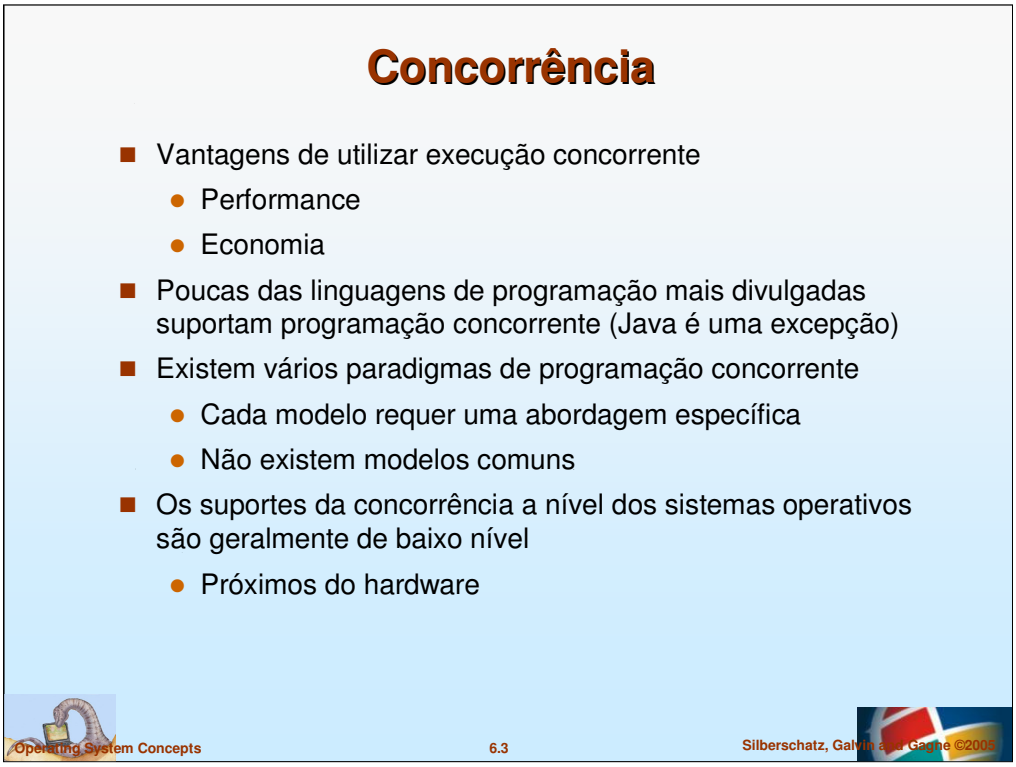
Sistemas Operativos
6ª parte - Sincronização de Processos

Prof. José Rogado
jrogado@ulusofona.pt
Prof. Pedro Gama
pedrogama@gmail.com
Universidade Lusófona
Adaptação LIG e Notas por Dr. Adriano Couto

Sincronização de Processos

- Objectivos:
  - Noções de concorrência, seus problemas e a necessidade de sincronização
  - Conhecer os principais mecanismos de sincronização (processos ou threads), seu funcionamento e implementação
  - Entender os problemas clássicos da sincronização
- Noções de Concorrência
- Secções Críticas
- Soluções Software
- Hardware de Sincronização
- Semáforos
- Problemas Clássicos de Sincronização
- Monitores
- Exemplos de Sincronização
- Transacções Atómicas



Já anteriormente se falou das vantagens da concorrência.

Desde 2005 que os abricantes de CPU (Intel e AMD) passarm a apostar na multi-programação (multi-processo – multi-thread) como única forma de contínuar a aumentar o desempenho, uma vez que não é viável continuar a apostar no aumento das velocidade de relógio.

Ver o site: **hhtp://www.go-parallel.com** da Intel.

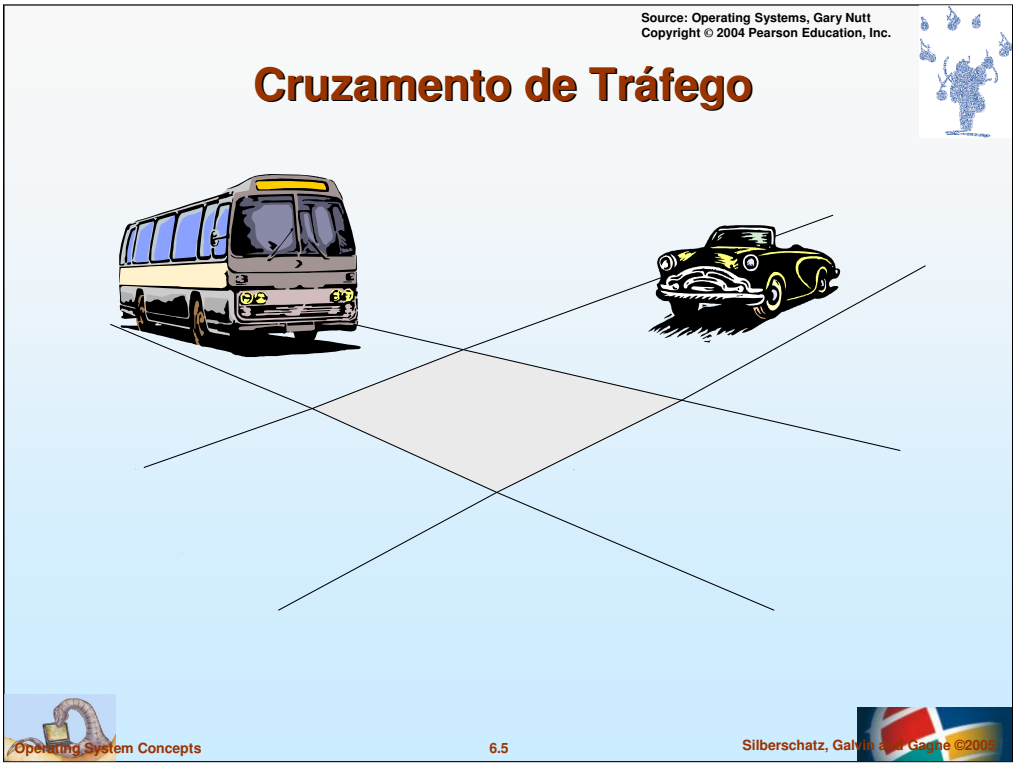
Daremos agora enfâse às soluções para os problemas associados à concorrência.

O aluno deve entender que há vários paradigmas para resolver esses problemas, e reconhecer os mecanismos básicos subjacentes. Esse mecanismos são geralmente garantidos (ao nível mais baixo) pelo próprio hardware. Deverá entender porque razão o hardware tem de fornecer este suporte!

Problemas

- Acessos simultâneos (concorrentes) a dados partilhados por vários fluxos de execução podem gerar inconsistências
  - Processos cooperando num mesma aplicação usando memória partilhada
  - Processos acedendo aos mesmos dispositivos de I/O
  - Threads dentro de um mesmo processo utilizando as mesmas varáveis globais ou o mesmo I/O
- Para manter a consistência dos dados são necessários mecanismos para garantir a serialização de fluxos de execução concorrentes

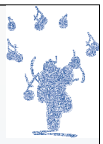
Nota: em computação há duas acepções para a palavra **serialização**: em estruturas de dados e programação OO é o processo de escrever e ler uma estrutura de dados/objecto por um canal série (um byte de cada vez); em sistemas concorrentes descreve a propriedade de as operações efectuadas concorrentemente terem o mesmo resultado se executadas uma de cada vez. Quando isso não acontece temos um problema...



Aqui há um problema se serialização: se os véiculos passam um de cada vez temos um resultado diferente e melhor.

Os 2 veículos **competem/concorrem** por uma região comum: querem estar ao mesmo tempo no mesmo espaço.

Secção Crítica: exemplo



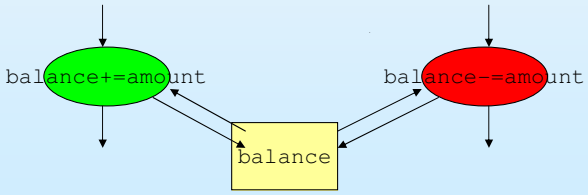
```
shared double balance;
```

Code for p_1

```
. . .
balance = balance + amount;
. . .
```

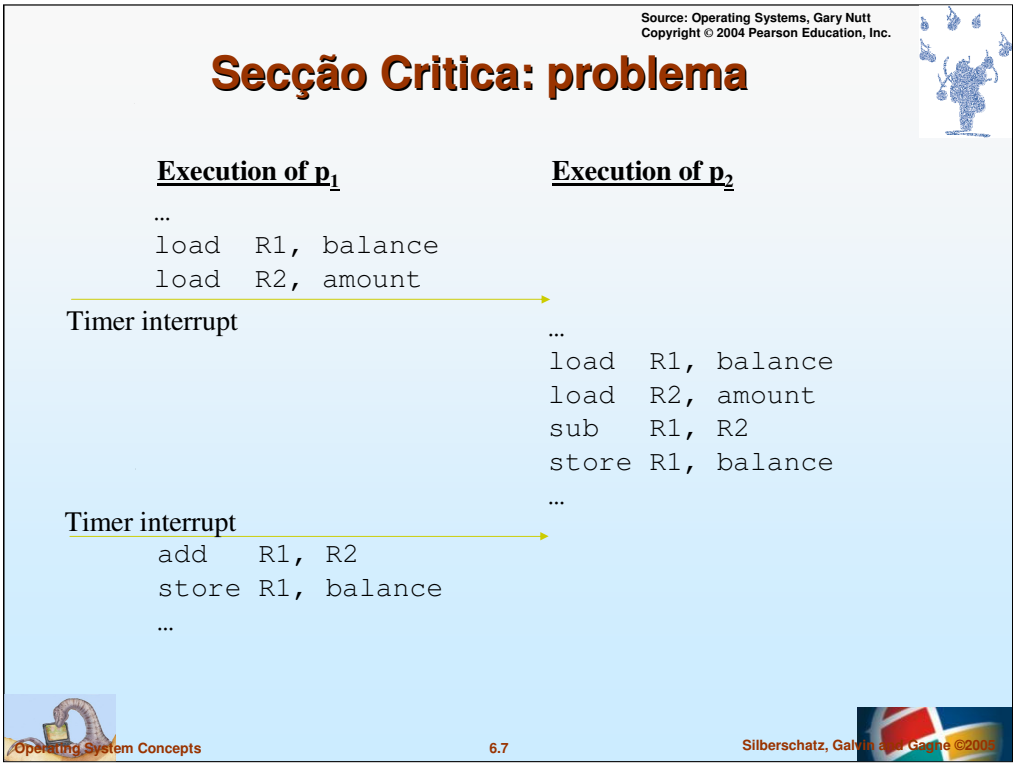
Code for p_2

```
. . .
balance = balance - amount;
. . .
```



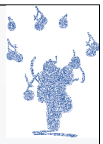
2 processos manipulam a **mesma varável ao mesmo tempo**.
Esta é uma situação que está sempre a acontecer nas nossas contas bnacárias!

-- "**ao mesmo tempo**"? Mas se existir apenas um processador?
Não esquecer a prempção!



A partir do momento em que há prrempção a comutação de processos pode ocorrer em qualquer momento!!!!

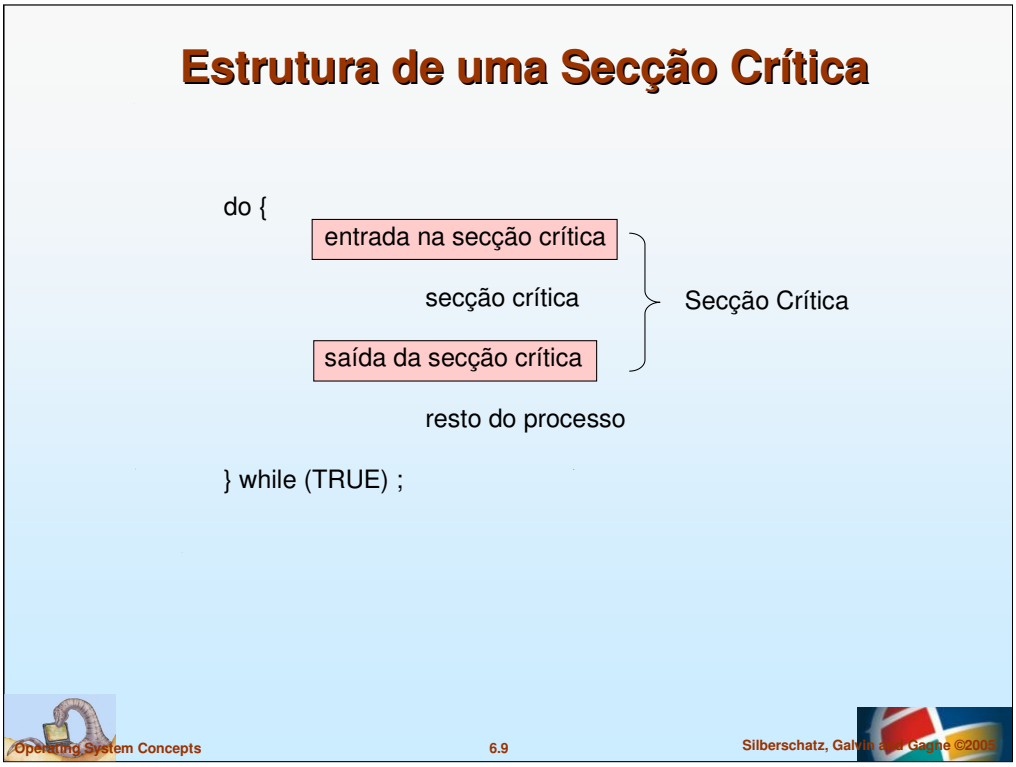
Como nenhum dos processos está a efectuar uma chamadade sistema, o Kernel não sabe que estáo a fazer algo crítico!



Secção Crítica: definição

- *Secção Crítica (SC):* Zona de código que só pode ser utilizada por um processo de cada vez, em *exclusão mútua*
- Existe uma competição entre processos para executar a secção crítica
- A secção crítica pode ter forma diferente (código) em cada processo
  - ∴ Não é facilmente detectavel por análise estatística
- Sem exclusão mútua, o resultado da execução dos processos é indeterminado
- São necessários mecanismos do SO para resolver os problemas levantados pela concorrência

Criamos o conceito de **Seccção Crítica.**



Protocolo para usar uma secção crítica: o processo ou thread avisa que quer entrar e avisa quando sai.

Cabe ao programador garantir este protocolo.

O programador só deve efectuar uma manipulação das variáveis (recursos em geral) que são objecto de concorrência depois de entrar na secção critica usando o protocolo.

Se o fizer sem respeitar o protocolo irá arder para sempre no inferno de Belzebu, isto é, a aplicação irá destruir dados de forma aleatória. (Se esse dados representarem dinheiro haverá lágrimas, baba e ranho na cara dos utilizadores e o programador passará a gestor de hedge funds).

Solução do Problema

1. Exclusão Mútua – Se o processo P_i está a executar uma secção crítica, então nenhum outro processo pode estar a executar a mesma secção crítica (SC) em simultâneo
2. Progresso – Só os processos que competem para uma secção crítica são elegíveis para entrar nessa secção, e a partir do momento em que um processo requisita o acesso à SC este não pode ser adiado indefinidamente
3. Espera Limitada – Depois de um processo ter requisitado o acesso à SC, só um número limitado de outros processos podem entrar na SC antes deste

Definição de uma SC. Repare-se na afirmação:

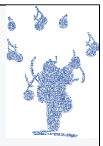
"*só processos que competem para uma secção crítica são elegíveis para entrar nessa secção*".

Ou seja não pode entrar quem não mostrou interesse em entrar (1ª zona colorida no slide anterior)

Algumas Soluções Possíveis

- Desactivar as interrupções
  - Só válido em mono-processadores
  - Kernel não preemptivo
  - Impacto no desempnho
  - Podem perder-se eventos p/ outros processos.
- Utilizar soluções de Software (Peterson)
  - Soluções complexas válidas para poucos processos
  - Situações de bloqueio difíceis de evitar
- Utilizar soluções de Hardware
  - Instruções específicas do processador
  - Geralmente utilizadas no kernel
  - Fornecidas às aplicações através de APIs de programação de mais alto nível
    - Pthreads
    - Java

Estas soluções são discutidas a seguir.



Desactivar as Interrupções

```
shared double balance;
```

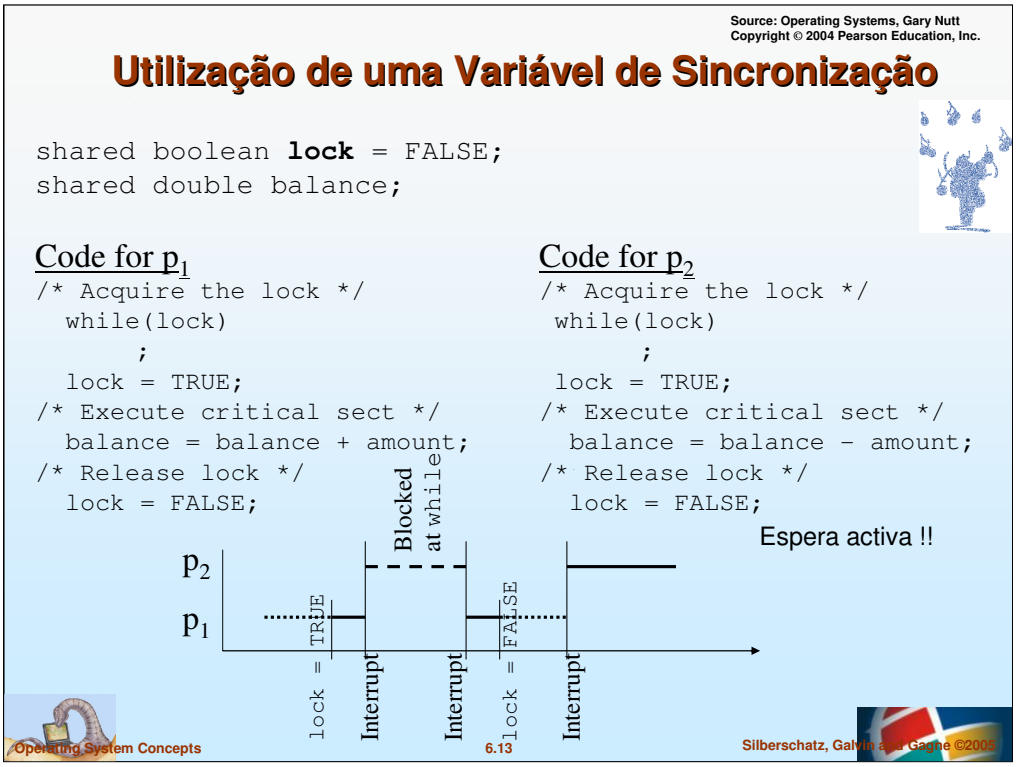
Code for p_1

```
disableInterrupts();
balance = balance + amount;
enableInterrupts();
```

Code for p_2

```
disableInterrupts();
balance = balance - amount;
enableInterrupts();
```

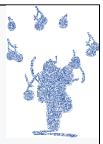
- As interrupções podem ficar desactivadas durante tempos demasiados longos
- Ao sincronizar p_1 e p_2 está-se a impedir que outros processos possam continuar a sua execução



Se a interrupção (preempção) ocorrer num momento "favorável", como represesentado, a solução funciona. Isto, se **p1** mudar a variável antes da interrupção, **p2** irá ficar bloqueado no **while,** em espera activa, isto é em execução contínua, usando o CPU.

O problema ocorre quando a interrupção ocorre depois do teste e antes da afectação da variável!!! Lembro-nos que estão em causa várias instruções máquina, apesar de ser um curto intervalo de tempo!!!

Falsa "Solução" !!



```
shared boolean lock = FALSE;
shared double balance;
```

Code for p_1

```
/* Acquire the lock */
  while(lock) ;
  lock = TRUE;
/* Execute critical sect */
  balance = balance + amount;
/* Release lock */
  lock = FALSE;
```

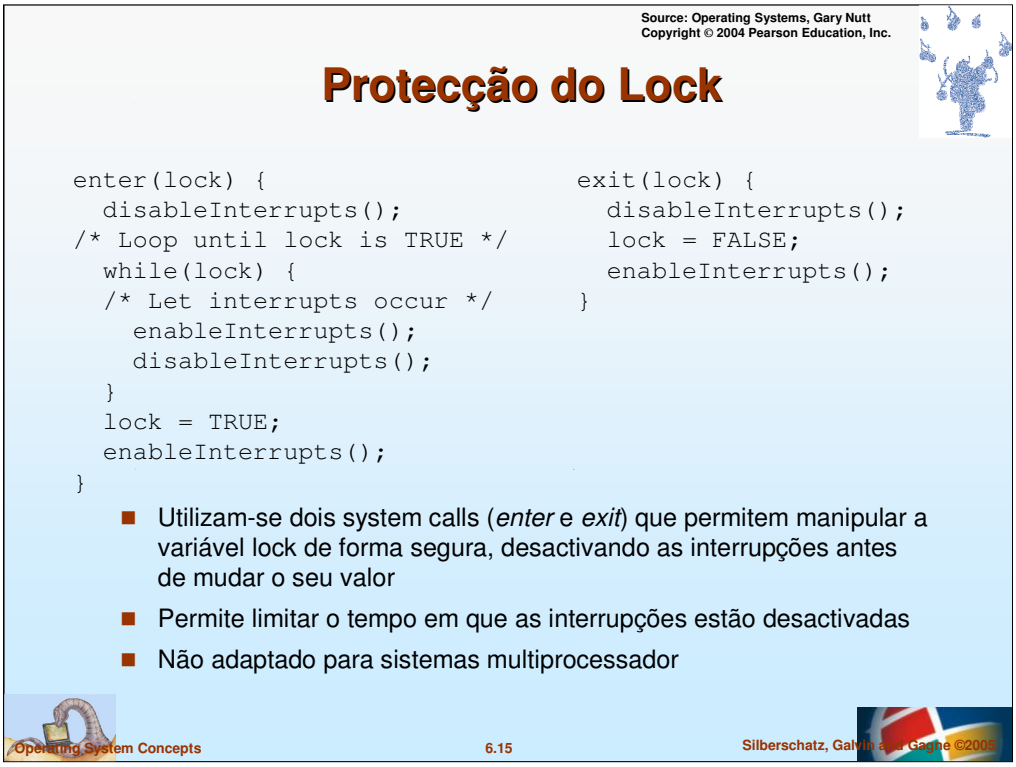
Code for p_2

```
/* Acquire the lock */
  while(lock) ;
  lock = TRUE;
/* Execute critical sect */
  balance = balance - amount;
/* Release lock */
  lock = FALSE;
```

- O processo p_1 … testa a variável, e encontra-a com o valor FALSE
  - Decide entrar na secção crítica, sai do while e é interrompido
- O processo p_2 … testa a variável e encontra-a também a FALSE
  - Decide entrar também na secção crítica
- Resultado: dois processos na mesma secção crítica…

Depois de efectuado o teste no **while** o processo CORRE para mudar o valor do lock. Nesta corrida o outro processo pode ganhar, mudando os resultados.

Trata-se de uma **corrida crítica** (em inglês **race condition**). Os resultados são aleatórios e esta solução não tem a propriedade de serialização.



Aqui estamos a avisar o kernel (atrvés de uma SysCall) que vamos manipular uma variável especial.

Mas se não há apio específico do hardware o kernel também vai ter que desactivar interrupções (para o processador onde a chamada foi efectuada).

Hardware de Sincronização

- A maioria dos processadores modernos fornecem instruções específicas para implementar secções críticas
  - Atómicas = não interruptíveis
- Dois tipos mais frequentes
  - Test-and-set
    - Teste de uma variável em memória e modificação do seu valor
  - Swap
    - Troca (swap) os valores de duas variáveis em memória

A arquitectura de processador x86 possuia a instrução LOCK que garante que a(s) instruções seguintes são atómicas.

Instrução Test-and-Set

- Definição:

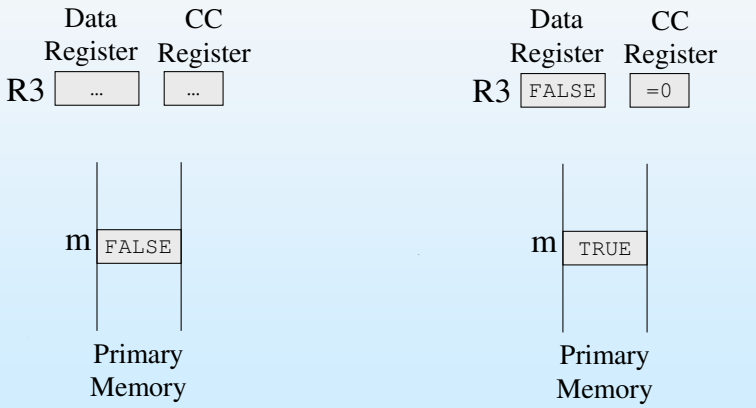
```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Este é a definição **equivalente**, em C, do TestAndSet.

O hardware garante a atomicidade.

Detalhe do Test and Set

- TS(m): [Reg_i = memory[m]; memory[m] = TRUE;]



No caso dos CPU x86 o opcode LOCK é aplicado às instruções aasembly indicadas em cima.

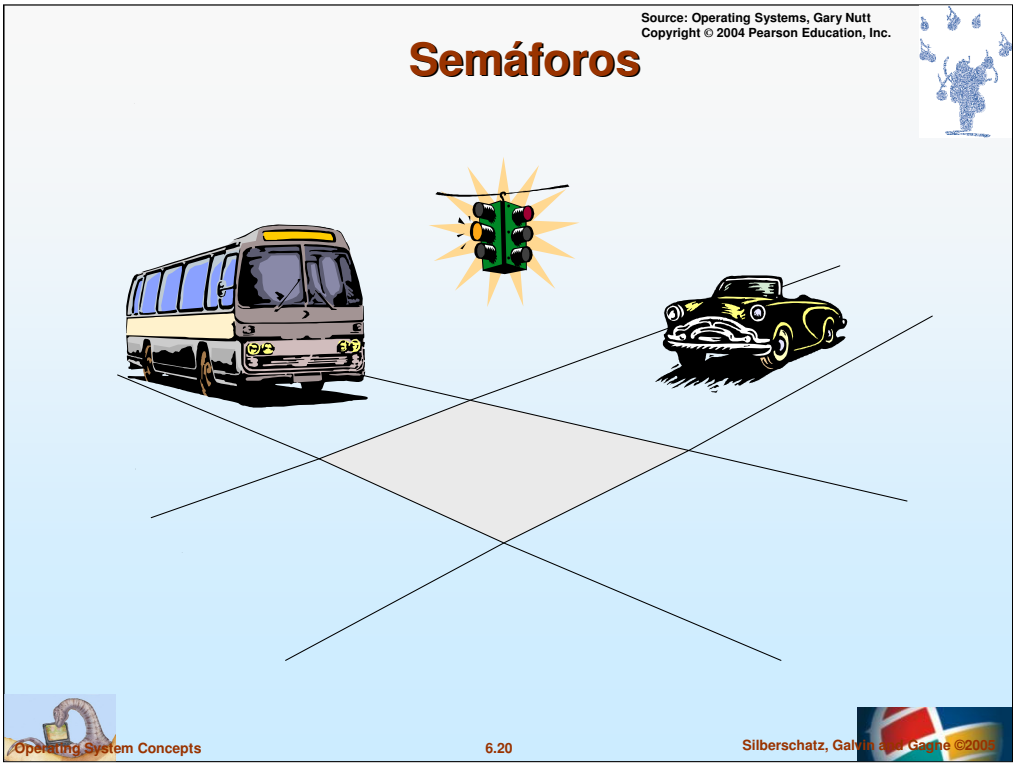
Solução utilizando Test-and-Set

- Variável booleana partilhada **lock**, inicializada a FALSE.
- Solução:

```
while (true) {
        while ( TestAndSet (&lock ))
                ;   // do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

}
```

A SC implementada com o TS têm agora a robustez desejada!

Consegue-se assim implementar uma Secção Crítica.

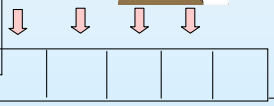


Este é um mecanismo mais sofisticado para sincronização. Ele permite resolver um outro tipo de problemas (por exemplo produtores e consumidores).

E veremos como ele é implementável à custa da primitiva de sincronização mais simples, facultada pelo hardware, que é a SC.

O Problema do Produtor e Consumidor(1)

```
producer() {
   while (true) {
      item = produceItem()

      if (itemCount ==
      BUFFER_SIZE) {
         sleep()
      }
      putItemIntoBuffer(item)
      itemCount = itemCount + 1

      if (itemCount == 1) {
         wakeup(consumer)
      }
   }
}
```



BUFFER_SIZE=5

```
consumer() {

while (true) {

      if (itemCount == 0) {
         sleep()
      }

      item = removeItemFromBuffer()
      itemCount = itemCount - 1

      if (itemCount == BUFFER_SIZE - 1) {
         wakeup(producer)
      }

      consumeItem(item)
   }
}
```

O problema do Produtor e Consumidor(2)

- O produtor depois de produzir um item e incrementar itemCount verifica que o número de itemCount==1 e conclui que o consumidor **pode estar a dormir** (código a vermelho)
- Porém o consumidor pode ainda não estar a dormir. Ele acabou de fazer o teste, decidiu ir dormir, mas foi comutado antes de chamar o **sleep() (**código a vermelho**)**
- Portanto, o wakeup do produtor não encontar **ainda** ninguém a dormir e perde-se!
- O consumidor finalmente entra no **sleep()**, mas o **respectivo wakeup já foi emitido** e não volta a ser repetido (itenCount <>0). Nunca mais acorda!!!
- O código a azul é a situação simétrica e equivalente
- Precisávamos de ter guardado o **wakeup!**
- O semáforo serve para isso: contabiliza **despertares!**

Definição de Semáforo

- A noção de Semáforo foi inicialmente proposta por Edsger Dijkstra em 1968.
- Um semáforo contém um valor inteiro que só pode ser modificado através de duas operações indivisíveis (atómicas):
  - Originalmente chamadas P() e V()
  - Proberen (testar) e Verhogen (incrementar) em holandês
- O nome das operações evoluíram para
  - wait() e signal()
  - Mais de acordo com as suas funcionalidades

Operações sobre um Semáforo

- Definição de P() - wait() e V() - signal()
  - atomic wait (S) {
    while (S <= 0)
    ; // wait
    S--;
    }
  - atomic signal (S) {
    S++;
    }

```
atomic wait (S) {
      waitfor (S > 0);
      S = S - 1;
}
```

Define-se a semântica do **wait():**

• decrementa o semáforo a menos que ele seja 0 e prosegue. Se é 0 bloqueia.

Mostram-se duas hipotéticas implementações uma com espera activa evidente (esq) outra em que a espera activa poderá ser evitada.

E do **signal():**

•que apenas incrementa o semáforo.

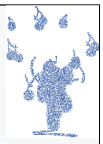
Repare-se que a atomicidade das operações tem de ser assegurada!!!

Ferramenta de Sincronização Genérica

- Os semáforos podem ser de contagem (*counting*) ou binários (*mutexes*)
  - Counting -> sincronização de n recursos distintos
    - O valor do semáforo é um inteiro entre 0 e N.
  - Mutexes -> exclusão mútua de secções críticas
    - O valor do semáforo só pode ser 1 ou 0
- O Mutex permite criar exclusão mútua em secções críticas
  - Semaphore S; // initialized to 1
  - wait (S);
    Critical Section
    signal (S);

A especilização do semáforo conhecida por **mutex** é usada para implementar as SC, pois apresenta para o programador uma API mais homegénea.

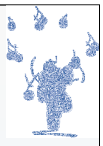
Exemplos de Utilização



```
semaphore mutex = 1;

fork(proc_0, 0);
fork(proc_1, 0);

proc_0() {                          proc_1() {
  while(TRUE) {                       while(TRUE {
    <compute section>;                  <compute section>;
    wait(mutex);                        wait(mutex);
      <critical section>;                 <critical section>;
    signal(mutex);                      signal(mutex);
  }                                   }
}                                   }
```

Compare-se com o protocolo da SC!

Problema do Balanço de Conta



```
semaphore mutex = 1;
shared int balance;

fork(proc_0, 0);
fork(proc_1, 0);
```

```
proc_0() {
   . . .
/* Enter the CS */
  wait(mutex);
    balance += amount;
  signal(mutex);
   . . .
}
```

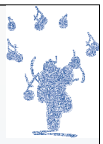
```
proc_1() {
   . . .
/* Enter the CS */
  wait(mutex);
    balance -= amount;
  signal(mutex);
   . . .
}
```

Partilha de 2 Variáveis

- Problema:
  - Dois processos P1 e P2 trabalham em cooperação
  - Utilizam duas variáveis partilhadas x e y para trocar valores
  - Os processos têm de se sincronizar para que os valores das variáveis sejam significativos
- Algoritmo:
  - P1 actualiza x e continua a execução
  - P2 deve ler o valor posto em x por P1 e actualizar y
  - P1 deve esperar que P2 actualize y e ler o seu valor
- Sincronização
  - Dois semáforos S1 e S2 associados a x e y
  - Os semáforos servem de sincronização entre P1 e P2
  - Podem também criar um *deadlock* se forem mal geridos

O problema da comunicação por memória partilhada! Qdo falamos deste mecanismo de IPC dissemos que tinha o problema da sincronização. Eis a solução!
Numa mesma máquina os processos de uma mesma máquina usam geralmente memória partilhada e estes mecanismo para se sincronizarem!

Partilha de 2 Variáveis (Cont.)



```
semaphore s1 = 0; // Protects x, initially wait
semaphore s2 = 0; // Protects y, initially wait
shared int x, y;

create_thread(proc_A, 0);
create_thread(proc_B, 0);
```

```
proc_A() {
  while(TRUE) {
    <compute section A1>;
    update(x);
  /* Signal proc_B */
    signal(s1);
    <compute section A2>;
  /* Wait for proc_B */
    wait(s2);
    retrieve(y);
  }
}
```

```
proc_B() {
  while(TRUE) {
  /* Wait for proc_A */
    wait(s1);
    retrieve(x);
    <compute section B1>;
    update(y);
  /* Signal proc_A */
    signal(s2);
    <compute section B2>;
  }
}
```


Implementação de Semáforos

- É preciso garantir que não há mais de um processo a executar wait () e signal () no mesmo semáforo ao mesmo tempo
- A implementação reduz-se ao problema da secção crítica onde os códigos de wait e signal estão dentro da SC
- O semáforo binário pode ser implementado simplesmente com uma instrução Test-and-Set e uma variável binária
- O valor FALSE indica que o semáforo está livre e TRUE ocupado

```
// Wait definition
boolean s = FALSE;

wait(s) {
   while(TS(s))
      ; // wait
}
```

```
// Signal Definition

signal(s) {
      s = FALSE;
}
```

Como é contruído um semáforo? É muito simples! O problema está nos detalhes e a beleza!

Começamos com o mutex.

Usamos as primitivas que já construimos, o TestAndSet.

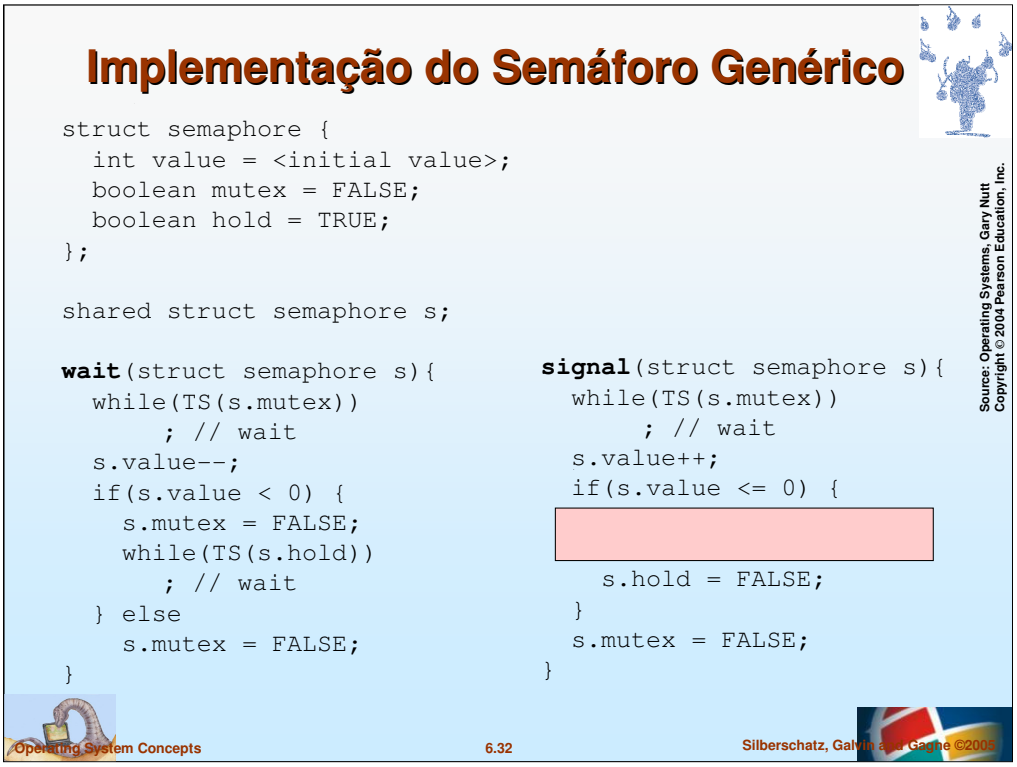
Nesta implementação continuamos a fazer espera activa no while.

Implementação do Semáforo Genérico

- A implementação de um semáforo com contador é mais complexa e necessita de uma estrutura com três campos:

```
struct semaphore {
  boolean mutex = FALSE;
  boolean hold = TRUE;
  int value = <initial value>;
};
```

- O campo mutex permite criar e proteger a secção crítica correspondente à modificação do valor do semáforo
- O campo hold permite colocar o processo (ou thread) em espera no caso do valor do semáforo ser <= 0
- O campo value contém o valor do semáforo (nº de recursos livres) que é negativo no caso de haver processos em espera
- Os campos mutex e hold são modificados através de instruções Test-and-Set
- Um semáforo binário é um semáforo genérico com valor inicial = 1



Para o aluno se aperceber da importância dos detalhes é útil seguir a lógica:

•Todo o acesso ao contador tem de ser exclusivo

•Se, no wait, passamos o contador abaixo de =, é porque ele estava em 0.

•Logo devemos bloquear (no **hold**).

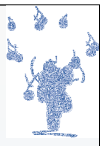
• Mas não podemos bloquear antes de sair da SC,

pois de outra forma paravamos os outros processos também!!!

•Mas isso significa que o TS(s.hold) é fora da SC, logo pode haver comutação!!!

• Temos de acrescentar um teste~adcional ao **signal()** para verificar se houve esta comutação.!!! É o código na zona colorida.

Implementação do Semáforo Genérico



```
struct semaphore {
  int value = <initial value>;
  boolean mutex = FALSE;
  boolean hold = TRUE;
};

shared struct semaphore s;

wait(struct semaphore s){
  while(TS(s.mutex))
       ; // wait
  s.value--;
  if(s.value < 0) {
    s.mutex = FALSE;
    while(TS(s.hold))
       ; // wait
  } else
    s.mutex = FALSE;
}
```

- ***Race condition:***
- No caso em que `s.value<=0`, a thread em signal() tem de aguardar que haja outra thread em espera em `s.hold`, pois pode ter entrado em wait(), mas estar no ponto indicado
- Assim seria libertado um recurso antes da outra thread esperar, e perdia-se uma sinalização !

```
signal(struct semaphore s){
  while(TS(s.mutex))
       ; // wait
  s.value++;
  if(s.value <= 0) {
    while(s.hold == FALSE)
       ; // wait
    s.hold = FALSE;
  }
  s.mutex = FALSE;
}
```


Race condition = corrida crítica.

Espera Activa

- Nas implementações de semáforos apresentadas, a operação wait() implica uma espera activa pelo valor positivo do semáforo
  - Busy wait ou Spin lock

```
atomic wait (S) {
        while (S <= 0)
              ; // busy wait
        S = S - 1;
}
```

- O processo testa o valor e caso este seja menor ou igual a zero, volta a testar, continuando a concorrer com os outros processos
  - Está a consumir recursos quando não tem condições para se executar

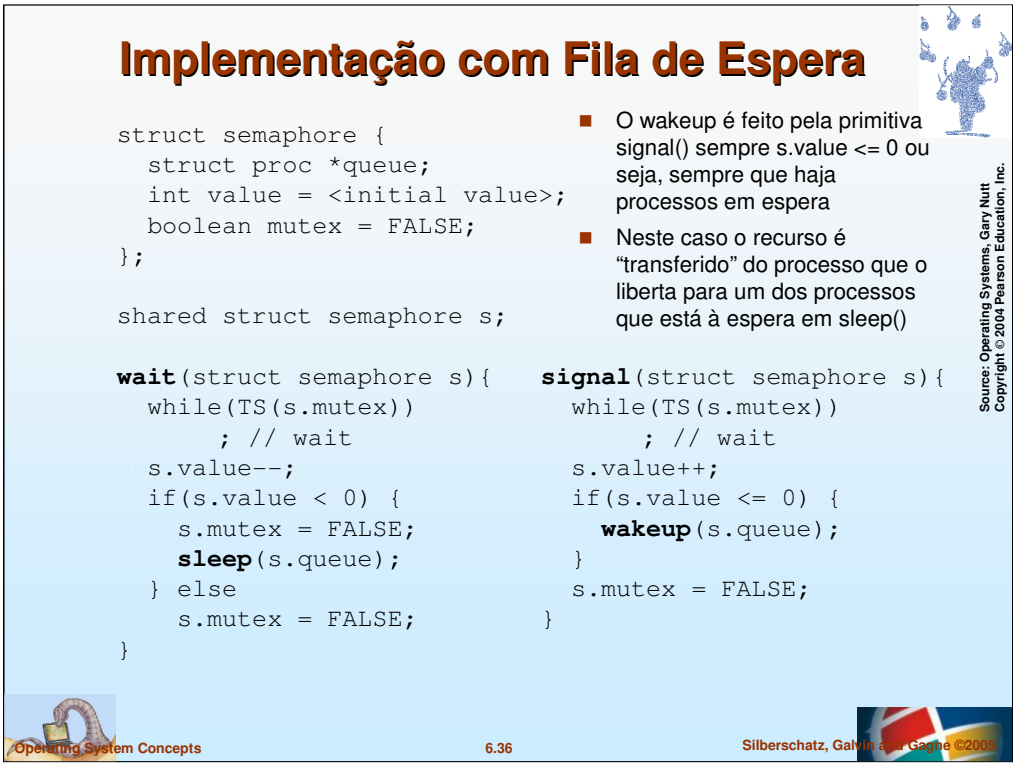
Quando há espera activa (busy wait) estamos a consumir CPU. Poderá ser aceitável durante um curto intervalo de tempo espacielmente se o processo ou thread tem algo mais para fazer.

Em SO com mais que em CPU a espera activa pode existir dentro de um wait, pois o overhead de comutação de processos pode ser superior ao tempo de espera.

Semáforos sem Espera Activa

- Uma solução mais correcta é pôr o processo em espera através de invocações explícitas do *Scheduler*
  - wait(): retirar o processo da Ready Queue no caso em que o semáforo esteja ocupado
  - signal(): voltar a colocar um dos processo em espera na Ready Queue sempre que haja processos bloqueados
- Assim a espera num semáforo assemelha-se a qualquer outra operação de espera por recursos (p.ex.: I/O)
- Ao semáforo pode ser associada uma fila de espera de processos e duas rotinas:
  - sleep() – retira o processo corrente da RQ e coloca-o na fila de espera associada ao semáforo.
  - wakeup() – retira um processo da fila de espera do semáforo e volta a inseri-lo na RQ.
  - O processo a eleger depende dos critérios de *scheduling* do sistema

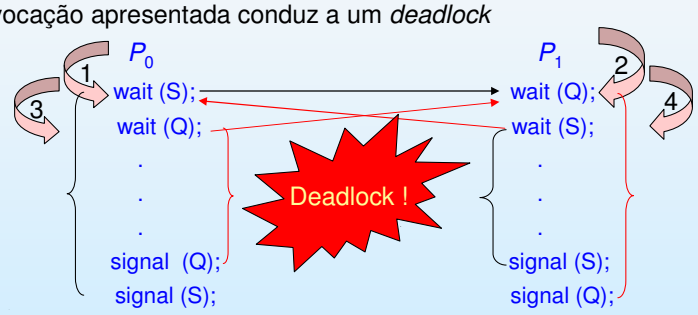
Vemos como o mecanismo para gestão e escalonamento de processos e threads é **reusável** para o Kernel implementar semáforos sem espera activa!



Aqui vemos como a implementação das operações do semáforo se faz com recursos às chamadas de sistema.

Deadlock e Starvation

- Deadlock – dois ou mais processos esperam para sempre por um evento que só um dos processos bloqueados pode produzir
- Se S e Q forem dois semáforos inicializados a 1, a ordem de invocação apresentada conduz a um *deadlock*



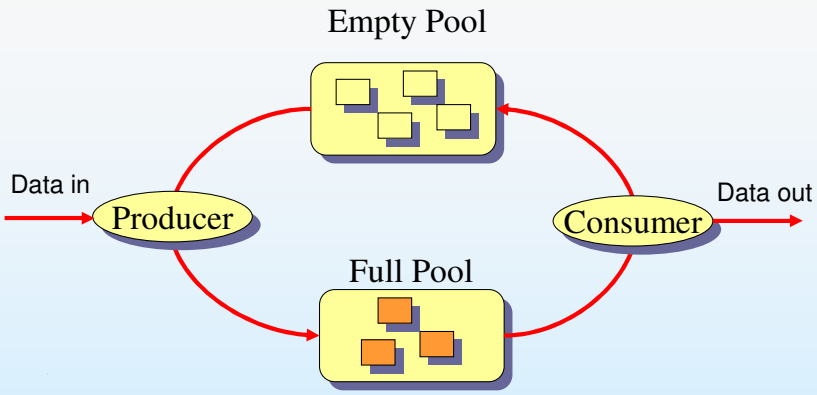
Deadlock é uma situação frequente em aplicações multiprocesso/multithread devido a **erros de concepção e implementação .**

Problemas Clássicos de Sincronização

- Problema do Buffer Limitado ou do Produtor e Consumidor
- Problema dos Leitores e Escritores
- Problema do Jantar de Filósofos

Estes problemas não são meramente alegóricos. Eles exemplificam situações bem Conbcretas.

Problema do Buffer Limitado



- Um processo produz dados que outro consome
- Utilizam uma *pool* de *buffers* para passar dados de um para o outro
- O produtor pode funcionar enquanto houver *buffers* vazios, e espera quando estiverem todos cheios, até que um esteja vazio
- O consumidor pode funcionar enquanto houver *buffers* cheios e espera quando estiverem todos vazios, até que um esteja cheio

Problema do Buffer Limitado (2)

```
semaphore full = 0;     /* A general (counting) semaphore */
semaphore empty = N;    /* A general (counting) semaphore */

producer() {
  while(TRUE) {
    produce_item();
  /* Get an empty buffer */
    wait(empty);
  put_item_in_buffer();
  /* Signal one more item
     is available */
    signal(full);
  }
}
```

```
consumer() {
  while(TRUE) {
  /* Get a full buffer */
    wait(full);
   remove_item_from_buffer();
  /* Signal an empty buffer */
    signal(empty);
    consume_item();
  }
}
```

- Solução para 1 consumidor e 1 produtor.
- Para mais produtores as funções put_item_in_buffer() concorrem.
  - Necessário esclusão mútua
- Para mais consumidores as funções remove_item_from_buffer() concorrem.
  - Necessário exclusão mútua.

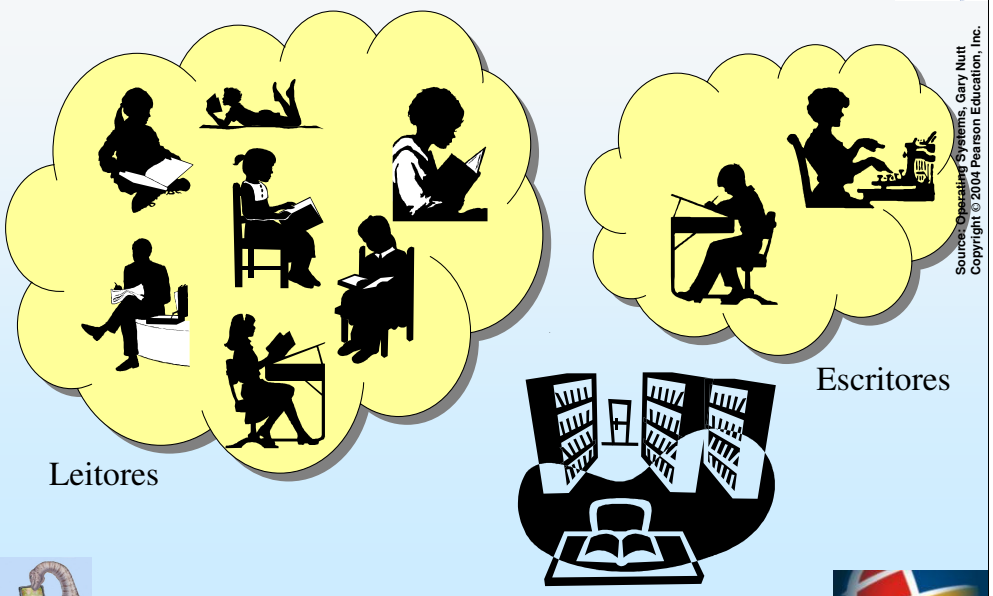
Problema do Buffer Limitado (3)
múltiplos produtores/consumidores

```
semaphore mutex = 1;
semaphore full = 0;
semaphore empty = N;

procedure producer() {
   while (true) {
      produceItem()
      wait(empty)
      wait(mutex)
      putItemIntoBuffer()
      signal(mutex)
      signal(full))
   }
}

procedure consumer() {
   while (true) {
      wait(full)
      wait(mutex)
      removeItemFromBuffer()
      signal(mutex)
      signal(empty)
      consumeItem()
   }
}
```

Esta implementação assume que existe mais do que um produtor que competem entre pelos buffers livres e tabém mais do que consumidor que também competem entre si.

Assim torna-se necessário que apenas um produtor aceda ao slot livre; para isso usa-se o mutex.

Problema dos Leitores e Escritores

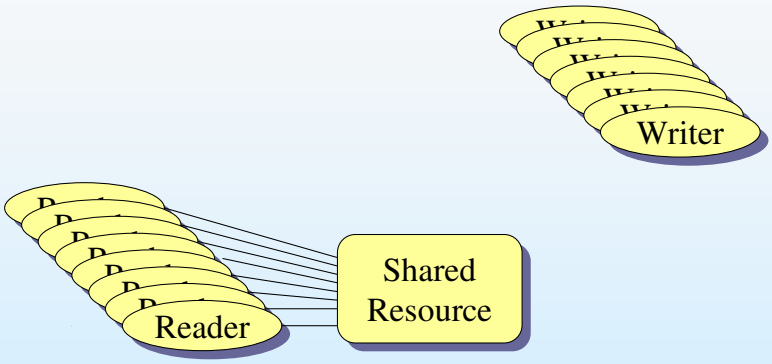


Problema dos Leitores e Escritores (2)

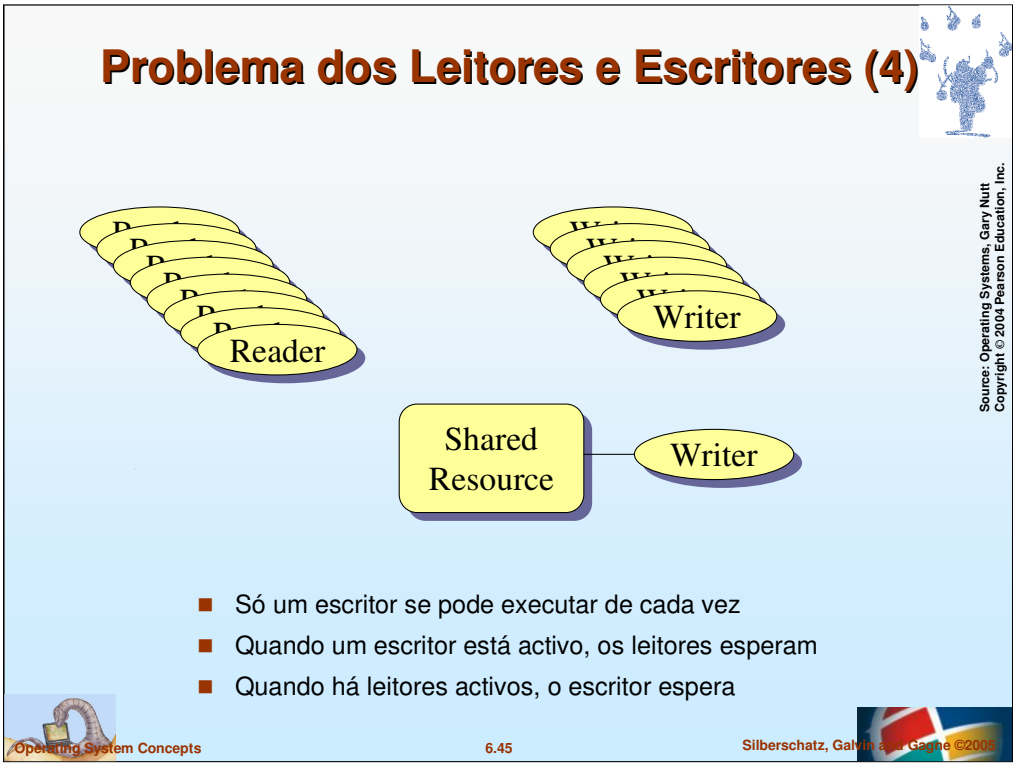
Shared Resource

- Um conjunto de processos realizam acessos a um recurso comum, uns para escrever, outros para ler o seu conteúdo
  - Ficheiro, BD, …
- É necessário garantir a coerência dos valores

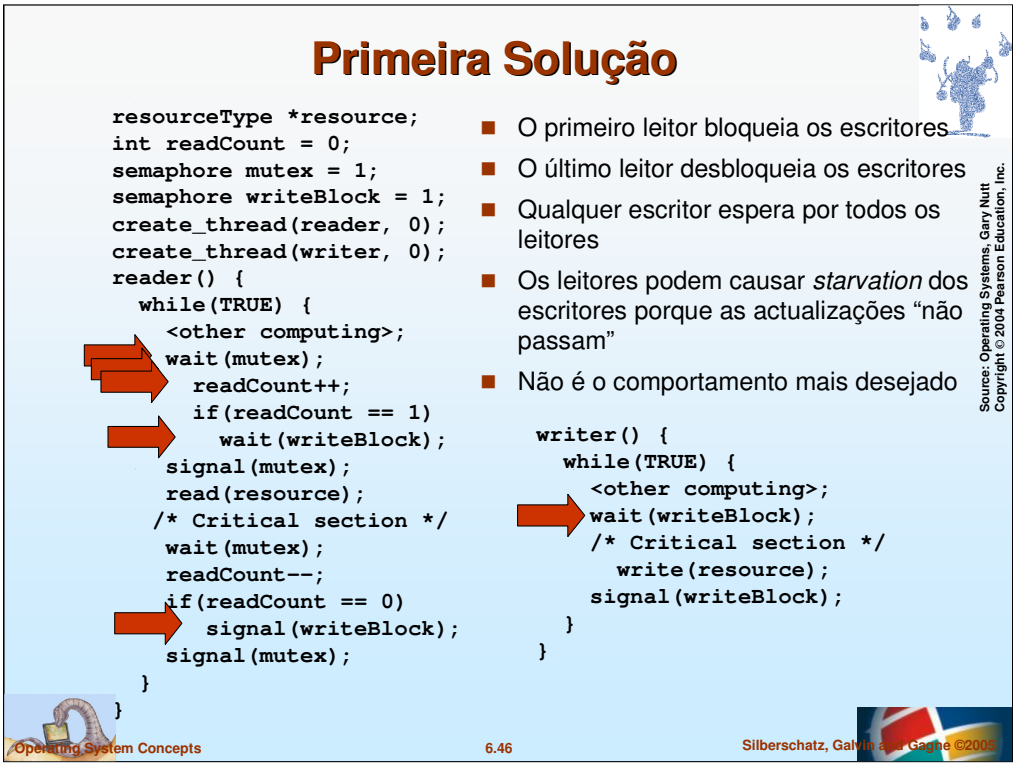
Problema dos Leitores e Escritores (3)



- N leitores podem-se executar simultaneamente



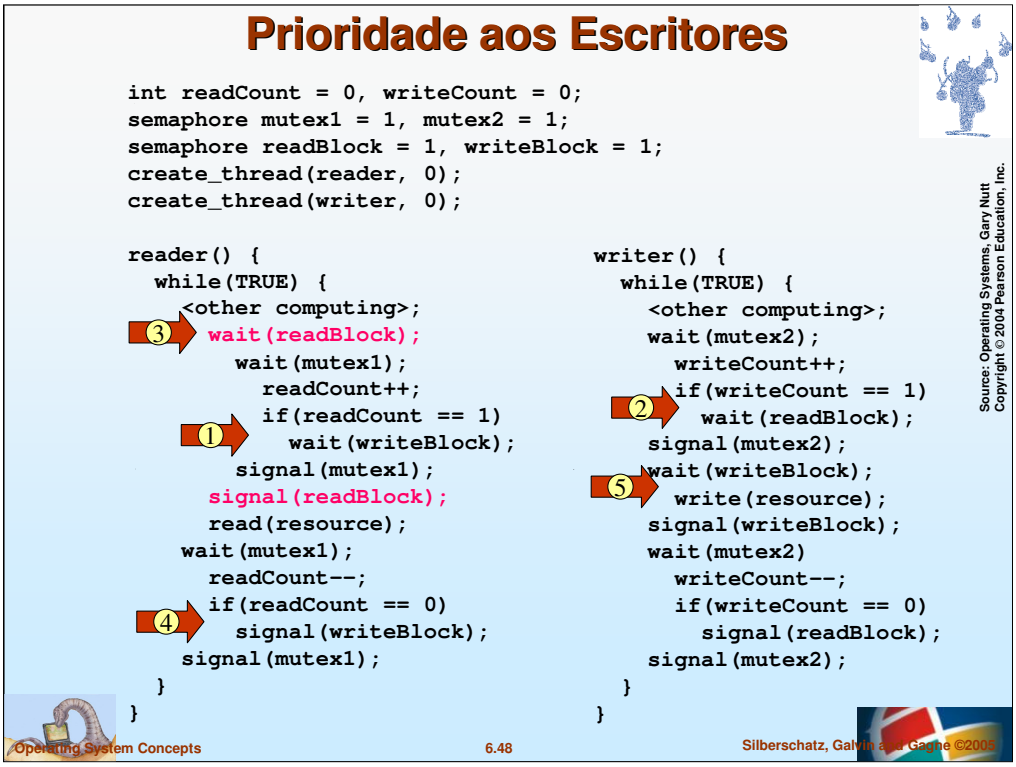
O estudante deve perceber a natureza do problema e de que forma as primitivas de sincronização aprendidas servem para o resolver.



Existem 2 soluções: uma dá prioridade aos leitores.

Segunda Solução

- O algoritmo é modificado para privilegiar os escritores
  - Os leitores continuam a poder ler sem exclusão mútua
  - Se aparece um escritor e está um leitor activo, o escritor espera
  - Se um leitor aparece com um escritor em espera, o leitor espera que o escritor saia
  - Se um segundo escritor aparece, tem prioridade sobre os leitores que podem esperar indefinidamente
- Esta solução é complexa e necessita uma implementação cautelosa para evitar deadlocks e starvation
- Veremos uma solução mais robusta baseada em monitores



a segunda dá prioridade aos escritores.

Problema dos Filósofos à Mesa



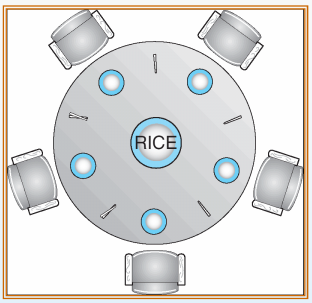
- Cinco filósofos estão sentados a uma mesa a comer massa:
  - Cada um tem um talher de cada lado, que partilha com o seu vizinho do lado
  - Cada um pensa, e só pode comer quando ficam livres os dois talheres de cada lado do prato
  - Dois filósofos sentados ao lado um do outro não podem comer ao mesmo tempo.





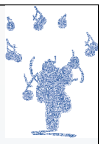


Problema dos Filósofos à Mesa

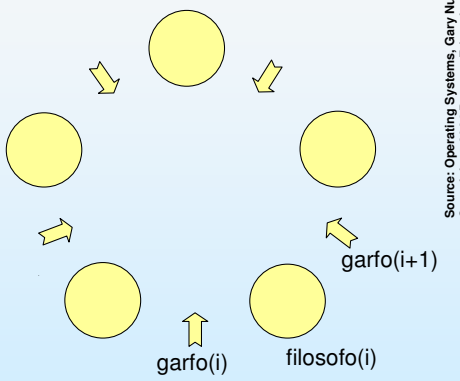


- Para resolver o problema vamos atribuir um índice a cada filósofo, com i variando de 0 a 4
  - A cada garfo associa-se um semáforo designado por garfo(i)
- O filósofo i pode comer quando estão livres simultaneamente os semáforos garfo(i) e garfo((i+1) mod 5), ou seja:
  - garfo(0) e garfo(1)
  - garfo(1) e garfo(2)
  - garfo(2) e garfo(3)
  - garfo(3) e garfo(4)
  - garfo(4) e garfo(0)

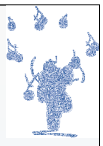
Primeira Solução



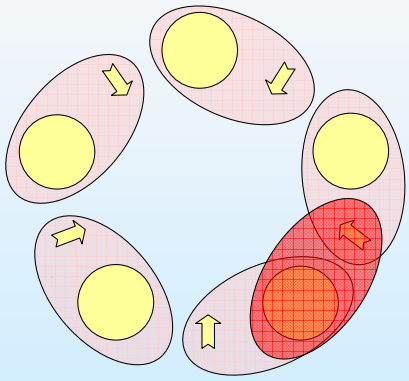
```
semaphore garfo[5] = (1,1,1,1,1);
fork (filosofo, 1, 0);
fork (filosofo, 1, 1);
fork (filosofo, 1, 2);
fork (filosofo, 1, 3);
fork (filosofo, 1, 4);

filosofo (int i) {
  while(TRUE) {
    // Think
    // Eat
    P(garfo[i]);
      P(garfo[(i+1) % 5]);
        eat();
      V(garfo[(i+1) % 5]);
    V(garfo[i]);
  }
}
```



Primeira Solução

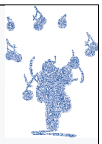


```
semaphore garfo[5] = (1,1,1,1,1);
fork (filosofo, 1, 0);
fork (filosofo, 1, 1);
fork (filosofo, 1, 2);
fork (filosofo, 1, 3);
fork (filosofo, 1, 4);

filosofo (int i) {
  while(TRUE) {
    // Think
    // Eat
    P(garfo[i]);
      P(garfo[(i+1) % 5]);
        eat();
      V(garfo[(i+1) % 5]);
    V(garfo[i]);
  }
}
```

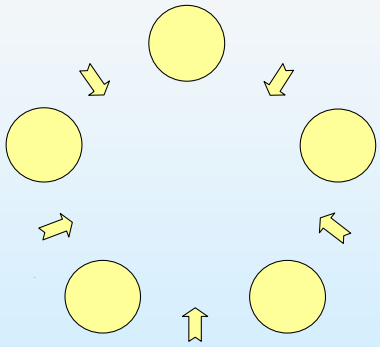


- Se todos os filósofos acederem simultaneamente aos talheres do mesmo lado (esquerdo ou direito), cria-se uma solução de *Deadlock*

Segunda Solução



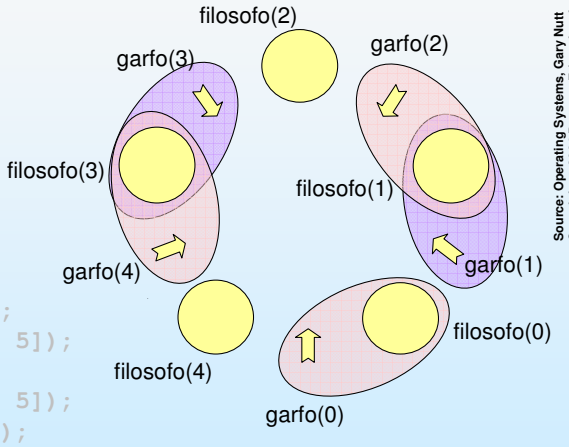
```
semaphore garfo[5] = (1,1,1,1,1);
fork (filosofo, 1, 0);
fork (filosofo, 1, 1);
fork (filosofo, 1, 2);
fork (filosofo, 1, 3);
fork (filosofo, 1, 4);

filosofo (int i) {
  while(TRUE) {
    // Think
    // Eat
    j = i % 2;
    P(garfo[(i+j) % 5]);
      P(garfo[(i+1-j) % 5]);
        eat();
      V(garfo[(i+1-j) % 5]);
    V(garfo[[(i+j) % 5]);
  }
}
```



- Na primeira escolha, os filósofos de índice par escolhem o garfo esquerdo, e os de índice ímpar, o garfo direito
- Na segunda escolha inverte-se a ordem

Segunda Solução

```
semaphore garfo[5] = (1,1,1,1,1);
fork (filosofo, 1, 0);
fork (filosofo, 1, 1);
fork (filosofo, 1, 2);
fork (filosofo, 1, 3);
fork (filosofo, 1, 4);

filosofo (int i) {
  while(TRUE) {
    // Think
    // Eat
    j = i % 2;
    P(garfo[(i+j) % 5]);
      P(garfo[(i+1-j) % 5]);
        eat();
      V(garfo[(i+1-j) % 5]);
    V(garfo[[(i+j) % 5]);
  }
}
```



- Na primeira escolha, os filósofos de índice par escolhem o garfo esquerdo, e os de índice ímpar, o garfo direito
- Na segunda escolha inverte-se a ordem

Problemas de Utilização dos Semáforos

- As operações sobre os semáforos têm de ser usadas correctamente
- Se há troca de ordem, pode haver vários processos na secção crítica

  wait (mutex)

  // Secção Crítica

  signal (mutex)
- Se houver troca de funções, gera-se um *deadlock*

  signal (mutex)

  // Secção Crítica

  wait (mutex)
- Se houver omissão de uma das funções, um dos dois casos anteriores pode acontecer
- É relativamente frequente em programas complexos acontecerem situações deste tipo que não são fáceis de detectar
- Para evitar estas situações, algumas linguagens utilizam ferramentas de sincronização mais fáceis de utilizar

Motivação para a necessiade de mecanismos mais amigos do programador, que, não obstante vão ser construídos à custa do que já aprendemos.

Monitores

- Uma extensão do conceito de semáforo de mais alto nível que fornece uma interface eficaz e fácil de utilizar para sincronização de threads ou processos
  - Conceito de programação Orientada aos Objectos
  - Embebida nas linguagens Java e Concurrent Pascal
  - Gerida pelo compilador
- O monitor encapsula (esconde) a sua implementação
  - Só são acessíveis métodos e dados públicos
  - Os métodos e dados privados são só utilizador pelo monitor
- O monitor garante exclusão mútua dos processos que o executam:

```
monitor mon {
private:
  semaphore mutex = 1;  // Implícito
  . . .
public:
  proc_i(…) {
    P(mutex);  // Implícito
      <code for proc_i>;
    V(mutex);  // Implícito
  };
};
```

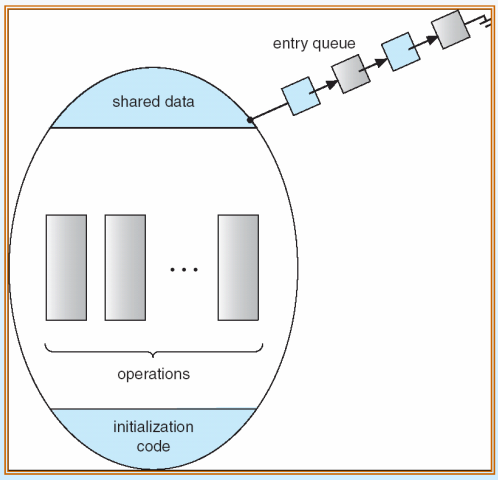
Esta maéria não vem para o tesde de 6/Janeiro.

Exemplo: Shared Balance

- O problema da partilha da variável balance é resolvido através da declaração de um monitor contendo:
  - Uma variável privada balance
  - Dois métodos públicos que são executados no interior do monitor com exclusão mútua implícita

```
monitor sharedBalance {
private:
  double balance;
public:
  credit(double amount){
       balance += amount;
  }
  debit(double amount){
       balance -= amount;
  }
  . . .
}
```

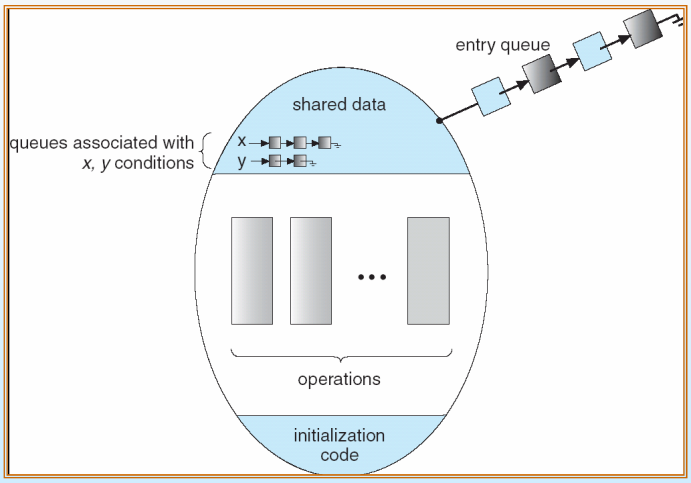
Representação de um Monitor



Variáveis de Condição

- Os monitores fornecem também mecanismos que permitem às threads esperar que se verifiquem condições específicas
  - Designam-se por Variáveis de Condição
- Geralmente, são definidas duas operações sobre uma VC x
  - x.wait () – permite a uma thread que invoca esta operação ser suspensa até a condição se verificar
  - x.signal () – permite a uma thread sinalizar o facto de que a condição se verificou, e activar uma das threads que estiverem em espera
- Não é geralmente garantido que a thread activada seja a que mais tempo tenha esperado
  - A forma como as threads são reactivadas por x.signal() depende da implementação do monitor e do algoritmo de scheduling

Um Monitor com Variáveis de Condição



Estrutura de um Monitor

- Variáveis de Sincronização do Monitor
  - Um mutex para exclusão mútua no monitor, inicializado a 1
  - Um mutex de sinalização no monitor, inicializado a 0
  - Um contador de processos em espera

```
semaphore mutex = 1;  // monitor mutex, initially  = 1
semaphore other = 0;    // signaling mutex, initially  = 0
int others_waiting = 0;   // number waiting for signaling
```

- Variáveis de Condição do Monitor: para cada condição é introduzida uma variável com a seguinte estrutura
  - Um semáforo de sinalização inicializado a 0
  - Um contador de processos em espera na variável

```
struct condvar x  {
        semaphore sem; // initially  = 0
        int count = 0; // waiting on x
};
```

Implementação do Monitor - 1

- Variável de condição ***x***:

```
struct condvar x  {
    semaphore sem; // initially  = 0
    int count = 0; // waiting on x
};
```

- Variáveis de sincronização:

```
semaphore mutex = 1;  // monitor mutex
semaphore other = 0;   // signaling mutex
int others_waiting = 0;   // waiting for signaling
```

- As operações sobre a variável de condição são executadas no interior dos métodos do monitor, com o semáforo de exclusão mutex tomado

- Operação x.wait:

```
x.wait () {
   x.count++;
   if (others_waiting > 0)
        signal (other);
   else
        signal (mutex);
   wait (x.sem);
   x.count--;
}
```

- Operação x.signal:

```
x.signal () {
   if (x.count > 0) {
      others_waiting++;
      signal (x.sem);
      wait (other);
      others_waiting--;
   }
```

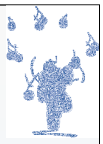
Implementação do Monitor - 2

- Cada método ***M*** do monitor deve obedecer à seguinte estrutura:

```
wait (mutex);
        ...
         código de M;
        ...
if (other_waiting)
      signal (other);
else
      signal (mutex);
```

- No caso de um método ***M*** invocar no seu interior uma operação de espera ou sinalização de uma variável de condição, e havendo outros processos em espera, deve à saída libertar o semáforo de sinalização e não o de sincronização

Readers & Writers com Monitores



```
monitor readerWriter {
  int numberOfReaders = 0;
  boolean busy = FALSE;
  condition okToRead, okToWrite;
public:
  startRead() {
    if(busy || (okToWrite.queue())
      okToRead.wait();
    numberOfReaders++;
    okToRead.signal();
  }
  finishRead() {
    numberOfReaders--;
    if(numberOfReaders == 0)
      okToWrite.signal();
  }
```

```
  startWrite() {
    if((numberOfReaders != 0)
       || busy)
         okToWrite.wait();
    busy = TRUE;
  }
  finishWrite() {
    busy = FALSE;
    if(okToRead.queue())
      okToRead.signal()
    else
      okToWrite.signal()
  }
}
```


Exemplos de Sincronização

- Solaris
- Windows XP
- Linux
- Pthreads

Sincronização em Solaris

- Implementa uma grande variedade de locks para suporte de multitasking, multithreading e multiprocessing
- Usa mutexes adaptativos para proteger dados partilhados em segmentos de código curtos
  - Comportamento do lock depende de haver ou não vários CPUs
- Utiliza condition variables e readers-writers locks quando as secções de código a proteger são mais extensas
- Utiliza um tipo de fila de espera específico (turnstiles) para ordenar a lista de threads que esperam para adquirir os vários tipos de locks.

Sincronização em Windows XP

- Em monoprocessador, utiliza desactivação de interrupções para proteger secção críticas curtas
- Utiliza spinlocks em ambiente multiprocessador
- Fornece dispatcher objects que podem funcionar como mutexes e semáforos
- Os Dispatcher objects também podem gerar eventos
  - Funcionam como variáveis de condição

Sincronização em Linux

- Em monoprocessador, utiliza a desactivação de interrupções para proteger secção críticas curtas
- Utiliza spinlocks baseados em TAS em ambiente multiprocessador
- O kernel é preemptível desde a versão 2.6, mas quando um processo kernel obtém um lock, esta possibilidade é suspensa através da manutenção de um contador do número de preempções em curso (*preempt_count*) por processo kernel
- Quando as secções críticas de código são extensas, utilizam-se semáforos
- System call futex() “Fast Userspace muTexes” permite a implementação de mutex e semáforos em modo utilizador com recurso mínimo a funções do kernel

http://lxr.linux.no/linux/kernel/futex.c

Sincronização nas Pthreads

- A API das pthreads é independente do Sistema Operativo
- A API fornece:
  - mutex locks
  - condition variables
- Extensões não dependentes do SO fornecem também:
  - read-write locks
  - spin locks

API pthreads para Sincronização

- Gestão de Mutexes
  - pthread_mutex_init (mutex,attr)
  - pthread_mutex_lock (mutex)
  - pthread_mutex_trylock (mutex)
  - pthread_mutex_unlock (mutex)
- Gestão de Condition Variables
  - pthread_cond_init (condition,attr)
  - pthread_cond_wait (condition,mutex)
  - pthread_cond_signal (condition)
  - pthread_cond_broadcast (condition)
- Utilização
  - pthread_cond_wait deve ser invocada com um mutex locked, a função liberta-o implicitamente
  - pthread_cond_signal deve ser invocada com um mutex locked que é necessário libertar para que uma thread que espera possa recomeçar

Fim do Capítulo 6