Sincronização
Parte I – Primitivas de Sincronização

Sistemas Operativos

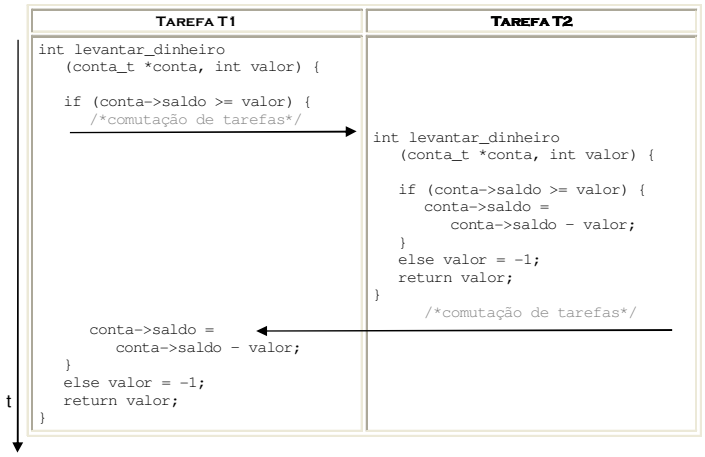
2011 / 2012

Execução Concorrente

```
struct {
   int saldo;
   /* outras variáveis,ex. nome do titular, etc. */
} conta_t;

int levantar_dinheiro (conta_t* conta, int valor) {
   if (conta->saldo >= valor)
      conta->saldo = conta->saldo – valor;
   else
      valor = –1; /* –1 indica erro ocorrido */
   return valor;
}
```

Problema se for multi-tarefa?

Execução Concorrente



Os programas em linguagem máquina têm acções mais elementares

;assumindo que a variável conta->saldo está na posição SALDO da memória

;assumindo que variável valor está na posição VALOR da memória

```
mov AX, SALDO ;carrega conteúdo da posição de memória
              ;SALDO para registo geral AX
mov BX, VALOR ;carrega conteúdo da posição de memória
              ;VALOR para registo geral BX
sub  AX, BX   ;efectua subtracção AX = AX - BX
mov SALDO, AX  ;escreve resultado da subtracção na
               ;posição de memória SALDO
```

Secção crítica

Em programação concorrente sempre que se testam ou se modificam estruturas de dados partilhadas é necessário efectuar esses acessos dentro de uma secção crítica.

Secção Crítica

```
int levantar_dinheiro (ref *conta, int valor)
{
  fechar(); /* lock() */
  if (conta->saldo >= valor) {
    conta->saldo = conta->saldo – valor;
  } else valor = -1
  abrir(); /* unlock */
  return valor;
}
```

Secção crítica (executada em exclusão mútua)

Secção Crítica: Propriedades

- Exclusão mútua
- Progresso (liveness)
  - Ausência de interblocagem (deadlock)
  - Ausência de míngua (starvation)
  - Eficiência

Secção Crítica: Implementações

- Algorítmicas
- Hardware
- Sistema Operativo

Tentativa de Solução #1

```
int trinco = ABERTO;

Fechar () {
  while (trinco == FECHADO)  {/* instrução vazia */};
  trinco = FECHADO;
}

Abrir () {
  trinco = ABERTO;
}
```

- quando T1 executa a função Fechar, podemos observar que o teste da variável trinco (efectuado no ciclo while) e a atribuição do valor FECHADO (feito na instrução seguinte) não é feito de forma atómica.
- pode ocorrer que T1 perca o processador imediatamente após ter saído do ciclo while e antes de executar a instrução trinco = FECHADO.
- se em seguida T2 executar a função Fechar, esta poderá entrar na secção crítica (pois a variável trinco ainda está com o valor ABERTO).

Tentativa de Solução #2

```
int trinco_t1 = ABERTO;
int trinco_t2 = ABERTO;
```

Tarefa T1

```
t1_fechar () {
  while (trinco_t2 == FECHADO);
  trinco_t1 = FECHADO;
}

t1_abrir() {trinco_t1 = ABERTO;}
```

Tarefa T2

```
t2_fechar ( ) {
  while (trinco_t1 == FECHADO);
  trinco_t2 = FECHADO;
}

t2_abrir() {trinco_t2 = ABERTO;}
```

- considerando T1, podemos observar que o teste do trinco trinco_t2 (efectuado no ciclo while) e a atribuição do valor FECHADO ao trinco trinco_t1 não é feito de forma atómica.
- pode ocorrer que T1 perca o processador imediatamente após ter saído do ciclo e antes de executar a atribuição trinco_t1=FECHADO.
- se em seguida a tarefa T2 executa a função t2_fechar, este poderá entrar na secção crítica (pois trinco_t1 ainda está com o valor ABERTO).

Tentativa de Solução #3 (igual à #2 mas com linhas trocadas)

```
int trinco_t1 = ABERTO;
int trinco_t2 = ABERTO;
```

Tarefa T1

```
t1_fechar () {
  trinco_t1 = FECHADO;
  while (trinco_t2 == FECHADO);
}

t1_abrir() {trinco_t1 = ABERTO;}
```

Tarefa T2

```
t2_fechar ( ) {
  trinco_t2 = FECHADO;
  while (trinco_t1 == FECHADO);
  }

t2_abrir() {trinco_t2 = ABERTO;}
```

- basta que T1 execute a instrução trinco_t1=FECHADO e perca o processador logo de seguida;
- quando T2 se executar, esta irá ficar em ciclo na instrução while(trinco_t1==FECHADO).
- o mesmo sucederá à tarefa T1 quando se voltar a executar, i.e. ficará em ciclo na instrução while(trinco_t2==FECHADO).

Tentativa de Solução #4

```
int trinco_vez = 1;
```

Tarefa T1

```
t1_fechar () {
  while (trinco_vez == 2);
}

t1_abrir () {trinco_vez = 2;}
```

Tarefa T2

```
t2_fechar () {
  while (trinco_vez == 1);
}

t2_abrir () {trinco_vez = 1;}
```

- a tarefa T2 (T1) pode bloquear-se fora da secção crítica o que, devido à necessidade de alternância, impede T1 (T2) de executar a secção crítica no futuro;
- na ausência de contenção, se a tarefa T1 quiser entrar na secção crítica, não o conseguirá fazer de uma forma eficiente, pois terá de esperar que T2 execute a secção crítica.
- portanto, esta solução tem os seguintes problemas:
  - progresso - não é cumprido;
  - eficiência - não é cumprido.

Algoritmo de Peterson

```
int trinco_t1 = ABERTO;
int trinco_t2 = ABERTO;
int tar_prio = 1;

t1_fechar () {
  A1: trinco_t1 = FECHADO;
  B1: tar_prio = 2;
  C1: while (trinco_t2 == FECHADO && tar_prio == 2);
}

t1_abrir () {trinco_t1 = ABERTO;}

t2_fechar () {
  A1: trinco_t2 = FECHADO;
  B1: tar_prio = 1;
  C1: while (trinco_t1 == FECHADO && tar_prio == 1);
}

t2_abrir () {trinco_t2 = ABERTO;}
```

T1 pretende aceder à secção crítica

T2 é mais prioritária

Algoritmo de Lamport (Bakery)

```
int senha[N];   // Inicializado a 0
int escolha[N];   // Inicializado a FALSE

Fechar (int i) {
  int j;
  escolha[i] = TRUE;
  senha [i] = 1 + maxn(senha);
  escolha[i] = FALSE;

  for (j=0; j<N; j++) {
    if (j==i) continue;
    while (escolha[j]) ;
    while (senha [j] && (senha [j] < senha [i]) ||
           (senha [i] == senha [j] && j < i));
  }
}

Abrir (int i) {senha [i] = 0;}
```

- senha contém o número da senha atribuído à tarefa
- escolha indica se a tarefa está a pretender aceder à secção crítica

•Pi indica que está a escolher a senha
•Escolhe uma senha maior que todas as outras
•Anuncia que escolheu já a senha

•Pi verifica se tem a menor senha de todos os Pj

•Se Pj estiver a escolher uma senha, espera que termine

•Neste ponto, Pj ou já escolheu uma senha, ou ainda não escolheu
  •Se escolheu, Pi vai ver se é menor que a sua
  •Se não escolheu, vai ver a de Pi e escolher uma senha maior

•Se a senha de Pi for menor, Pi entra
•Se as senhas forem iguais, entra o que tiver o menor identificador

Soluções Algorítmicas

- Conclusão:
  - Complexas → Latência
  - Só contemplam **espera activa**
- Solução:
  - Introduzir instruções hardware para facilitar a solução

Soluções com Suporte do Hardware

- Abrir( ) e Fechar( ) usam instruções especiais oferecidas pelos processadores:
  - Inibição de interrupções
  - Exchange (xchg no Intel)
  - Test-and-set (cmpxchg no Intel)

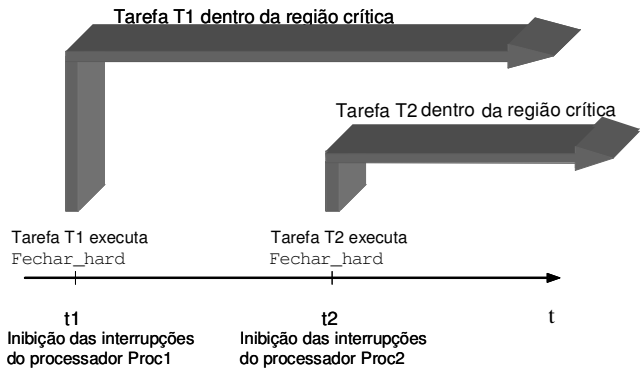
Exclusão Mútua com Inibição de Interrupções

```
int mascara;
```

Fechar_hard () { ** mascara = mascarar_int ();** **}**	**Abrir_hard () {** ** repoe_int (mascara);** **}**

- Este mecanismo só deve ser utilizado dentro do sistema operativo em secções críticas de muito curta duração
  - inibição das interrupções impede que se executem serviços de sistema (I/O, etc)
  - se o programa se esquecer de chamar abrir(), as interrupções ficam inibidas e o sistema fica parado
- Não funciona em multiprocessadores

Inibição das Interrupções não Funciona em Multiprocessadores



Test-and-Set

```
ABERTO  EQU 0              ; ABERTO equivale ao valor 0
FECHADO EQU 1              ; FECHADO equivale ao valor 1

Fechar_hard:
L1:     MOV AX, 0          ; AX é inicializado com o valor 0 (ABERTO)
        BTS trinco, AX     ; testa se trinco e AX são iguais
        JC L1              ; a carry flag fica com o valor inicial do trinco
                           ; se carry flag ficou a 1, trinco estava FECHADO
                           ; implica voltar a L1 e tentar de novo
                           ; se carry flag ficou a 0, trinco estava ABERTO
        RET                ; trinco fica a 1 (FECHADO)

Abrir_hard:
        MOV     AX, ABERTO
        MOV     trinco, AX
        RET
```

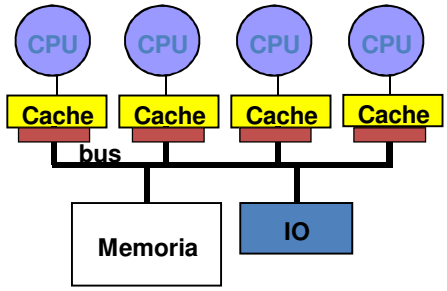

XCHG

```
ABERTO  EQU     0       ; ABERTO equivale ao valor 0
FECHADO EQU     1       ; FECHADO equivale ao valor 1

trinco DW       0       ; trinco com valor inicial ABERTO

Fechar_hard:
        MOV  AX, FECHADO
L1:     XCHG AX, trinco
        CMP  AX, ABERTO  ; compara o valor de AX com 0
        JNZ  L1          ; se AX é diferente de zero volta para L1
        RET

Abrir_hard:
        MOV AX, ABERTO
        MOV trinco, AX
        RET
```

- a instrução XCHG assegura que o trinco fica fechado:
  - pois escreveu o valor um (que estava no registo AX) na posição de memória trinco (bus trancado)
- qualquer outra tarefa que posteriormente execute a instrução cmp AX,ABERTO vai detectar que o valor do trinco é um (está fechado)

XCHG em Multiprocessadores

	P1	P2
Instante 1	P1 inicia exchange e tranca o bus	
Instante 2	P1 completa exchange e tranca a secção crítica	P2 tenta fazer exchange mas bloqueia-se a tentar obter o bus
Instante 3	P1 entra secção crítica	P2 verifica que o trinco está trancado e fica em espera activa



Soluções com Suporte do Hardware

- Conclusão:
  - Oferecem os mecanismos básicos para a implementação da exclusão mútua, mas...
  - Algumas não podem ser usadas directamente por programas em modo utilizador
    - e.g., inibição de interrupções
  - Outras só contemplam espera activa
    - e.g., exchange, test-and-set

Soluções com Suporte do SO

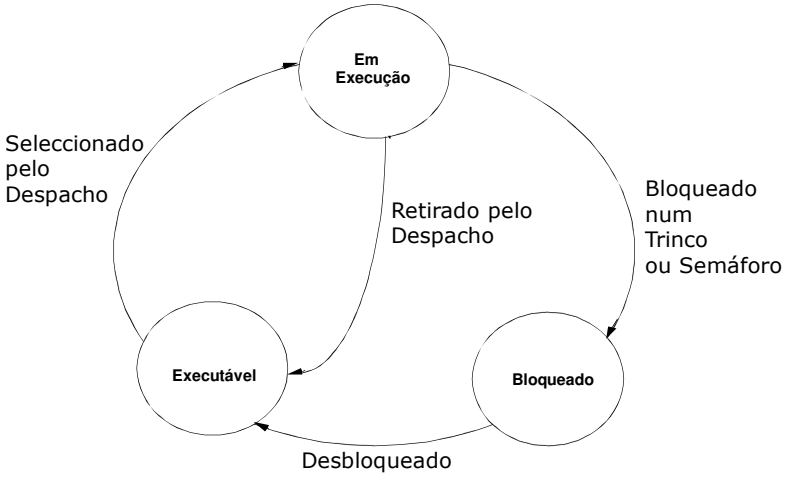
- Primitivas de sincronização são chamadas ao SO:
  - Software trap (interrupção SW)
  - Comutação para modo núcleo
  - Estruturas de dados e código de sincronização pertence ao núcleo
  - Usa o suporte de hardware (exch. / test-and-set)
- Veremos duas primitivas de sincronização:
  - Trincos
  - Semáforos

Soluções com Suporte do SO

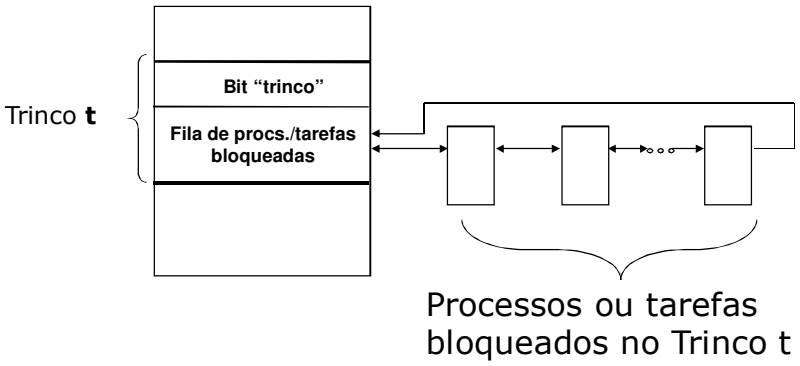
- Vantagens (intuitivas):
  - se SO souber quais processos estão à espera de secção crítica, nem sequer lhes dá tempo de processador
  - evita-se a espera activa!
- Soluções hardware podem ser usadas, mas pelo código do SO:
  - dentro das chamadas sistema que suportam o abrir(), fechar(), etc.

Sistemas Operativos – DEI - IST

Diagrama de Estado dos Processos / Tarefas



Estruturas de Dados Associadas aos Trincos



Alocador de Memória – exemplo de programação concorrente

```
#define MAX_PILHA 100
char* pilha[MAX_PILHA];
int topo = MAX_PILHA-1;

char* PedeMem() {
 ptr = pilha[topo];
 topo--;
 return ptr;
}

void DevolveMem(char* ptr) {
 topo++;
 pilha[topo]= ptr;
}
```

O programa está errado porque as estruturas de dados não são actualizadas de forma atómica

Alocador de Memória com Secção Crítica

```
#define MAX_PILHA 100
char* pilha[MAX_PILHA];
int topo = MAX_PILHA-1;
trinco_t mutex = ABERTO;

char* PedeMem() {
  Fechar_mutex(mutex);
  ptr = pilha[topo];
  topo--;
  Abrir_mutex(mutex);
  return ptr;
}

void DevolveMem(char* ptr) {
  Fechar_mutex(mutex);
  topo++;
  pilha[topo]= ptr;
  Abrir_mutex(mutex);
}
```

Um trinco é criado sempre no estado ABERTO

No início da secção crítica, os processos têm que chamar **Fechar_mutex**
Se o trinco estiver FECHADO, o processo espera que o processo abandone a secção crítica. Se estiver ABERTO, passa-o ao estado FECHADO. Estas operações executam-se **atomicamente**.

No fim da secção crítica, os processos têm que chamar **Abrir_mutex**
Passa o trinco para o estado ABERTO ou desbloqueia um processo que esteja à sua espera de entrar na secção crítica

Fechar_Mutex e Abrir_Mutex

```
trinco_t = t;
t.var=ABERTO; // trinco inicialmente ABERTO
t.numTarefasBloqueadas = 0;


Fechar_mutex (trinco_t t) {
    Fechar_hard(t);
    if (t.var == FECHADO) bloqueia_tarefa();
    numTarefasBloqueadas++;
    t.var = FECHADO;
    Abrir_hard(t);
}

 Abrir_mutex (trinco_t t) {
    Fechar_hard(t);
    numTarefasBloqueadas--;
    t.var = ABERTO;
    if (numTarefasBloqueadas >= 0)desbloqueia_tarefa();
    Abrir_hard(t)
 }
```

Fechar_mutex
Se o trinco estiver FECHADO, o processo espera que o processo abandone a secção crítica. Se estiver ABERTO, passa-o ao estado FECHADO. Estas operações executam-se **atomicamente**.

- Retirar tarefa de execução
- Salvaguardar o seu contexto
- Marcar o seu estado como bloqueada
- Colocar a estrutura de dados que descreve a tarefa na fila de espera associada ao trinco
- Invocar o algoritmo de escalonamento

Abrir_mutex
Passa o trinco para o estado ABERTO ou desbloqueia um processo que esteja à sua espera de entrar na secção crítica

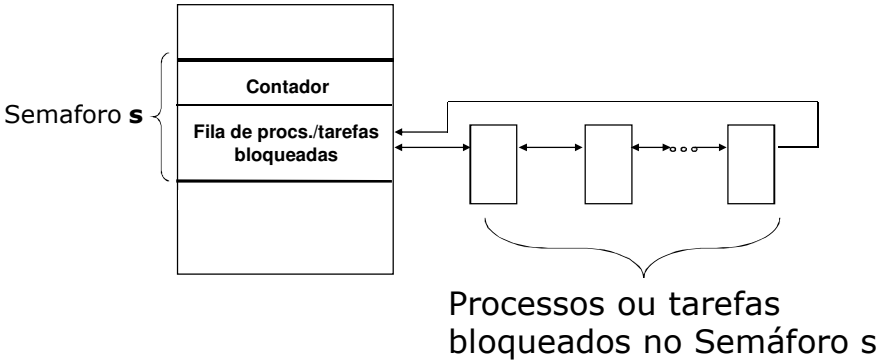
- Existem tarefas bloqueadas
- Marcar o estado de uma delas como "executável"
- Retirar a estrutura de dados que descreve a tarefa da fila de espera associada ao trinco

•É necessário assegurar que variáveis partilhadas são acedidas em exclusão mútua

Trincos – Limitações

- Trincos não são suficientemente expressivos para resolver alguns problemas de sincronização
- Ex: no alocador de memória, bloquear tarefas se a pilha estiver cheia
  - necessário um "contador de recursos" → só bloqueia se o número de pedidos exceder um limite

Semáforos

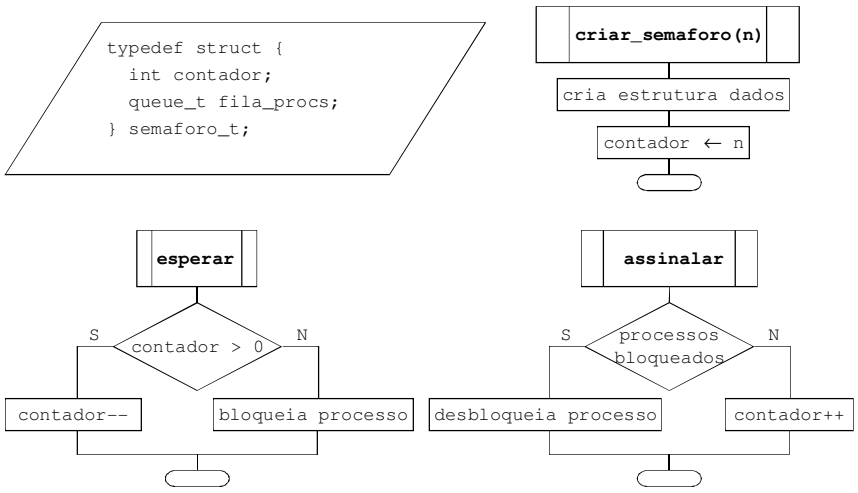


- Nunca fazer analogia com semáforo de trânsito !!!

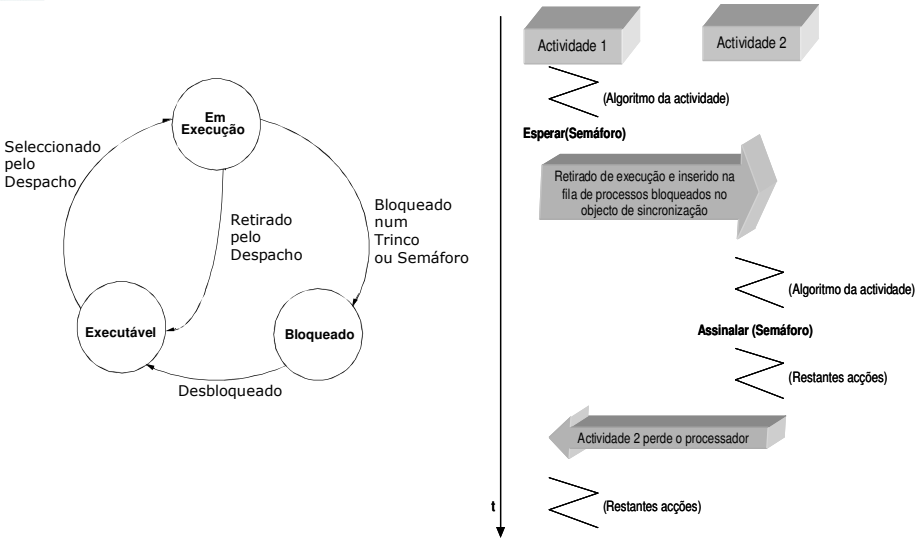
Semáforos: Primitivas

- `s = criar_semaforo(num_unidades)`
  - cria um semáforo e inicializa o contador
- `esperar(s)`
  - bloqueia o processo se o contador for menor ou igual a zero; senão decrementa o contador
- `assinalar(s)`
  - se houver processos bloqueados, liberta um; senão, incrementa o contador
- Todas as primitivas se executam atomicamente
- `esperar()` e `assinalar()` podem ser chamadas por processos diferentes

Semáforos: Primitivas



Semáforos: Primitivas



Alocador de Memória com Secção Crítica

```
#define MAX_PILHA 100
char* pilha[MAX_PILHA];
int topo = MAX_PILHA-1;
trinco_t t = ABERTO;

char* PedeMem() {
  Fechar_mutex(mutex);
  ptr = pilha[topo];
  topo--;
  Abrir_mutex(mutex);
  return ptr;
}

void DevolveMem(char* ptr) {
  Fechar_mutex(mutex);
  topo++;
  pilha[topo]= ptr;
  Abrir_mutex(mutex);
}
```

Um trinco é criado sempre no estado ABERTO

No início da secção crítica, os processos têm que chamar **Fechar_mutex**
Se o trinco estiver FECHADO, o processo espera que o processo abandone a secção crítica. Se estiver ABERTO, passa-o ao estado FECHADO. Estas operações executam-se **atomicamente**.

No fim da secção crítica, os processos têm que chamar **abrir_mutex**
Passa o trinco para o estado ABERTO ou desbloqueia um processo que esteja à sua espera de entrar na secção crítica

E se a pilha estiver completa?

Semáforos (exemplo)

```
#define MAX_PILHA 100
char* pilha[MAX_PILHA];
int topo = MAX_PILHA-1;
  semáforo_t SemMem;
  semáforo_t SemExMut;

char* PedeMem() {
  Esperar(SemMem);
  Esperar(SemExMut);
   ptr = pilha[topo];
   topo--;
  Assinalar(SemExMut);
   return ptr;
}

void DevolveMem(char* ptr) {
  Esperar(SemExMut);
   topo++;
   pilha[topo]= ptr;
  Assinalar(SemExMut);
  Assinalar(SemMem);
}
```

Alocador de memória com o semáforo SemMem a controlar a existência de memória livre

```
main(){
/*...*/
  semExMut = CriarSemaforo(1);
  semMem = CriarSemaforo(MAX_PILHA);
}
```

O semáforo é inicializado com o valor dos recursos disponíveis

Semáforos (exemplo)

```
#define MAX_PILHA 100
char* pilha[MAX_PILHA];
int topo = MAX_PILHA-1;
  semáforo_t SemMem;
  semáforo_t SemExMut;

char* PedeMem() {
  Esperar(SemMem);
  Esperar(SemExMut);
  ptr = pilha[topo];
  topo--;
  Assinalar(SemExMut);
  return ptr;
}

void DevolveMem(char* ptr) {
  Esperar(SemExMut);
  topo++;
  pilha[topo]= ptr;
  Assinalar(SemExMut);
  Assinalar(SemMem);
}
```

?

```
main(){
/*...*/
  semExMut = CriarSemaforo(1);
  semMem = CriarSemaforo(MAX_PILHA);
}
```

Alocador de memória com o semáforo SemMem a controlar a existência de memória livre

Interblocagem
A troca das operações sobre os semáforos cria uma situação de erro

Semáforos: Variantes

- Genérico:
  - assinalar() liberta um processo qualquer da fila
- FIFO:
  - assinalar() liberta o processo que se bloqueou há mais tempo
- Semáforo com prioridades:
  - o processo especifica em esperar() a prioridade,
  - assinalar() liberta os processos por ordem de prioridades
- Semáforo com unidades:
  - as primitivas esperar() e assinalar() permitem especificar o número de unidades a esperar ou assinalar

Interface POSIX: Mutexes

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            const struct timespec *timeout);
```

Exemplo:

```
pthread_mutex_t count_lock;

pthread_mutex_init(&count_lock, NULL);
pthread_mutex_lock(&count_lock);
count++;
pthread_mutex_unlock(&count_lock);
```



```
#include <stdio.h>
#include <pthread.h>
#define MAX_PILHA 100
char * pilha[MAX_PILHA];
int topo = MAX_PILHA-1;
pthread_mutex_t semExtMut; /* semáforo exclusão mútua */

char* PedeMem()
{
  char* ptr;

  pthread_mutex_lock(&semExtMut);
  if (topo >= 0) { /* verificar se há memória livre */
    ptr = pilha[topo]; /* devolve bloco livre */
  topo--;
  }
  else {ptr = NULL /* indica erro */}
  pthread_mutex_unlock(&semExtMut);
  return ptr;
}
void DevolveMem(char * ptr)
{
  pthread_mutex_lock(&semExtMut);
  topo++;
  pilha[topo] = ptr;
  pthread_mutex_unlock(&semExtMut);
}
int main(int argc, char *argv[]) {
  pthread_mutex_init(&semExtMut, NULL); /* attr default */
  /* programa */
  pthread_mutex_destroy(&semExtMut);
  return 0;

}
```

Alocador com secção crítica programado com mutex da biblioteca Posix

Interface POSIX: Semáforos

```
int sem_init(sem_t *sem, int pshared, unsigned value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);

Exemplo:

sem_t sharedsem;
sem_init(&sharedsem, 0, 1);
sem_wait(&sharedsem);
count++;
sem_post(&sharedsem);
```