

Sincronismo com Semáforos

Espera activa & espera passiva

Podemos classificar a espera pelo acesso a um recurso como :

- **espera activa:** se consome ciclos de cpu
- **espera passiva:** se não o faz

Espera activa

```
while (! Cond) ;  
...
```

Espera passiva

```
if (! Cond) suspender_processo  
quando Cond for true acordar_processo
```



Exemplo de um mecanismo de espera passiva

Este mecanismo possui:

Uma *flag* indicadora de estado (livre, ocupado)

Uma lista de processos bloqueados

Operações (atômicas) sobre o mecanismo:

Iniciar: *flag* recebe “livre”;

Obter acesso ao/pelo mecanismo :

se mecanismo “livre”, colocar mecanismo como ocupado e prosseguir
se não, bloquear a *thread* na cauda da lista do mecanismo

Libertar acesso :

se há *threads* bloqueadas, desbloquear uma
se não, colocar o mecanismo como “livre”

Livre / ocupado



Semáforo [Dijkstra, 1965]

Um semáforo \underline{S} é um objecto com um atributo \underline{C} (ou valor do semáforo), inteiro não negativo, sobre o qual são permitidas duas **operações indivisíveis**, designadas por: .

Proberen (testar) - **Wait** (esperar)

Vermagen (avisar) - **Signal** (sinalizar)

O atributo \underline{C} ou valor do semáforo tem um valor inicial que depende de caso para caso.

Funcionalidade do P/Wait e V/Signal

```
Wait( ) {  
    if (C > 0) // semáforo com unidades  
        C = C-1; // decrementa uma unidade  
    else Bloqueia_task_na_fila;  
}
```

```
Signal( ) {  
    if (Tasks_bloqueadas_na_fila)  
        Desbloqueia_Task_da_fila  
    else C=C+1; // incrementa uma unidade  
}
```

Invariante de um Semáforo:

N° de Waits terminados \leq Valor Inicial + N° Signals



Classe Semáforo

```
class Semaforo {  
public:  
    Semaforo(int numUnidades=1, int maxUnidades=MAXLONG);  
    ~Semaforo();  
    void P();  
    void V();  
    void Wait() { P(); }  
    void Signal() { V(); }  
};
```

Semântica textual das funções Wait e Signal:

Wait:

se houver uma unidade: retira-a e prossegue
caso contrário: fica bloqueada, quando houver uma unidade retira-a e prossegue

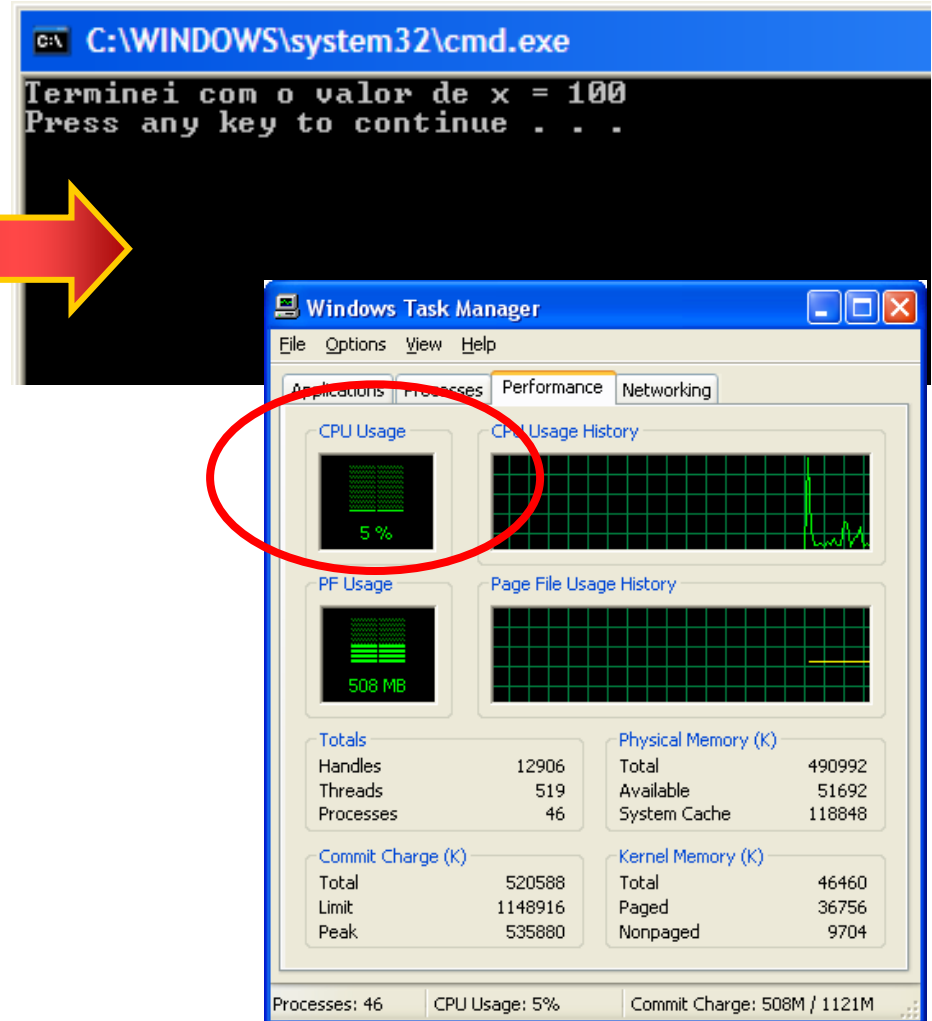
Signal:

coloca uma unidade
se houver alguém na fila de espera, acorda um
Prossegue sempre



Exemplo com garantia de exclusão mútua utilizando semáforos

```
DWORD WINAPI IncFunc(LPVOID args)
{
    int tmp;
    for(int i=0; i<50; i++){
        sem.Wait();
        /* Inicio da região critica */
        tmp = x;
        tmp++;
        x = tmp;
        /* Fim da região critica */
        sem.Signal();
    }
    return 0;
}
```



Exercício de sincronismo

Controlador de acesso concorrente

Pretende-se um controlador que deixe entrar em acesso simultâneo somente N threads

Interface pretendida:

```
class SharedAccess {  
protected:  
    unsigned MAX;  
  
public:  
    virtual void Entrar() = 0;  
    virtual void Sair() = 0;  
};
```

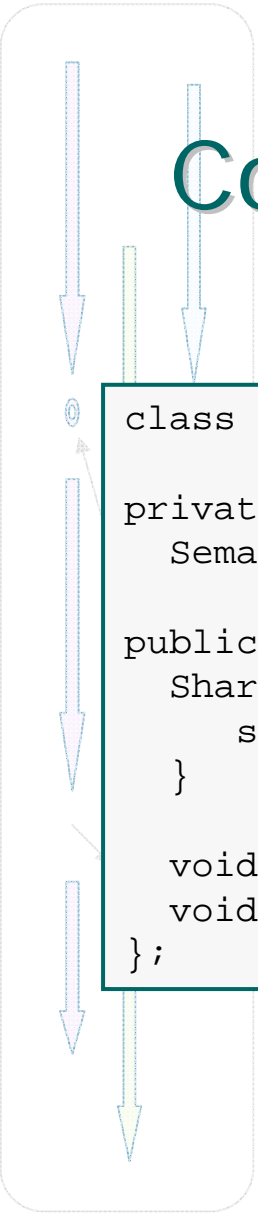
Código das threads

```
SharedAccess SharedSpace;  
  
unsigned ThreadFunc() {  
  
    while(!terminar) {  
        // vai entrar  
        SharedSpace.Entrar();  
        // está dentro  
        SharedSpace.Sair();  
        // saiu  
    }  
}
```



Exercício de sincronismo

Controlador de acesso concorrente



```
class SharedSpace : private SharedAccess {  
private:  
    Semaforo * sem;  
  
public:  
    SharedSpace(unsigned max=4) {  
        sem = new Semaforo(max, max);  
    }  
  
    void Entrar(void) { ... }  
    void Sair(void) { ... }  
};
```

```
void Entrar(void) {  
    sem->Wait();  
}  
  
void Sair(void) {  
    sem->Signal();  
}
```



Exercício de sincronismo

Controlador de acesso concorrente

Versão: Transferência da exclusão mútua

```
class SharedSpace : private SharedAccess {  
private:  
    unsigned n, ntb;  
    Semaforo * sMutex;  
    Semaforo * sBlock;  
  
public:  
    SharedSpace(unsigned max=4) {  
        MAX = max; n = ntb = 0;  
        sMutex = new Semaforo(1,1);  
        sBlock = new Semaforo(0,1);  
    }  
  
    void Entrar(void) { ... }  
    void Sair(void) { ... }  
};
```

Transferência da exclusão mútua
para a tarefa libertada que entra

```
void Entrar(void) {  
    sMutex->Wait();  
    if ( n == MAX) {  
        ++ntb;  
        sMutex->Signal();  
        sBlock->Wait();  
        --ntb;  
    }  
    ++n;  
    sMutex->Signal();  
}
```

Recebi a exclusão
mútua de quem saiu

```
void Sair(void) {  
    sMutex->Wait();  
    --n;  
    if(ntb>0) sBlock->Signal();  
    else sMutex->Signal();  
}
```



Exercício de sincronismo

Controlador de acesso concorrente

Versão: Obter explicitamente a exclusão mútua sempre que necessário

```
class SharedSpace : private SharedAccess {  
private:  
    unsigned n, ntb;  
    Semaforo * sMutex;  
    Semaforo * sBlock;  
  
public:  
    SharedSpace(unsigned max=4) {  
        MAX = max; n = ntb = 0;  
        sMutex = new Semaforo(1,1);  
        sBlock = new Semaforo(0,1);  
    }  
  
    void Entrar(void) { ... }  
    void Sair(void) { ... }  
};
```

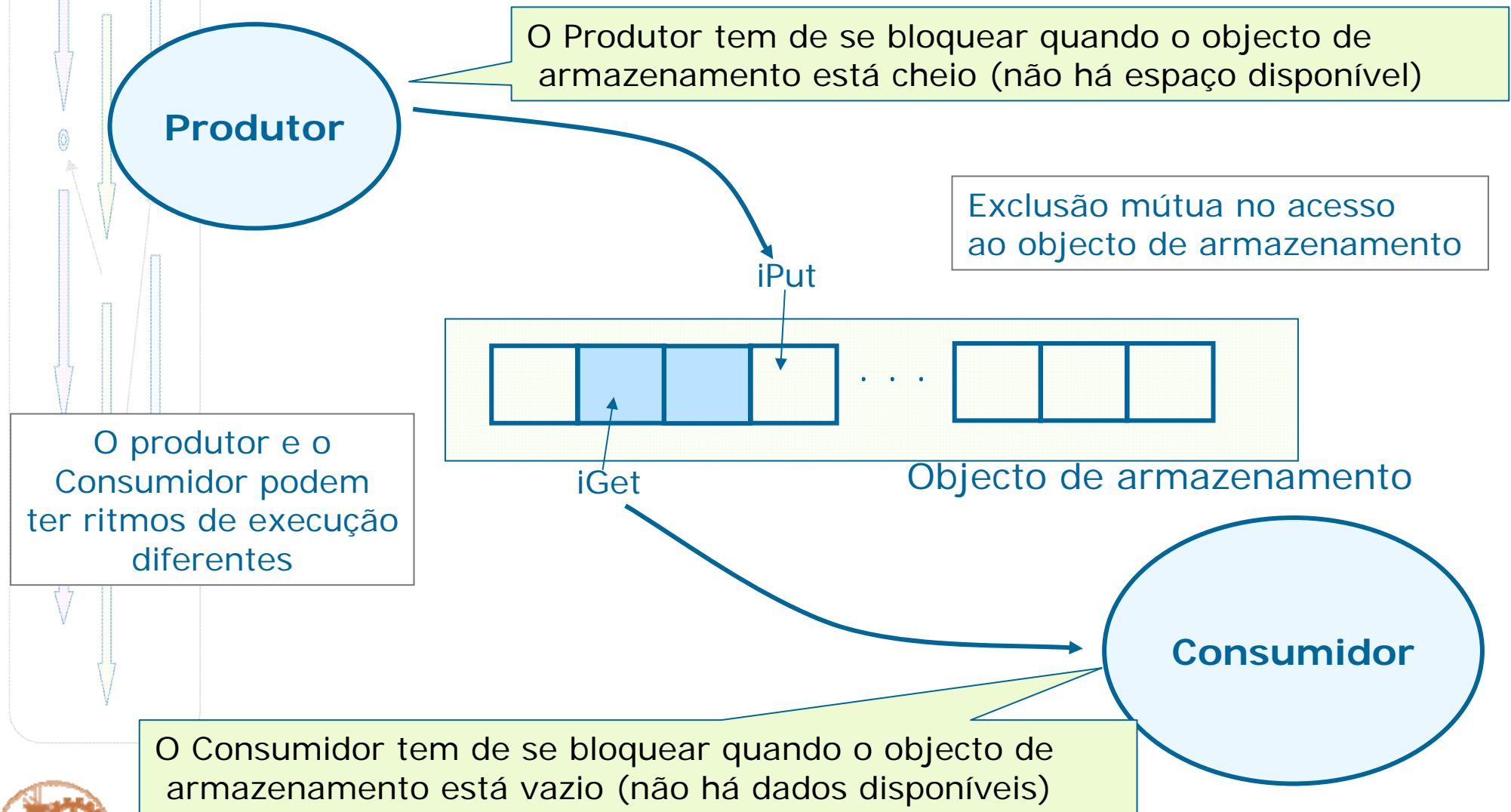
Liberto a exclusão mútua. A tarefa que libertei terá de concorrer para ganhar a exclusão juntamente com todas as outras interessadas.

```
void Entrar(void) {  
    sMutex->Wait();  
    while ( n == MAX) {  
        ++ntb;  
        sMutex->Signal();  
        sBlock->Wait();  
        sMutex->Wait();  
        --ntb;  
    }  
    ++n;  
    sMutex->Signal();  
}  
  
void Sair(void) {  
    sMutex->Wait();  
    --n;  
    if(ntb>0) sBlock->Signal();  
    sMutex->Signal();  
}
```

Obtém a exclusão mútua



Problema do Buffer de dimensão limitada ou Produtor/Consumidor



Produtor/Consumidor

Versão: com variáveis de estado

```
void put(TCHAR ch) {
    sMutex->Wait();
    if (nelems==SIZEBUF) {
        ++nputw;
        sMutex->Signal();
        sPutWait->Wait();
        --nputw;
    }
    ++nelems;

    // colocar elemento no buffer
    buf[iput] = ch;
    iput = (iput+1) % SIZEBUF;

    // se Get em espera acordá-lo
    if (ngetw>0) sGetWait->Signal();
    else sMutex->Signal();
} // put
```

```
TCHAR get() {
    TCHAR ch;
    sMutex->Wait();
    if (nelems==0) {
        ++ngetw;
        sMutex->Signal();
        sGetWait->Wait();
        --ngetw;
    }
    --nelems;

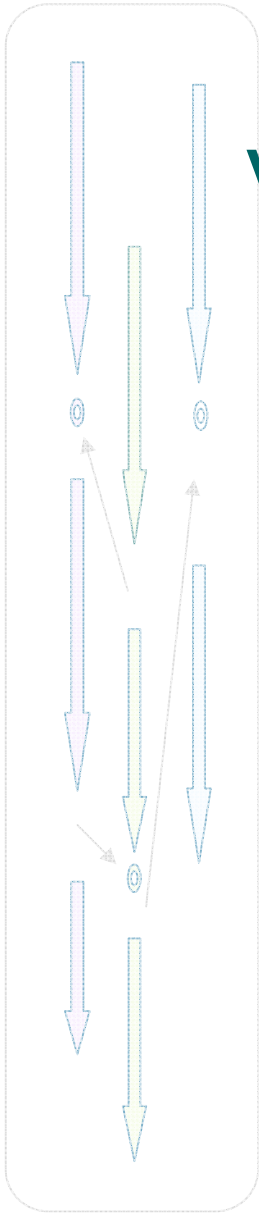
    // retirar elemento do buffer
    ch = buf[iget];
    iget = (iget+1) % SIZEBUF;

    if (nputw) sPutWait->Signal();
    else sMutex->Signal();
    return ch;
} // get
```



Produtor/Consumidor

Versão: com variáveis de estado

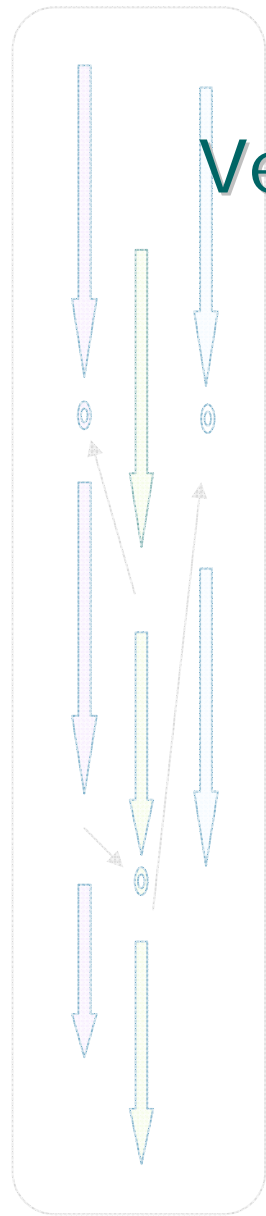


```
class SharedBuffer {  
private:  
    TCHAR buf[SIZEBUF];           // Buffer  
    int iput, iget;                // posicao Put e get  
    int nelems, nputw, ngetw;      // puts gets em espera  
    Semaforo *sMutex,             // sem para exclusão mútua  
            *sGetWait,            // sem para bloquear Gets  
            *sPutWait;            // sem para bloquear Puts  
  
public:  
    SharedBuffer() {  
        pput      = pget  = currsize = 0;  
        nelems = nputw = ngetw = 0;  
  
        sMutex    = new Semaforo(1, 1);  
        sGetWait  = new Semaforo(0, 1);  
        sPutWait  = new Semaforo(0, 1);  
    }  
    ~SharedBuffer() {  
        delete sMutex;  
        delete sGetWait ;  
        delete sPutWait ;  
    }  
}
```



Produtor/Consumidor

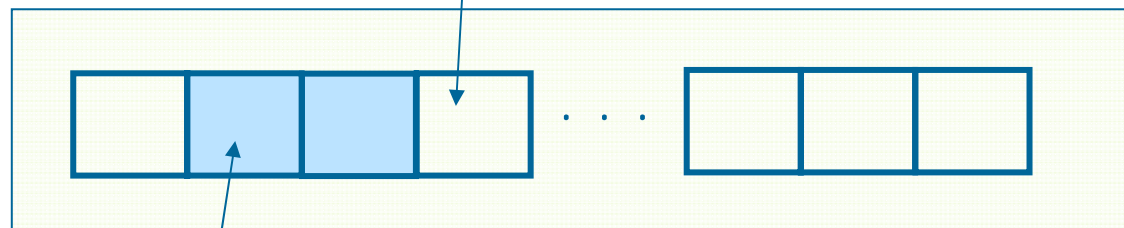
Versão: controle de recursos por semáforos



sVazios

Esperar por espaços vazios
para poder produzir

iPut



iGet

sCheios

Esperar por espaços cheios
para poder consumir

Put

1. sVazios->Wait()
2. Colocar
3. sCheios->Signal()

Get

1. sCheios->Wait()
2. Retirar
3. sVazios->Signal()



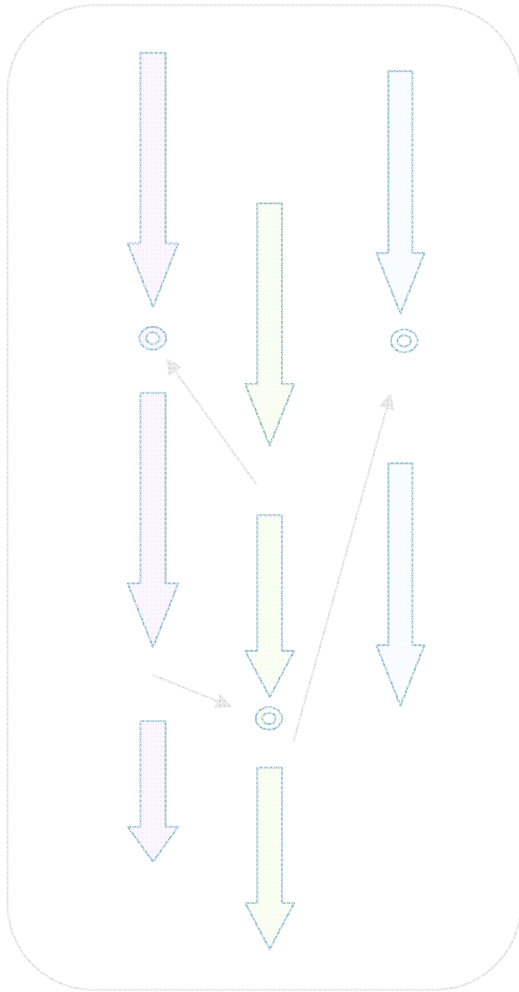
Produtor/Consumidor

Versão: controle de recursos por semáforos

```
class SharedBuffer {  
private:  
    char buf[SIZEBUF];  
    int iut, iget;    // posicao Put e Get  
    Semaforo *sMutex;  
    Semaforo *sCheios;  
    Semaforo *sVazios;  
public:  
    SharedBuffer() {  
        iput      = iget = 0;  
        sMutex    = new Semaforo (1, 1);  
        sCheios   = new Semaforo (0, SIZEBUF);  
        sVazios   = new Semaforo (SIZEBUF, SIZEBUF);  
    }  
  
    ~SharedBuffer() {  
        delete sMutex;  
        delete sCheios;  
        delete sVazios;  
    }  
}
```

```
void put(char ch) {  
    sVazios->Wait();  
    sMutex->Wait();  
    buf[iput] = ch;  
    iput = (iput+1) % SIZEBUF;  
    sMutex->Signal();  
    sCheios->Signal();  
} // put  
  
char get() {  
    char ch;  
    sCheios->Wait();  
    sMutex->Wait();  
    ch = buf[pget];  
    iget = (iget+1) % SIZEBUF;  
    sMutex->Signal();  
    sVazios->Signal();  
    return ch;  
} // get  
  
}; // end class SharedBuffer
```





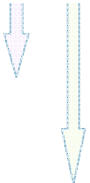
Exercícios propostos



Problema: leitores escritores

Vão existir thread leitoras e threads escritoras. As threads leitoras podem aceder em simultâneo entre si. As threads escritoras têm de aceder sempre em exclusividade.

Faça um controlador que implemente este comportamento mas que dê prioridade total às threads leitoras.



```
class ReadersWriters {  
public:  
    virtual void EntrarRd() = 0;  
    virtual void SairRd() = 0;  
    virtual void EntrarWr() = 0;  
    virtual void SairWr() = 0;  
};
```



Problema: controle de entradas

Tendo em conta que irão existir threads de tipos A e B. Pretende-se um controlador que deixe entrar em acesso as threads A e B tendo em conta que existem NA e NB entradas directas para threads do tipo A e B respectivamente.

```
class ControlAccess {  
protected:  
    unsigned NA, NB;  
public:  
    virtual void EntrarTA() = 0;  
    virtual void SairTA() = 0;  
    virtual void EntrarTB() = 0;  
    virtual void SairTB() = 0;  
};
```

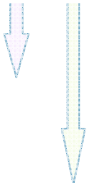




Problema: acesso alternado

Tendo em conta que irão existir threads de tipos A e B. Pretende-se um controlador que deixe entrar em modo exclusivo e alternadamente threads do tipo A e B.

Faça uma segunda versão deste controlador de modo a que a alternância só se verifique em caso de conflito de acesso.



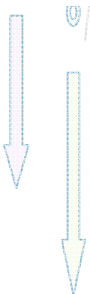
```
class AlternatedAccess {  
public:  
    virtual void EntrarTA() = 0;  
    virtual void SairTA() = 0;  
    virtual void EntrarTB() = 0;  
    virtual void SairTB() = 0;  
};
```





Problema: ponto de encontro

O mecanismo de Ponto de Encontro permite trocar informação entre duas threads. Uma thread coloca informação e a outra retira. Mas a troca tem de ser síncrona entre o emissor e o receptor. Não havendo portanto cache da informação pelo mecanismo.



```
template <class T>
class PENCONTRO {
    T* msgPt;    // Mensagem a trocar
public:
    virtual void Enviar    (T& M)=0;
    virtual void Receber   (T& M)=0;
};
```

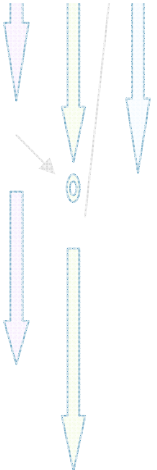




Problema:

Semáforo com WaitUntilZero

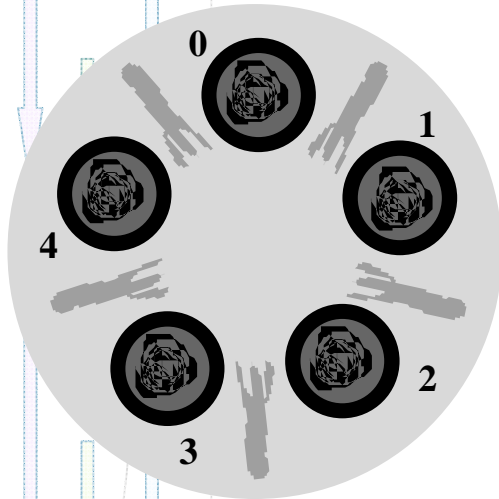
Pretende-se a implementação de uma classe semáforo que possua uma função chamada de waitUntilZero, que bloqueará as threads até que o número de unidades no semáforo seja zero.



```
class WUZSemaforo {  
public:  
    virtual void wait() = 0;  
    virtual void signal() = 0;  
    virtual void waitUntilZero() = 0;  
};
```



Problema Clássico do Jantar dos Filósofos



(*Dining Philosophers Problem*, Diskstra, 1965)
Problema clássico onde a utilização de semáforos permite resolver os problemas de sincronização e evitar situações de bloqueio infinito (*Dead Lock*).

- Existem vários filósofos que passam a vida a PENSAR, COMER e DORMIR;
- Os filósofos só comem espaguete quando conseguirem agarrar os dois garfos que se encontram colocados ao lado do seu prato;
- A solução deve garantir:
 - Ausência de deadlock, (exº: se cada um agarrar um garfo e tentar agarrar o segundo, todos vão ficar bloqueados)

