Sistemas Operacionais

4º período

Professora: Michelle Hanne

PUC Minas

Semáforos

Diferença entre Semáforo e Mutex

PUC Minas

Considere o problema padrão do **produtor-consumidor.** Suponha que temos um buffer de comprimento de **4.096 bytes**. Um encadeamento de **produtor coleta os dados e os grava no buffer.** Um encadeamento do consumidor **processa os dados coletados do buffer.** O objetivo é que ***os dois threads não sejam executados ao mesmo tempo.***

Fonte: https://acervolima.com/mutex-vs-semaphore/

Usando Mutex

PUC Minas

Um mutex fornece exclusão mútua, tanto o produtor quanto o consumidor podem ter a chave (mutex) e prosseguir com seu trabalho. Enquanto o buffer for preenchido pelo produtor, o consumidor precisará esperar e vice-versa.

Em qualquer ponto do tempo, apenas um thread pode funcionar com *todo o* buffer. O conceito pode ser generalizado usando semáforo.

Fonte: https://acervolima.com/mutex-vs-semaphore/

Usando Semáforo

PUC Minas

Um semáforo é um mutex generalizado. Em vez de um único buffer, podemos dividir o buffer de 4 KB em quatro buffers de 1 KB (recursos idênticos). Um semáforo pode ser associado a esses quatro buffers. O consumidor e o produtor podem trabalhar em buffers diferentes ao mesmo tempo.

Fonte: https://acervolima.com/mutex-vs-semaphore/

Diferença entre Semáforo e Mutex

PUC Minas

Um mutex é um **mecanismo de bloqueio** usado para sincronizar o acesso a um recurso.

O semáforo é **um mecanismo de sinalização** (tipo de sinal “Pronto, você pode continuar”)

Fonte: https://acervolima.com/mutex-vs-semaphore/

Diferença entre Semáforo e Mutex

PUC Minas

O semáforo e o mutex devido a semelhança na implementação podem ter o conceito parecido, mas são diferentes.

O **mutex** é um **mecanismo de bloqueio** utilizado para sincronizar o acesso a um recurso, ou seja, apenas uma tarefa pode adquirir o recurso. Já o **semáforo** é um **mecanismo** de **sinalização** para **interrupções** de tarefas ocorridas no sistema.

Diferença entre Semáforo e Mutex

PUC Minas

Imagine dois usuários enviando arquivos para uma impressora, mas **somente um processo terá o direito de impressão (mutex)**, enquanto os demais ficam à espera de aquisição do recurso e impressão.

Já em uma outra situação, suponha que esteja ouvindo música no celular e receba uma ligação. Neste caso, será **sinalizado uma rotina de interrupção para que possa atender a ligação (semáforo).**

Semáforo de Contagem

PUC Minas

Um semáforo de contagem é um objeto de sincronização utilizado para implementar a concorrência limitada.

Esses semáforos não precisam ser adquiridos e liberados pelo mesmo encadeamento, eles podem ser usados para notificação de **eventos assíncronos** (como em manipuladores de sinais).

Semáforo de Contagem

PUC Minas

Esse semáforo de contagem contém um valor e suporta duas operações: ***wait e post***.

Post *incrementa o semáforo e retorna imediatamente;*

Wait irá esperar se a contagem for zero. Se a contagem for diferente de zero, o semáforo diminui a contagem e retorna imediatamente.

PUC Minas

Monitores

Monitores

PUC Minas

Semáforos: solução elegante e eficiente
Mas... necessita de programação cuidadosa e interfere diretamente na lógica do problema

Propostas de Hansen(1973) e Hoare(1974)

- Solução de alto nível de abstração, provida pela linguagem
- Agrupamento unidades do programa (regiões críticas)
- Dados e/ou métodos
- Somente um processo pode estar ativo em um monitor em um instante de tempo

Monitores

PUC Minas

Esse conceito foi proposto para resolver o **problema de sincronismo de comunicação** entre processos de alto nível, ou seja, **um conjunto de rotinas, variáveis e estrutura de dados definido dentro de um módulo de programação que tem o objetivo de garantir a exclusão mútua entre seus procedimentos.**

Em um determinado tempo**, somente um processo pode executar um procedimento do monitor.** Quando um **processo faz a chamada de um desses procedimentos, o monitor verifica se há outro processo executando o procedimento. Caso exista, o procedimento fica aguardando até obter permissão para execução.**

Esquema geral de Hansen (1973)

- Variáveis de condição para controlar a espera
  - **Dois sinais: wait e signal**
  - Esperar por uma condição específica: **condition x**;
  - Esperar por uma condição: **x.wait();**
    - Processo espera até ser sinalizado
  - Sinalização: **x.signal();**
    - Se não há processos esperando, não tem efeito
    - Caso contrário, acorda um único processo.

Monitores

PUC Minas

Monitores: produtor e consumidor

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Monitores

PUC Minas

Monitores: produtor e consumidor

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Monitores

PUC Minas

Monitores: produtor e consumidor

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Monitores

PUC Minas

Monitores: produtor e consumidor

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Monitores

PUC Minas

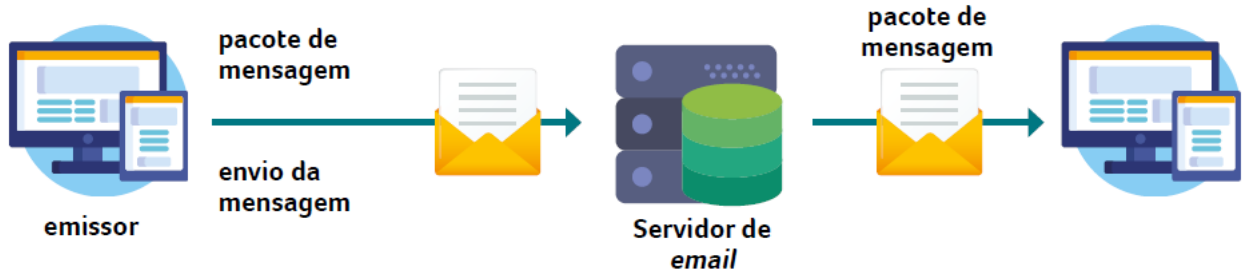
Em resumo: exclusão mútua “automática”

- Solução simples
- Menos probabilidades de erro
- MAS dependem da linguagem de programação
  - Java, C#, ObjectPascal apresentam soluções próprias aproximadas ao conceito de monitor

Troca de Mensagens

PUC Minas

O sistema operacional implementa duas rotinas de sistemas **SEND** e **RECEIVE** para solucionar problemas de sincronismo e comunicação entre processos.
O mecanismo faz a sincronização porque uma mensagem somente pode ser lida após ser enviada, **e restringe a ordem que dois eventos podem ocorrer.** Em um sistema de rede, mensagens podem ser perdidas causando um problema no sincronismo.
Essa comunicação pode ser síncrona (direto) e assíncrona (indireto). O e-mail exemplifica uma forma de troca de mensagens indireta.



Os problemas são os mais variados possíveis devido à complexidade dos sistemas operacionais.

- Cenários que modelam problemas recorrentes em sistemas operacionais e outros sistemas de software
- Metáforas para componentes e recursos do sistema

Problemas clássicos

PUC Minas

É comum haver fontes de dados que podem ser consultadas por determinados processos e alteradas por outros

Ex: lançamento de notas no Canvas

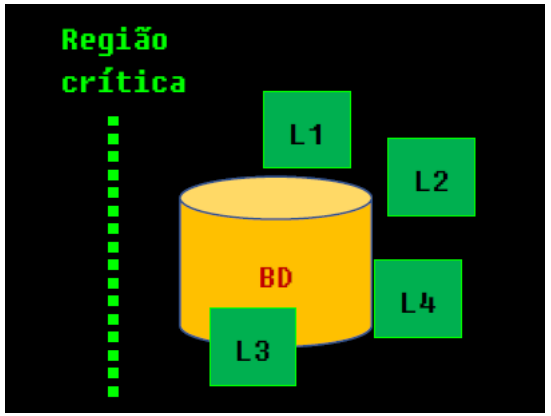
- Processos leitores - É aceitável múltiplas leituras simultâneas
- Processos escritores - A escrita é uma operação exclusiva

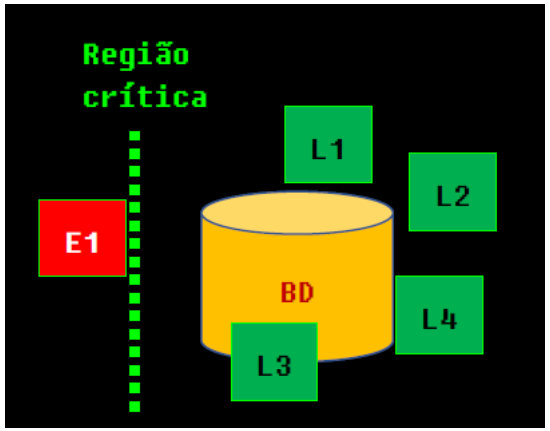
Problemas clássicos

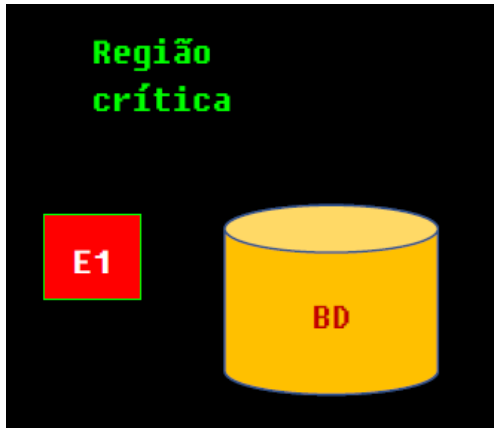
PUC Minas

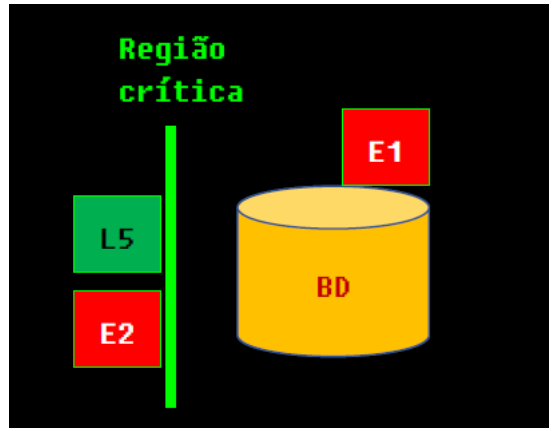
Premissa para solução básica:

- Leitores novos podem entrar se já há um ou mais leitores
- Escritores só entram se os dados estiverem totalmente liberados









Solução com semáforos:

- Um semáforo para controlar o acesso ao banco de dados
- Um mutex para a região crítica (contador de leitores)

Problemas clássicos

PUC Minas

Leitores, escritores e semáforos

```
Semaforo mutex = new Semaforo(1);
Semaforo bd = new Semaforo(1);
int qtLeitores = 0

escritor(){
    produz_dados();
    down(bd);
        escreve_dados();
    up(bd);
}
```

```
leitor(){
    down(mutex);
        qtLeitores++;
        if(qtLeitores==1) down(bd);
    up(mutex);
    ler_dados();
    down(mutex);
        qtLeitores--;
        if(qtLeitores==0) up(bd);
    up(mutex);
}
```

Esta solução funciona, porém....

- Starvation de escritores

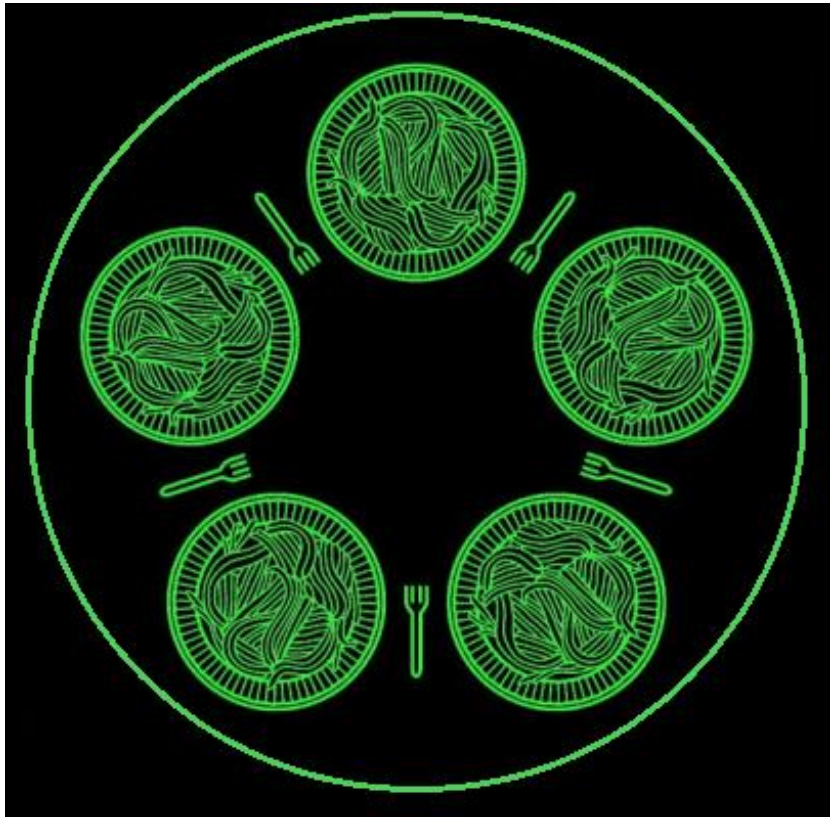
Proposta de solução:

- Dar prioridade aos escritores
- Semáforo adicional de fila de espera
- Diminui concorrência e garante funcionamento de escritores
- Escritores costumam ser menos frequentes que leitores

Jantar com os Filósofos

PUC Minas

Proposto por Dijkstra, 1965, esse algoritmo resolveu o problema de sincronização de processos em sistema com número limitado de recursos, como dispositivo de I/O. O problema consiste em:



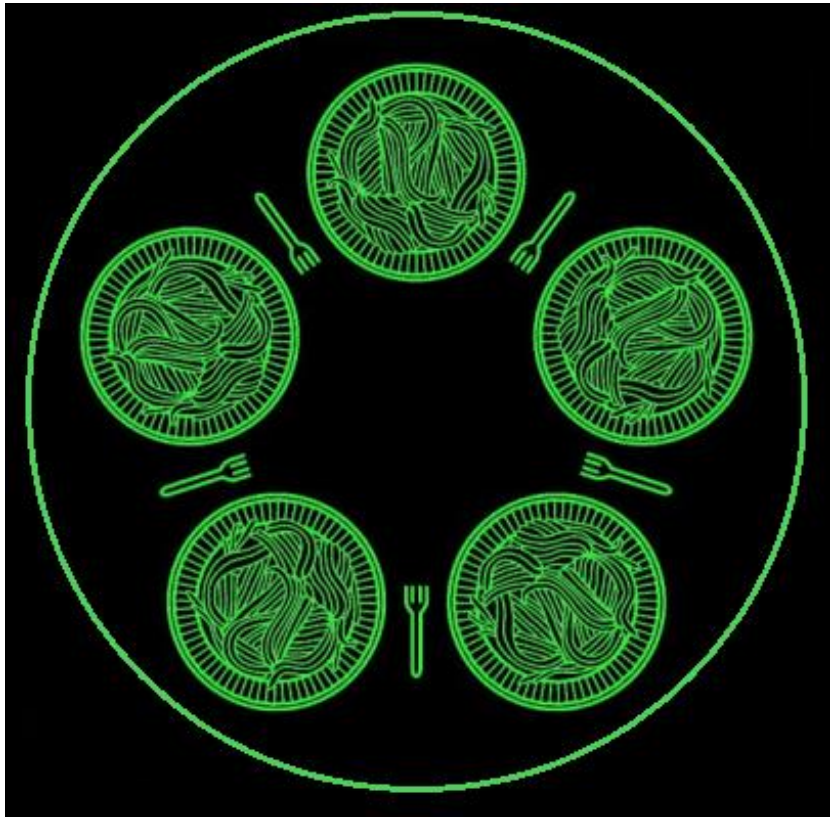
- **Cinco filósofos estão sentados em torno de uma mesa circular. Cada filósofo tem um prato de espaguete. O espaguete está tão escorregadio que um filósofo precisa de dois garfos para comê-lo. Entre cada par de pratos está um garfo.**

Jantar com os Filósofos

PUC Minas

A vida do filósofo consiste em **alternar períodos de comer e pensar**. Quando o filósofo fica **com fome ele tenta pegar os garfos da direita e da esquerda**, um por vez, em qualquer ordem. **Se conseguir acesso aos garfos, terá um determinado tempo para comer,** e **posteriormente votará a pensar, liberando os garfos**.

O algoritmo tem que ser eficiente para que todos os filósofos sejam atendidos, ou seja, comerem todo o espaguete.



Jantar com os Filósofos

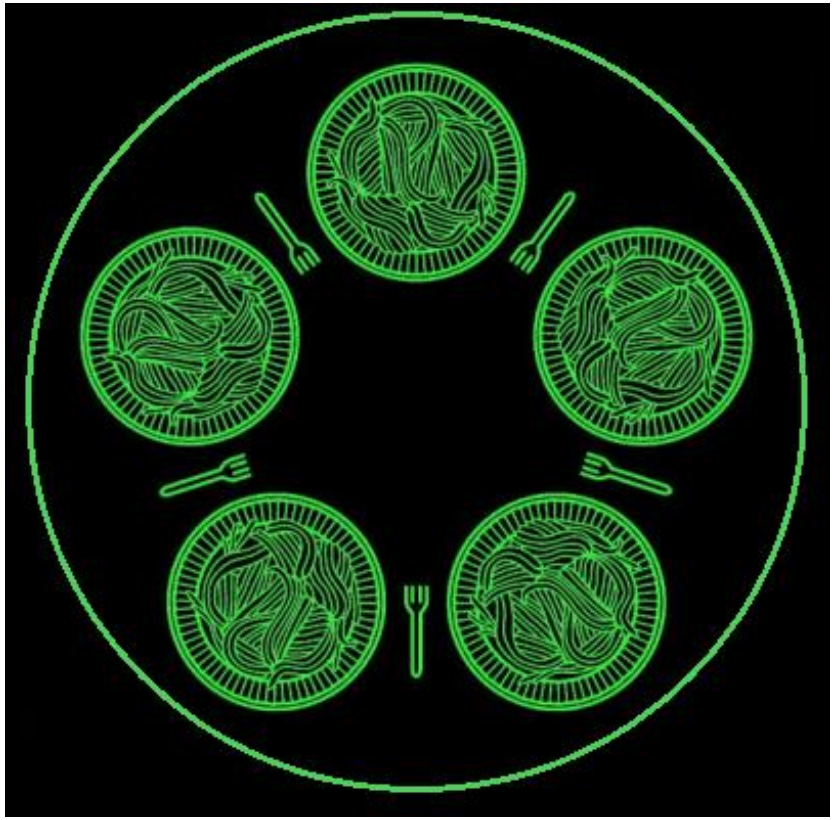
PUC Minas

Solução mais óbvia:

- tentar pegar um garfo de cada vez sendo bloqueado até conseguir
- caso não consiga, devolva os garfos

ERRO:

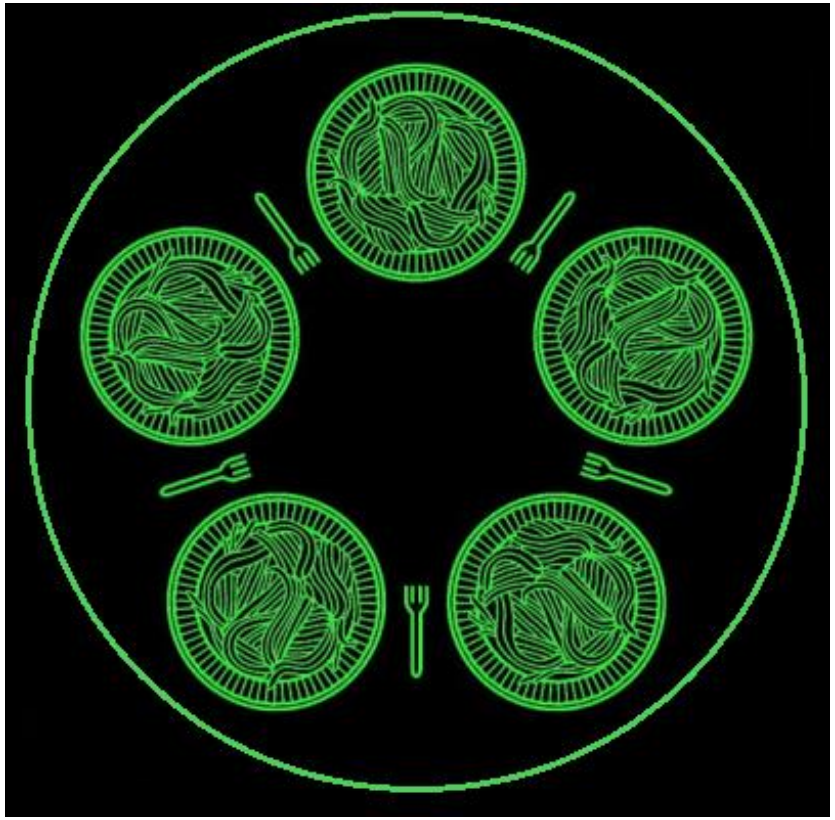
- tentativas simultâneas podem levar a travamento - Starvation



Jantar com os Filósofos

PUC Minas

O problema foi resolvido utilizando **um arranjo de semáforo**, pois assim **um filósofo ao comer, bloqueia outro filósofo**, e **ao término do tempo,** o outro filósofo tem acesso ao garfo e come, assim sucessivamente. **Esse algoritmo permite não ter starvation ou deadlock.**



Jantar com os Filósofos

Algoritmo com uma solução para o problema jantar dos filósofos.

Uma matriz state, é usada para controlar se um filósofo está comendo, pensando ou faminto tentando pegar os garfos. Um filósofo só pode mudar para o estado “comendo” se nenhum dos vizinhos estiver comendo. Os vizinhos do filósofo “i” são definidos pelas macros LEFT e RIGHT. Se “i” for 2, LEFT será 1 e RIGHT será 3. Uma matriz de semáforos, um por filósofo.
Assim, filósofos famintos podem ser bloqueados se os garfos necessários estiverem ocupados.
Observe que cada processo executa o procedimento *philosopher* como seu código principal,
mas outros procedimentos – *take, forks, put_forks e test*, são procedimentos ordinários, e não processos separados.

```
#define N
#define LEFT
#define RIGHT
#define THINKING
#define HUNGRY
#define EATING
typedf int semaphore;
int state[N];
void philosopher(int i)
{       while (true){
        think();
        take_forks(i);
        eat();
        put_forks(i);} }
void take_forks(int i)
{       down(@mutex);
        state[i]= HUNGRY;
        test(i);
        up(&mutex);
        down(@s[i]);}
void put_forks(i){
        down(@mutex);
        state[i]= THINKING;
        test(LEFT);
        test(RIGHT);
        UP(&mutex);}
void test(i){
        if (state[i] == HUNGRY && state[i] == EATING &&
state[RIGHT] != EATING){
        state[i] = EATING;
        up(&s[i]);       }}
```

PUC Minas

Barbeiro Sonolento

PUC Minas

Em ciência da computação, outro problema clássico de IPC (*Inter-Process Communication*) é denominado o barbeiro sonolento (SILBERSCHATZ,2015). O problema faz uma analogia a manter o barbeiro ocupado enquanto há clientes, e descansando quando não há nenhum (ordenado).

O barbeiro e seus clientes correspondem a CPU e os processos. O problema consiste em:

Na barbearia há um barbeiro, uma cadeira de barbeiro e "n" cadeiras para eventuais clientes esperarem a vez.
Quando não há clientes, o barbeiro senta-se na cadeira de barbeiro e cai no sono. Quando chega um cliente, ele precisa acordar o barbeiro. Se outros clientes chegarem enquanto o barbeiro estiver cortando o cabelo de um cliente, eles se sentarão (se houver cadeiras vazias) ou sairão da barbearia (se todas as cadeiras estiverem ocupadas) (TANENBAUM, 2016).

Barbeiro Sonolento

PUC Minas

A solução proposta utiliza semáforos: customers, que conta os clientes à espera de atendimento (exceto o cliente que está na cadeira de barbeiro, que não está esperando); **barbers**, o número de barbeiros (0 ou 1) que estão ociosos à espera de clientes, e mutex, que é usado para exclusão mútua (TANENBAUM, 2016).

Esse problema é semelhante ao problema de múltiplas filas. Pode ser exemplificado por um sistema de atendimento de telemarketing com diversos atendentes e com um sistema computadorizado de chamadas em espera, atendendo a um número limitado de chamadas que chegam (TANENBAUM, 2016).

Barbeiro Sonolento

Pela manhã, o babeiro executa o procedimento **barber**, que o leva a **bloquear sobre o semáforo customers**, que inicialmente está em 0. O barbeiro então vai dormir, e permanece dormindo até que o primeiro cliente apareça.

Quando chega, o **cliente executa customers** e inicia obtendo o **mutex para entrar em uma região crítica**. Se um outro cliente chega, sem seguida, o segundo nada pode fazer até que o **primeiro libere o mutex**. O cliente então verifica **se o número de clientes a espera é menor que o número de cadeiras**. Caso sim, **ele liberará o mutex e sairá sem cortar o cabelo.**
Se houver uma cadeira disponível, o cliente incrementará a variável inteira waiting. Ele faz um **up** no semáforo **customers**, que acorda o barbeiro. **O cliente e o barbeiro estão ambos acordados.**
Quando o cliente libera o mutex, o barbeiro o pega, faz alguma limpeza e começa a cortar o cabelo.
Quando termina o corte, o cliente sai do procedimento e deixa a barbearia. Cada cliente fará somente um corte de cabelo. O barbeiro contém um laço para tentar obter o próximo cliente. Caso houver outro, um novo corte de cabelo será iniciado, caso contrário, o barbeiro cairá no sono.

```
#define CHAIRS 5
typedf int semaphore;
semaphore customers=0;
semaphore barbers=0;
semaphore mutex= 1;
int waiting =0;
void barber(void)
{
                    while (true){
                    down(&customers);
                    down(&mutex);
                    waiting = waiting -1;
                    up(&barbes);
                    up(&mutex);
                    cut_hair();
                    }
}
void customers(void)
{           down(&mutex);
            if(waiting < CHAIRS){
            waiting = waiting +1;
            up(&customers);
            up(&mutex);
            down(&barbers);
            get_haircut();
             }
             else {
             up(&mutex);
             }
}
```

PUC Minas