Sincronização
Parte I – Primitivas de Sincronização

Sistemas Operativos

2012 / 2013

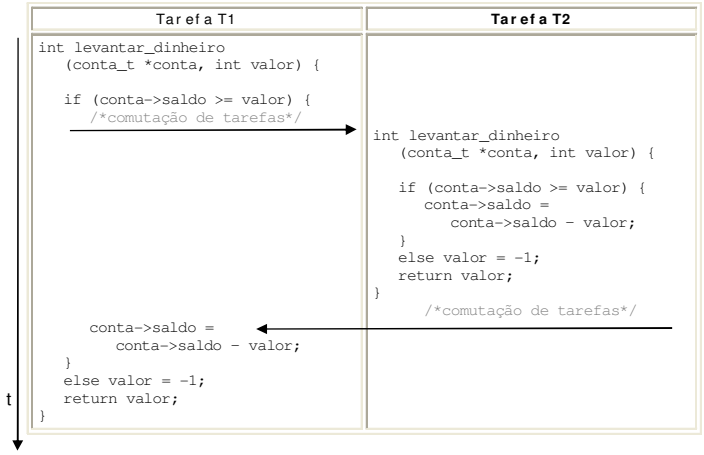
Execução Concorrente

```
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



E se simplificarmos?...

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

int levantar_dinheiro (conta_t* conta, int valor) {

   conta->saldo = conta->saldo - valor;

   return valor;
}
```

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)

Trincos lógicos (mutexes)

Trincos lógicos: Propriedades

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

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

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;
}
```

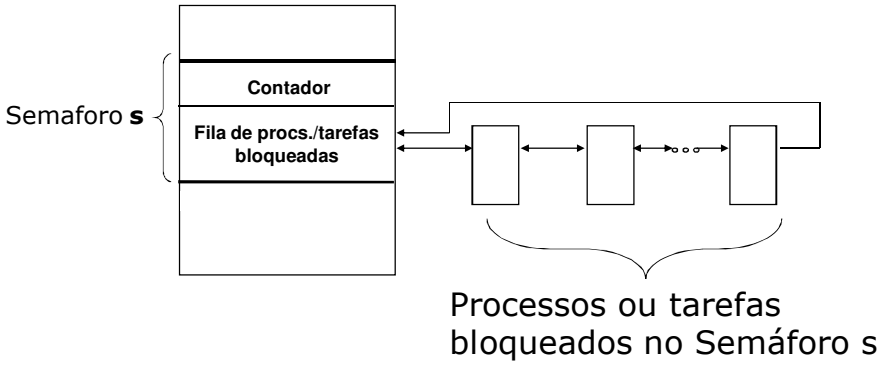
Alocador com secção critica Programado com mutex da biblioteca Posix

Trincos – Limitações

- Trincos não são suficientemente expressivos para resolver alguns problemas de sincronização
  - Ex: Bloquear tarefas se a pilha estiver cheia
  - Necessário um "contador de recursos" → só bloqueia se o número de pedidos exceder um limite

Semáforos

Semáforos



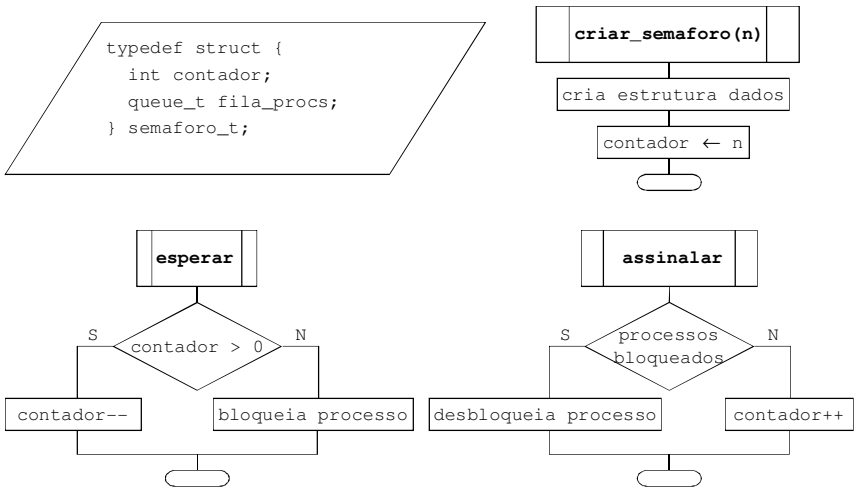
- Nunca fazer paralelo com semáforo de trânsito

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



Para que serve um semáforo?

Uso 1: gerir conjunto limitado de recursos partilhados

Usar semáforos para gerir conjunto limitado de recursos partilhados

- Um processo requisita um recurso
  - Executa Esperar (SemRecursos)
- Um processo liberta um recurso
  - Executa Assinalar (SemRecurso)
- O semáforo que controla o algoritmo é inicializado com o número de recursos disponíveis
  - SemRecurso = CriarSemaforo (NUM_RECURSOS)

Usar semáforos para gerir conjunto limitado de recursos partilhados

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

char* PedeMem() {
    Esperar(SemMem);
    Fechar(mutex);
    ptr = pilha[topo];
    topo--;
    Sair(mutex);
    return ptr;
}

void DevolveMem(char
    Fechar(mutex);
    topo++;
    pilha[topo]= ptr;
    Sair(mutex);
    Assinalar(SemMem);
}
```

```
main(){
/*...*/
  mutex = CriarMutex();
  semMem = CriarSemaforo(MAX_PILHA);
}
```

>?

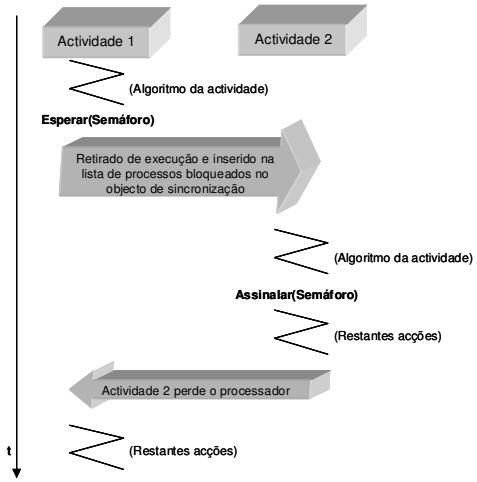
Interblocagem
A troca das operações sobre os semáforos criaria uma situação de erro

O semáforo é inicializado com o valor dos recursos disponíveis

Para que serve um semáforo?

Uso 2: cooperação entre tarefas

Situação mais simples de cooperação



Cooperação usando semáforo

```
Proci
semEvent = CriarSemaforo(0);
...
void EsperarAcontecimento() {
   Esperar (semEvent);
}

Procj
void AssinalarAcontecimento(){
   Assinalar (semEvent);
}
```

O semáforo é inicializado a zero

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: 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);
```