Programação multi-tarefa em Memória Partilhada

Parte II – Programação avançada

Sistemas Operativos

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*)

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;
conta_t contas[N];

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 teoria, permite maior paralelismo
- Mas, atenção, compensa mesmo?
  - Custo de memória do trinco
    - vs. tamanho do objeto a proteger
  - Custo computacional
    - Tempo para fechar/abrir vs. duração da secção crítica
  - Há grandes ganhos de paralelismo na prática?
    - Por exemplo, se objetos partilhados forem raramente acedidos, haverá pouca diferença entre trincos finos vs. 1 trinco global
- E programar com trincos finos é mais difícil

E no nosso projeto?

- Tabela de ficheiros abertos
- Tabela de i-nodes
- Vetor de blocos
- Mapas de alocação
  - ficheiros abertos
  - i-nodes
  - blocos

Um trinco global?
ou
um trinco por cada elemento?

Que tipo de trinco?

- Mutex: mais leve mas mais restritivo
- Trinco de leitura escrita (rwlock): mais pesado mas maior paralelismo com operações de leitura-apenas

- Objetos partilhados acedidos por operações de leitura-apenas?
- Essas operações têm impacto relevante no desempenho global do sistema?
- Então usar rwlock, caso contrário mutex.

E no nosso projeto?

- Tabela de ficheiros abertos
- Tabela de i-nodes
- Vetor de blocos
- Mapas de alocação
  - ficheiros abertos
  - i-nodes
  - blocos

Um trinco global?
ou
um trinco por cada elemento?

Mutex ou rwlock?

“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

Outras dicas (a pensar no projeto)

- Inicializar os trincos no início do *main*, destruir os trincos no final do *main*
  - Operações concorrentes fazem lock/unlock, não init/destroy
- Caso uma operação aceda a mais que um objeto partilhado
  - Tipicamente, obter o trinco de cada objeto antes deste ser acedido
  - Libertar todos os trincos no final da operação
    - Não esquecer os *return* de erros!

Exemplo com trincos finos

```
transferir(conta a, conta b, int montante) {
  fechar(a.trinco);
  debitar(a, montante);
  fechar(b.trinco);
  creditar(b, montante);
  abrir(a.trinco);
  abrir(b.trinco);
}
```

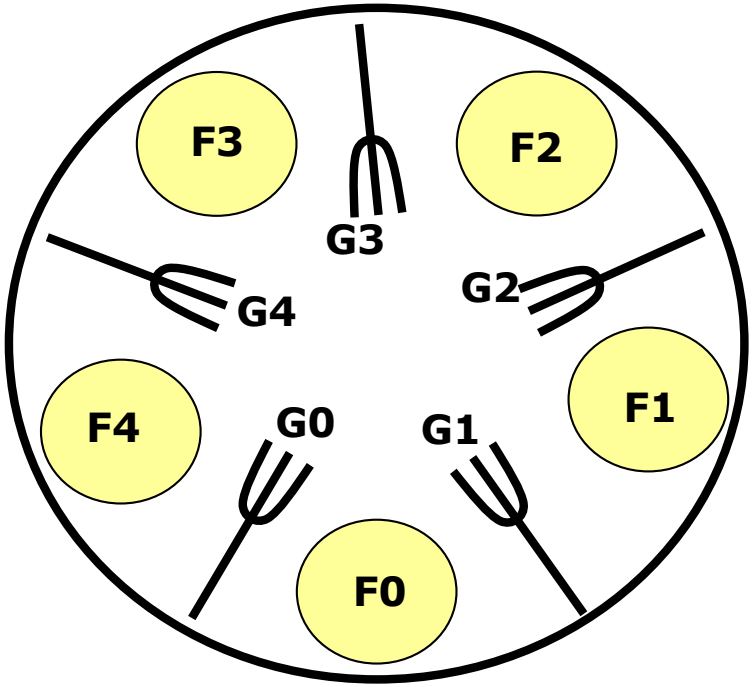
- O que pode correr mal?

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

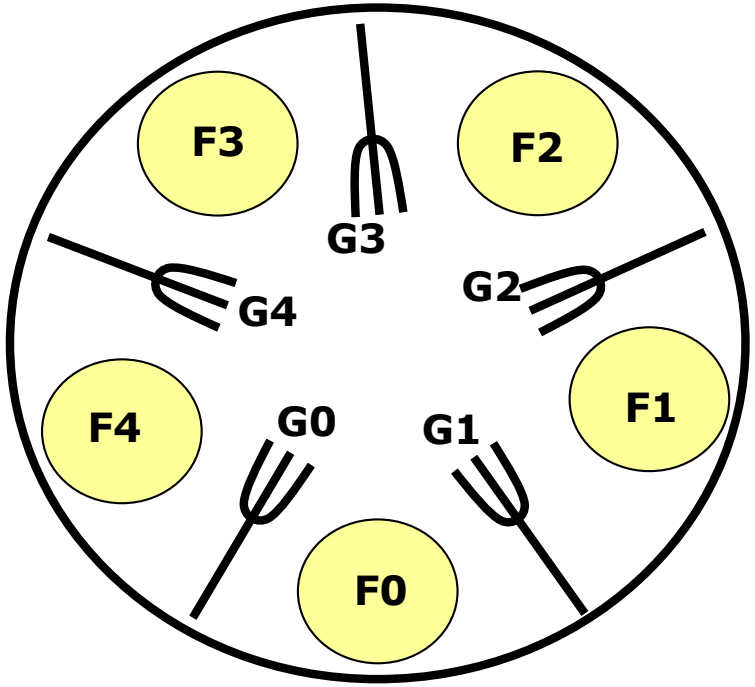


Jantar dos Filósofos

```
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]);
    }
}
```

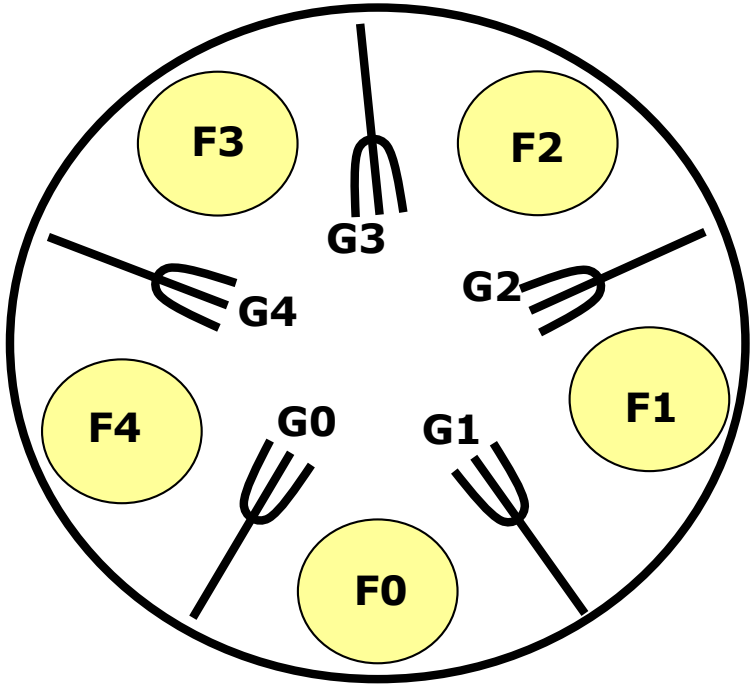


Problema?

Como prevenir interblocagem?

Prevenir de interblocagem: uma solução

```
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 segundo uma ordem total pré-definida

Prevenir de interblocagem: outra solução

```
mutex_t garfo[5] = {…};

filosofo(int id)
{
   int garfos;
   while (TRUE) {
      pensar();
      garfos = FALSE;
      while (!garfos){
         if (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
               sleep(random([0, MAX]);}
      }
      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

Recapitulando: como prevenir interblocagem?

- Garantir que os recursos são todos adquiridos segundo uma ordem total pré-definida
- 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?

Trincos – Limitações

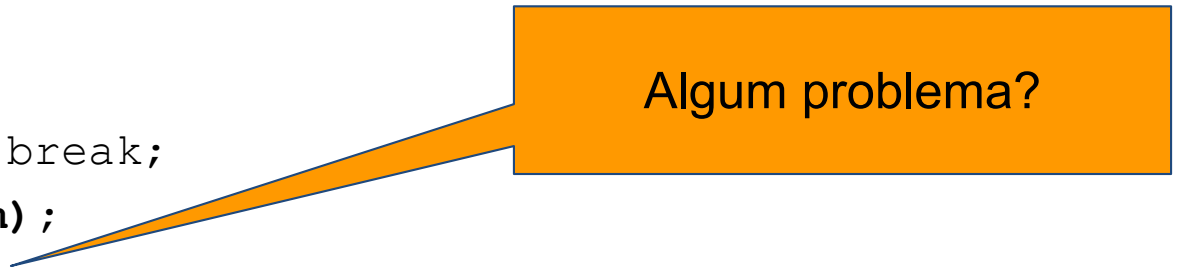
- Só servem para proteger secções críticas
  - Não são suficientemente expressivos para resolver outros problemas de sincronização
- Exemplo:
  - Num dado ponto do código, tarefa só quer avançar quando uma condição se verificar

Exemplo: acesso a parque de estacionamento

```
int vagas = N;

void entrar() {
  if (vagas==0)
    esperar até haver vaga
  vagas --;
}

void sair() {
  vagas ++;
}
```

Exemplo: acesso a parque de estacionamento

```
int vagas = N; mutex m;

void entrar() {
  do {
    lock(m);
    if (vagas>0) break;
    else unlock(m);
  } while (1);
  vagas --;
  unlock(m);
}

void sair() {
  lock(m);
  vagas ++;
  unlock(m);
}
```



Variáveis de Condição

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

Exemplo: acesso a parque de estacionamento

```
int vagas = N;

void entrar() {
  if (vagas==0)
    esperar até haver vaga
  vagas --;
}

void sair() {
  vagas ++;
}
```

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

- pthread_cond_t
- Criação/destruição de variáveis de condição;
  - pthread_cond_init (condition,attr)
  - pthread_cond_destroy (condition)
- Assinalar e esperar nas variáveis de condição:
  - pthread_cond_wait (condition,mutex)
  - pthread_cond_signal (condition)
  - pthread_cond_broadcast (condition)

Exemplo: acesso a parque de estacionamento

```
int vagas = N; mutex m;

void entrar() {
  lock(m);
  if (vagas == 0)
    //esperar até condição mudar
  vagas --;
  unlock(m);
}

void sair() {
  lock(m);
  vagas ++;
  unlock(m);
}
```

Tentemos concretizar este programa usando variáveis de condição: https://tinyurl.com/so-condvars1

Exemplo: acesso a parque de estacionamento

```
int vagas = N; mutex m; cond c;

void entrar() {
  lock(m);
  while (vagas == 0)
    wait(c, m);
  vagas --;
  unlock(m);
}

void sair() {
  lock(m);
  vagas ++;
  signal(c);
  unlock(m);
}
```

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 == N)
    return -1;
  buffer[prodptr] = item;
  prodptr ++;
  if (prodptr == N) prodptr=0;
  count++;
  return 1;
}

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

  item = buffer[consptr];
  consptr ++;
  if (consptr ==N) consptr = 0;
  count--;
  return item;
}
```

Problemas caso enviar/receber sejam chamadas concorrentemente?

Como estender para suportar envio/recepção síncrona?

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

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

```
1.  enviar(int item) {
2.    pthread_mutex_lock(&mutex);
3.    //AQUI QUERO:
4.    //Esperar enquanto buffer cheio
5.    buffer[prodptr] = item;
6.    prodptr ++;
7.    if (prodptr == N) prodptr=0;
8.    count++;
9.    pthread_mutex_unlock(&mutex);
10.   return 1;
11. }

12. int receber(){
13.   int item;
14.   pthread_mutex_lock(&mutex);
15.   //AQUI QUERO:
16.   //Esperar enquanto buffer vazio
17.   item = buffer[consptr];
18.   consptr ++;
19.   if (consptr ==N) consptr = 0;
20.   count--;
21.   pthread_mutex_unlock(&mutex);
22.   return item;
23. }
```

Produtor – Consumidor com Variáveis Condição

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

pthread_mutex_t mutex;
pthread_cond_t podeProd, podeCons;
```

```
produtor() {
  while(TRUE) {
    int item = produz();
    pthread_mutex_lock(&mutex);
    while (count == N) pthread_cond_wait(&podeProd,&mutex);
    buf[prodptr] = item;
    prodptr ++; if(prodptr==N) prodptr = 0;
    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 ++; if (consptr == N) consptr = 0;
      count--;
      pthread_cond_signal(&podeProd);
      pthread_mutex_unlock(&mutex);
   }
}
```

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 *esperc*

**Uma tarefa só pode chamar
trinco associado à var**

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)
```

Exemplo: acesso a parque de estacionamento

```
int vagas = N; mutex m; cond c;

void entrar() {
  lock(m);
  while (vagas == 0)
    wait(c, m);
  vagas --;
  unlock(m);
}

void sair() {
  lock(m);
  vagas ++;
  signal(c);
  unlock(m);
}
```

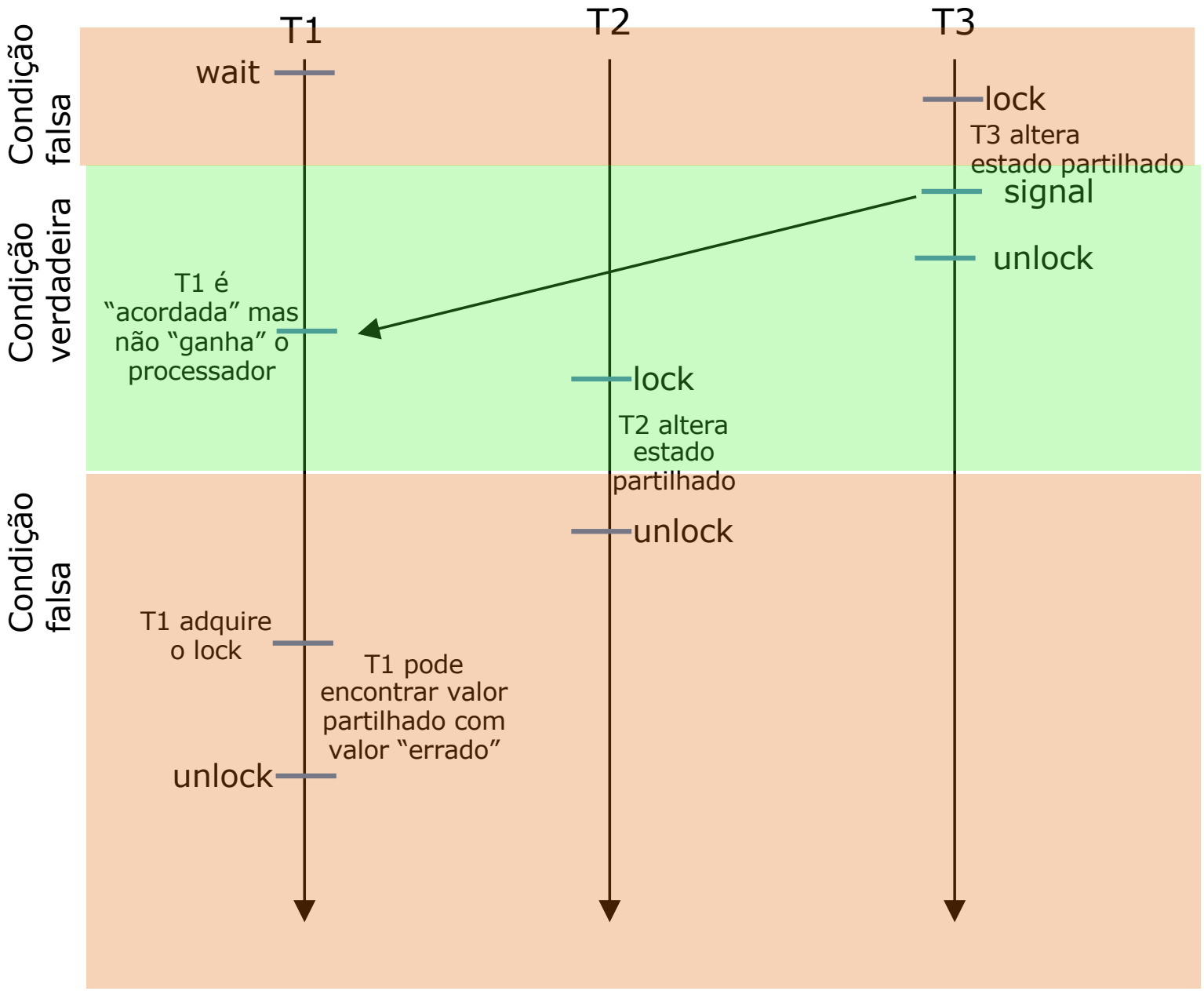
Variável de Condição: discussão (I)

- 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)

- 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(II)

- 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(II)

- 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*