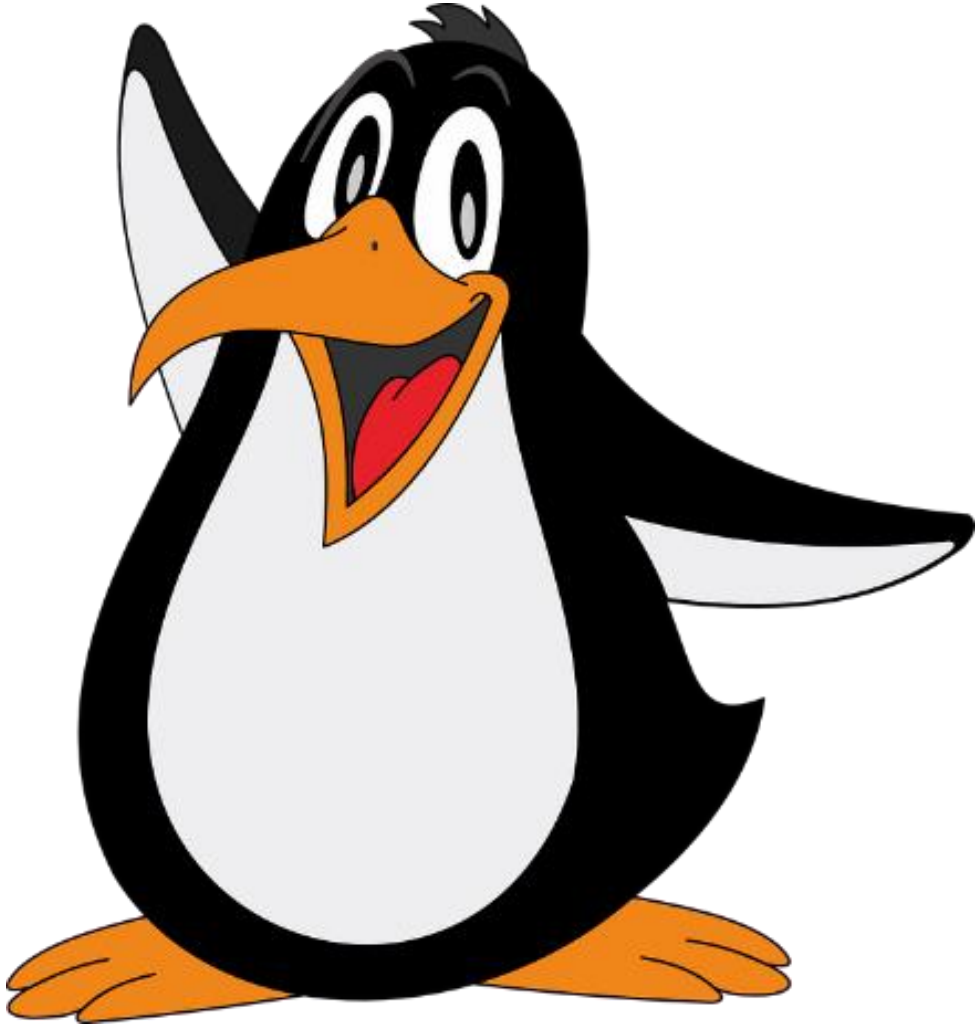
Sistemas Operacionais II

Threads – Parte 2

Fabricio Breve
fabricio.breve@unesp.br
https://www.fabriciobreve.com

Sumário

- Sincronização e Seções Críticas
- Condições de Corrida
- *Mutexes*
- *Deadlocks*
- Teste de *mutex* não bloqueante
- Semáforos
- Variáveis de Condição
- Implementação de *Threads* no Linux
- Processos versus *Threads*

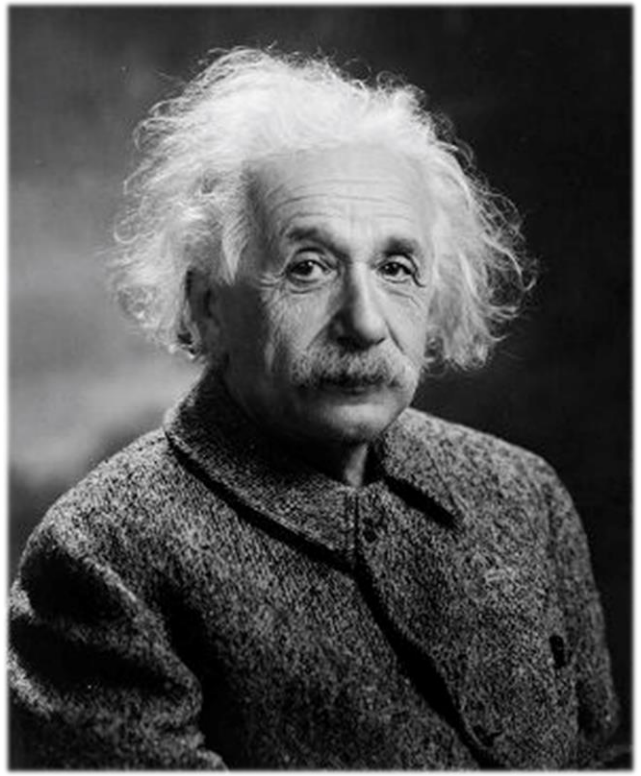


Sincronização e Seções Críticas

- Programar usando *threads* é uma tarefa cheia de truques pois a maioria dos programas que usam *threads* são também programas que usam programação concorrente.
  - Não há como saber quando uma determinada *thread* será executada.
  - Em sistemas com múltiplos processadores e/ou múltiplos núcleos, *threads* podem rodar literalmente ao mesmo tempo.

Sincronização e Seções Críticas

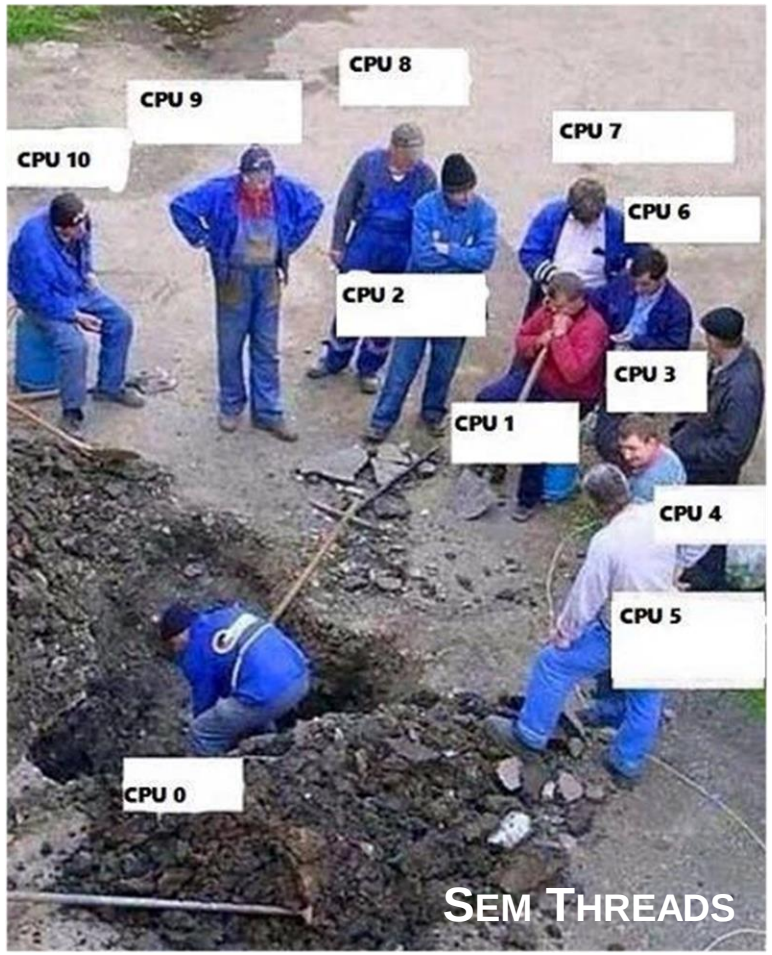
- Depurar um programa que usa *threads* é difícil, pois nem sempre é possível reproduzir o comportamento que causou o problema.
  - Tudo pode correr bem em uma execução e causar um travamento na próxima.
- Muitos problemas ocorrem envolvendo *threads* que acessam os mesmos dados.
  - O aspecto poderoso e perigoso das *threads*.



“Insanidade é continuar fazendo sempre a mesma coisa e esperar resultados diferentes.”

Albert Einstein nunca disse isso!
Pare de acreditar em qualquer bobagem que escrevem na Internet! ☺

Essa não é a definição de insanidade, é apenas o que acontece quando você executa um programa *multi-thread* que tem algum *bug*.

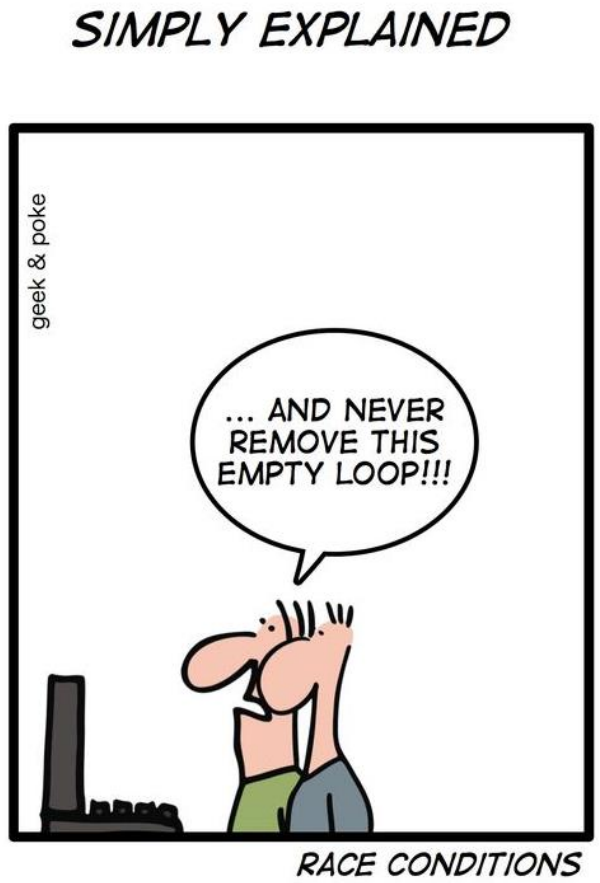

CPU 8
CPU 9
CPU 7
CPU 10
CPU 6
CPU 2
CPU 3
CPU 1
CPU 4
CPU 5
CPU 0

SEM THREADS


MULTITHREADING
AS *THREADS* NÃO VÃO SE SINCRONIZAR SOZINHAS

Condições de Corrida

- São bugs que ocorrem quando *threads* “correm” para alterar a mesma estrutura de dados.
  - Os programas só funcionam se uma *thread* executar mais frequentemente ou mais cedo que outra.



Condições de Corrida

- Exemplo:
  - Suponha que seu programa tem uma série de tarefas em fila que são processadas por várias *threads* concorrentes.
  - A lista de tarefas é representada por uma lista ligada de objetos do tipo **`struct job`**
  - Depois que cada *thread* termina uma operação, ela checa a fila para ver se outro trabalho está disponível.
  - Se **`job_queue`** não é nulo, a *thread* remove a primeira tarefa da fila e ajusta **`job_queue`** para a próxima tarefa da fila.

```c
struct job {
  /* Campo de link para a lista encadeada.  */
  struct job* next;

  /* Outros campos descrevendo o trabalho a ser realizado... */
};

/* Uma lista ligada de tarefas pendentes.  */
struct job* job_queue;

extern void process_job (struct job*);
```

job-queue1.c

Função de *Thread* para Processar Tarefas da Fila

```c
/* Processa as tarefas na fila até que a fila esteja vazia.  */

void* thread_function (void* arg)
{
  while (job_queue != NULL) {
    /* Pegue a próxima tarefa disponível.  */
    struct job* next_job = job_queue;
    /* Remove esta tarefa da lista.  */
    job_queue = job_queue->next;
    /* Faça o trabalho.  */
    process_job (next_job);
    /* Limpeza.  */
    free (next_job);
  }
  return NULL;
}
```

Condições de Corrida

- Que problema pode ocorrer no exemplo anterior?
  - Duas *threads* podem acabar suas tarefas quase ao mesmo tempo.
  - Existe mais um único trabalho na fila.
  - A primeira *thread* verifica se **`job_queue`** não é nulo, vê que não é e entra no *loop* armazenando o ponteiro para o objeto em **`next_job`**.
  - Neste momento, o Linux interrompe a primeira *thread* e executa a segunda.
  - A segunda *thread* também checa **`job_queue`** e, vendo que ele não é nulo, também ajusta seu ponteiro **`next_job`** para ela.
  - **Agora temos duas *threads* executando a mesma tarefa.**
  - E não para por aí, a primeira *thread* removerá o objeto da fila, deixando-a em nulo. Quando a outra *thread* tentar acessar **`job_queue -> next`**, uma falha de segmentação ocorrerá.

Condições de Corrida

- O exemplo anterior é um exemplo de condição de corrida.
  - Com sorte a situação do exemplo nunca ocorrerá e a condição de corrida nunca aparecerá.
  - O *bug* poderá aparecer em uma circunstância diferente, como uma alta carga no sistema.
- Para eliminar condições de corrida, é preciso tornar operações *atômicas*.
  - **Indivisíveis e ininterruptas** até que sejam completadas.
  - Nenhuma outra operação será executada neste tempo.
  - No exemplo anterior, a checagem de **`job_queue`** e a remoção da primeira tarefa (se não nula) deveriam ser uma operação atômica.

Mutexes

- Para implementar operações atômicas o GNU/Linux oferece *mutexes*.
  - *MUTual EXclusion locks*
  - Tipo especial de trava que apenas uma *thread* pode travar de cada vez:
    - Se uma *thread* trava uma *mutex* e uma segunda *thread* tenta travá-la também, a segunda é bloqueada e colocada na espera.
    - Apenas quando a primeira *thread* destravar a *mutex*, a segunda *thread* é desbloqueada.

Mutexes

- Para criar uma *mutex*, crie uma variável do tipo **`pthread_mutex_t`** e chame a função **`pthread_mutex_init`** passando um ponteiro para a variável criada.
  - O segundo argumento é um ponteiro para um *objeto de atributos mutex*.
    - Se o ponteiro é **`NULL`**, os atributos padrões são assumidos.
    - A variável *mutex* deve ser inicializada apenas uma vez.
    - Exemplo:
      - **`pthread_mutex_t mutex;`**
      - **`pthread_mutex_init (&mutex, NULL);`**

Mutexes

- Outra alternativa é chamar o valor especial **`PTHREAD_MUTEX_INITIALIZER`**, dispensando a chamada a **`pthread_mutex_init`**
  - Útil para variáveis globais.
  - Exemplo:
    - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`

Mutexes

- Para tentar travar uma *mutex* chame **`pthread_mutex_lock`** na variável *mutex*.
  - Se estiver destravada, será travada e a função retornará imediatamente.
  - Se outra *thread* já travou a *mutex*, a *thread* bloqueia e só retorna quando a *mutex* for destravada pela outra *thread*.
    - Mais de uma *thread* pode ficar bloqueada em uma *mutex* travada de uma vez só.
      - Quando ocorre o destravamento da *mutex*, apenas uma das *threads* bloqueadas é desbloqueada (não dá para prever qual) e travará a *mutex;* as demais permanecem bloqueadas.

Mutexes

- **`pthread_mutex_unlock`** destrava a *mutex*.
  - Sempre deve ser chamada pela mesma *thread* que bloqueou a *mutex*.

- Agora vamos ver como fica o exemplo das tarefas usando *mutexes*...

```c
#include <malloc.h>
#include <pthread.h>

struct job {
  /* Campo de link para a lista encadeada.  */
  struct job* next;

  /* Outros campos descrevendo o trabalho a ser realizado... */
};

/* Uma lista encadeada de tarefas pendentes.  */
struct job* job_queue;

extern void process_job (struct job*);

/* Uma mutex protegendo a fila de tarefas.  */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
```

job-queue2.c

Função de *Thread* de Fila de Tarefas, Protegida por uma *Mutex*

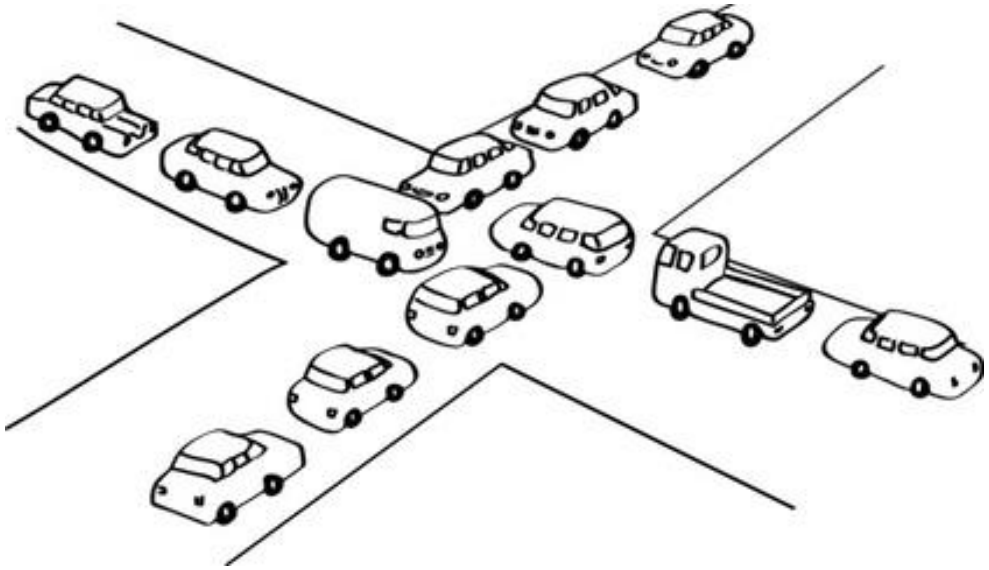
```c
/* Tarefas enfileiradas até que a fila esteja vazia.  */

void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;

    /* Trave a mutex na fila de tarefas.  */
    pthread_mutex_lock (&job_queue_mutex);
    /* Agora é seguro checar se a fila está vazia.  */
    if (job_queue == NULL)
      next_job = NULL;
    else {
      /* Pegue a próxima tarefa disponível.  */
      next_job = job_queue;
      /* Remova esta tarefa da lista.  */
      job_queue = job_queue->next;
    }
    /* Destrave a mutex na fila de tarefas, pois já terminamos com
       a fila por enquanto.  */
    pthread_mutex_unlock (&job_queue_mutex);

    /* A fila estava vazia?  Se sim, encerre a thread.  */
    if (next_job == NULL)
      break;

    /* Faça o trabalho.  */
    process_job (next_job);
    /* Limpeza.  */
    free (next_job);
  }
  return NULL;
}
```

Mutexes

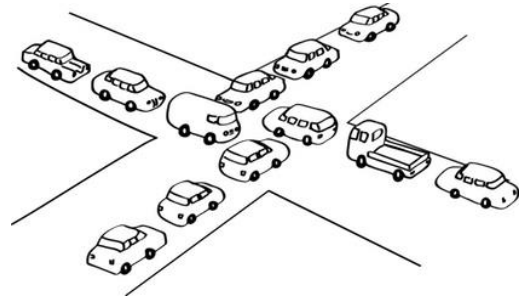
- Importante:
  - **`next_job`** só é acessado fora da região entre o travamento e o destravamento da *mutex* após ter sido removido da fila, estando agora inacessível para outras *threads*.
  - No caso de fila vazia, o *break* ocorre só depois de destravar a *mutex*, do contrário ela ficaria permanentemente travada, de forma que nenhuma outra *thread* voltaria a acessar **`job_queue`**

Deadlocks

- *Mutexes* fornecem um mecanismo para que uma *thread* possa bloquear outras.
  - Isto abre a possibilidade de uma nova classe de *bugs*, chamada *deadlocks*.
    - Ocorre quando uma ou mais *threads* ficam emperradas esperando por algo que nunca acontecerá.



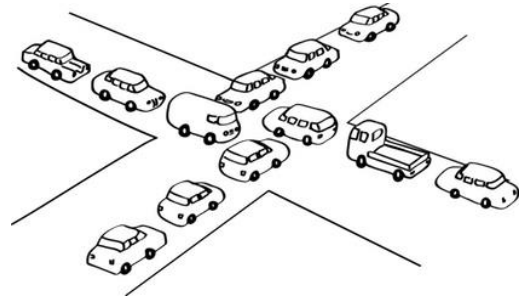
Deadlocks



- Um tipo simples de *deadlock* ocorre quando a mesma *thread* tenta travar uma *mutex* duas vezes em seguida. O comportamento neste caso depende do tipo de *mutex* usado. Existem três tipos:
  - *Mutex* rápida (padrão)
  - *Mutex* recursiva
  - *Mutex* com checagem de erros

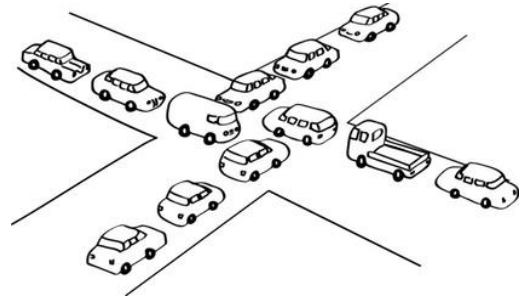
https://www.oficinadanet.com.br/post/12786-sistemas-operacionais-o-que-e-deadlock

Deadlocks



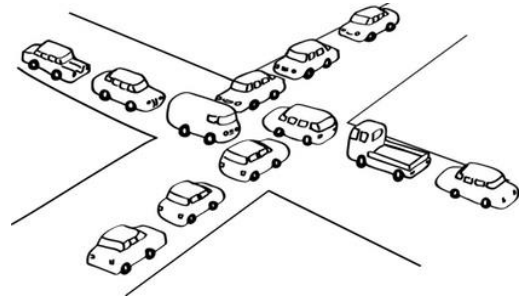
- O que acontece ao tentar travar novamente a mesma *mutex* dentro da mesma *thread* que a travou?
  - *Mutex rápida:*
    - Causa um *deadlock.*
      - Uma tentativa de travar a *mutex* bloqueia até que a *mutex* seja destravada, mas como a *thread* que travou está bloqueada na mesma *mutex*, a trava nunca será removida.
  - *Mutex recursiva:*
    - Não causa um *deadlock.*
      - A *mutex recursiva* pode ser travada várias vezes pela mesma thread, pois ela lembra quantas vezes **`pthread_mutex_lock`** foi chamada nela pela *thread* a quem pertence a trava. Tal *thread* deve fazer o mesmo número de chamadas para **`pthread_mutex_unlock`** antes que a trava seja realmente removida e outra *thread* possa travá-la.
  - *Mutex com checagem de erros:*
    - Retorna o código de erro **`EDEADLK`**
      - Ao tentar travar uma *mutex* já travada pela mesma *thread*, o que geraria um *deadlock*, retorna o código de erro **`EDEADLK`**

Deadlocks



- Por padrão, o GNU/Linux cria *mutexes* do tipo rápido.
- Para criar os demais tipos:
  - Crie um *objeto de atributos de mutex* declarando uma variável **`pthread_mutexattr_t`** e chamando **`pthread_mutexattr_init`** com um ponteiro para a variável criada.
  - Configure o tipo de *mutex* chamando **`pthread_mutexattr_setkind_np`**
    - O primeiro argumento é um ponteiro para o *objeto de atributos de mutex*.
    - O segundo é **`PTHREAD_MUTEX_RECURSIVE_NP`** para uma *mutex recursiva,* ou **`PTHREAD_MUTEX_ERRORCHECK_NP`** para uma *mutex com checagem de erros*.
  - Passe um ponteiro do objeto de atributos para **`pthread_mutex_init`** para criar *mutexes* deste tipo, e então destrua o objeto de atributos com **`pthread_mutexattr_destroy`**

Deadlocks



- Exemplo de criação de *mutex com checagem de erros* :

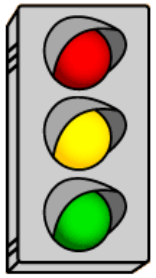
```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;

pthread_mutexattr_init (&attr);
pthread_mutexattr_setkind_np (&attr, PTHREAD_MUTEX_ERRORCHECK_NP);
pthread_mutex_init (&mutex, &attr);
pthread_mutex_destroy (&attr);
```

- O sufixo **NP** dos tipos *recursivo* e *com checagem de erros* informa que eles são específicos do Linux e, portanto, **N**ão **P**ortáveis. Desta forma, não é recomendável utilizá-los em programas, porém os mesmos são úteis para depuração.

Testes de *Mutex* Não Bloqueantes

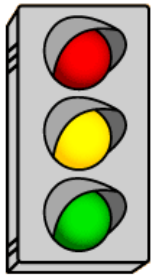
- As vezes é útil testar se uma *mutex* está travada, mas sem bloquear caso ela esteja.
  - Exemplo: uma *thread* deseja travar uma *mutex*, mas tem outro trabalho a fazer caso não consiga a trava.
    - **`pthread_mutex_lock`** não serve para isso...
      - bloqueia esperando a *mutex* destravar e não retorna enquanto não obtém a trava.
    - **`pthread_mutex_trylock`** é a solução!
      - Se a *mutex* estiver destravada, ele a trava e retorna zero.
      - Se a *mutex* estiver travada, ele não bloqueia e retorna imediatamente com o código de erro **EBUSY**. A trava obtida por outra *thread* não será afetada.

Semáforos para Threads



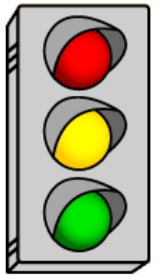
- No exemplo da fila de tarefas, onde várias *threads* processam tarefas de uma fila, a função de *thread* principal carrega a próxima tarefa da fila até que não haja mais tarefas, e então sai.
  - Esse esquema funciona se todas as tarefas são colocadas na fila antecipadamente ou se novas tarefas são colocadas na fila ao menos tão rápido quanto as *threads* as processam.
  - Porém, se as *threads* trabalharem muito rápido, a fila ficará vazia e a *thread* encerrará.
    - Se chegarem novas tarefas depois, não haverá *threads* para processá-las.
    - Precisamos de um mecanismo que bloqueie as *threads* quando a fila encerra, até que novas tarefas se tornem disponíveis.

Semáforos para Threads



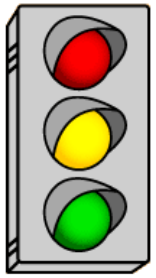
- Um *semáforo* é uma maneira de resolver tal problema.
- *Semáforo* é um contador que pode ser usado para sincronizar múltiplas *threads*.
  - O GNU/Linux garante que a checagem e modificação do valor de um semáforo pode ser feito de maneira segura, sem criar uma condição de corrida.

Semáforos para Threads



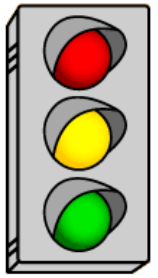
- Cada semáforo tem um valor de contador, que é um inteiro não negativo, e tem duas operações básicas:
  - **`wait`**
    - Decrementa o valor do semáforo em 1.
      - Se o valor já é zero, a operação bloqueia até que o valor do semáforo se torne positivo.
        - » Quando o valor se torna positivo, o valor é decrementado em 1 e a operação **`wait`** retorna.
  - **`post`**
    - Incrementa o valor do semáforo em 1.
      - Se o valor do semáforo era zero e outras *threads* estão bloqueadas em uma operação **`wait`** neste semáforo, uma das *threads* é desbloqueada e a operação **`wait`** completa.
        - » Neste caso, o valor do semáforo volta a ser zero.

Semáforos para Threads



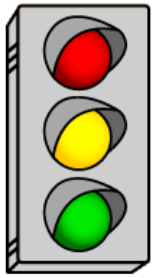
- O GNU/Linux oferece duas implementações de semáforo ligeiramente diferentes.
  - A que veremos na aula de hoje é a implementação de semáforos do padrão POSIX.
    - Use-a na comunicação entre *threads*.
  - A outra veremos nas aulas de comunicação entre processos.
- Para usar semáforos, inclua **`<semaphore.h>`**

Semáforos para Threads



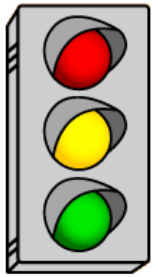
- Um semáforo é representado por uma variável **`sem_t`**
  - Antes de usá-lo, inicialize-o com a função **`sem_init`**
    - Argumentos:
      - Ponteiro para a variável **`sem_t`**
      - Zero.
        - » não zero indica um semáforo que pode ser compartilhado entre processos.
      - Valor inicial do semáforo.
  - Se não precisa mais de um semáforo, desaloque-o com **`sem_destroy`**

Semáforos para Threads



- **`sem_wait`**
  - Decrementa um semáforo (-1).
  - Bloqueia se estiver zerado.
- **`sem_post`**
  - Incrementa um semáforo (+1).
- **`sem_trywait`**
  - Decrementa semáforo (-1).
  - Se estiver zerado retorna imediatamente, sem bloquear, com o valor de erro **`EAGAIN`**

Semáforos para Threads



- **`sem_getvalue`**
  - Recupera o valor atual do semáforo.
  - **Não use** para tomar decisões sobre postar/esperar.
    - Poderia levar a condições de corrida.
    - Use **`post`**/**`wait`** que são atômicos.
  - Argumentos:
    - Variável de semáforo **`sem_t`**
    - Ponteiro para **`int`** que receberá o valor lido.

```c
#include <malloc.h>
#include <pthread.h>
#include <semaphore.h>

struct job {
  /* Campo de ligação para a fila encadeada.  */
  struct job* next;

  /* Outros campos descrevendo a tarefa a ser realizada... */
};

/* Uma fila encadeada de trabalhos pendentes.  */
struct job* job_queue;

extern void process_job (struct job*);

/* Uma mutex protegendo a fila de tarefas.  */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;

/* Um semáforo contando o número de tarefas na fila.  */
sem_t job_queue_count;

/* Realizar a iniciação da fila de tarefas uma única vez.  */

void initialize_job_queue ()
{
  /* A fila está inicialmente vazia.  */
  job_queue = NULL;
  /* Inicializar o semáforo que conta as tarefas na fila. Seu
     valor inicial deve ser zero.  */
  sem_init (&job_queue_count, 0, 0);
}
```

job-queue3.c

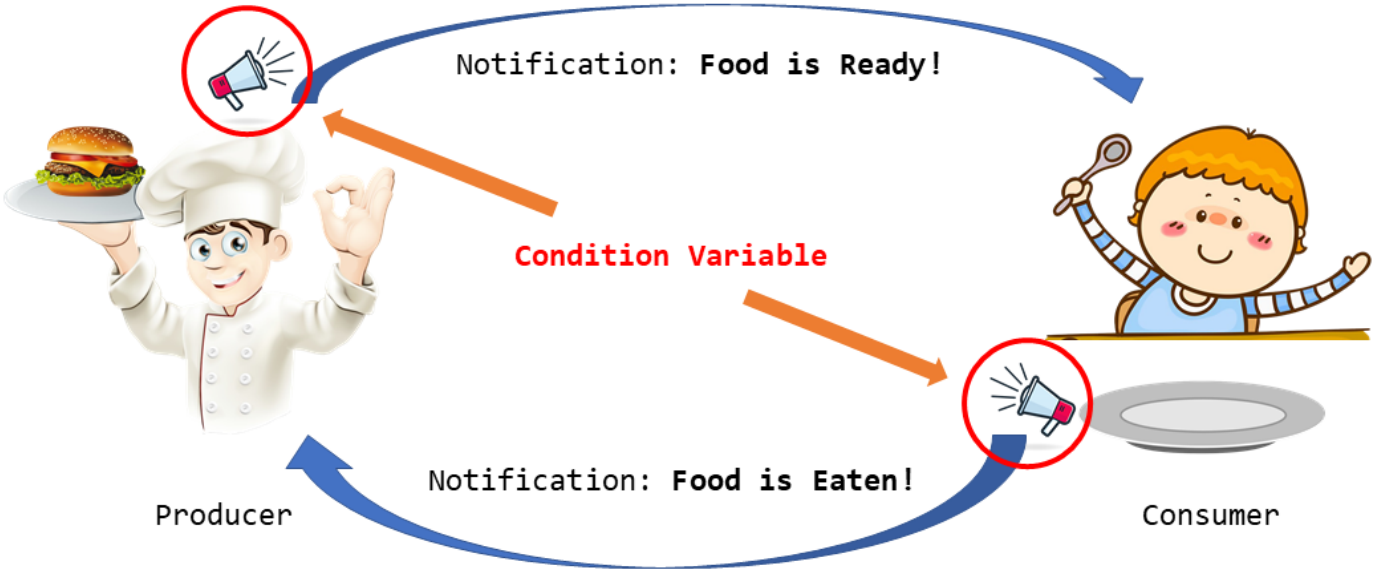
Fila de Tarefas controlada por um Semáforo

Obs: o que está sendo chamado de **fila** neste código na verdade está funcionando como uma **pilha**

```c
/* Processa tarefas até que a fila esteja vazia.  */
void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;

    /* Espere no semáforo da fila de tarefas. Se o valor é positivo, indicando que a fila não
       está vazia, decremente o contador em um. Se a fila está vazia, bloqueie até uma
       nova tarefa entrar na fila.  */
    sem_wait (&job_queue_count);
    /* Trave a mutex na fila de tarefas.  */
    pthread_mutex_lock (&job_queue_mutex);
    /* Por conta do semáforo, sabemos que a fila não está vazia. Pegue á próxima tarefa
       disponível.  */
    next_job = job_queue;
    /* Remova este trabalho da lista.  */
    job_queue = job_queue->next;
    /* Destrave o mutex na fila de tarefas, pois já terminamos com a fila por agora.  */
    pthread_mutex_unlock (&job_queue_mutex);
    /* Realizar a tarefa.  */
    process_job (next_job);
    /* Limpeza.  */
    free (next_job);
  }
  return NULL;
}

/* Adiconar uma nova tarefa na frente da fila de tarefas.  */
void enqueue_job (/* Passe dados específicos da tarefa aqui...  */)
{
  struct job* new_job;
  /* Alocar o novo objeto de tarefa.  */
  new_job = (struct job*) malloc (sizeof (struct job));
  /* Ajustar outros campos para a estrutura da tarefa aqui...  */

  /* Travar a mutex na fila de tarefas antes de acessá-la.  */
  pthread_mutex_lock (&job_queue_mutex);
  /* Colocar a nova tarefa na cabeça da fila.  */
  new_job->next = job_queue;
  job_queue = new_job;

  /* Postar para o semáforo para indicar que outra tarefa está disponível. Se threads estão
     bloqueadas, esperando o semáforo, uma será desbloqueada e processará a tarefa.  */
  sem_post (&job_queue_count);

  /* Destravar a mutex da fila de tarefas.  */
  pthread_mutex_unlock (&job_queue_mutex);
}
```

Variáveis de Condição

- É um terceiro dispositivo de sincronização fornecido pelo GNU/Linux.
- Permite condições mais complexas para execução de *threads*.



Variáveis de Condição

- Exemplo:
  - Imagine uma *thread* que execute um *loop* infinito, fazendo algum trabalho em cada iteração.
  - Porém, o *loop* da *thread* é controlado por uma *flag*:
    - executa somente quando a *flag* está ligada.
    - pausa quando a *flag* está desligada.
- Veja a seguir uma implementação ***ineficiente*** deste exemplo.

```c
#include <pthread.h>

extern void do_work ();

int thread_flag;
pthread_mutex_t thread_flag_mutex;

void initialize_flag ()
{
  pthread_mutex_init (&thread_flag_mutex, NULL);
  thread_flag = 0;
}
```

spin-condvar.c

Uma implementação simples
de variável de condição

```c
/* Chama do_work repetidamente enquanto a flag da thread
   está ligada; caso contrário fica apenas girando.  */

void* thread_function (void* thread_arg)
{
  while (1) {
    int flag_is_set;

    /* Proteger a flag com uma trava mutex.  */
    pthread_mutex_lock (&thread_flag_mutex);
    flag_is_set = thread_flag;
    pthread_mutex_unlock (&thread_flag_mutex);

    if (flag_is_set)
      do_work ();
    /* Senão, não faça nada. Apenas faça o loop novamente.  */
  }
  return NULL;
}

/* Ajustar o valor da flag da thread para FLAG_VALUE.  */

void set_thread_flag (int flag_value)
{
  /* Proteger o flag com uma trava mutex.  */
  pthread_mutex_lock (&thread_flag_mutex);
  thread_flag = flag_value;
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

Variáveis de Condição

- Por que a implementação anterior é ineficiente?
  - A *thread* gasta muito tempo da CPU quando a *flag* não está ligada, apenas checando e rechecando a *flag*, travando e destravando a *mutex*.
  - O que necessitamos é uma forma de colocar a *thread* para dormir quando a *flag* não está ligada, até que alguma circunstância faça a *flag* ligar.
    - É exatamente isso que fazem as variáveis de condição!

Variáveis de Condição

- Com variáveis de condição você pode criar uma condição que:
  - Quando verdadeira, a *thread* execute.
  - Quando falsa, a *thread* é bloqueada.
- O Linux garante que a *thread* bloqueada será desbloqueada quando a condição mudar.

Variáveis de Condição

- Como com semáforos, uma *thread* pode *esperar* (**`wait`**) em uma variável de condição:
  - Se a *thread* A *espera* em uma variável de condição, ela é bloqueada até que alguma outra *thread* sinalize a mesma variável de condição.
  - Ao contrário de um semáforo, uma variável de condição não tem contador ou memória, ou seja, a *thread* A deve *esperar* **antes** que outra *thread* sinalize.
  - Se uma *thread* sinaliza a variável de condição antes que a *thread* A esteja esperando, o sinal é perdido, e a *thread* A bloqueia até que alguma *thread* sinalize a variável de condição novamente.

Variáveis de Condição

- No exemplo anterior, usaríamos variáveis de condição assim:
  - O *loop* em **`thread_function`** checa a *flag*. Se a *flag* não está ligada, a *thread* espera na variável de condição.
  - A função **`set_thread_flag`** sinaliza a variável de condição após ligar a *flag.*
    - Desta forma, se **`thread_function`** está bloqueada na variável de condição, ela será desbloqueada e irá checar a condição novamente.
- Mas temos um problema:
  - Há uma condição de corrida entre checar o valor da *flag* e esperar na variável de condição.
    - **`thread_function`** poderia checar o valor da *flag* e ela estar desligada.
    - Neste momento o escalonador alterna para a *thread* principal que liga a *flag* e sinaliza a variável de condição, antes que **`thread_function`** acione **`wait`**, ou seja, o sinal é descartado.
    - Ao voltar para a **`thread_function`**, esta bloqueará em **`wait`** para sempre.

Variáveis de Condição

- Para resolver o problema, precisamos de uma forma de travar a *flag* e a variável de condição juntas com uma mesma *mutex*.
  - Felizmente o Linux fornece esse mecanismo!
  - Cada variável de condição deve ser usada em conjunto com uma *mutex*, para evitar este tipo de condição de corrida.
  - Usando este esquema, a função de *thread* fica assim:
    1. O loop em **`thread_function`** trava a *mutex* e lê o valor da *flag.*
    2. Se a *flag* está ligada, ele desbloqueia a *mutex* e executa a função de trabalho.
    3. Se a *flag* está desligada, ele **atomicamente** desbloqueia a *mutex* e espera a variável de condição.

Variáveis de Condição

- São representadas por uma instância de **`pthread_cond_t`**
- As funções para manipular as variáveis de condição são:
  - **`pthread_cond_init`**
    - Inicializa a variável de condição.
    - Argumentos:
      - Ponteiro para a instância de **`pthread_cond_t`**.
      - Ponteiro para um objeto de atributos de variável de condição. **Ignorado no Linux**.
  - **`pthread_cond_signal`**
    - Sinaliza a variável de condição.
      - Uma única *thread* que está bloqueada na variável de condição será desbloqueada.
        - » Se não houver nenhuma, o sinal é ignorado.
    - O argumento é um ponteiro para a instância de **`pthread_cond_t`**
  - **`pthread_cond_broadcast`**
    - Similar a **`pthread_cond_signal`** mas desbloqueia **todas** as *threads* bloqueadas na variável de condição.
  - **`pthread_cond_wait`**
    - Bloqueia a *thread* chamadora até que a variável de condição seja sinalizada.
    - Argumentos:
      - Ponteiro para a instância de **`pthread_cond_t`**
      - Ponteiro para a instância de **`pthread_mutex_t`**
    - Quando chamada, a *mutex* já deve estar travada pela *thread* chamadora.
    - Atomicamente destrava a *mutex* e bloqueia a variável de condição.
    - Quando a variável de condição é sinalizada e a *thread* chamadora é desbloqueada, **`pthread_cond_wait`** automaticamente readquire uma trava na *mutex.*

Variáveis de Condição

- Sempre que seu programa realizar alguma ação que possa mudar algo sendo protegido com a variável de condição (a *flag* em nosso exemplo), ele deve seguir os seguintes passos:
  1. Travar a *mutex* que acompanha a variável de condição.
  2. Tomar a ação que pode mudar o que está sendo protegido (no nosso exemplo, ajustar a *flag*).
  3. Sinalizar (ou difundir) a variável de condição, dependendo do comportamento desejado.
  4. Destravar a *mutex* que acompanha a variável de condição.

```c
#include <pthread.h>

extern void do_work ();

int thread_flag;
pthread_cond_t thread_flag_cv;
pthread_mutex_t thread_flag_mutex;

void initialize_flag ()
{
  /* Inicializar a mutex e a variável de condição.  */
  pthread_mutex_init (&thread_flag_mutex, NULL);
  pthread_cond_init (&thread_flag_cv, NULL);
  /* Inicializar o valor da flag.  */
  thread_flag = 0;
}
```

condvar.c

Controle uma Thread usando uma Variável de Condição

```c
/* Chamar do_work repetidamente até que a flag da thread seja
   ativada; bloqueia se a flag estiver desativada.  */

void* thread_function (void* thread_arg)
{
  /* Loop infinito.  */
  while (1) {
    /* Trava a mutex antes de acessar o valor da flag.  */
    pthread_mutex_lock (&thread_flag_mutex);
    while (!thread_flag)
      /* A flag está desligada.  Espere por um sinal na variável de
         condição, indicando que o valor da flag mudou. Quando o
         sinal chega e a thread desbloqueia, faça o loop e cheque a
         flag novamente.  */
      pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
    /* Quando chegamos aqui, sabemos que a flag deve estar ativada.
       Destrave a mutex.  */
    pthread_mutex_unlock (&thread_flag_mutex);
    /* Faça algum trabalho.  */
    do_work ();
  }
  return NULL;
}

/* Ajuste o valor da flag para FLAG_VALUE.  */

void set_thread_flag (int flag_value)
{
  /* Travar a mutex antes de acessar o valor da flag.  */
  pthread_mutex_lock (&thread_flag_mutex);
  /* Ajustar o valor da flag, e então sinalizar para o caso da função de
thread
     estar bloqueada, esperando pela ativação da flag.  Porém,
     a função de thread não poderá realmente checar o valor da flag até que
     a mutex seja desbloqueada.  */
  thread_flag = flag_value;
  pthread_cond_signal (&thread_flag_cv);
  /* Destravar a mutex.  */
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

Variáveis de Condição

- Podem ser usadas sem uma condição, apenas como mecanismo para bloquear uma *thread* até que outra a acorde.
  - Um semáforo também pode ser usado para este propósito, com algumas diferenças:
    - Semáforo:
      - Lembra da chamada mesmo que não existam *threads* bloqueadas no momento.
      - Só podem acordar uma *thread* de cada vez, mesmo que existam várias na espera.
    - Variáveis de Condição:
      - Descartam a chamada se não existe nenhuma *thread* na espera.
      - Pode acordar todas as *threads* que estiverem na espera.

Deadlocks com Duas ou Mais *Threads*

- Acontecem quando duas (ou mais) *threads* estão bloqueadas, esperando por uma condição ocorrer, que apenas outra *thread* bloqueada pode causar.
  - Exemplos:
    - *Thread* A está bloqueada em uma variável de condição esperando sinalização da *Thread* B, que por sua vez está bloqueada em variável de condição esperando sinalização da *Thread* A.
    - *Thread* A e *Thread* B precisam travar *mutex* 1 e *mutex* 2. *Thread* A trava *mutex* 1 e *Thread* B trava *mutex* 2. Ambas ficarão bloqueadas para sempre esperando a outra *mutex.*
      - Solução: certifique-se de que todas as *threads* adquiram as travas na mesma ordem.

Implementação de *Threads* no Linux

- **LinuxThreads**
  - A implementação original de *threads* no GNU/Linux (chamada **LinuxThreads**) era diferente da implementação de *threads* em muitos outros sistemas UNIX e similares.
    - *Threads* eram implementadas como processos.
      - O Linux cria um novo processo cada vez que uma *thread* é criada.
      - Porém não é um processo igual aos criados com **`fork`**.
      - Ele compartilha o mesmo espaço de endereços e recursos do processo original.
      - Cada *thread* tem seu próprio **`pid`**.

Implementação de *Threads* no Linux

- NTPL (Native POSIX Thread Library)
  - NTPL é a implementação de *threads* no Linux que, a partir da versão 2.6 do *kernel*, é utilizada por padrão.
    - Versão 2.6 do *kernel* lançada em 2003.
    - Cada *thread* de um mesmo processo é colocada em um mesmo grupo de *threads* e compartilha o mesmo **`pid`** .
    - Corrige uma série de problemas de desempenho, escalabilidade, usabilidade e não-conformidades com o padrão POSIX.

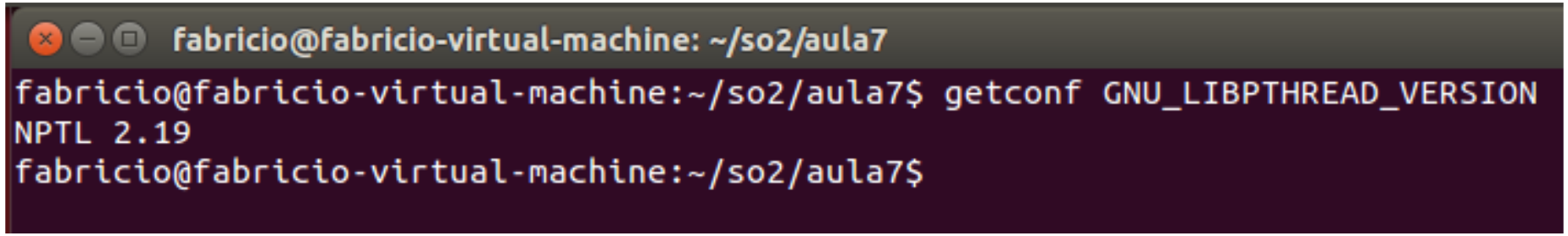
thread-pid.c

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void* thread_function (void* arg)
{
  fprintf (stderr, "pid do thread filho é %d\n", (int) getpid ());
  /* Rode para sempre.  */
  while (1);
  return NULL;
}

int main ()
{
  pthread_t thread;
  fprintf (stderr, "pid do thread principal é %d\n", (int) getpid ());
  pthread_create (&thread, NULL, &thread_function, NULL);
  /* Rode para sempre.  */
  while (1);
  return 0;
}
```

Veja em execução:
https://youtu.be/B_8kAQslFbg

Implementação de *Threads* no Linux

- Para determinar qual biblioteca de *threads* está sendo usada em seu sistema, use:
  - `getconf GNU_LIBPTHREAD_VERSION`



Veja em execução:
https://youtu.be/VDRAF9oOx60

Processos versus *Threads*

Processos

- O processo filho pode rodar um diferente executável chamando a função **exec**.
- Um processo errante não afeta memória e recursos de outros processos.
- Compartilhar memória requer comunicação entre processos.
- Mais usado em paralelismo com tarefas com diferenças significativas.

Threads

- Todas as *threads* de um programa devem rodar o mesmo executável.
- Uma *thread* errante pode causar danos a outras *threads* alterando a memória e recursos em comum.
- Compartilhar memória é trivial.
- Mais usado em paralelismo com tarefas quase idênticas.

Exercício

- Considere o seguinte problema:
  - Uma barbearia tem ***n*** barbeiros com suas respectivas cadeiras de barbeiro e ***m*** cadeiras para clientes esperarem por sua vez na sala de espera.
  - Quando não há clientes, o barbeiro se senta na cadeira e dorme.
  - Quando chega um cliente:
    - Se um barbeiro está disponível, ele precisa acordá-lo.
    - Se todos os barbeiros estão ocupados, o cliente senta-se em uma das cadeiras na sala de espera e espera sua vez.
    - Se não há cadeiras disponíveis na sala de espera, o cliente vai embora.
- Implemente um programa usando *threads*, uma fila, uma única *mutex* e uma única variável de condição (ou semáforo) para simular este problema.
  - Argumentos:
    - Quantidade de barbeiros, quantidade de cadeiras de espera, tempo de corte de cabelo (segundos), intervalo entre chegada de clientes (segundos).
  - Utilize uma *thread* para cada barbeiro.

Veja em execução:
https://youtu.be/-IL3vELlEXQ

Exercício

- Exemplos de saída esperada:

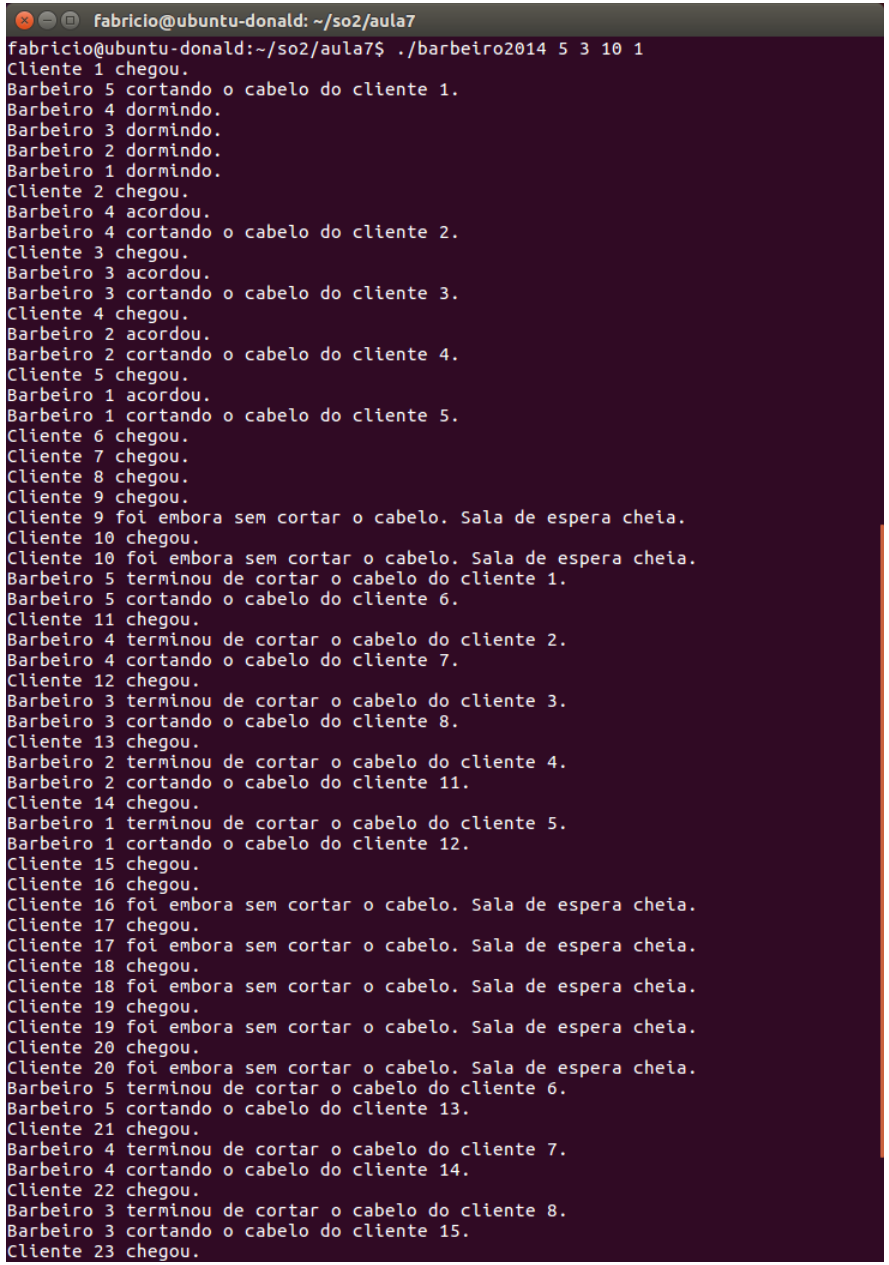
```
fabricio@fabricio-virtual-machine:~/so2/aula7$ ./barbeiro2014 5 3 10 1
Cliente 1 chegou.
Barbeiro 5 cortando o cabelo do cliente 1.
Barbeiro 4 dormindo.
Barbeiro 3 dormindo.
Barbeiro 2 dormindo.
Barbeiro 1 dormindo.
Cliente 2 chegou.
Barbeiro 4 acordou.
Barbeiro 4 cortando o cabelo do cliente 2.
Cliente 3 chegou.
Barbeiro 3 acordou.
Barbeiro 3 cortando o cabelo do cliente 3.
Cliente 4 chegou.
Barbeiro 2 acordou.
Barbeiro 2 cortando o cabelo do cliente 4.
Cliente 5 chegou.
Barbeiro 1 acordou.
Barbeiro 1 cortando o cabelo do cliente 5.
Cliente 6 chegou.
Cliente 7 chegou.
Cliente 8 chegou.
Cliente 9 chegou.
Cliente 9 foi embora sem cortar o cabelo. Sala de espera cheia.
Cliente 10 chegou.
Cliente 10 foi embora sem cortar o cabelo. Sala de espera cheia.
Barbeiro 5 terminou de cortar o cabelo do cliente 1.
```

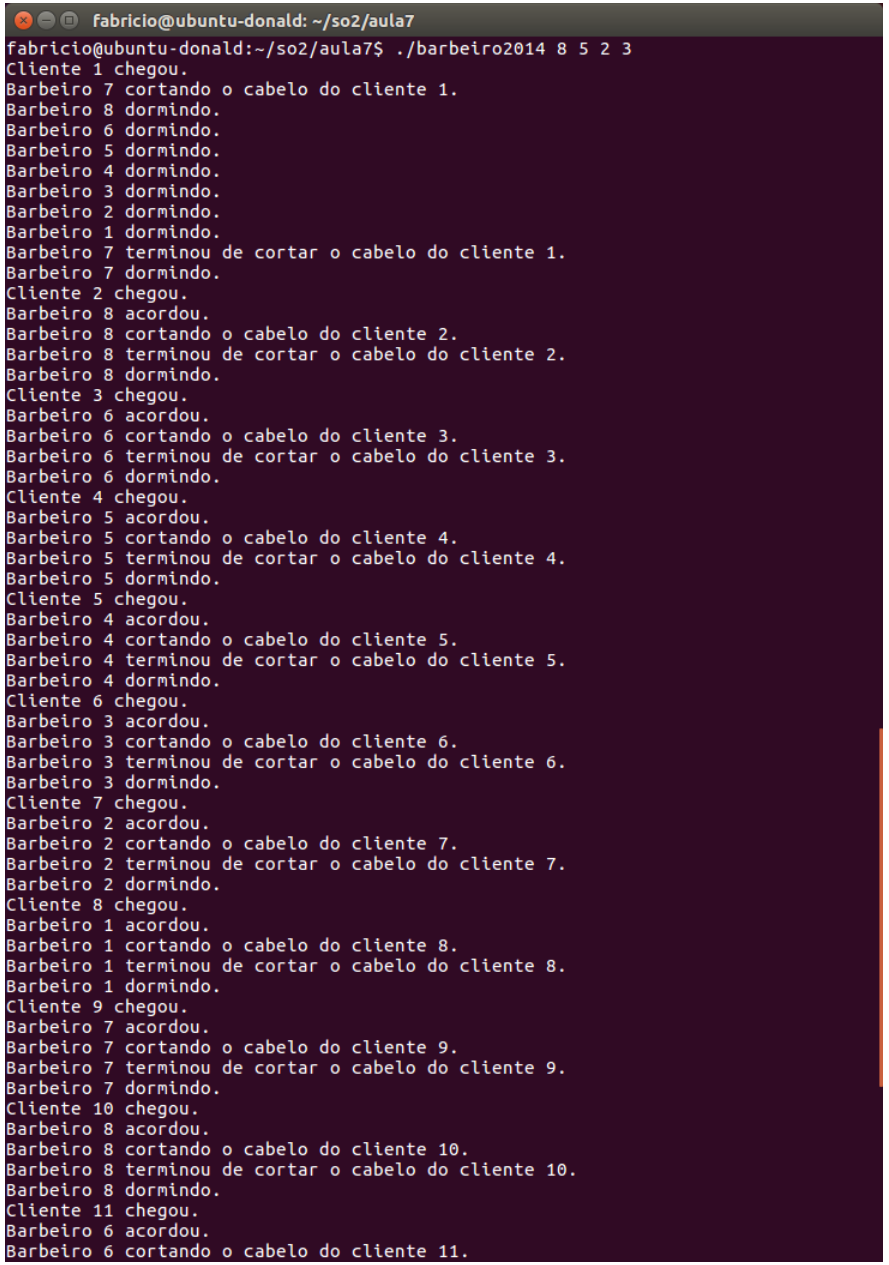
```
Barbeiro 5 cortando o cabelo do cliente 6.
Cliente 11 chegou.
Barbeiro 4 terminou de cortar o cabelo do cliente 2.
Barbeiro 4 cortando o cabelo do cliente 7.
Cliente 12 chegou.
Barbeiro 3 terminou de cortar o cabelo do cliente 3.
Barbeiro 3 cortando o cabelo do cliente 8.
Cliente 13 chegou.
Barbeiro 2 terminou de cortar o cabelo do cliente 4.
Barbeiro 2 cortando o cabelo do cliente 11.
Cliente 14 chegou.
Barbeiro 1 terminou de cortar o cabelo do cliente 5.
Barbeiro 1 cortando o cabelo do cliente 12.
Cliente 15 chegou.
Cliente 16 chegou.
Cliente 16 foi embora sem cortar o cabelo. Sala de espera cheia.
Cliente 17 chegou.
Cliente 17 foi embora sem cortar o cabelo. Sala de espera cheia.
Cliente 18 chegou.
Cliente 18 foi embora sem cortar o cabelo. Sala de espera cheia.
Cliente 19 chegou.
Cliente 19 foi embora sem cortar o cabelo. Sala de espera cheia.
Cliente 20 chegou.
Cliente 20 foi embora sem cortar o cabelo. Sala de espera cheia.
Barbeiro 5 terminou de cortar o cabelo do cliente 6.
Barbeiro 5 cortando o cabelo do cliente 13.
Cliente 21 chegou.
...
```

Exercício

- Exemplos de saída esperada:

```
fabricio@fabricio-virtual-machine:~/so2/aula7$ ./barbeiro2014 8 5 2 3
Cliente 1 chegou.
Barbeiro 7 cortando o cabelo do cliente 1.
Barbeiro 8 dormindo.
Barbeiro 6 dormindo.
Barbeiro 5 dormindo.
Barbeiro 4 dormindo.
Barbeiro 3 dormindo.
Barbeiro 2 dormindo.
Barbeiro 1 dormindo.
Barbeiro 7 terminou de cortar o cabelo do cliente 1.
Barbeiro 7 dormindo.
Cliente 2 chegou.
Barbeiro 8 acordou.
Barbeiro 8 cortando o cabelo do cliente 2.
Barbeiro 8 terminou de cortar o cabelo do cliente 2.
Barbeiro 8 dormindo.
Cliente 3 chegou.
Barbeiro 6 acordou.
Barbeiro 6 cortando o cabelo do cliente 3.
Barbeiro 6 terminou de cortar o cabelo do cliente 3.
Barbeiro 6 dormindo.
Cliente 4 chegou.
Barbeiro 5 acordou.
Barbeiro 5 cortando o cabelo do cliente 4.
Barbeiro 5 terminou de cortar o cabelo do cliente 4.
Barbeiro 5 dormindo.
Cliente 5 chegou.
Barbeiro 4 acordou.
Barbeiro 4 cortando o cabelo do cliente 5.
Barbeiro 4 terminou de cortar o cabelo do cliente 5.
Barbeiro 4 dormindo.
Cliente 6 chegou.
Barbeiro 3 acordou.
Barbeiro 3 cortando o cabelo do cliente 6.
Barbeiro 3 terminou de cortar o cabelo do cliente 6.
Barbeiro 3 dormindo.
Cliente 7 chegou.
Barbeiro 2 acordou.
Barbeiro 2 cortando o cabelo do cliente 7.
Barbeiro 2 terminou de cortar o cabelo do cliente 7.
Barbeiro 2 dormindo.
Cliente 8 chegou.
Barbeiro 1 acordou.
Barbeiro 1 cortando o cabelo do cliente 8.
Barbeiro 1 terminou de cortar o cabelo do cliente 8.
Barbeiro 1 dormindo.
Cliente 9 chegou.
Barbeiro 7 acordou.
Barbeiro 7 cortando o cabelo do cliente 9.
Barbeiro 7 terminou de cortar o cabelo do cliente 9.
Barbeiro 7 dormindo.
Cliente 10 chegou.
Barbeiro 8 acordou.
Barbeiro 8 cortando o cabelo do cliente 10.
Barbeiro 8 terminou de cortar o cabelo do cliente 10.
Barbeiro 8 dormindo.
Cliente 11 chegou.
Barbeiro 6 acordou.
Barbeiro 6 cortando o cabelo do cliente 11.
Barbeiro 6 terminou de cortar o cabelo do cliente 11.
Barbeiro 6 dormindo.
Cliente 12 chegou.
Barbeiro 5 acordou.
Barbeiro 5 cortando o cabelo do cliente 12.
Barbeiro 5 terminou de cortar o cabelo do cliente 12.
Barbeiro 5 dormindo.
Cliente 13 chegou.
Barbeiro 4 acordou.
Barbeiro 4 cortando o cabelo do cliente 13.
Barbeiro 4 terminou de cortar o cabelo do cliente 13.
Barbeiro 4 dormindo.
Cliente 14 chegou.
Barbeiro 3 acordou.
Barbeiro 3 cortando o cabelo do cliente 14.
Barbeiro 3 terminou de cortar o cabelo do cliente 14.
Barbeiro 3 dormindo.
Cliente 15 chegou.
Barbeiro 2 acordou.
Barbeiro 2 cortando o cabelo do cliente 15.
Barbeiro 2 terminou de cortar o cabelo do cliente 15.
Barbeiro 2 dormindo.
Cliente 16 chegou.
Barbeiro 1 acordou.
Barbeiro 1 cortando o cabelo do cliente 16.
Barbeiro 1 terminou de cortar o cabelo do cliente 16.
Barbeiro 1 dormindo.
Cliente 17 chegou.
Barbeiro 7 acordou.
Barbeiro 7 cortando o cabelo do cliente 17.
Barbeiro 7 terminou de cortar o cabelo do cliente 17.
Barbeiro 7 dormindo.
Cliente 18 chegou.
Barbeiro 8 acordou.
Barbeiro 8 cortando o cabelo do cliente 18.
Barbeiro 8 terminou de cortar o cabelo do cliente 18.
Barbeiro 8 dormindo.
Cliente 19 chegou.
Barbeiro 6 acordou.
Barbeiro 6 cortando o cabelo do cliente 19.
Barbeiro 6 terminou de cortar o cabelo do cliente 19.
Barbeiro 6 dormindo.
Cliente 20 chegou.
Barbeiro 5 acordou.
Barbeiro 5 cortando o cabelo do cliente 20.
...
```


fabricio@ubuntu-donald: ~/so2/aula7
fabricio@ubuntu-donald:~/so2/aula7$ ./barbeiro2014 5 3 10 1
Cliente 1 chegou.
Barbeiro 5 cortando o cabelo do cliente 1.
Barbeiro 4 dormindo.
Barbeiro 3 dormindo.
Barbeiro 2 dormindo.
Barbeiro 1 dormindo.
Cliente 2 chegou.
Barbeiro 4 acordou.
Barbeiro 4 cortando o cabelo do cliente 2.
Cliente 3 chegou.
Barbeiro 3 acordou.
Barbeiro 3 cortando o cabelo do cliente 3.
Cliente 4 chegou.
Barbeiro 2 acordou.
Barbeiro 2 cortando o cabelo do cliente 4.
Cliente 5 chegou.
Barbeiro 1 acordou.
Barbeiro 1 cortando o cabelo do cliente 5.
Cliente 6 chegou.
Cliente 7 chegou.
Cliente 8 chegou.
Cliente 9 chegou.
Cliente 9 foi embora sem cortar o cabelo. Sala de espera cheia.
Cliente 10 chegou.
Cliente 10 foi embora sem cortar o cabelo. Sala de espera cheia.
Barbeiro 5 terminou de cortar o cabelo do cliente 1.
Barbeiro 5 cortando o cabelo do cliente 6.
Cliente 11 chegou.
Barbeiro 4 terminou de cortar o cabelo do cliente 2.
Barbeiro 4 cortando o cabelo do cliente 7.
Cliente 12 chegou.
Barbeiro 3 terminou de cortar o cabelo do cliente 3.
Barbeiro 3 cortando o cabelo do cliente 8.
Cliente 13 chegou.
Barbeiro 2 terminou de cortar o cabelo do cliente 4.
Barbeiro 2 cortando o cabelo do cliente 11.
Cliente 14 chegou.
Barbeiro 1 terminou de cortar o cabelo do cliente 5.
Barbeiro 1 cortando o cabelo do cliente 12.
Cliente 15 chegou.
Cliente 16 chegou.
Cliente 16 foi embora sem cortar o cabelo. Sala de espera cheia.
Cliente 17 chegou.
Cliente 17 foi embora sem cortar o cabelo. Sala de espera cheia.
Cliente 18 chegou.
Cliente 18 foi embora sem cortar o cabelo. Sala de espera cheia.
Cliente 19 chegou.
Cliente 19 foi embora sem cortar o cabelo. Sala de espera cheia.
Cliente 20 chegou.
Cliente 20 foi embora sem cortar o cabelo. Sala de espera cheia.
Barbeiro 5 terminou de cortar o cabelo do cliente 6.
Barbeiro 5 cortando o cabelo do cliente 13.
Cliente 21 chegou.
Barbeiro 4 terminou de cortar o cabelo do cliente 7.
Barbeiro 4 cortando o cabelo do cliente 14.
Cliente 22 chegou.
Barbeiro 3 terminou de cortar o cabelo do cliente 8.
Barbeiro 3 cortando o cabelo do cliente 15.
Cliente 23 chegou.


fabricio@ubuntu-donald: ~/so2/aula7
fabricio@ubuntu-donald:~/so2/aula7$ ./barbeiro2014 8 5 2 3
Cliente 1 chegou.
Barbeiro 7 cortando o cabelo do cliente 1.
Barbeiro 8 dormindo.
Barbeiro 6 dormindo.
Barbeiro 5 dormindo.
Barbeiro 4 dormindo.
Barbeiro 3 dormindo.
Barbeiro 2 dormindo.
Barbeiro 1 dormindo.
Barbeiro 7 terminou de cortar o cabelo do cliente 1.
Barbeiro 7 dormindo.
Cliente 2 chegou.
Barbeiro 8 acordou.
Barbeiro 8 cortando o cabelo do cliente 2.
Barbeiro 8 terminou de cortar o cabelo do cliente 2.
Barbeiro 8 dormindo.
Cliente 3 chegou.
Barbeiro 6 acordou.
Barbeiro 6 cortando o cabelo do cliente 3.
Barbeiro 6 terminou de cortar o cabelo do cliente 3.
Barbeiro 6 dormindo.
Cliente 4 chegou.
Barbeiro 5 acordou.
Barbeiro 5 cortando o cabelo do cliente 4.
Barbeiro 5 terminou de cortar o cabelo do cliente 4.
Barbeiro 5 dormindo.
Cliente 5 chegou.
Barbeiro 4 acordou.
Barbeiro 4 cortando o cabelo do cliente 5.
Barbeiro 4 terminou de cortar o cabelo do cliente 5.
Barbeiro 4 dormindo.
Cliente 6 chegou.
Barbeiro 3 acordou.
Barbeiro 3 cortando o cabelo do cliente 6.
Barbeiro 3 terminou de cortar o cabelo do cliente 6.
Barbeiro 3 dormindo.
Cliente 7 chegou.
Barbeiro 2 acordou.
Barbeiro 2 cortando o cabelo do cliente 7.
Barbeiro 2 terminou de cortar o cabelo do cliente 7.
Barbeiro 2 dormindo.
Cliente 8 chegou.
Barbeiro 1 acordou.
Barbeiro 1 cortando o cabelo do cliente 8.
Barbeiro 1 terminou de cortar o cabelo do cliente 8.
Barbeiro 1 dormindo.
Cliente 9 chegou.
Barbeiro 7 acordou.
Barbeiro 7 cortando o cabelo do cliente 9.
Barbeiro 7 terminou de cortar o cabelo do cliente 9.
Barbeiro 7 dormindo.
Cliente 10 chegou.
Barbeiro 8 acordou.
Barbeiro 8 cortando o cabelo do cliente 10.
Barbeiro 8 terminou de cortar o cabelo do cliente 10.
Barbeiro 8 dormindo.
Cliente 11 chegou.
Barbeiro 6 acordou.
Barbeiro 6 cortando o cabelo do cliente 11.

Referências Bibliográficas

1. NEMETH, Evi.; SNYDER, Garth; HEIN, Trent R.;. *Manual Completo do Linux: Guia do Administrador*. São Paulo: Pearson Prentice Hall, 2007. Cap. 4.
2. DEITEL, H. M.; DEITEL, P. J.; CHOFFNES, D. R.; *Sistemas Operacionais: terceira edição*. São Paulo: Pearson Prentice Hall, 2005. Cap. 20.
3. MITCHELL, Mark; OLDHAM, Jeffrey; SAMUEL, Alex; *Advanced Linux Programming*. New Riders Publishing: 2001. Cap. 4.
4. TANENBAUM, Andrew S.; *Sistemas Operacionais Modernos.* 3ed. São Paulo: Pearson Prentice Hall, 2010. Cap. 10.