

Cooperação entre Actividades Semáforos

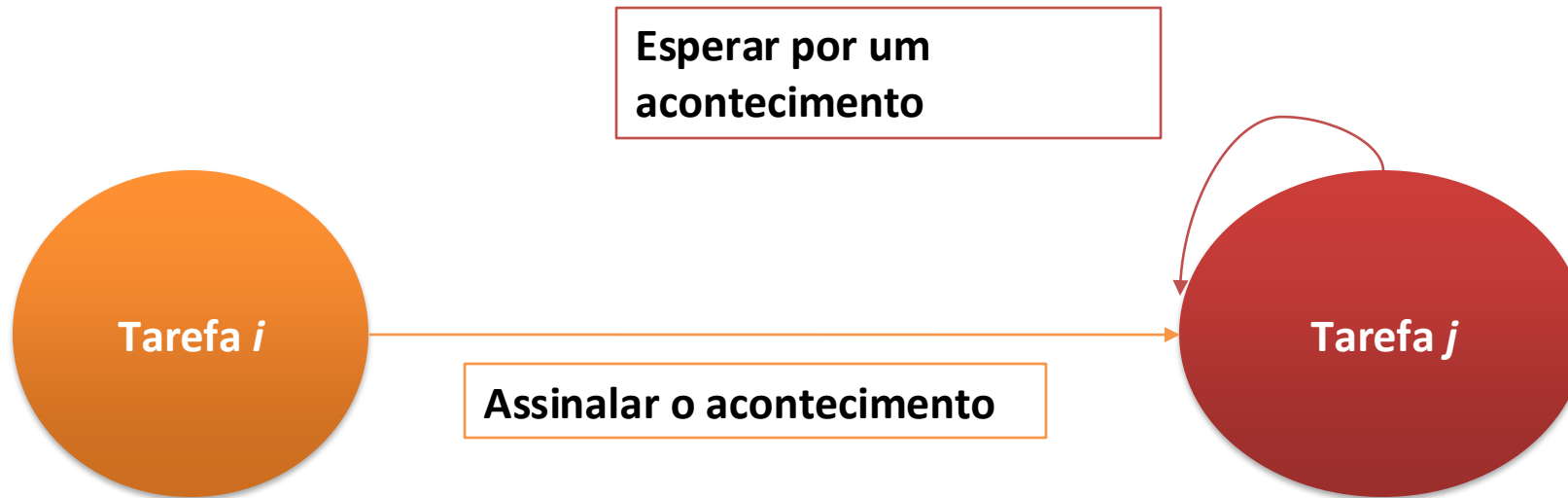
Programação concorrente

Cooperação

- Tal como nas actividades humanas, numa aplicação paralela as tarefas necessitam de colaborar para sinalizarem qual o estado de evolução da sua execução ou para trocarem informações
- Os exemplos são inúmeros :
 - uma tarefa quer saber quando um servidor (tarefa ou processo) acabou um processamento que lhe requisitou;
 - um servidor ser informado quando um cliente lhe envia uma mensagem;
 - um processo que actualiza uma base de dados notificar aos que pretendem ler que terminou.

Sincronizar com um evento

Uma tarefa j quer esperar que outra tarefa i complete uma dada actividade



Notificar um Evento

- Já vimos um caso especial de um processo ou uma tarefa pretender esperar pela terminação de um processo filho/tarefa com as funções `wait` e `thread_join`, mas estas pressupõem sempre a terminação da outra actividade.
- Precisamos de um mecanismo mais genérico

Semântica associada ao evento de assinalar um acontecimento

- Tal como no mundo real, podemos considerar diversas semânticas associadas a um evento
- Cenário do mundo real: numa loja os clientes esperam para poder entrar:
 - Entram todos os clientes à espera
 - Entra só um cliente
 - Entram clientes sempre que na loja estão menos de 5 clientes
 - Entram crianças prioritariamente

**Vamos procurar resolver
todas as semânticas**

Semáforos

- O conceito de semáforo é dos mais antigos na programação concorrente [Dijkstra 1965] e um dos que mais influenciou os mecanismos dos sistemas operativos
- Os semáforos permitem programar os padrões de sincronização mais usuais
- Praticamente todos os sistemas operativos disponibilizam este objecto

Semáforo

- Na ideia original um semáforo é um objecto do sistema operativo que mantem um contador (s)
- E tem duas operações Esperar e Assinalar
 - Esperar
 - Se s é maior que zero então a tarefa pode prosseguir e decrementa o contador
 - senão a tarefa é bloqueada
 - Assinalar
 - Se existir uma tarefa bloqueada coloca-a em execução
 - senão incrementa s

Não tem nada a ver com o funcionamento do semáforo de trânsito

Objecto Semáforo

Propriedades

Identificador

Contador

Lista das tarefas
bloqueadas

Operações

CriarSemáforo

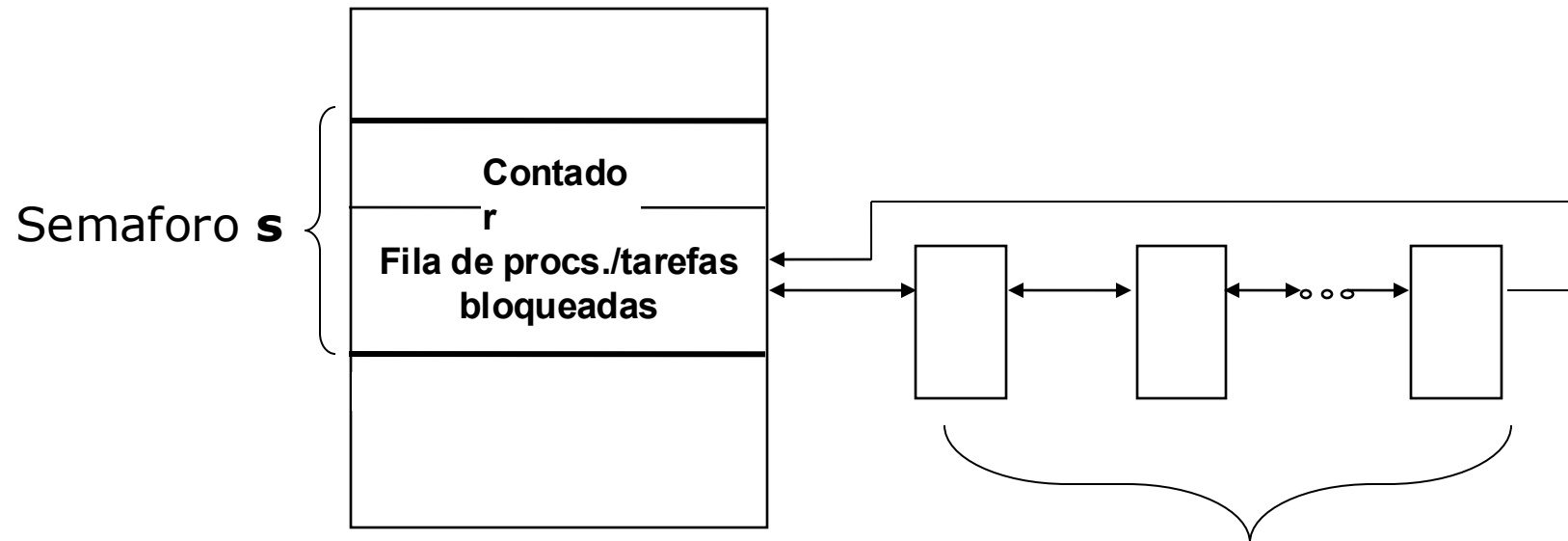
Esperar /wait / down / P

Assinalar /post / signal / up / V

EliminarSemáforo

**P e V são as designações originais de
incrementar/decrementar em holandês**

Semáforos – estrutura de dados no núcleo



Se a tarefa não pode prosseguir é retirada de execução para a fila do semáforo pelo que não há espera activa

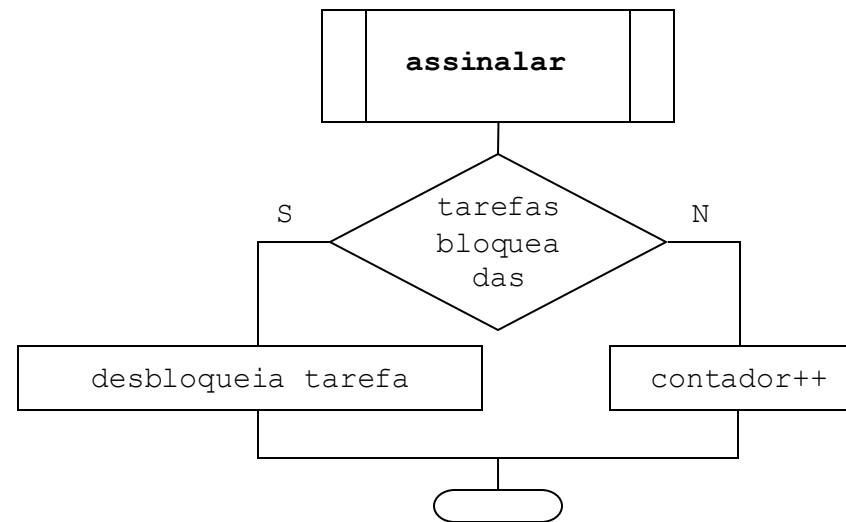
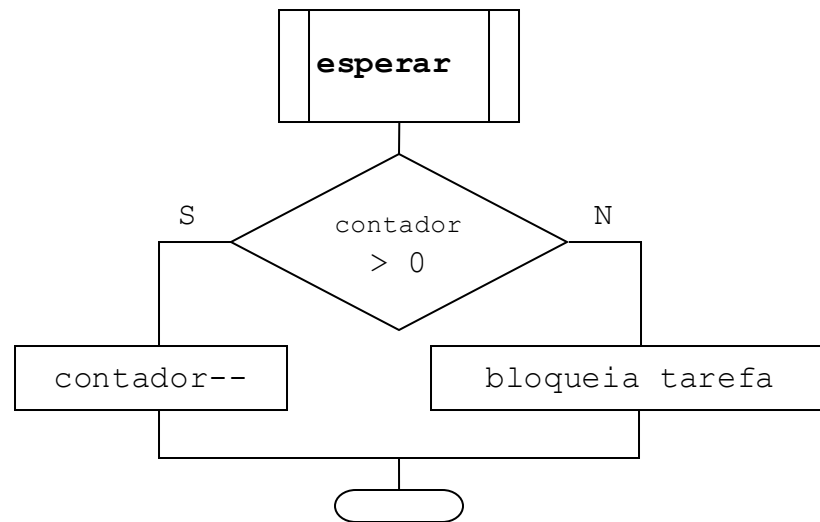
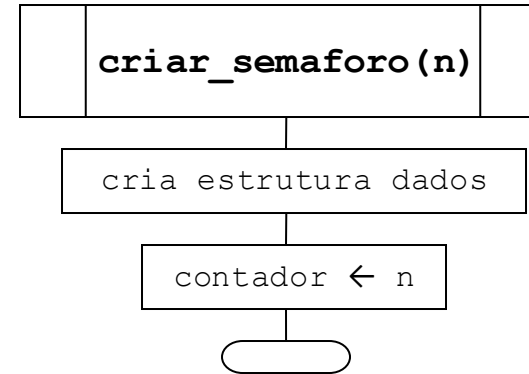
Semáforo: Operações

```
s = CriarSemaforo(num_unidades)
```

- Cria um semáforo e inicializa o contador. O valor inicial do contador é muito importante no funcionamento do semáforo porque define à partida quantas tarefas vão poder fazer
Esperar sem se bloquearem
- Todas as primitivas se executam atomicamente (garantindo a exclusão mútua) no núcleo.

Semáforos: Primitivas

```
typedef struct {
    int contador;
    queue_t fila_procs;
} semaforo_t;
```

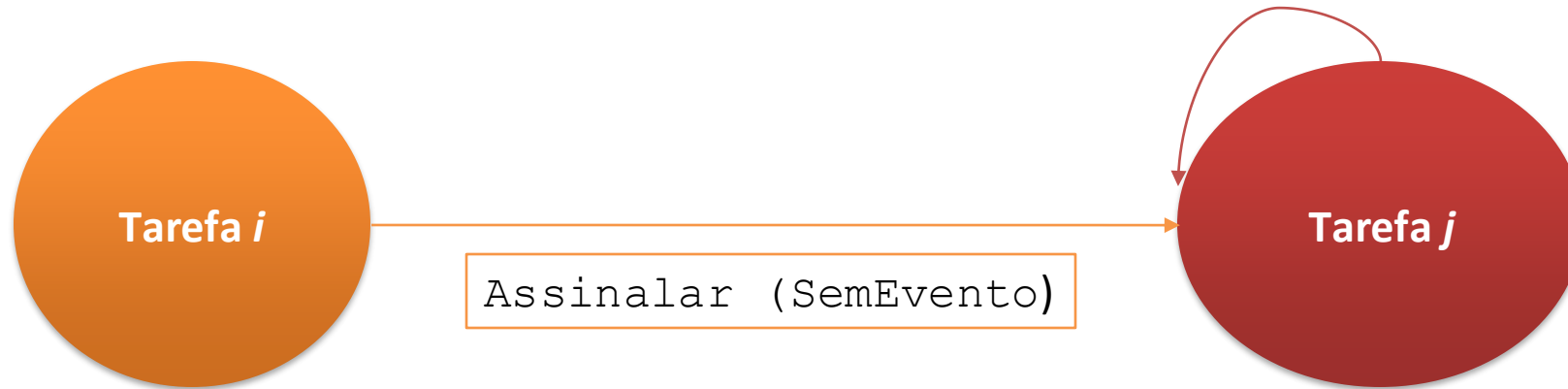


Sincronizar com um evento

Uma tarefa j quer esperar que outra tarefa i complete uma dada actividade

```
CriarSemaforo (SemEvento, 0)
```

```
Esperar (SemEvento)
```



Cenários de utilização do semáforo

- Cenário 1

Tarefa *j* faz Esperar => Semáforo com valor 0 bloqueia-se

Tarefa *i* faz Assinalar => Tarefa *j* é desbloqueada
semáforo permanece com valor 0

- Cenário 2

Tarefa *i* faz Assinalar => incrementa o valor do
semáforo que fica com o valor 1

O contador faz com que o
semáforo memorize os
eventos

Tarefa *j* faz Esperar => Semáforo com valor 1,
decrementa o semáforo e continua,
semáforo fica com o valor 0

Semáforos em Unix

```
#include <semaphore.h>
```

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

```
int sem_wait(sem_t *sem);
```

```
int sem_post(sem_t *sem);
```

pshared = 0 semáforo só pode ser usado entre threads do mesmo processo.
pshared ≠ 0 pode ser usado entre processos independentes

Colaboração: Gestão de Recursos

- Uma situação típica de colaboração é a necessidade de diversas tarefas usarem recursos finitos e terem de se bloquear quando os recursos se esgotam, continuando naturalmente quando os mesmos ficam de novo disponíveis
- Exemplos:
 - As tarefas cliente têm um conjunto de *buffers* limitado para colocarem mensagens
 - O número de ficheiros abertos é limitado

Condições da gestão de recursos

- Uma tarefa quer um recurso
 - Se recursos disponíveis são em número > 0 a tarefa decrementa o numero de recursos e pode prosseguir
 - Se recursos disponíveis = 0 a tarefa é bloqueada
- Uma tarefa liberta um recurso
 - Se existirem tarefas bloqueadas liberta uma
 - Senão incrementa o número de recursos livres

Semáforo na gestão de recursos

- Inicialização:
 - Semáforo criado com o número de recursos disponíveis
- Alocar (reservar) um recurso:
 - Esperar sobre o semáforo
- Libertar um recurso:
 - Assinalar o semáforo

Semáforos e Mutexes

- Um semáforo inicializado a 1 comporta-se como um *mutex* podendo ser usado para implementar uma secção critica (na verdade um recurso unitário que só pode ter uma tarefa a usar de cada vez)
- Funciona, mas não é recomendado fazê-lo (pois a implementação do mutex é mais otimizada, como veremos mais tarde)

A secção crítica como um recurso unitário

```

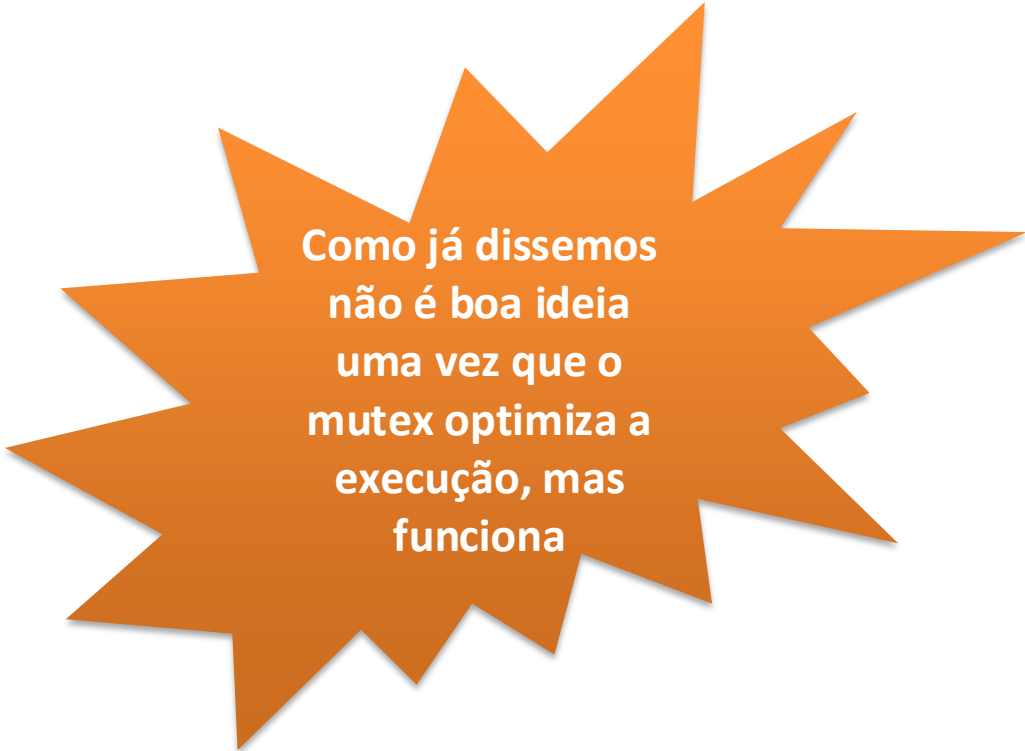
if (sem_init(&sem, 0, 1) == -1)
    errExit("sem_init");

for (j = 0; j < loops; j++) {
    if (sem_wait(&sem) == -1)
        errExit("sem_wait");

    loc = glob;
    loc++;
    glob = loc;

    if (sem_post(&sem) == -1)
        errExit("sem_post");
}

```



Como já dissemos
não é boa ideia
uma vez que o
mutex otimiza a
execução, mas
funciona

Gestão de recursos : Um parque de estacionamento

- Queremos programar um parque de estacionamento
- Os requisitos são simples:
 - Parque dispõe de N lugares (recurso limitado a N)
 - O cliente entra no parque se houver lugares livres, senão bloqueia-se à espera de ter um lugar
 - Quando sai do parque se houver clientes à espera tem de acordar um, senão incrementa o número de lugares livres

Programa Parque de Estacionamento

- Criação de um semáforo `lugaresLivres` inicializado com o número de lugares livres
- Cliente (tarefa) que entra no parque executa `wait`
 - se há lugares livres entra,
 - senão fica boqueado à espera
- Cliente (tarefa) que sai do parque executa `post`
 - se há clientes à espera liberta um,
 - senão incrementa o semáforo indicando que há mais um lugar livre

O contador do semáforo permite modelar directamente a gestão dos recursos



Entrar e Sair do Parque

```
void entrarNoParque() {  
    sem_wait(&lugaresLivres);  
    return;  
}
```

```
void sairDoParque() {  
    sem_post(&lugaresLivres);  
    return;  
}
```



Procedimento de um cliente do estacionamento

```
void * Proccliente (int i){
```

```
    int espera;
```

```
    entrarNoParque();
```

```
    espera = rand() % 20;
```

```
    printf ("cliente # %d estacionado. Vai demorar %d seg \n", i, espera);
```

```
    sleep (espera);
```

```
    sairDoParque();
```

```
    printf ("cliente # %d saiu do parque \n", i);
```

```
    return 0;
```

```
}
```

Gera um valor aleatório para o tempo de estacionamento

Estacionado fica suspenso

Programa principal do estacionamento

```
#define NUM_LUGARES 5

#define MAX_CLIENTES 10

sem_t lugaresLivres;
pthread_t t_cliente[MAX_CLIENTES];

int main () {
    int i;
    void * resul;
    time_t t;

    /* Inicializa o gerador de números aleatórios */
    srand((unsigned) time(&t));

    /* Cria e inicializa o semáforo */
    sem_init(&lugaresLivres, 0, NUM_LUGARES);

    ....
}
```




Programa principal do estacionamento

```
/* Cria os clientes */
for (i=1; i < MAX_CLIENTES; i++)
{
    pthread_create(&t_cliente[i],0, Proccliente, i);
    printf ("criou o cliente numero %d \n",i);
};

/* Espera que os clientes terminem */
printf ("\n espera terminação dos clientes \n \n");
for (i=1; i < MAX_CLIENTES; i++) {
    pthread_join(t_cliente[i], &resul);
    printf ("terminou cliente %d \n", i);
};

printf (" sistema terminou \n");
}
```

Produtores e Consumidores

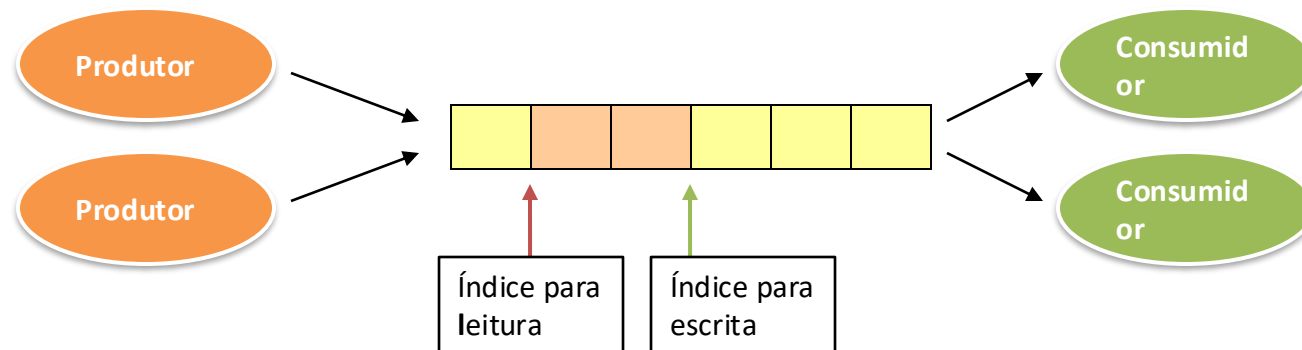
Problemas típicos de Cooperação

- Estes algoritmos aparecem em numerosas situações na programação de sistemas e são importantes como algoritmos de programação paralela, mas também como visão dos algoritmos que o núcleo implementa internamente para disponibilizar algumas destas funcionalidades.
- Vamos ver mais dois em detalhe

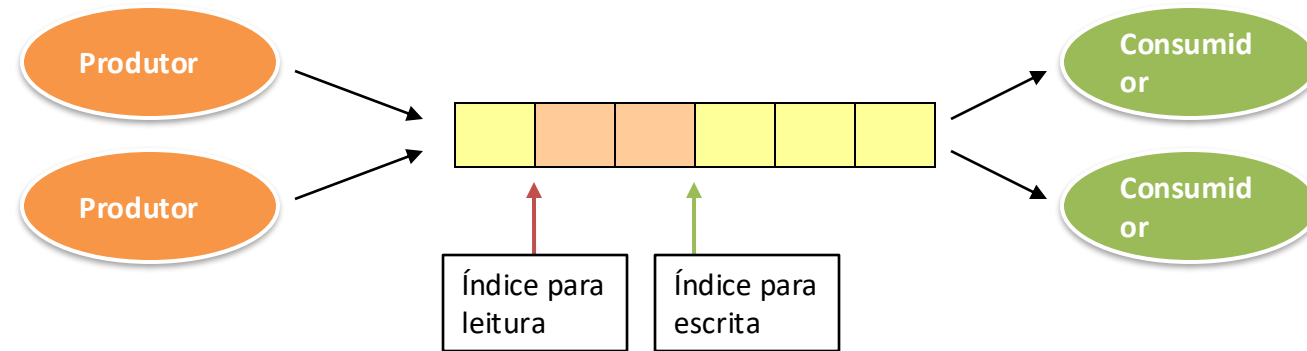
O algoritmo dos Produtores/Consumidores
O algoritmo dos Leitores/Escritores

Produtores e Consumidores

- Definição do problema
 - Pretende-se um mecanismo de cooperação entre tarefas que sirva para transferir informação (mensagens) entre tarefas
- Duas classes de tarefas:
 - Produtores - produzem informação que colocam num *buffer*
 - Consumidores – que consomem/lêem a informação a partir do *buffer*



Produtores e Consumidores



```
void *produtor(void *pi) {
    while (emActividade) {
        < preenche mensagem>
        EscreveMsg(Msg);
    };
}
```

```
void *consumidor(void *pi) {
    while (emActividade){
        LeMsg(&Msg);
        < processa a mensagem >
    };
}
```

Condições de sincronização

- Se uma mensagem for escrita \Rightarrow um consumidor deve ser acordado
- Se o buffer estiver cheio \Rightarrow os produtores devem ser bloqueados
- Se o número de mensagens for igual a 0 \Rightarrow os consumidores devem ser bloqueados
- Se uma mensagem for lida e houver produtores bloqueados \Rightarrow um deve ser acordado

Para simplificar vamos considerar as mensagens todas do mesmo tamanho

Semáforos

- A condição de assinalar a escrita de uma mensagem aos consumidores segue o modelo de utilizar um semáforo para **assinalar um evento com memória**: semáforo `MsgparaLer` inicializado a zero.
- A condição sobre os produtores pode ser modelada como a **gestão de recursos** das posições no buffer que estão limitadas a N mensagens: semáforo `bufferCheio` inicializado a N.

Algoritmo básico: Produtor - Consumidor

```
/* criação dos objectos de sincronização */  
  
sem_init(&MsgparaLer, 0, 0);  
sem_init(&bufferCheio, 0, N);
```

```
void EscreveMsg(t_msg Msg) {  
    sem_wait(&bufferCheio);  
    buffer[indiceEscrita] = Msg;  
    indiceEscrita= (indiceEscrita+1)%N;  
    sem_post(&MsgparaLer);  
}
```

```
void LeMsg(t_msg *Msg) {  
    sem_wait(&MsgparaLer);  
    *Msg = buffer[indiceLeitura];  
    indiceLeitura=(indiceLeitura+1)%N;  
    sem_post(&bufferCheio);  
}
```

Buffer circular de
mensagens

Algoritmo básico :Produtor - Consumidor

```
/* criação dos objectos de sincronização */
```

```
sem_init(&MsgparaLer, 0, 0);
```

```
sem_init(&bufferCheio, 0, N);
```

```
void EscreveMsg(t_msg Msg) {
```

```
sem_wait(&bufferCheio);
```

```
buffer[indiceEscrita] = Msg;
```

```
indiceEscrita= (indiceEscrita+1)%N;
```

```
sem_post(&MsgparaLer);
```

```
}
```

Escreve a mensagem
se o buffer não está
cheio

Assinala que escreveu
uma mensagem

```
void LeMsg(t_msg *Msg) {
```

```
sem_wait(&MsgparaLer);
```

```
*Msg = buffer[indiceLeitura];
```

```
indiceLeitura= (indiceLeitura+1)%N;
```

```
sem_post(&bufferCheio);
```

```
}
```

Algoritmo básico :Produtor - Consumidor

```
/* criação dos objectos de sincronização */
```

```
sem_init(&MsgparaLer, 0, 0);
```

```
sem_init(&bufferCheio, 0, N);
```

```
void EscreveMsg(t_msg Msg) {
```

```
sem_wait(&bufferCheio);
```

```
buffer[indiceEscrita] = Msg;
```

```
indiceEscrita= (indiceEscrita+1)%N;
```

```
sem_post(&MsgparaLer);
```

```
}
```

Bloqueia-se se não há mensagens

Assinala que libertou uma posição do *buffer*

```
void LeMsg(t_msg *Msg) {
```

```
sem_wait(&MsgparaLer);
```

```
*Msg = buffer[indiceLeitura];
```

```
indiceLeitura=(indiceLeitura+1)%N;
```

```
sem_post(&bufferCheio);
```

```
}
```

Funções que escrevem e lêem as mensagens

- Infelizmente a solução anterior está errada!
- Porquê?

Os índices de leitura e escrita são variáveis partilhadas

**Só podem ser testados
ou modificados em
exclusão mútua**



Funções que escrevem e lêem as mensagens

```
/* criação dos objectos de sincronização */  
  
pthread_mutex_t semExMut = PTHREAD_MUTEX_INITIALIZER;  
  
sem_init(&MsgparaLer, 0, 0);  
  
sem_init(&bufferCheio, 0, N);
```

```
void EscreveMsg(t_msg Msg) {  
    sem_wait(&bufferCheio);  
  
    pthread_mutex_lock(&semExMut);  
    buffer[indiceEscrita] = Msg;  
  
    indiceEscrita= (indiceEscrita+1)%N;  
  
    pthread_mutex_unlock(&semExMut);  
  
    sem_post(&MsgparaLer);  
}
```

```
void LeMsg(t_msg *Msg) {  
    sem_wait(&MsgparaLer);  
  
    pthread_mutex_lock(&semExMut);  
  
    *Msg = buffer[indiceLeitura];  
  
    indiceLeitura=(indiceLeitura+1)%N;  
  
    pthread_mutex_unlock(&semExMut);  
  
    sem_post(&bufferCheio);  
}
```

Precaução na utilização dos semáforos

```
void EscreveMsg(t_msg Msg) {
    pthread_mutex_lock(&semExMut);
    sem_wait(&bufferCheio);
    buffer[indiceEscrita] = Msg;
    indiceEscrita= (indiceEscrita+1)%N;
    pthread_mutex_unlock(&semExMut);
    sem_post(&MsgparaLer);
}
```

Atenção esta sequência de instruções está errada!

Interblocagem

```
void EscreveMsg(t_msg Msg) {
    sem_wait(&bufferCheio);
    pthread_mutex_lock(&semExMut);
    buffer[indiceEscrita] = Msg;
    indiceEscrita= (indiceEscrita+1)%N;
    pthread_mutex_unlock(&semExMut);
    sem_post(&MsgparaLer);
}
```

Regra:
Não se pode ter a operação `sem_wait` dentro de uma secção critica

Precaução na utilização dos semáforos

```
void EscreveMsg(t_msg Msg) {
    sem_wait(&bufferCheio);
    pthread_mutex_lock(&semExMut);
    buffer[indiceEscrita] = Msg;
    indiceEscrita= (indiceEscrita+1)%N;
    sem_post(&MsgparaLer);
    pthread_mutex_unlock(&semExMut);
}
```



Problema?

Na operação `sem_post` não há o risco de interblocagem, mas podemos estar a colocar em execução uma tarefa que se irá bloquear no *mutex* que ainda não foi libertado, conduzindo a uma eventual comutação desnecessária de tarefas

Aumentar o paralelismo

- Programámos correctamente, colocando um trinco que protege as secções críticas das duas funções.
- Mas se analisarmos com atenção este programa os produtores e os consumidores não trabalham sobre os mesmos índices, um é exclusivo dos produtores e outro dos consumidores
- Poderíamos ter dois *mutexes* diferentes: criando uma secção crítica para os produtores e outra secção crítica para os consumidores
- Aumentávamos o paralelismo da solução

Tentem efectuar esta alteração

Leitores e Escritores de uma estrutura de dados partilhada

Problema de base

- Uma estrutura de dados é partilhada por múltiplas tarefas
 - (pode ser muito simples ou complexa como as contas do banco de um exemplo anterior)
- Algumas tarefas vão actualizar/modificar a estrutura de dados (ler, eventualmente efectuar algum processamento e escrever)
- Outras vão apenas ler/consultar a estrutura de dados
- Como é uma estrutura de dados partilhada, necessita de sincronização, como vimos em todos os exemplos anteriores

Diferença entre os escritores e leitores

- Ao contrário das secções críticas temos duas classes de tarefas:
- Os leitores podem aceder em paralelo, ou seja, N tarefas podem ler simultaneamente, mas não pode nenhuma estar a modificar os dados
- Há ainda um outro aspecto importante deste problema, normalmente as leituras são muito mais frequentes do que as escrita
- E também normalmente as leituras demoram muito menos tempo que a modificação dos dados

Leitores Escritores: programação

- Duas classes de tarefas:
 - Leitores: apenas lêem a estrutura de dados
 - Escritores: modificam a estrutura de dados
- Condições de sincronização
 - Escritores: só podem aceder em **exclusão mútua**. Não pode haver leitores porque poderiam ler valores inconsistentes, ou outros escritores porque estes poderiam modificar valores testados ou usados na computação pelo escritor
 - Leitores: podem aceder **simultaneamente com outros leitores**, mas em **exclusão mútua com os escritores**

Esta é a semântica oferecida
pelos Read-Write Locks!
Vamos aprender como implementá-los
usando semáforos!

Semáforos

- Necessitamos de dois semáforos:
 - Um para os leitores que se bloqueiam até que lhes seja assinalado que podem ler. Conceptualmente assinala aos leitores o evento “podem ler” , inicializado a 0
 - Um para os escritores que se bloqueiam até que lhes seja assinalado que podem escrever. Conceptualmente assinala aos escritores o evento “podem escrever”, inicializado a 0
- Vamos considerar 4 operações:
 - fechar_leitura; abrir_leitura
 - fechar_escrita; abrir_escrita

Condições lógicas de sincronização

Condições de bloqueio mais complexas

- Como vimos um escritor bloqueia-se se houver um leitor ou um escritor em simultâneo
- Mas quando termina uma escrita?
 - deve ser assinalado o leitor seguinte (se houver) ou o escritor seguinte (se houver).
 - e se não estiver ninguém à espera?
- Os semáforos não têm estas semânticas necessitamos de programa-las com condições lógicas com variáveis de estado (em_escrita, nleitores, nescritores, etc.)

Problema dos Leitores - Escritores

```
void *Leitor(void *param)
{
    fechar_leitura();
    < ler dados>
    abrir_leitura();
    < processar dados >
}
```

```
void *Escritor(void
*param)
{
    fechar_escrita();
    < escrever dados>
    abrir_escrita();
    < outras actividades>
}
```

fechar_leitura

Primeira versão

```
fechar_leitura()
{
    if (em_escrita)
        leitores_espera++;

    sem_wait(&leitores); /* leitores_espera--; */

    else nleitores++;
}
```

Se está em_escrita
os leitores esperam

Decrementado por um
escritor antes de
desbloquear o leitor

Senão incrementam um
contador de leitores que
permite controlar o
número de leitores

Errado sem secções críticas

Segunda versão com secção crítica

```
fechar_leitura()
{
    pthread_mutex_lock(&secCritica);
    if (em_escrita) {
        leitores_espera++;
        pthread_mutex_unlock(&secCritica);
        sem_wait(&leitores); /* leitores_espera--; */
        pthread_mutex_lock(&secCritica);
    }
    else nleitores++;
    pthread_mutex_unlock(&secCritica);
}
```

abrir_leitura

```

abrir_leitura()
{
    pthread_mutex_lock(&secCritica);

    nleitores--;
    if (nleitores == 0 & escritores_espera > 0) {
        sem_post(&escritores);
        em_escrita=TRUE;
        escritores_espera--;
    }
    pthread_mutex_unlock(&secCritica);
}

```

Acabou a leitura abre o trinco

Se não há nenhum leitores e há escritores à espera então liberta um escritor

Senão apenas decrementa os leitores

NOTA: o último leitor está a transferir o controlo para os escritores, caso haja algum à espera, para que estes tenham oportunidade de aceder à estrutura de dados antes do eventual aparecimento de um leitor

Fechar para escrita

```
fechar_escrita()
{
    pthread_mutex_lock(&secCritica);
    if (em_escrita || nleitores > 0) {
        escritores_espera++;
        pthread_mutex_unlock(&secCritica);
        sem_wait(&escritores); /*escritores_espera--;*/
        pthread_mutex_lock(&secCritica);
    }
    em_escrita = TRUE;
    pthread_mutex_unlock(&secCritica);
}
```

Condição que impede os escritores de prosseguir porque há uma tarefa em_escrita ou há leitores

Incrementa o contador de leitores à espera

abrir_escrita()

- Condições mais complexas quando a escrita termina
- Como acabou um escritor, vamos dar oportunidade aos leitores se estiverem à espera (`leitores_espera > 0`)
- Mas não podemos libertar só um, pois todos os leitores deveriam poder continuar
- Hipótese de solução, um ciclo para libertar todos os leitores, mas talvez seja possível melhorar...

```
abrir_escrita()
{
    pthread_mutex_lock(&secCritica);
    em_escrita = FALSE;
    if (leitores_espera > 0)
        {for (int i=0; i<leitores_espera; i++) {
            sem_post(&leitores);
            nleitores++; }
        leitores_espera=0;
    }
    else if (escritores_espera > 0) {
        sem_post(&escritores);
        em_escrita=TRUE;
        escritores_espera--;
    }
    pthread_mutex_unlock(&secCritica);
}
```

O ciclo liberta as tarefas do semáforo leitores mas vão bloquear-se na secção critica

Evitar a Mingua dos escritores

```
fechar_leitura()  
{  
    pthread_mutex_lock(&secCritica);  
    if (em_escrita) {  
        leitores_espera++;  
        pthread_mutex_unlock(&secCritica);  
        sem_wait(&leitores); /* leitores_espera--; */  
        pthread_mutex_lock(&secCritica);  
    }  
    else nleitores++;  
    pthread_mutex_unlock(&secCritica);  
}
```

- Uma coligação de leitores, ou seja, se existir sempre pelo menos um leitor activo, impede os escritores de entrar => *starvation*
- Para evitar a mingua dos escritores necessitamos de condicionar a entrada dos leitores

Evitar a Mingua dos escritores

```

fechar_leitura()
{
    pthread_mutex_lock(&secCritica);
    if (em_escrita || escritores_espera > 0) {
        leitores_espera++;
        pthread_mutex_unlock(&secCritica);
        sem_wait(&leitores); /* leitores_espera--; */
        pthread_mutex_lock(&secCritica);
    }
    else nleitores++;
    pthread_mutex_unlock(&secCritica);
}

```

- Uma coligação de leitores, ou seja, se existir sempre pelo menos um leitor activo, impede os escritores de entrar => *starvation*

- Para evitar a mingua dos escritores, podemos condicionar a entrada a não existirem escritores em espera, estamos a impedir a mingua dos escritores

Libertar todos os leitores no final da escrita

```

fechar_leitura()
{
    pthread_mutex_lock(&secCritica);
    if (em_escrita || escritores_espera > 0) {
        leitores_espera++;
        pthread_mutex_unlock(&secCritica);
        sem_wait(&leitores); /* leitores_espera++ */
        pthread_mutex_lock(&secCritica);
        if (leitores_espera > 0) {
            nleitores++;
            leitores_espera--;
            sem_post(&leitores);
        }
    }
    else
        nleitores++;
    pthread_mutex_unlock(&secCritica);
}

```

```

abrir_escrita()
{
    int i;
    pthread_mutex_lock(&secCritica);
    em_escrita = FALSE;
    if (leitores_espera > 0) {
        sem_post(&leitores);
        nleitores++;
        leitores_espera--;
    }
    else if (escritores_espera > 0) {
        sem_post(&escritores);
        escritores_espera--;
    }
    em_escrita = TRUE;
    pthread_mutex_unlock(&secCritica);
}

```

**Os leitores libertam-se
sucessivamente, solução melhor
que o ciclo dentro da secção
crítica**

Recapitulando

- Variáveis de controlo

```
int nleitores=0, leitores_espera=0, ;
```

```
int em_escrita=FALSE;
```

```
int escritores_espera=0;
```

- Semáforo: evento “pode escrever”

```
sem_init(&escritores, 0, 0);
```

- Semáforo: evento “pode ler”

```
sem_init(&leitores, 0, 0);
```

- *Mutex* para garantir a secção crítica

```
pthread_mutex_t secCritica = PTHREAD_MUTEX_INITIALIZER
```

Trincos de leitura escrita

- Este programa constitui o essencial dos “trincos de leitura escrita” (dai o fechar e o abrir)
- São muito utilizados para programar algoritmos deste tipo em particular nas bases de dados
- Como já vimos, em Linux existe um objecto de sincronização com esta semântica: `pthread_rwlock`

Tudo o que é mais complexo, é mais lento

- Esta implementação mostra que a maior flexibilidade dos *Read-write locks*, comparados com *mutexes*, vem a custo de uma maior complexidade!
- Estes trincos são úteis quando existe elevado paralelismo e:
 - As tarefas leitor são em número muito superior as escritoras.
 - Os escritores detêm o trinco por períodos relativamente longos

Conclusão

Vimos vários exemplos de algoritmos que permitem a colaboração entre tarefas ou processos.

A utilização de semáforos e *mutexes* conjugada com condições lógicas permite resolver qualquer problema

Mas, atenção que os programas são complexos e sujeitos a numerosos erros, em particular a possibilidade de provocarem interblocagem

Por esta razão, os mais utilizados são oferecidos como objectos do sistema operativo ou das linguagens de programação