entrada */
register int i = 1;
while(<swap(i,lock)>,i);

/*** seção crítica ***/

/* saída */
lock = 0;
```

```
entrada:
    xor eax,eax
    inc eax
    xchg eax,$lock
    test eax,eax
    jnz entrada

secao_critica:
    /*** seção crítica ***/

saída:
    lea eax,$lock
    mov [eax],0h
```

# “Busy waiting”

- Estas soluções são baseadas em espera ociosa (“spin lock”, “spin wait”, etc) e acabam tomando tempo útil de CPU
  - Processos podem usar todo o seu time quantum e cair na fila de escalonamento
  - Como alternativa podemos mandar o processo dormir por alguns milisegundos
    - Instrução pause (redução do consumo de energia)
  - Processos do kernel normalmente se utilizam deste mecanismo de sincronização
  - Não gostaríamos que processos do usuário gastassem ciclos de CPU de forma ociosa.

# Inversão de prioridades

- Um processo de alta prioridade precisa de um recurso e é bloqueado
- Processo de menor prioridade está segurando o recurso
  - O que fazer ?
- Redução do tempo de resposta
- Problema da sonda Mars Pathfinder
  - Reinicialização frequente, pois um timer era ativado quando um processo de alta prioridade ficava bloqueado.

# Falhas não-recuperáveis

- Se todos as 3 propriedades forem verificadas, uma solução válida será robusta à falhas somente fora da sua seção crítica.
- Nenhuma solução válida pode garantir a robutez quando um processo falha dentro da sua SC.
- Desta forma, um processo que falha na sua SC deve *sinalizar* esse fato para outros processos.

# Desvantagem das travas

- **Problem with lock:** mutual exclusion, but no ordering
- **Producer-consumer problem:** need order
  - \$ cat 1.txt | sort | uniq | wc
  - **Producer:** creates a resource
  - **Consumer:** uses a resource
  - **bounded buffer between them**
  - **Scheduling order:** producer waits if buffer full, consumer waits if buffer empty

# Semáforo

- Um semáforo  $S$  é uma variável inteira que, exceto na sua inicialização, pode ser acessada somente através de suas operações *atômicas e mutualmente exclusivas*:
  - **wait( $S$ )**
    - Também chamado de  $P(S)$ , do holandês “proberen”, testar acesso ao semáforo
  - **signal( $S$ )**
    - Também conhecido como  $v(S)$ , do holandês “verhogen”, incrementar valor do semáforo.

# Possível implementação ?

- Útil quando as SC não demoram muito para completar, ou temos sistemas SMP.
- O valor S normalmente é inicializado com um valor positivo.

```
semaphore S = 1;  
  
wait(S):  
    while(S <= 0) {};  
    S--;  
  
signal(S):  
    S++;
```

Problemas: (i) S-- e S++ não são atômicos, e o próprio while deve ser protegido. Também sem garantias de ordenação no ganho de entrada da SC. Solução: usar os construtos atômicos do x86 para a guarda e implementação da fila para ordenação.

# Exemplo: exclusão mútua

```
semaphore mutex = 1;

void main()
{
    for(;;) {
        /* verifica se podemos entrar a SC */
        wait(mutex);

        /* altera dados compartilhados */
        seção_crítica();

        /* avisa que deixamos a SC */
        signal(mutex);

        /* restante das instruções */
        seção_final();
    }
}
```

# Semáforos em Linux/Unix

- `sem_init (sem_t *s, int pshared, unsigned int value)`
- Has two operations to manipulate this integer
  - `sem_wait (or down(), P())`
  - `sem_post (or up(), V())`

```
int sem_wait(sem_t *s) {  
    wait until value of semaphore s  
    is greater than 0  
    decrement the value of  
    semaphore s by 1  
}
```

```
int sem_post(sem_t *s) {  
    increment the value of  
    semaphore s by 1  
    if there are 1 or more  
    threads waiting, wake 1  
}
```

# Exclusão mútua

## □ Mutual exclusion

- Semaphore as mutex
- Binary semaphore: X=1

```
// initialize to X  
sem_init(s, 0, X)  
  
sem_wait(s);  
// critical section  
sem_post(s);
```

## □ Mutual exclusion with more than one resources

- Counting semaphore: X>1

# Sincronização

## □ Scheduling order

- One thread waits for another
- What should initial value be?

```
//thread 0  
... // 1st half of computation  
sem_post(s);
```

```
// thread 1  
sem_wait(s);  
... //2nd half of computation
```



# Sincronização

- Dois processos  $P_1, P_2$  e  $P_3$  devem sincronizar suas atividades
- $P_1$  deve executar suas operações necessariamente antes que  $P_2$  inicie as suas, que deve fazer suas instruções antes de  $P_3$  executar
- Precisamos fazer com que  $P_2$  espere até que  $P_1$  diga que podemos continuar, e que  $P_3$  espere até que  $P_2$  termine suas tarefas ?
- Como solucionar este problema utilizando instruções `wait()` e `signal()` ?

# Observações

- Quando  $S > 0$ 
  - $S$  significa o número de processos que podem executar  $\text{wait}(S)$  sem bloquear
- Quando  $S = 0$ 
  - Um ou mais processos estão “esperando” no semáforo  $S$
  - Quando  $S$  se torna  $> 0$ , o primeiro processo que testa  $S$  entra na sua SC
    - “Race condition”
    - Não atende a propriedade de espera limitada.

# Semáforos bloqueantes

- Na prática, `wait()` e `signal()` são system calls
  - O S.O. é quem implementa o tipo semáforo
- Para evitar “busy waiting”, quando um processo precisar esperar em um semáforo, ele é colocado em uma fila de espera (FIFO)
- Criamos então uma fila por semáforo, assim como existem filas para E/S.

# Implementação

- Um semáforo pode ser visto como uma estrutura:

```
typedef struct {  
    int count;  
    struct PCB *queue;  
} semaphore;  
  
semaphore S;
```

- Quando um processo deve esperar em um semáforo, ele é bloqueado e colocado na fila de espera.
- signal(S) remove um processo da fila de Execução e coloca na fila de Pronto.

# Implementação

- Note que tanto `wait()` e `signal()` contém seções críticas! Como implementá-las ?
- Estas operações são trechos pequenos de código (seções críticas de curta duração).
- Soluções:
  - Máquinas uniprocessadas: podemos desligar interrupções
  - Sistemas multiprocessados: técnicas de “busy-wait”

# Semáforos e deadlocks

P0

```
wait(S); <.....  
wait(Q);
```

...

```
signal(S);  
signal(Q);
```

P1

```
wait(Q);  
wait(S);
```

...

```
signal(Q);  
signal(S);
```

# Semáforos binários

- Os semáforos estudados até agora são chamados de semáforos contadores
- Podemos também ter semáforos binários
  - O valor de count pode ser somente 0 ou 1.
  - Simples implementação
- Pode também ser utilizado em situações de contagem
  - Ex: exclusão mútua de uma variável contadora de recursos.

# Semáforos binários

```
void wait_binary(S) {
    if (S.value == 1) {
        S.value = 0;
    } else {
        /* insere este processo em S.queue e bloqueia */
    }
}

void signal_binary(semaphore S) {
    if (empty(S.queue)) {
        s.value = 1;
    } else {
        /* retira um processo de S.queue e coloca-o na
         fila de pronto */
    }
}
```

# Problemas clássicos de sincronização

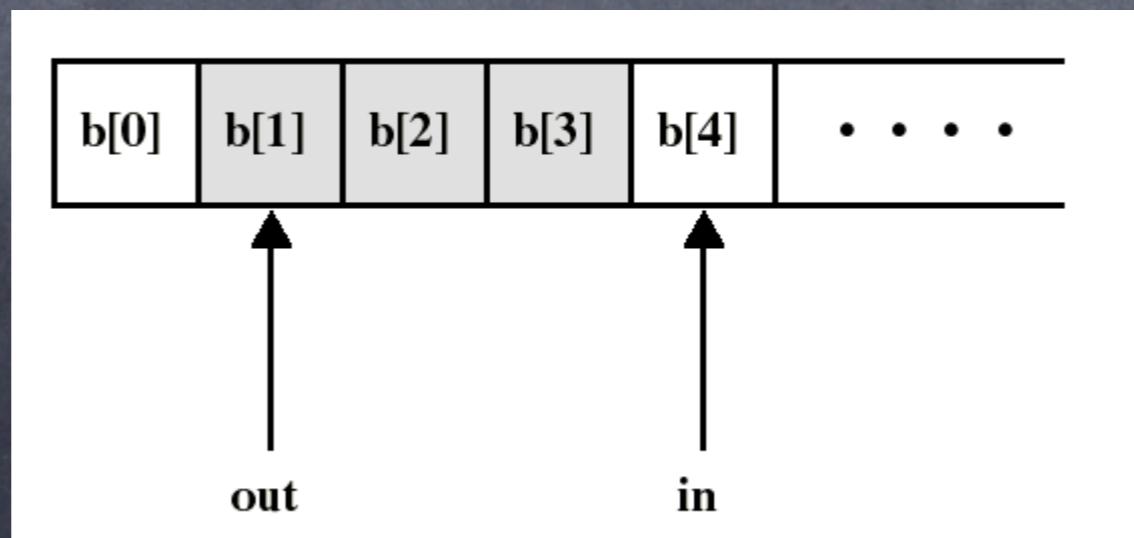
- Padrões de comportamento para cooperação entre processos
- Produtor-Consumidor
  - Acesso à buffer compartilhado
- Jantar dos Filósofos
  - Distribuição de recursos
- Leitores-Escritores
  - Acesso simultâneo à base de dados
  - Prioridades diferentes

# Produtor-Consumidor

- Um processo dito “produtor” produz informação que será consumida por um processo consumidor
  - Implementação de pipes em Unix
- Utilizaremos um buffer para guardar itens que são produzidos e eventualmente consumidos
  - Buffer de tamanho limitado
- Precisamos então coordenar o acesso à este buffer que é compartilhado

# Buffer ilimitado

- Considere inicialmente um buffer sem limite de espaço que consiste de um array de elementos
- Variável **in** aponta para o próximo item a ser produzido
- Variável **out** aponta para o próximo item a ser consumido
- Número de elementos =  $\text{in}-\text{out}$



Área sombreada representa porção do buffer que está ocupado.

# Solução inicial

```
/* variáveis compartilhadas */
int in = 0, out = 0;
int *b = ... /* espaço infinito */

/* PRODUTOR */                                /* CONSUMIDOR */
void main()                                     void main()
{
    for(;;) {
        b[in] = produz_item();
        in++;
    }
}

for(;;) {
    while(in == out);
    consome_item(b[out]);
    out++;
}
```

# Observações

- Caso somente o produtor altere a variável **in**, somente o consumidor altere **out**, e somente o processo produtor escreve no buffer, não precisaremos de exclusão mútua
  - Se formos cuidadosos!
  - O produtor escreve quando quiser, porém o consumidor deve verificar se buffer não está “vazio” (**in==out**)
  - O consumidor poderá ter que entrar em “busy wait”.

# Solução alternativa

```
/* variáveis compartilhadas */
int in = 0, out = 0;
int *b = ... /* espaço infinito */

/* PRODUTOR */                                /* CONSUMIDOR */
void main()                                     void main()
{
    for(;;) {
        in++;
        b[in-1] = produz_item();
    }
}

for(;;) {
    while(in == out);
    consome_item(b[out]);
    out++;
}
```

Problemas ?

# Solução com semáforos

- Declarar buffer e seus apontadores como dados críticos
  - Seção crítica para protegê-los
- Usar um semáforo para controlar a exclusão mútua no acesso ao buffer e aos apontadores!

# Solução com semáforos

```
/* variáveis compartilhadas */
int in = 0, out = 0;
int *b = ... /* espaço infinito */
semaphore mutex = 1;

/* PRODUTOR */                                /* CONSUMIDOR */
void main()                                     void main()
{
    for(;;) {
        wait(mutex);
        b[in] = produz_item();
        in++;
        signal(mutex);
    }
}

for(;;) {
    wait(mutex);
    consome_item(b[out]);
    out++;
    signal(mutex);
}
```

Funciona ? É eficiente ?

# Solução com semáforos

```
/* variáveis compartilhadas */
int in = 0, out = 0;
int *b = ... /* espaço infinito */
semaphore mutex = 1, total = 0;

/* PRODUTOR */                                /* CONSUMIDOR */
void main()                                     void main()
{
    for(;;) {
        wait(mutex);
        b[in] = produz_item();
        in++;
        signal(mutex);
        signal(total);
    }
}

for(;;) {
    wait(total);
    wait(mutex);
    consome_item(b[out]);
    out++;
    signal(mutex);
}
```

Funciona ? É eficiente ?

# Solução com semáforos

```
/* variáveis compartilhadas */
int in = 0, out = 0;
int *b = ... /* espaço infinito */
semaphore mutex = 1, total = 0;

/* PRODUTOR */                                /* CONSUMIDOR */
void main()                                     void main()
{
    for(;;) {
        wait(mutex);
        b[in] = produz_item();
        in++;
        ? ↳ signal(total);
        ? ↳ signal(mutex);
    }
}

for(;;) {
    ? ↳ wait(mutex);
    wait(total);
    consome_item(b[out]);
    out++;
    signal(mutex);
}
```