Programação multi-tarefa em Memória Partilhada
Parte II – Programação avançada

Sistemas Operativos

2018 / 2019

Semáforos

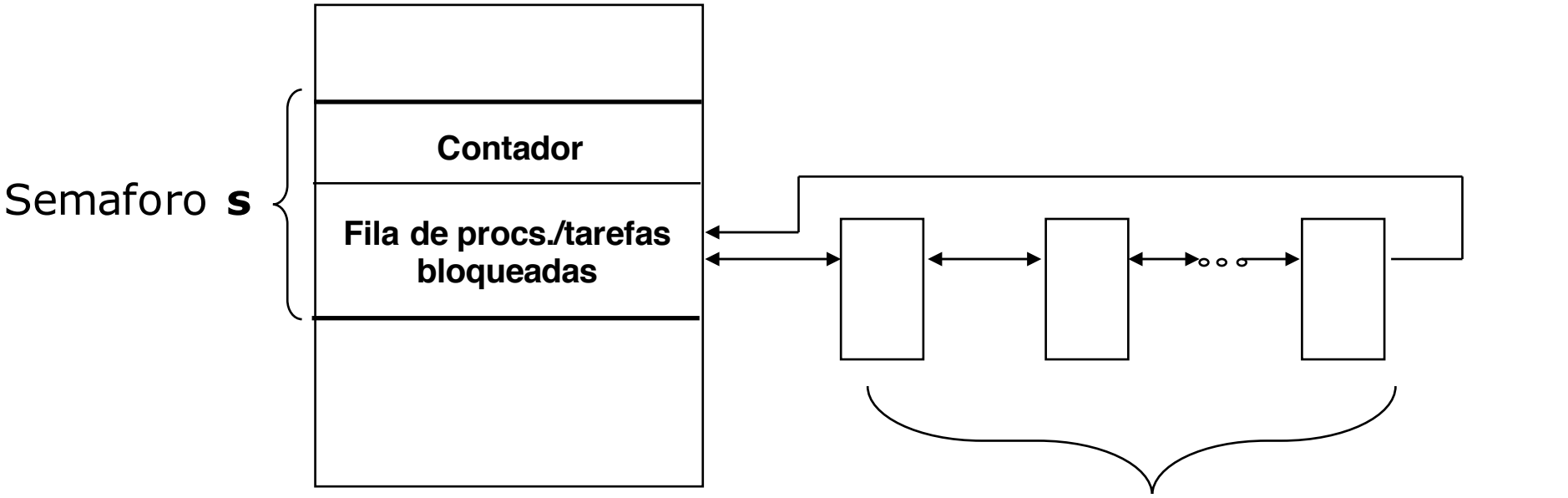
Como resolver estas situações?

1. Apenas posso ter N tarefas em simultâneo numa dada secção de código
2. Tarefa i esperar até que tarefa j complete determinada atividade

Trincos – Limitações

- Trincos não são suficientemente expressivos para resolver alguns problemas de sincronização

Semáforos



- Nunca fazer analogia com semáforo de trânsito !!!

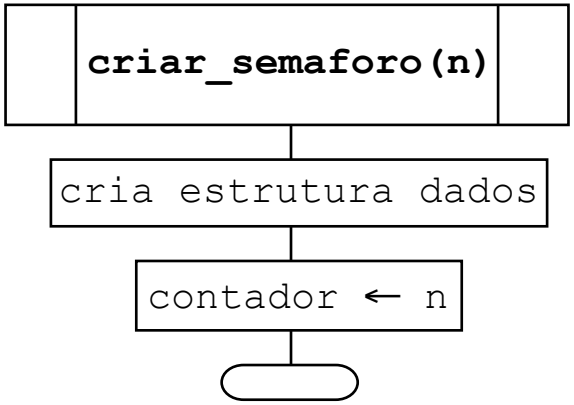
Semáforos: Primitivas

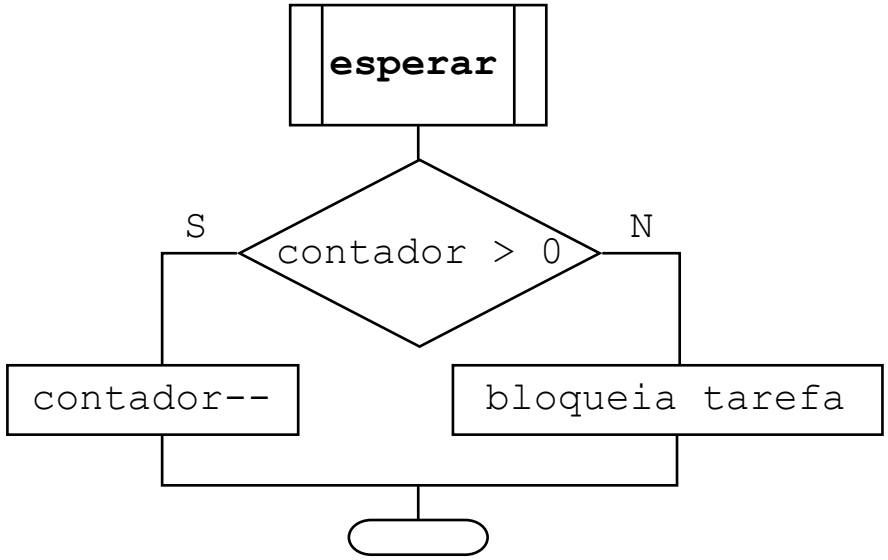
- `s = criar_semaforo(num_unidades)`
  - cria um semáforo e inicializa o contador
- `esperar(s)`
  - bloqueia a tarefa se o contador for menor ou igual a zero; senão decrementa o contador
- `assinalar(s)`
  - se houver tarefas bloqueadas, liberta um; senão, incrementa o contador

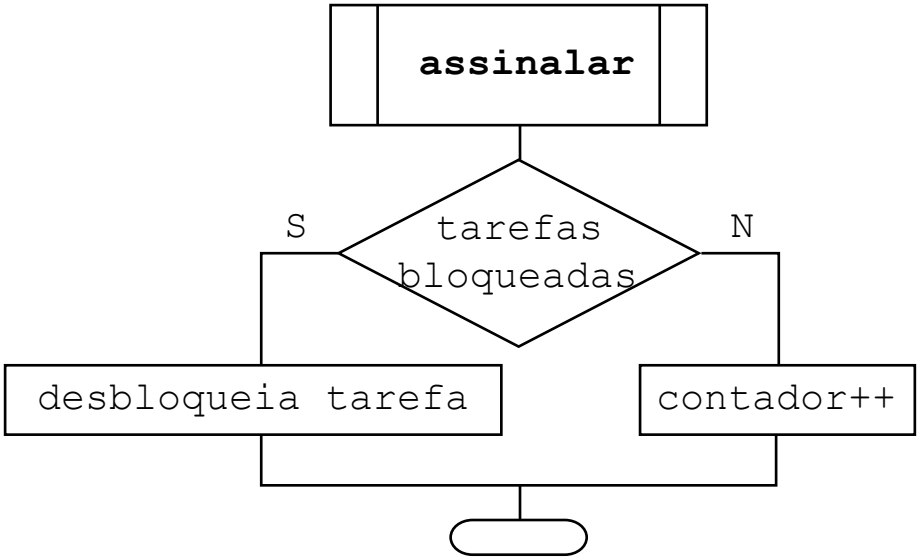
- Todas as primitivas se executam atomicamente
- `esperar()` e `assinalar()` podem ser chamadas por tarefas diferentes

Semáforos: Primitivas

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







Semáforos em Unix

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

int sem_post(sem_t *sem);

int sem_wait(sem_t *sem);
```

Semáforos como forma de limitar acesso a recursos de quantidade limitada

Exemplo: Parque de estacionamento

```
sem_t lugaresLivres;
sem_init(&lugaresLivres, 0, NUM_LUGARES);

entrarNoParque() {
   sem_wait(&lugaresLivres);
   //...
}

sairDoParque() {
   //...
   sem_post(&lugaresLivres);
}
```

Exemplo: Alocador de Memória

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

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

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

Um trinco é criado sempre no estado ABERTO

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

No fim da secção crítica, as tarefas têm que chamar **abrir_mutex**
Passa o trinco para o estado ABERTO ou desbloqueia uma tarefa que esteja à sua espera de entrar na secção crítica

E se a pilha estiver completa?

Exemplo: Alocador de Memória (II)

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

main(){
/*...*/
  ExMut = CriarTrinco();
  semMem = CriarSemaforo(MAX_PILHA);
}
```

O semáforo é inicializado com o valor dos recursos disponíveis

```
char* PedeMem() {
    Esperar(SemMem);
    Fechar(ExMut);
    ptr = pilha[topo];
    topo--;
    Abrir(ExMut);
    return ptr;
}

void DevolveMem(char* ptr) {
    Fechar(ExMut);
    topo++;
    pilha[topo]= ptr;
    Abrir(ExMut);
    Assinalar(SemMem);
}
```

Semáforos: Variantes

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

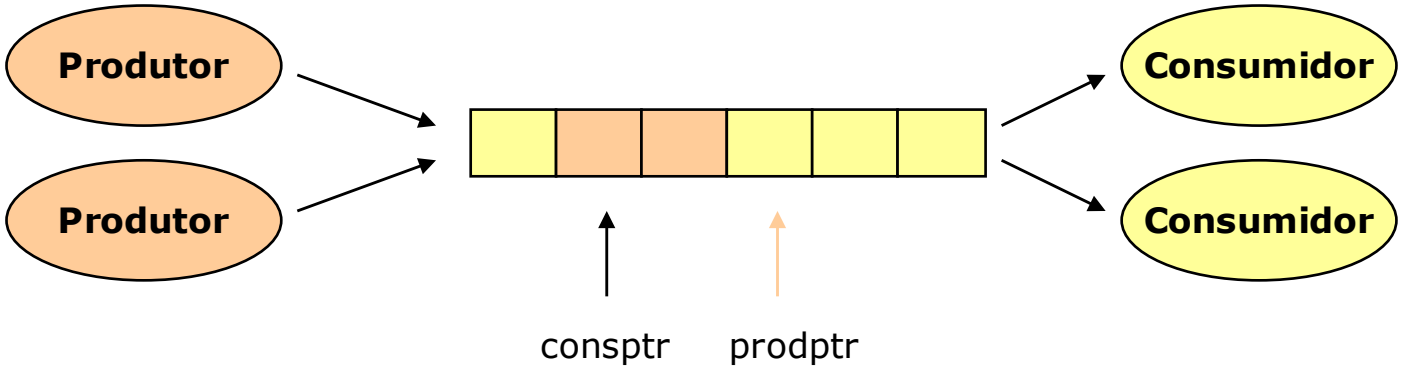
Semáforos como forma de coordenação entre tarefas

Exemplo: tarefa *i* espera que tarefa *j* complete uma dada atividade

```
Semaforo_t sem = CriaSemaforo(0);

Tarefa i:
/* quer esperar pela atividade que a tarefa j
está a executar*/
Esperar(sem);
/* se chegou aqui, tarefa j já completou o que
estava a fazer */

Tarefa j:
executaAtividade(...);
Assinala(sem);
/* continua a sua execução */
```

Problema dos produtores-consumidores

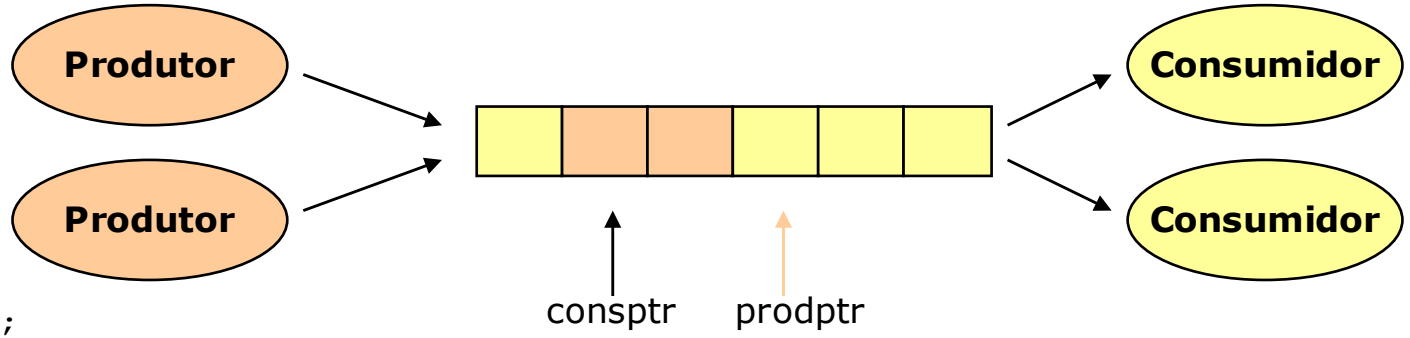
Exemplo de Cooperação entre tarefas: Produtor - Consumidor




```
/* ProdutorConsumidor */
int buf[N];
int prodptr=0, consptr=0;

produtor()
  while(TRUE) {
    int item = produz();
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
  }
}
```

```
consumidor(){
  while(TRUE) {
    int item;
    item = buf[consptr];
    consptr = (consptr+1) % N;
    consome(item);
  }
}
```

Exemplo de Cooperação entre tarefas: Produtor - Consumidor




```
int buf[N];
int prodptr=0, consptr=0;
trinco_t trinco;
```

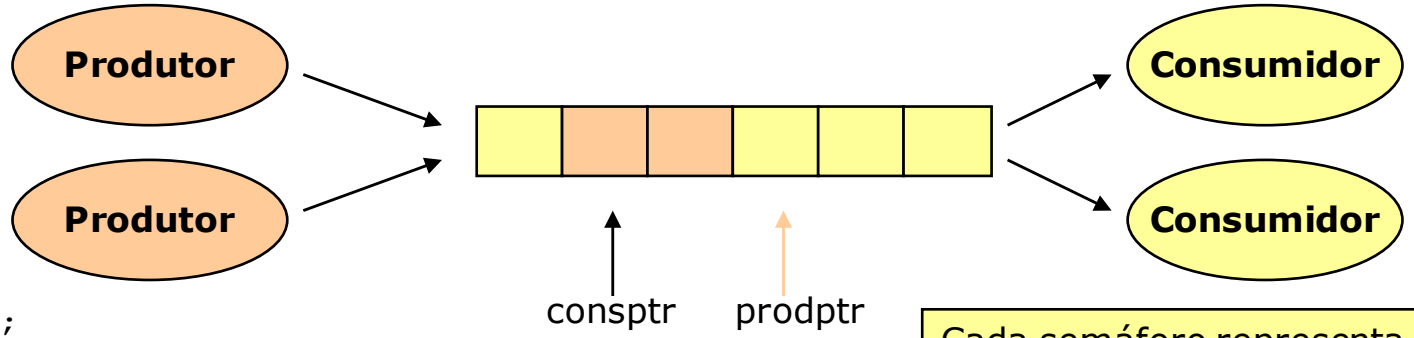
```
produtor() {
  while(TRUE) {
    int item = produz();
    fechar(trinco);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    abrir(trinco);

  }
}
```

Que acontece se o buffer estiver cheio ?

```
consumidor(){
  while(TRUE) {
    int item;
    fechar(trinco);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    abrir(trinco);
   consome(item);
  }
}
```

Que acontece se não houver itens no buffer ?

Exemplo de Cooperação entre tarefas: Produtor - Consumidor



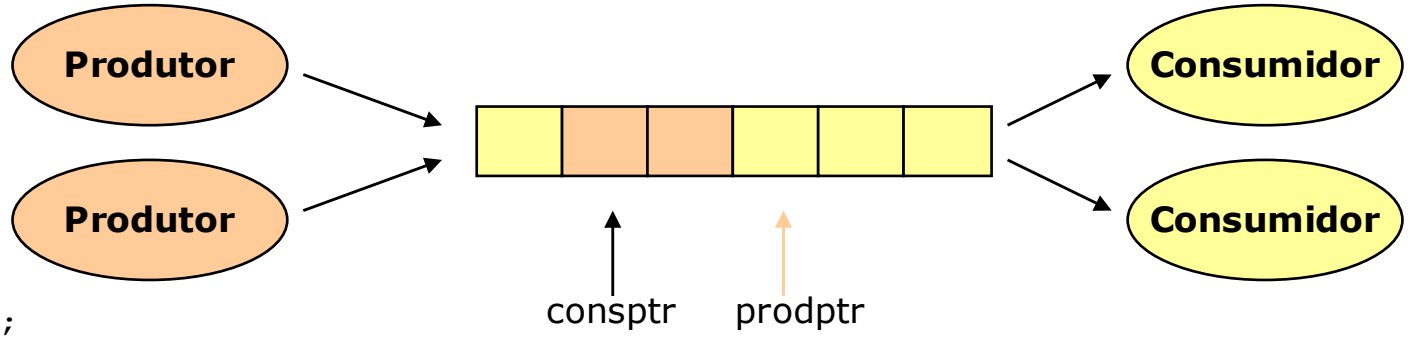

```
int buf[N];
int prodptr=0, consptr=0;
trinco_t trinco;
semaforo_t pode_prod = criar_semaforo(N),
           pode_cons = criar_semaforo(0);
```

Cada semáforo representa um recurso:
pode_produzir: espaços livres, inicia a N
pode_consumir: itens no buffer, inicia a 0

```
produtor() {
  while(TRUE) {
    int item = produz();
    esperar(pode_prod);
    fechar(trinco);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    abrir(trinco);
    assinalar(pode_cons);
  }
}
```

```
consumidor(){
  while(TRUE) {
    int item;
    esperar(pode_cons);
    fechar(trinco);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    abrir(trinco);
    assinalar(pode_prod);
    consome(item);
  }
}
```

Exemplo de Cooperação entre tarefas: Produtor - Consumidor

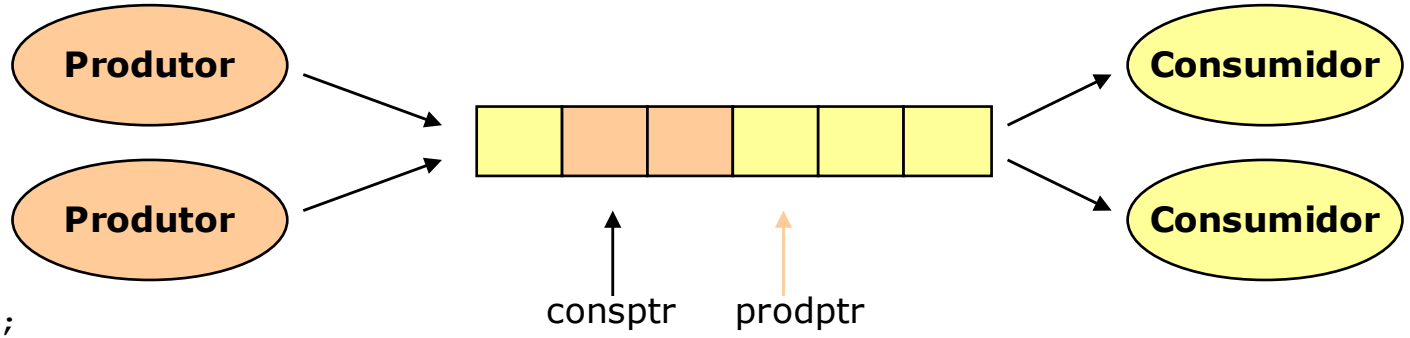



```
int buf[N];
int prodptr=0, consptr=0;
trinco_t trinco;
semaforo_t pode_prod = criar_semaforo(N),
           pode_cons = criar_semaforo(0);

produtor() {
  while(TRUE) {
    int item = produz();
    esperar(pode_prod);
    fechar(trinco);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    abrir(trinco);
    assinalar(pode_cons);
  }
}
```

```
consumidor(){
  while(TRUE) {
    int item;
    esperar(pode_cons);
    fechar(trinco);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    abrir(trinco);
    assinalar(pode_prod);
    consome(item);
  }
}
```

Problema?

Exemplo de Cooperação entre tarefas: Produtor - Consumidor

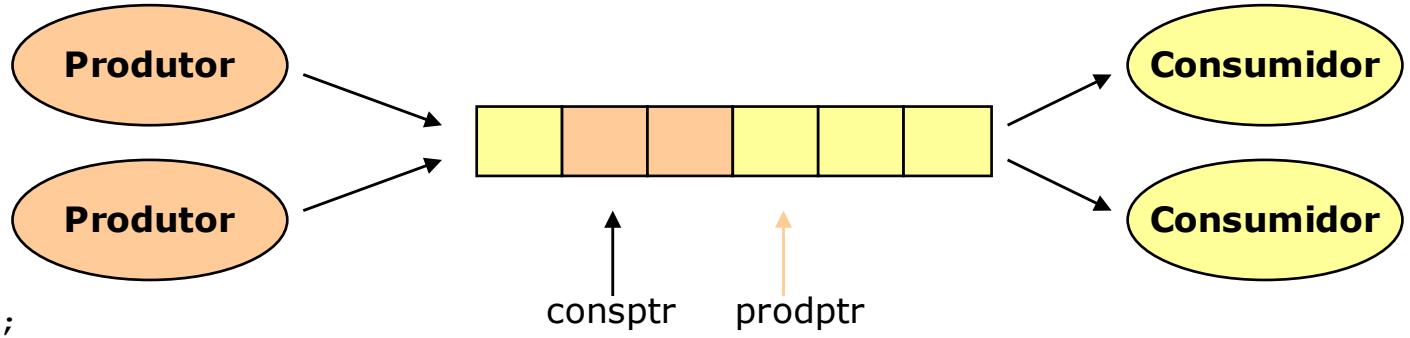



```
int buf[N];
int prodptr=0, consptr=0;
trinco_t trinco;
semaforo_t pode_prod = criar_semaforo(N),
           pode_cons = criar_semaforo(0);

produtor() {
  while(TRUE) {
    int item = produz();
    esperar(pode_prod);
    fechar(trinco);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    abrir(trinco);
    assinalar(pode_cons);
  }
}
```

```
consumidor(){
  while(TRUE) {
    int item;
    esperar(pode_cons);
    fechar(trinco);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    abrir(trinco);
    assinalar(pode_prod);
    consome(item);
  }
}
```

Problema?

Exemplo de Cooperação entre tarefas: Produtor - Consumidor

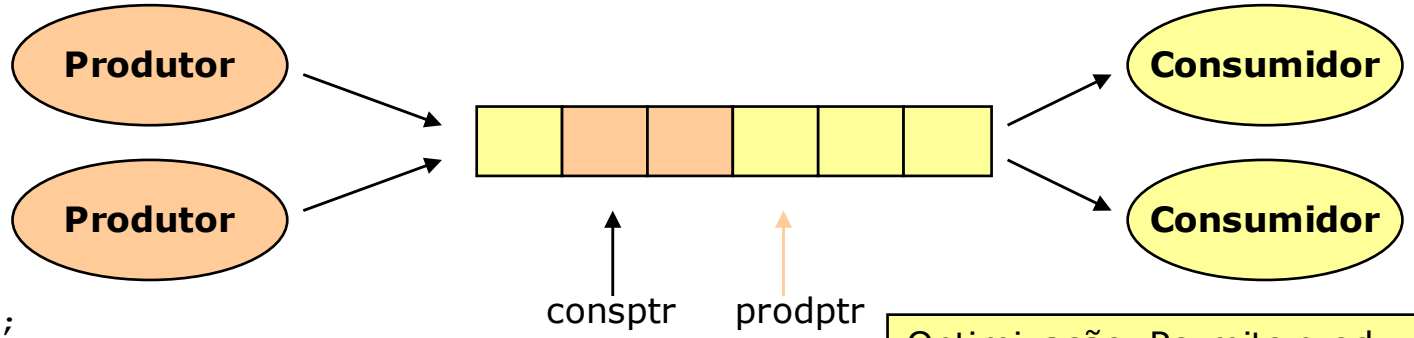



```
int buf[N];
int prodptr=0, consptr=0;
trinco_t trinco;
semaforo_t pode_prod = criar_semaforo(N),
           pode_cons = criar_semaforo(0);

produtor() {
  while(TRUE) {
    int item = produz();
    esperar(pode_prod);
    fechar(trinco);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    abrir(trinco);
    assinalar(pode_cons);
  }
}
```

```
consumidor(){
  while(TRUE) {
    int item;
    esperar(pode_cons);
    fechar(trinco);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    abrir(trinco);
    assinalar(pode_prod);
    consome(item);
  }
}
```

E podemos ter solução que permita produzir e consumir ao mesmo tempo em partes diferentes do buffer ?

Exemplo de Cooperação entre tarefas: Produtor - Consumidor




```
int buf[N];
int prodptr=0, consptr=0;
trinco_t trinco_p, trinco_c;
semaforo_t pode_prod = criar_semaforo(N),
           pode_cons = criar_semaforo(0);
```

Optimização: Permite produzir e consumir ao mesmo tempo em partes diferentes do buffer

```
produtor() {
  while(TRUE) {
    int item = produz();
    esperar(pode_prod);
    fechar(trinco_p);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    abrir(trinco_p);
    assinalar(pode_cons);
  }
}
```

```
consumidor(){
  while(TRUE) {
    int item;
    esperar(pode_cons);
    fechar(trinco_c);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    abrir(trinco_c);
    assinalar(pode_prod);
    consome(item);
  }
}
```

Programar com objetos partilhados

Programar com objetos partilhados

- Até agora, aprendemos esta receita para programar concorrentemente com memória partilhada:
  - Identificar secções críticas
  - Sincronizar cada secção crítica com trinco (*mutex*)

  e
  - Caso haja outras situações que impliquem coordenação entre tarefas, usar os semáforos apropriados

Hoje vamos discutir esta parte em maior detalhe

Aspetos avançados para discutir hoje

- Um trinco global ou múltiplos trincos finos?
- Preciso mesmo usar trinco?

Como sincronizar esta função?

```
struct {
   int saldo;
   …
} conta_t;

int levantar_dinheiro (conta_t* conta, int valor) {

   mutex_lock(____________________);

   if (conta->saldo >= valor)
      conta->saldo = conta->saldo - valor;
   else
      valor = -1; /* -1 indica erro ocorrido */

   mutex_unlock(____________________);
   return valor;
}
```

Trinco global

- Normalmente é a solução mais simples
- Mas limita o paralelismo
  - Quanto mais paralelo for o programa, maior é a limitação
- Exemplo: “big kernel lock” do Linux
  - Criado nas primeiras versões do Linux (versão 2.0)
  - Grande barreira de escalabilidade
  - Finalmente removido na versão 2.6

Trincos finos: programação com objetos partilhados

- Objeto cujos métodos podem ser chamados em concorrência por diferentes tarefas
- Devem ter:
  - Interface dos métodos públicos
  - Código de cada método
  - Variáveis de estado
  - **Variáveis de sincronização**
    - **Um trinco para garantir que métodos críticos se executam em exclusão mútua**
    - Opcionalmente: semáforos, variáveis de condição
      - as variáveis de condição serão introduzidas na próxima aula

Programar com trincos finos

- Em geral, maior paralelismo
- **Mas pode trazer problemas…**

Acesso a múltiplos objetos partilhados

```
transferir(conta a, conta b, int montante) {
  debitar(a, montante);
  creditar(b, montante);
}
```

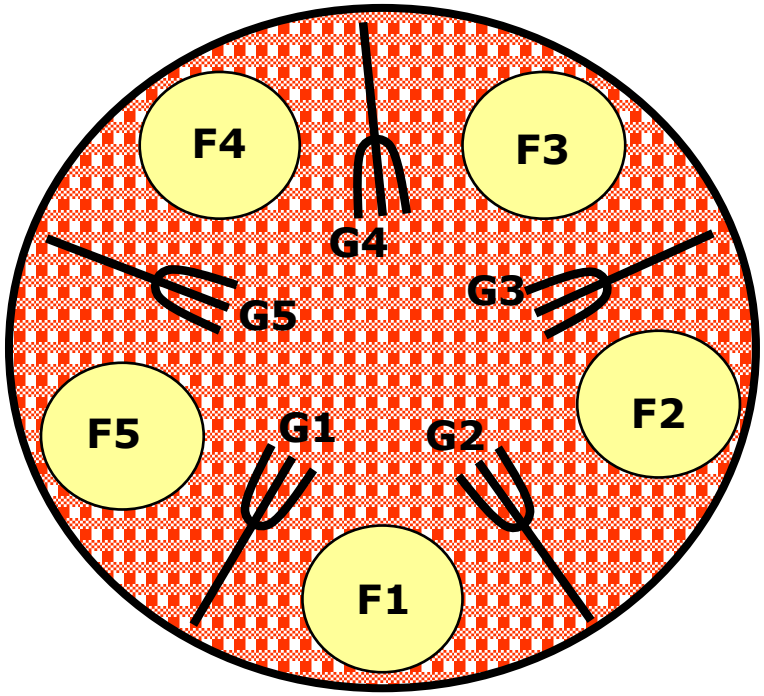
- Um trinco global
  - Fechar no início da função
  - Abrir no retorno
- Um trinco por cada conta
  - Fechar trinco de a, fechar trinco de b
  - Abrir ambos no retorno

Jantar dos Filósofos

- Cinco Filósofos estão reunidos para filosofar e jantar spaghetti:
  - Para comer precisam de dois garfos, mas a mesa apenas tem um garfo por pessoa.

- Condições:
  - Os filósofos podem estar em um de três estados : Pensar; Decidir comer ; Comer.
  - O lugar de cada filósofo é fixo.
  - Um filósofo apenas pode utilizar os garfos imediatamente à sua esquerda e direita.

Jantar dos Filósofos

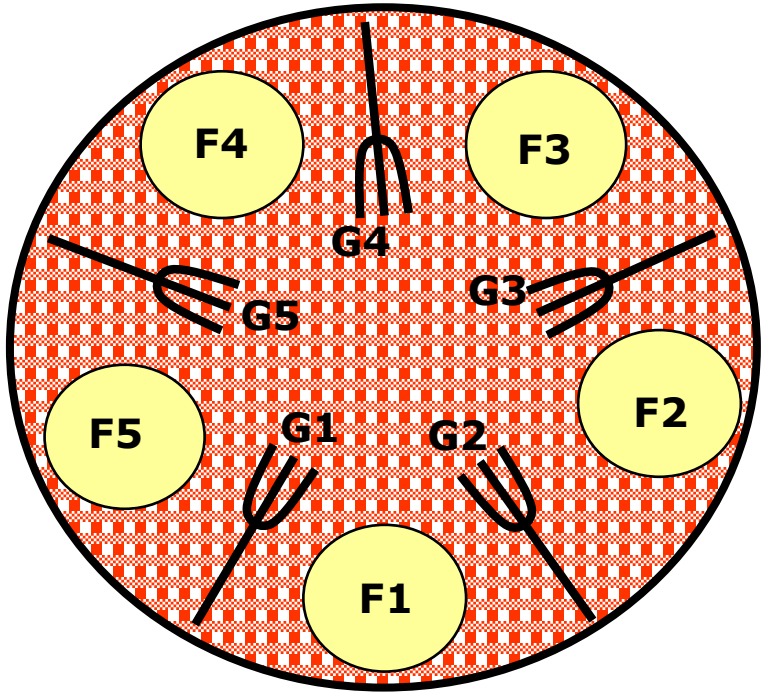


Jantar dos Filósofos

```
filosofo(int id){
    while (TRUE) {
        pensar();
        <adquirir os garfos>
        comer();
        <libertar os garfos>
    }
}
```

Jantar dos Filósofos com Semáforos, versão #1

```
mutex_t garfo[5] = {…};

filosofo(int id)
{
    while (TRUE) {
        pensar();
        fechar(garfo[id]);
        fechar (garfo[(id+1)%5]);
        comer();
        abrir(garfo[id]);
        abrir(garfo[(id+1)%5]);
    }
}
```



Problema?

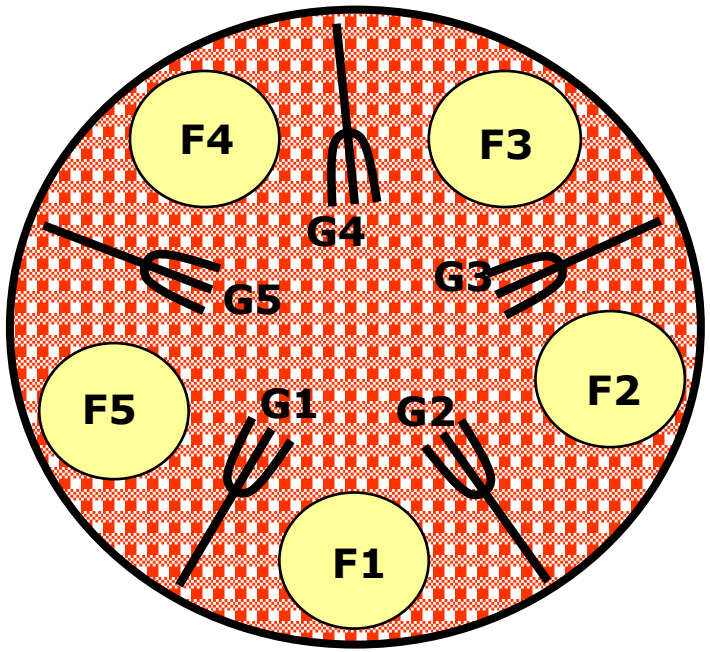
Interblocagem

- Uma situação de interblocagem pode aparecer se as quatro condições seguintes forem simultaneamente verdadeiras:
  - Pelo menos um recurso é usado de uma forma não partilhável;
  - Existe pelo menos uma tarefa que detém um recurso e que está à espera de adquirir mais recursos;
  - Os recursos não são “preemptíveis”, ou seja, os recursos apenas são libertados voluntariamente pelas tarefas que os detêm;
  - Existe um padrão de sincronização em que a tarefa T_1 espera por um recurso de T_2 e circularmente T_{n-1} espera por um recurso de T_1

Como prevenir interblocagem?

Jantar dos Filósofos com Semáforos, versão #2

```
mutex_t garfo[5] = {…};

filosofo(int id)
{
   while (TRUE) {
      pensar();
         if (id < 4) {
            fechar(garfo[id]);
            fechar (garfo[(id+1)%5]);
         else {
            fechar (garfo[(id+1)%5]);
            fechar(garfo[id]);
         }
         comer();
         abrir(garfo[id]);
         abrir(garfo[(id+1)%5]);
    }
}
```



Princípio base: garantir que os recursos são todos adquiridos pela mesma ordem

Jantar dos Filósofos com Semáforos, versão #3

```
#define PENSAR 0
#define FOME 1
#define COMER 2
#define  N 5
int estado[N] = {0, 0, 0, 0, 0};
semaforo_t semfilo[N] = {0, 0, 0, 0, 0};
trinco mutex;

Testa(int k){
  if (estado[k] == FOME &&
      estado[(k+1)%N] != COMER &&
      estado[(k-1+N)%N] != COMER){
       estado[k] = COMER;
       assinalar(semfilo[K]);
  }
}
```

```
filosofo(int id)
{
    while (TRUE) {
        pensar();
        fechar(mutex);
        estado[id] = FOME;
        Testa(id);
        abrir(mutex);
        esperar(semfilo[id]);
        comer();
        fechar(mutex);
        estado[id] = PENSAR;
        Testa((id-1+N)%N);
        Testa((id+1)%N);
        abrir(mutex);
    }
}
```

Princípio base: Requisitar todos os recursos que a tarefa necessita no início da sua execução, de forma atómica

Jantar dos Filósofos com Semáforos, versão #4

```
mutex_t garfo[5] = {…};
int garfos;

filosofo(int id)
{
   while (TRUE) {
      pensar();
      garfos = FALSE;
      while (!garfos){
         if (try_lock(garfo[id]))
            if (try_lock(garfo[(id+1)%5])
               garfos = TRUE;
            else  // adquisição 2° trinco falhou
               unlock(garfo[id]); // abre 1°trinco e tenta outra vez
      }
      comer();
      unlock(garfo[id]);
      unlock(garfo[(id+1)%5]);
   }
}
```

Resolvemos o problema da interblocagem!
...e o da míngua?

Evitar míngua: recuo aleatório!

- Pretende-se evitar que dois filósofos vizinhos possam conflituar indefinidamente

Introduzir uma fase de espera/recuo (*back-off*) entre uma tentativa e outra de cada filósofo.

- Como escolher a duração da fase de espera?
  - Inúmeras politicas propostas na literatura
  - Vamos ilustrar apenas os princípios fundamentais das políticas mais genéricas e simples

Evitar míngua: escolha da duração do recuo

- Duração aleatória entre [0, MAX]
  - ➔ por quê aleatória?
- Como escolher o valor MAX?
  - Quanto maior o valor de MAX:
    - ➔ **menor desempenho**
    - ➔ **maior probabilidade de evitar contenção**
  - ➔ Adaptar o valor de MAX consoante o número de tentativas:
    - ➔ *Linear back-off:* MAX = constante * num_tentativas
    - ➔ *Exponential back-off:* MAX = constante * 2^num_tentativas

Políticas de recuo: variantes

- Ter em conta a prioridade da tarefa na definição do período máximo de recuo
  - Tarefas mais prioritárias esperam menos
- Adaptar a duração máxima ao número máximo de tarefas concorrentemente ativas
- Adaptar o período de espera à duração prevista das tarefas com as quais se entra em contenção:
  - as tarefas devem anunciar o progresso atingido ao executar na secção critica

Recapitulando: como prevenir interblocagem?

- Garantindo que os recursos são todos adquiridos pela mesma ordem;
- Requisitando todos os recursos que a tarefa necessita no início da sua execução, de forma atómica
- Quando a aquisição de um recurso não é possível, libertando todos os recursos detidos e anulando as operações realizadas até esse momento

Vantagens/desvantagens de cada abordagem?

Preciso mesmo usar trinco?

Preciso mesmo usar trinco? Exemplo 1

```
struct {
    int saldo;
    int cliente;
    ...
} conta_t;
conta_t contas[N] = ...; //Mantém as contas todas do banco

int transferir_dinheiro(conta_t *a, conta_t *b, valor) {...}

int saldoTotalDeCliente(int cliente) {
  int total = 0;
  for (int i=0; i<N; i++) {
    if (contas[i].cliente == cliente) {
      total += contas[i].saldo;
    }
  }
  return total;
}
```

Se esta função só lê, é mesmo preciso trinco?

Não usar trinco: arriscamos resultado inconsistente!
Usar trinco: proíbe que duas tarefas executem a função em paralelo

Trincos de leitura-escrita

- Podem ser fechados de duas formas:
  - *fechar para ler* ou *fechar para escrever*
- Semântica:
  - Os **escritores** só podem aceder em **exclusão mútua**
  - Os **leitores** podem aceder **simultaneamente com outros leitores** mas em **exclusão mútua com os escritores**
- Mais pesado que trinco lógico, mas mais paralelo. Então, quando usar?
  - Vantajoso quando acessos a secções críticas de leitura são dominantes

Trincos de leitura-escrita em POSIX

- Tipo de dados: `pthread_rwlock_t`
- Fechar para ler:

```
int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
```

- Fechar para escrever:

```
int pthread_rwlock_wrlock(pthread_rwlock_t *lock);
```

- Abrir:

```
int pthread_rwlock_unlock(pthread_rwlock_t *lock);
```

- Mais variantes disponíveis
  - trylock, timedlock, etc. Ver *man pages.*

Exemplo 1: Usando trincos leitores-escritores

```
struct {
    int saldo;
    int cliente;
    pthread_rwlock_t rw_lock;
    ...
} conta_t;
conta_t contas[N] = ...; //Mantém as contas todas do banco

int transferir_dinheiro(conta_t *a, conta_t *b, valor) {...}
```

Usar pthread_rwlock_wrlock (e pthread_rwlock_unlock)

```
int saldoTotalDeCliente(int cliente) {...}
```

Usar pthread_rwlock_rdlock (e pthread_rwlock_unlock)

Preciso mesmo usar trinco? Exemplo 2

```
struct {
    int saldo;
    int cliente;
    char nomeCliente[N];
    ...
} conta_t;
conta_t contas[N] = ...; //Mantém as contas todas do banco

int transferir_dinheiro(conta_t *a, conta_t *b, valor) {...}

int lerSaldo (conta_t *c) {
    return c->saldo;
}
```

Como só estamos a ler um campo, é mesmo preciso sincronização?

“Como só estamos a ler um campo, é mesmo preciso sincronização?”

- Muita atenção ao que se passa no baixo nível:
  - Acesso a um valor em memória nem sempre é feitos de uma só vez
  - Optimizações do processador/compilador podem executar algumas instruções fora de ordem
- Receita para evitar bugs:
  - Sincronizar acesso de leitura com trinco (simples ou leitura/escrita) protege-nos das situações acima

Variáveis de Condição

Alternativa aos semáforos

Variável de Condição

- Permite a uma tarefa esperar por uma condição que depende da ação de outra tarefa
  - Condição é boleano determinado em função do estado de variáveis partilhadas

Variável de Condição

- Variável de condição sempre associada a um trinco
  - O trinco que protege as secções críticas com acessos às variáveis partilhadas que definem a condição da espera
  - Pode haver mais que uma variável de condição associada ao mesmo trinco
- O conjunto trinco + variáveis de condição é normalmente chamado um *monitor*

Variável de Condição: primitivas (semântica Mesa)

- *wait(conditionVar, mutex)*
  - **Atomicamente**, liberta o trinco associado e bloqueia a tarefa
    - Tarefa é colocada na fila de espera associada à variável de condição
  - Quando for desbloqueada, a tarefa re-adquire o trinco e só depois é que a função *esperar* retorna

Uma tarefa só pode chamar wait quando detenha o trinco associado à variável de condição

Variável de Condição: primitivas (semântica Mesa)

- *signal(conditionVar)*
  - Se houver tarefas na fila da variável de condição, desbloqueia uma
    - Tarefa que estava bloqueada passa a executável
  - Se não houver tarefas na fila da variável de condição, não tem efeito
- *broadcast(conditionVar)*
  - Análogo ao signal mas desbloqueia todas as tarefas na fila da variável de condição

Normalmente estas primitivas são chamadas quando a tarefa ainda não libertou o trinco associado à variável de condição

Padrões habituais de programação com variável de condição

Código que espera por condição

```
lock(trinco);
/* ..acesso a variáveis partilhadas.. */
while (! condiçãoSobreEstadoPartilhado())
        wait(varCondicao, trinco);
/* ..acesso a variáveis partilhadas.. */
unlock(trinco);
```

Código que muda ativa condição

```
lock(trinco);
/* ..acesso a variáveis partilhadas.. */

/* se o estado foi modificado de uma forma
que pode permitir progresso a outras tarefas,
chama signal (ou broadcast) */
signal/broadcast(varCondicao);

unlock(trinco);
```

Variáveis de Condição - POSIX

- Criação/destruição de variáveis de condição;
  - pthread_cond_init (condition,attr)
  - pthread_cond_destroy (condition)
  - pthread_condattr_init (attr)
  - pthread_condattr_destroy (attr)

- Assinalar e esperar nas variáveis de condição:
  - pthread_cond_wait (condition,mutex)
  - pthread_cond_signal (condition)
  - pthread_cond_broadcast (condition)

Exercício: canal de comunicação (a.k.a. problema do produtor-consumidor)

```
int buffer[N];
Int prodptr=0, consptr=0, count=0;

enviar(int item) {
    if (count == MAX)
          return -1;
    buffer[prodptr] = item;
     prodptr = (prodptr+1) % N;
     count++;
    return 1;
}

int receber(){
    int item;
    if (count == 0)
          return -1;

    item = buffer[consptr];
    consptr = (consptr+1) % N;
    count--;
    return item;
}
```

Problemas caso enviar/receber sejam chamadas concorrentemente?

Como estender para suportar envio/recepção síncrona?

Variável de Condição: discussão (I)

- Tarefa que chama wait liberta o trinco e entra na fila de espera **atomicamente**
  - Consequência: caso a condição mude e haja *signal*, pelo menos uma tarefa na fila será desbloqueada

O que aconteceria se não houvesse a garantia?

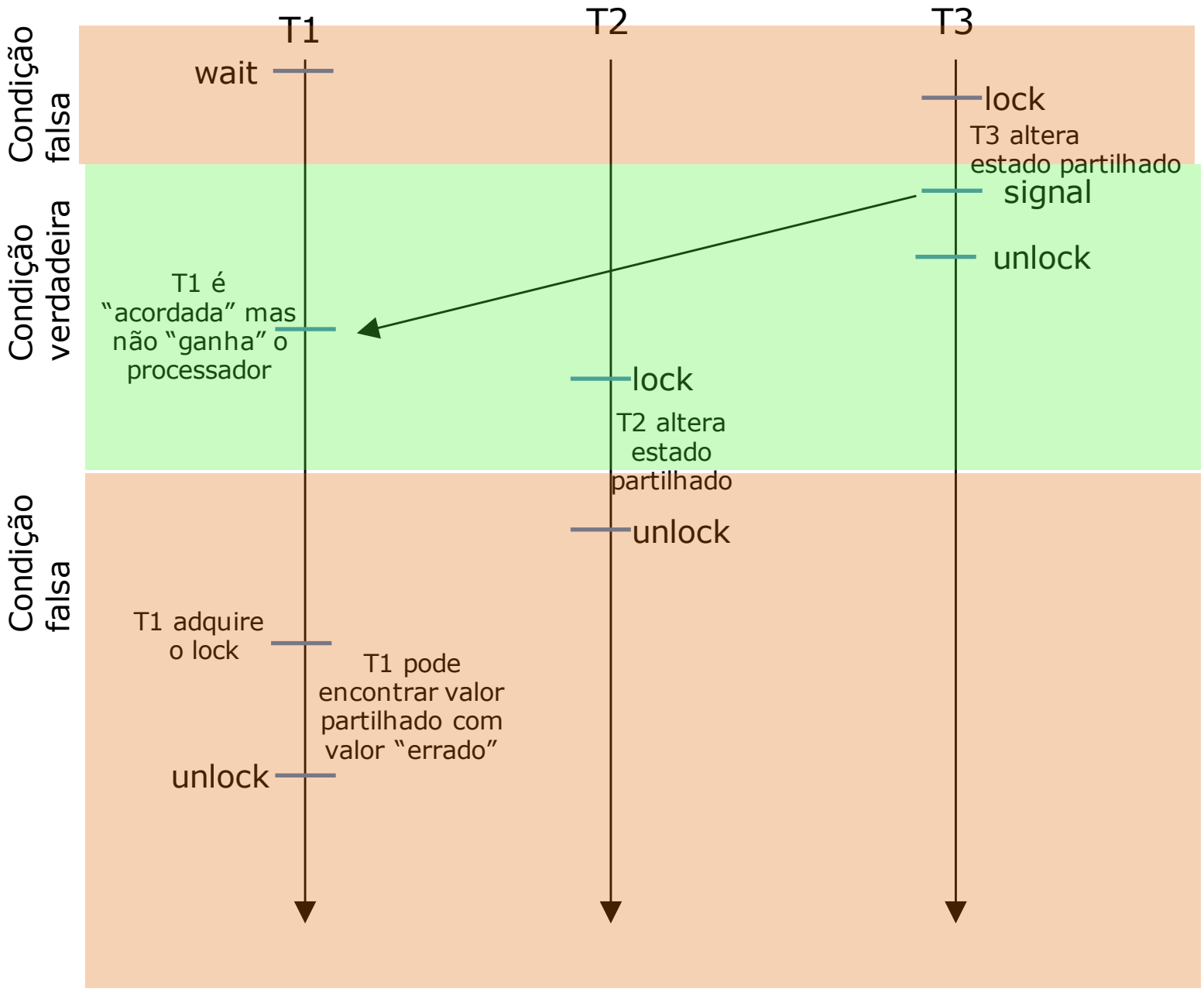
Variável de Condição: discussão(II)

- Tarefa em espera que seja desbloqueada por signal/broadcast não corre imediatamente
  - Simplesmente é tornada executável
  - Para que *wait* retorne, tem de re-adquirir o trinco

Consequência?

Variável de Condição: discussão(II)

- durante o tempo que medeia entre o signal (feito por T3) e uma tarefa ser "acordada" (T1) adquirindo o trinco
- a variável de condição pode ser alterada por outra tarefa (T2) !!!



Variável de Condição: discussão(III)

- Retorno do wait não garante que condição que lhe deu origem se verifique
  - Tarefa pode não ter sido a primeira tarefa a entrar na secção crítica depois da tarefa que assinalou a ter libertado
- Logo, após retorno do wait, re-verificar a condição:
  - Não fazer: if (testa variável partilhada) wait
  - Fazer: while (testa variável partilhada) wait

Variável de Condição: discussão(IV)

- Algumas implementações de variáveis de condição permitem que tarefa retorne do *wait* sem ter ocorrido *signal/broadcast*
  - "Spurious wakeups"
- Mais uma razão para testar condição com *while* em vez de *if*

Problema dos Leitores - Escritores

- Pretende-se gerir o acesso a uma estrutura de dados partilhada em que existem duas classes de tarefas:
  - Leitores – apenas lêem a estrutura de dados
  - Escritores – lêem e modificam a estrutura de dados

- Condições
  - Os escritores só podem aceder em exclusão mútua
  - Os leitores podem aceder simultaneamente com outros leitores mas em exclusão mútua com os escritores
  - Nenhuma das classes de tarefas deve ficar à mingua

Problema dos Leitores - Escritores

```
leitor() {
  while (TRUE) {
    inicia_leitura();
    leitura();
    acaba_leitura();
  }
}

escritor() {
  while (TRUE) {
    inicia_escrita();
    escrita();
    acaba_escrita();
  }
}
```

Problema dos Leitores – Escritores: hipótese 1

```
leitor() {
  while (TRUE) {
    fechar(mutex);
    leitura();
    abrir(mutex);
  }
}

escritor() {
  while (TRUE) {
    fechar(mutex);
    escrita();
    abrir(mutex);
  }
}
```

Demasiado forte!
É possível permitir mais paralelismo!

Leitores – Escritores: Dificuldades

- Condições de bloqueio mais complexas:
  - escritor bloqueia se houver um leitor ou um escritor em simultâneo
- Com quem deve ser feita a sincronização?
  - 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?
- Solução:
  - ler variáveis antes de efectuar esperar/assinalar

Leitores-Escritores: esboço da solução

```
int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;
```

```
inicia_leitura()
{

  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;
```

Bloquear até não haver ninguém a escrever

```
    leitores_espera--;


  }
  nleitores++;

}
```

```
acaba_leitura()
{

  nleitores--;
```

Desbloquear quem esteja à espera para escrever

```
}
```

```
inicia_escrita()
{

  if (em_escrita || nleitores > 0) {
    escritores_espera++;
```

Bloquear até não haver ninguém a escrever ou a ler

```
    escritores_espera--;
  }
  em_escrita = TRUE;

}
```

```
acaba_escrita()
{

  em_escrita = FALSE;
```

Desbloquear quem esteja à espera para ler ou para escrever

```
}
```

Leitores-Escritores: esboço da solução

```
int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;
```

```
inicia_leitura()
{
  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;
    esperar(leitores);
    leitores_espera--;
  }
  nleitores++;
}
```

Bloquear até não haver ninguém a escrever

```
acaba_leitura()
{
  nleitores--;
  if (nleitores == 0 && escritores_espera > 0)
    assinalar(escritores);
}
```

Desbloquear quem esteja à espera para escrever

```
semaforo_t leitores=0, escritores=0;
```

```
inicia_escrita()
{
  if (em_escrita || nleitores > 0) {
    escritores_espera++;
    esperar(escritores);
    escritores_espera--;
  }
  em_escrita = TRUE;
}
```

Bloquear até não haver ninguém a escrever ou a ler

```
acaba_escrita()
{
  em_escrita = FALSE;
  if (leitores_espera > 0)
    for (i=0; i<leitores_espera; i++)
      assinalar(leitores);
  else if (escritores_espera > 0)
    assinalar(escritores);
}
```

Desbloquear quem esteja à espera para ler ou para escrever

Não existem secções críticas???

Leitores-Escritores: esboço da solução

```
int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;
```

```
inicia_leitura()
{
  fechar(m);
  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;
    esperar(leitores);
    leitores_espera--;
  }
  nleitores++;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever

```
acaba_leitura()
{
  fechar(m);
  nleitores--;
  if (nleitores == 0 && escritores_espera > 0)
    assinalar(escritores);
  abrir(m);
}
```

Desbloquear quem esteja à espera para escrever

```
semaforo_t leitores=0, escritores=0;
trinco_t m;
```

```
inicia_escrita()
{
  fechar(m);
  if (em_escrita || nleitores > 0) {
    escritores_espera++;
    esperar(escritores);
    escritores_espera--;
  }
  em_escrita = TRUE;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever ou a ler

```
acaba_escrita()
{
  fechar(m);
  em_escrita = FALSE;
  if (leitores_espera > 0)
    for (i=0; i<leitores_espera; i++)
      assinalar(leitores);
  else if (escritores_espera > 0)
    assinalar(escritores);
  abrir(m);
}
```

Desbloquear quem esteja à espera para ler ou para escrever

Leitores-Escritores: esboço da solução

```
int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;
```

```
inicia_leitura()
{
  fechar(m);
  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;
    abrir(m);
    esperar(leitores);
    fechar(m);
    leitores_espera--;
  }
  nleitores++;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever

```
acaba_leitura()
{
  fechar(m);
  nleitores--;
  if (nleitores == 0 && escritores_espera > 0)
    assinalar(escritores);
  abrir(m);
}
```

Desbloquear quem esteja à espera para escrever

```
semaforo_t leitores=0, escritores=0;
trinco_t m;
```

```
inicia_escrita()
{
  fechar(m);
  if (em_escrita || nleitores > 0) {
    escritores_espera++;
    abrir(m);
    esperar(escritores);
    fechar(m);
    escritores_espera--;
  }
  em_escrita = TRUE;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever ou a ler

```
acaba_escrita()
{
  fechar(m);
  em_escrita = FALSE;
  if (leitores_espera > 0)
    for (i=0; i<leitores_espera; i++)
      assinalar(leitores);
  else if (escritores_espera > 0)
    assinalar(escritores);
  abrir(m);
}
```

Desbloquear quem esteja à espera para ler ou para escrever

Leitores-Escritores: esboço da solução

```
int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;
```

```
inicia_leitura()
{
  fechar(m);
  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;
    abrir(m);
    esperar(leitores);
    fechar(m);
    leitores_espera--;
  }
  nleitores++;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever

```
acaba_leitura()
{
  fechar(m);
  nleitores--;
  if (nleitores == 0 && escritores_espera > 0)
    assinalar(escritores);
  abrir(m);
}
```

Desbloquear quem esteja à espera para escrever

```
semaforo_t leitores=0, escritores=0;
trinco_t m;
```

```
inicia_escrita()
{
  fechar(m);
  if (em_escrita || nleitores > 0) {
    escritores_espera++;
    abrir(m);
    esperar(escritores);
    fechar(m);
    escritores_espera--;
  }
  em_escrita = TRUE;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever ou a ler

```
acaba_escrita()
{
  fechar(m);
  em_escrita = FALSE;
  if (leitores_espera > 0)
    for (i=0; i<leitores_espera; i++)
      assinalar(leitores);
  else if (escritores_espera > 0)
    assinalar(escritores);
  abrir(m);
}
```

Desbloquear quem esteja à espera para ler ou para escrever

**Se existe leitor L1 à espera, é assinalado;
pode perder proc. antes de fechar(m);
novo escritor E1 tenta aceder e consegue;
L1 executa-se e acede em leitura ☹**

Problema: e se uma nova tarefa obtém acesso antes das tarefas assinaladas?

Leitores-Escritores: esboço da solução

```
semaforo_t leitores=0, escritores=0;
trinco_t m;

int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;
```

```
inicia_leitura()
{
  fechar(m);
  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;
    abrir(m);
    esperar(leitores);
    fechar(m);
    leitores_espera--;
  }
  else nleitores++;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever

```
acaba_leitura()
{
  fechar(m);
  nleitores--;
  if (nleitores == 0 && escritores_espera > 0){
    assinalar(escritores);
    em_escrita=TRUE;
    escritores_espera--;}

  abrir(m);
}
```

Desbloquear quem esteja à espera para escrever

```
inicia_escrita()
{
  fechar(m);
  if (em_escrita || nleitores > 0) {
    escritores_espera++;
    abrir(m);
    esperar(escritores);
    fechar(m);
    escritores_espera--;
  }
  em_escrita = TRUE;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever ou a ler

```
acaba_escrita()
{
  fechar(m);
  em_escrita = FALSE;
  if (leitores_espera > 0)
    for (i=0; i<leitores_espera; i++) {
        assinalar(leitores);
        nleitores++; }
    leitores_espera-=i;
    }
else if (escritores_espera > 0) {
    assinalar(escritores);
    em_escrita=TRUE;
    escritores_espera--;
  }
  abrir(m);
}
```

Desbloquear quem esteja à espera para ler ou para escrever

Leitores-Escritores: esboço da solução

```
semaforo_t leitores=0, escritores=0;
trinco_t m;

int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;
```

```
inicia_leitura()
{
  fechar(m);
  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;
    abrir(m);
    esperar(leitores);
    fechar(m);
    leitores_espera--;
  }
  else nleitores++;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever

```
acaba_leitura()
{
  fechar(m);
  nleitores--;
  if (nleitores == 0 && escritores_espera > 0){
    assinalar(escritores);
    em_escrita=TRUE;
    escritores_espera--;}

  abrir(m);
}
```

Desbloquear quem esteja à espera para escrever

```
inicia_escrita()
{
  fechar(m);
  if (em_escrita || nleitores > 0) {
    escritores_espera++;
    abrir(m);
    esperar(escritores);
    fechar(m);
    escritores_espera--;
  }
  em_escrita = TRUE;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever ou a ler

```
acaba_escrita()
{
  fechar(m);
  em_escrita = FALSE;
  if (leitores_espera > 0)
    for (i=0; i<leitores_espera; i++) {
        assinalar(leitores);
        nleitores++; }
    leitores_espera-=i;
    }
else if (escritores_espera > 0) {
    assinalar(escritores);
    em_escrita=TRUE;
    escritores_espera--;
  }
  abrir(m);
}
```

Desbloquear quem esteja à espera para ler ou para escrever

Problema?

Leitores-Escritores: esboço da solução

```
semaforo_t leitores=0, escritores=0;
trinco_t m;

int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;
```

```
inicia_leitura()
{
  fechar(m);
  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;
    abrir(m);
    esperar(leitores);
    fechar(m);
    leitores_espera--;
  }
  else nleitores++;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever

```
acaba_leitura()
{
  fechar(m);
  nleitores--;
  if (nleitores == 0 && escritores_espera > 0){
    assinalar(escritores);
    em_escrita=TRUE;
    escritores_espera--;}
  abrir(m)
}
```

Desbloquear quem esteja à espera para escrever

```
inicia_escrita()
{
  fechar(m);
  if (em_escrita || nleitores > 0) {
    escritores_espera++;
    abrir(m);
    esperar(escritores);
    fechar(m);
    escritores_espera--;
  }
  em_escrita = TRUE;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever ou a ler

```
acaba_escrita()
{
  fechar(m);
  em_escrita = FALSE;
  if (leitores_espera > 0)
    for (i=0; i<leitores_espera; i++) {
        assinalar(leitores);
        nleitores++; }
    leitores_espera-=i;
    }
else if (escritores_espera > 0) {
    assinalar(escritores);
    em_escrita=TRUE;
    escritores_espera--;
  }
  abrir(m);
}
```

Desbloquear quem esteja à espera para ler ou para escrever

Problema?

- No acaba_escrita libertar apenas um leitor
- No inicia_leitura cada leitor que é desbloqueado, desbloqueia o que ainda está bloqueado (ver livro)

Eficiência: liberta leitor mas este fica bloqueado no trinco m (inicia_leitura)

Leitores-Escritores: esboço da solução

```
semaforo_t leitores=0, escritores=0;
trinco_t m;

int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;
```

```
inicia_leitura()
{
  fechar(m);
  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;
    abrir(m);
    esperar(leitores);
    fechar(m);
    leitores_espera--;
  }
  else nleitores++;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever

```
acaba_leitura()
{
  fechar(m);
  nleitores--;
  if (nleitores == 0 && escritores_espera > 0){
    assinalar(escritores);
    em_escrita=TRUE;
    escritores_espera--;}

  abrir(m);
}
```

Desbloquear quem esteja à espera para escrever

```
inicia_escrita()
{
  fechar(m);
  if (em_escrita || nleitores > 0) {
    escritores_espera++;
    abrir(m);
    esperar(escritores);
    fechar(m);
    escritores_espera--;
  }
  em_escrita = TRUE;
  abrir(m);
}
```

Bloquear até não haver ninguém a escrever ou a ler

```
acaba_escrita()
{
  fechar(m);
  em_escrita = FALSE;
  if (leitores_espera > 0)
    for (i=0; i<leitores_espera; i++) {
        assinalar(leitores);
        nleitores++; }
    leitores_espera-=i;
    }
else if (escritores_espera > 0) {
    assinalar(escritores);
    em_escrita=TRUE;
    escritores_espera--;
  }
  abrir(
}
```

Desbloquear quem esteja à espera para ler ou para escrever

E há míngua ?

No acaba_escrita, teste de leitores_espera impede míngua dos leitores

No inicia_leitura, teste de escritores_espera impede míngua dos escritores

Produtor – Consumidor com Variáveis Condição

```
int buf[N], prodptr=0, consptr=0, count=0;

mutex_t mutex;
cond_t podeProd, podeCons;
```

```
produtor() {
  while(TRUE) {
    int item = produz();
    pthread_mutex_lock(&mutex);
    while (count == MAX) pthread_cond_wait(&podeProd,&mutex);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    count++;
    pthread_cond_signal(&podeCons);
    pthread_mutex_unlock(&mutex);
  }
}

consumidor(){
   while(TRUE) {
      int item;
      pthread_mutex_lock(&mutex);
      while (count == 0) pthread_cond_wait(&podeCons,&mutex);
      item = buf[consptr];
      consptr = (consptr+1) % N;
      count--;
      pthread_cond_signal(&podeProd);
      pthread_mutex_unlock(&mutex);
   }
}
```

Leitores–Escritor com Variáveis Condição

```
int leitores=0, escritores=0;
int leitores_espera=0, escritores_espera=0;
trinco_t trinco;
cond_t cond_read, cond_write;
```

```
inicia_escrita() {
    fechar (trinco);
    while (escritores+leitores > 0) {
        escritores_espera++;
        cond_write.wait (trinco);
        escritores_espera--;
    }
    escritores++;
    abrir(trinco);
}
```

```
inicia_leitura(){
   fechar (trinco);
   while (escritores+escritores_espera > 0) {
       leitores_espera++;
       cond_read.wait(trinco);
       leitores_espera--;
   }
   leitores++;
   abrir(trinco);
}
```

```
acaba-escrita(){
    fechar (trinco);
    escritores--;
    if (escritores_espera > 0)
        cond_write.signal(trinco);
    else
        cond_read.broadcast(trinco);
   abrir (trinco);
}
```

```
acaba_leitura(){
    fechar (trinco);
    leitores--;
    if (leitores == 0 && escritores_espera >0)cond_write.signal(trinco);
    abrir (trinco);
}
```