

# Aula 11 – Comunicação Interprocessos e Problemas Clássicos de Comunicação entre Processos

Norton Trevisan Roman  
Clodoaldo Aparecido de Moraes Lima

11 de outubro de 2014

# Mensagens – Outros Mecanismos

- Existem diversos mecanismos de comunicação atuais baseados na troca de mensagens (além do envio direto ao processo). Os mais representativos são:
  - Caixas Postais (Mailboxes)
  - Portos ou Portas (Ports)
- Caixas Postais
  - Estrutura de dados: São filas de mensagens não associadas, a princípio, com nenhum processo.

# Mensagens – Outros Mecanismos

- Caixas Postais

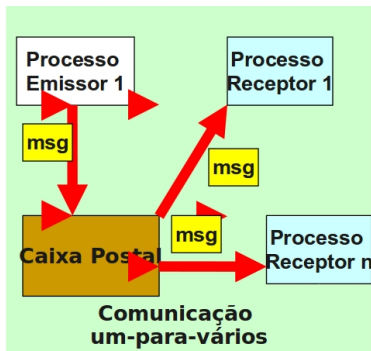
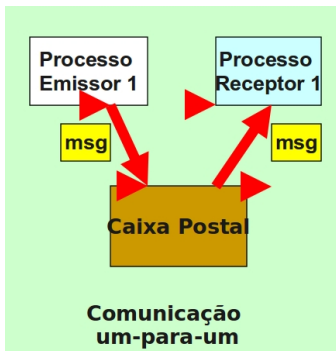
- Lugar para se colocar temporariamente um certo número de mensagens
  - Mensagens são enviadas para ou lidas da caixa postal, e não diretamente para os/dos processos
  - Quando um processo tenta enviar para uma caixa cheia, ele é suspenso até que uma mensagem seja removida da caixa
- Ex: Produtor-consumidor
  - Tanto o produtor quanto o consumidor criam caixas postais grandes o suficiente para conter N mensagens
  - O produtor envia mensagens com dados à caixa postal do consumidor
  - O consumidor envia mensagens vazias à caixa do produtor

# Mensagens – Outros Mecanismos

- Caixas Postais
  - Os processos podem enviar e receber mensagens das caixas postais
    - Mecanismo adequado para comunicar diversos remetentes a diversos receptores
    - Cada processo pode ter uma caixa exclusiva
  - Restrição
    - Necessidade de envio de duas mensagens para comunicar o remetente com o receptor: uma do remetente à caixa postal, e outra da caixa postal para o receptor

# Mensagens – Outros Mecanismos

- Caixas Postais



# Mensagens – Outros Mecanismos

- Caixas Postais em Posix:
  - Implementadas como uma fila de mensagens
    - `msgget()` – faz a requisição (ao kernel) de uma fila de mensagens usando o nome da caixa de mensagens passado por parâmetro (ou criando uma, não existir)
    - `msgsnd()` – envia uma mensagem a uma caixa particular
    - `msgrcv()` – obtém uma mensagem de uma determinada caixa
    - Em qualquer caso, se não houver espaço (ou mensagem), o processo é bloqueado – é adicionado à fila de processos em espera associada à caixa.
  - É verificado sempre se o processo pode acessar a fila

# Mensagens – Outros Mecanismos

- Portas

- Computadores, em geral, tem uma única conexão de rede
  - Todo dado chega por esta conexão
- Como o computador sabe a que aplicação enviar os dados que chegam?
  - Usando portas.
  - Os dados transmitidos são acompanhados por um endereço que identifica o computador (IP) e a porta de destino

# Mensagens – Outros Mecanismos

- Portas

- São elementos do sistema (em software) que permitem a comunicação entre conjuntos de processos
  - Conexão de dados virtual (ou lógica), usada para que programas troquem informação diretamente
- Ex: TCP – numerados de 1 a 65.535 (16-bits)
  - 0 – 1.023 são restritos: reservadas para serviços conhecidos, como HTTP e FTP
  - Aplicações de usuários devem usar as demais (embora possam usar essas)
- Cada porta é como uma caixa postal, porém com um dono, que será o processo que o criar

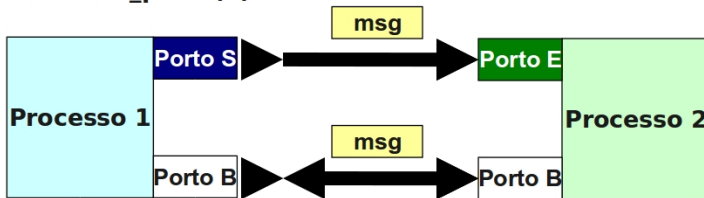


# Mensagens – Outros Mecanismos

- Portas

**Cria\_porto(S, saída, msg)**  
**conecta\_porto(S, E)**  
**envia\_porto(S, msg)**  
**desconecta\_porto(S, E)**  
**destruir\_porto(S)**

**Cria\_porto(E, entrada, msg)**  
**recebe\_porto(E, msg)**  
**destruir\_porto(E)**



# Mensagens – Outros Mecanismos

- Portas: Outras Características
  - A criação e a interligação de portas e caixas postais pode ser feita de maneira dinâmica
    - A necessidade de enfileiramento das mensagens enviadas torna necessário o uso de “buffers”, para o armazenamento intermediário
  - A comunicação entre processos locais ou remotos, em um sistema estruturado com portas, será feita pela execução de primitivas síncronas (ou assíncronas) do tipo envia e recebe

# Outros Mecanismos de Comunicação

- Alguns outros mecanismos para comunicação inter-processos são:
  - Sockets
  - Pipes
  - Sinais (Signals)
  - Barreiras

# Outros Mecanismos de Comunicação

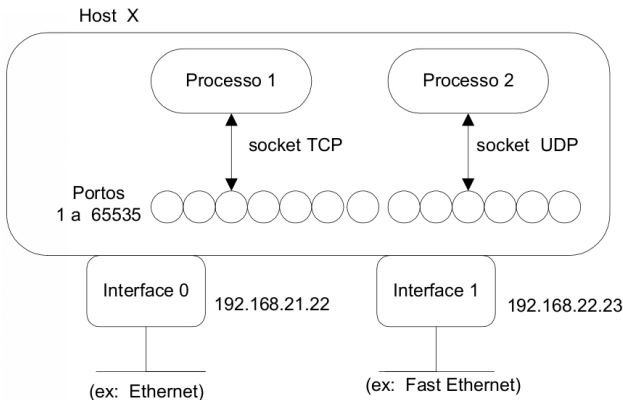
- Sockets

- Par endereço IP e porta utilizado para comunicação entre processos em máquinas diferentes
  - Host X (192.168.1.1:1065) Server Y (192.168.1.2:80)
- Para que possa ocorrer a troca de mensagens, uma aplicação no servidor liga um socket a uma porta específica
  - Registra, no SO, a aplicação, para que esta receba todo dado destinado àquela porta

# Outros Mecanismos de Comunicação

- Sockets

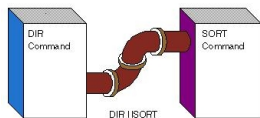
- Em unix, sockets são um tipo de arquivo que conecta processos a suas respectivas portas



# Outros Mecanismos de Comunicação

- Pipes

- Fluxo de bytes de mão única entre processos
- Usados no sistema operacional UNIX para permitir a comunicação entre processos
  - Uma pipe é um modo de conectar a saída de um processo com a entrada de outro processo, sem o uso de arquivos temporários
  - Ao tentar ler uma entrada da pipe, um processo é bloqueado até que o dado esteja disponível
  - Todo dado escrito por um processo na pipe é dirigido, pelo kernel, a outro processo, que pode então lê-lo



# Outros Mecanismos de Comunicação

- Pipes
  - Uma pipeline é uma conexão de dois ou mais programas ou processos através de pipes
    - Permite a criação de filas de processos
    - Ex: `ps -ef | grep alunos | sort`
  - Existe enquanto o processo existir
  - Exemplos:
    - Sem uso de pipes (usando arquivos temporários)  
`$ ls > temp`  
`$ sort < temp`
    - Com uso de pipes  
`$ ls | sort`

# Outros Mecanismos de Comunicação

- Pipes

- Podem ser consideradas arquivos abertos sem imagem correspondente no sistema de arquivos
- Funcionamento:
  - Um processo cria uma nova pipe, via a chamada `pipe()`
  - `pipe()` retorna um par de descritores de arquivo
  - O processo passa esses descritores a seus filhos (via `fork()`)
  - Cada filho pode ler da pipe fazendo um `read()` com o primeiro descritor, e escrever com um `write()` no segundo.



# Outros Mecanismos de Comunicação

## • Pipes – Exemplo

- Usa a chamada ao sistema `dup`
- `int dup(int oldfd);`
  - Cria uma cópia do descritor `oldfd`
  - Usa o descritor sem uso de menor número como novo descritor

```
#include <unistd.h>

void main(void) {
    int aux[2]; /* auxiliar para a criação das pipes */

    /* Cria a pipe, recebendo os dois descritores */
    if (pipe(aux)==-1) exit(1);

    switch (fork()) {
        case -1 : /*erro*/
            exit(1); /*Falha no fork*/
        case 0 : /* processo filho */
            /* fecho o stdin do filho */
            close(0);
            /* abro o descritor 0 (stdin) para a leitura
            da pipe */
            if (dup(aux[0])!=0) exit(1);
            /* fecho a ponta de escrita da pipe */
            close(aux[1]);
            /* executo o programa no filho */
            execlp("./filho","filho",NULL);
        }
    /* se não for -1 nem 0, é o pai */
    /* fechando a ponta de leitura da pipe */
    close(aux[0]);

    /* escrevendo no filho */
    write(aux[1], "Oi", 2);
}
```

# Outros Mecanismos de Comunicação

- Problemas com pipes:
  - Não há como abrir uma pipe já existente
    - Dois processos não compartilham a mesma pipe, a menos que sejam irmãos, e ela tenha sido criada pelo pai desses processos
- Named pipe (ou FIFO)
  - Extensão de pipe que soluciona esse problema
  - É um tipo de arquivo especial em Unix
    - Pode ser aberto por qualquer processo
    - Continua existindo mesmo depois que o processo terminar
    - Criado com chamadas de sistemas

# Outros Mecanismos de Comunicação

- Sinais (Linux e Unix)
  - São interrupções de software (traps), usadas para que um processo possa enviar um sinal a outro processo
  - Pode-se também dizer ao sistema receptor o que deve fazer quando um sinal é recebido:
    - Ignorá-lo
    - Capturá-lo: deve-se definir um procedimento para tratá-lo. Quando o sinal é recebido, o controle passa ao tratador. Quando esse finaliza e retorna, o controle volta para onde estava – como tratamento de interrupções
    - Deixá-lo matar o processo
  - Processos podem enviar sinais apenas a seu grupo de processos (ancestrais, irmãos e descendentes)

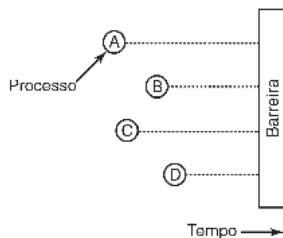
# Outros Mecanismos de Comunicação

- Barreiras
  - Algumas aplicações podem ser divididas em fases
    - Nenhum processo pode avançar à próxima fase até que todos os processos possam fazê-lo
  - Introduz-se uma barreira ao final de cada fase
    - Quando alcança a barreira, um processo fica bloqueado até que todos os outros alcancem a barreira.
    - A barreira garante que as operações colocadas antes dela são terminadas antes de começar as operações localizadas após ela.

# Outros Mecanismos de Comunicação

- Barreiras – funcionamento

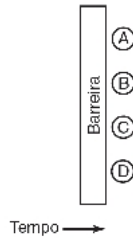
- Ao chegar ao ponto em que deve haver a barreira, o processo executa a primitiva `barrier()` e é suspenso



Processos se aproximando de uma barreira.



Todos os processos, exceto um, estão bloqueados pela barreira.



Quando o último processo chega à barreira, todos passam por ela.

# Outros Mecanismos de Comunicação

- Barreiras – Linux
  - O processo inicializa uma barreira, chamando `pthread_barrier_init()`
    - Passando, dentre outros parâmetros, o número de threads que sincronizarão na barreira
  - Cada thread, ao chegar na barreira, chama `pthread_barrier_wait()`
    - Ela bloqueia, até que o número de threads especificado tenha chamado `wait()` nessa mesma barreira
    - Quando o número requerido chamou `wait()`, elas são desbloqueadas

# Referências Adicionais

- Bovet, D.P., Cesati, M.: Understanding the Linux Kernel. O'Reilly. 1st Ed. 2000.
- <http://macboypro.wordpress.com/2009/05/15/posix-message-passing-in-linux/>
- <http://www.linux-tutorial.info/modules.php?name=MContent&pageid=292>
- [http://linux.die.net/man/3/pthread\\_barrier\\_wait](http://linux.die.net/man/3/pthread_barrier_wait)
- [http://linux.die.net/man/3/pthread\\_barrier\\_init](http://linux.die.net/man/3/pthread_barrier_init)

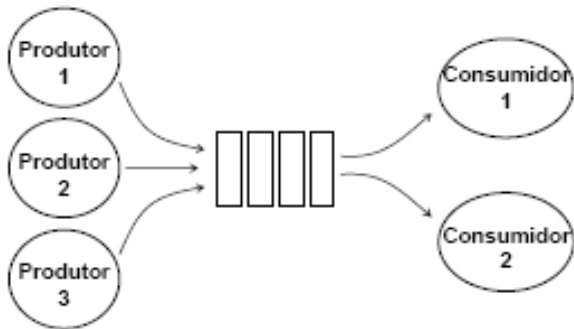
# Problemas Clássicos



# Produtor – Consumidor

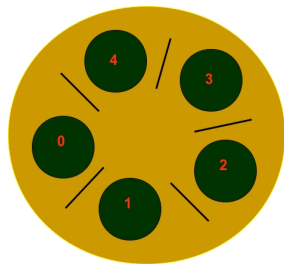
- Um sistema é composto por entidades produtoras e entidades consumidoras
- Entidades produtoras
  - Responsáveis pela produção de itens que são armazenados em um buffer (ou em uma fila)
  - Itens produzidos podem ser consumidos por qualquer consumidor
- Entidades consumidoras
  - Consomem os itens armazenados no buffer (ou na fila)
  - Itens consumidos podem ser de qualquer produtor

# Produtor – Consumidor

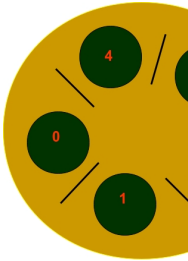


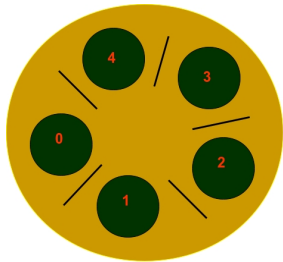
# Jantar dos Filósofos (Dijkstra, 1965)

- Cinco filósofos estão sentados ao redor de uma mesa circular para o jantar.
  - Cada filósofo possui um prato para comer macarrão
  - Além disso, eles dispõem de hashis, em vez de garfos
    - Cada um precisa de 2 hashis
  - Entre cada par de pratos existe apenas um hashi
    - Hashis precisam ser compartilhados de forma sincronizada

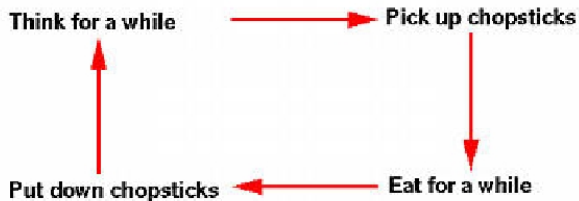


# Jantar dos Filósofos

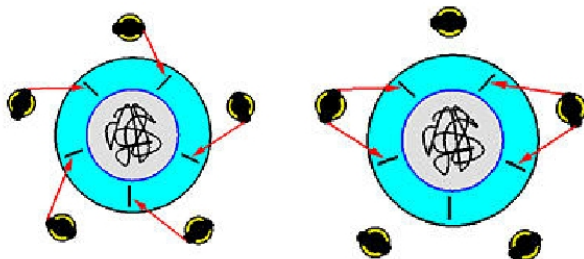
- Os filósofos comem e pensam, alternadamente
    - Não se atém a apenas uma das tarefas
  - Além disso, quando comem, pegam apenas um hashi por vez
    - Quem consegue pegar os dois, come por alguns instantes e depois larga os hashis
  - Como evitar que fiquem bloqueados?
- 



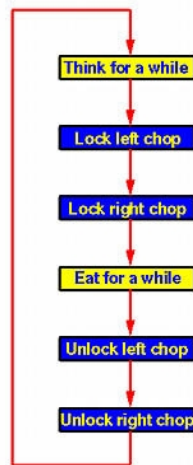
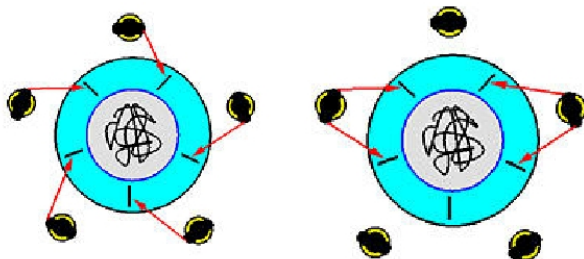
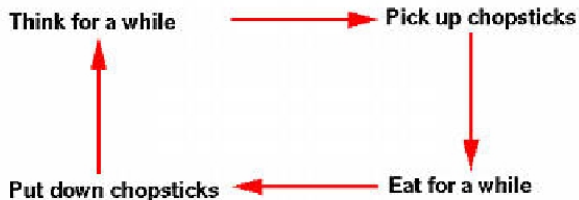
# Jantar dos Filósofos



Que fazer?



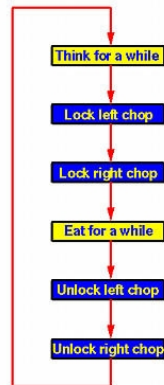
# Jantar dos Filósofos



# Jantar dos Filósofos

```
#define N 5                                /* número de filósofos */

void philosopher(int i)                    /* i: número do filósofo, de 0 a 4 */
{
    while (TRUE) {
        think();                          /* o filósofo está pensando */
        take_fork(i);                     /* pega o garfo esquerdo */
        take_fork((i+1) % N);             /* pega o garfo direito; % é o operador módulo */
        eat();                            /* hummm! Espaguete */
        put_fork(i);                      /* devolve o garfo esquerdo à mesa */
        put_fork((i+1) % N);             /* devolve o garfo direito à mesa */
    }
}
```

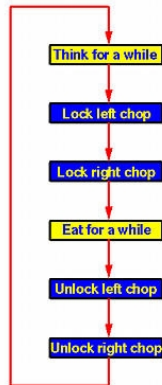


# Jantar dos Filósofos

```
#define N 5                                /* número de filósofos */

void philosopher(int i)                    /* i: número do filósofo, de 0 a 4 */
{
    while (TRUE) {
        think();                          /* o filósofo está pensando */
        take_fork(i);                     /* pega o garfo esquerdo */
        take_fork((i+1) % N);             /* pega o garfo direito; % é o operador módulo */
        eat();                            /* hummm! Espagete */
        put_fork(i);                      /* devolve o garfo esquerdo à mesa */
        put_fork((i+1) % N);             /* devolve o garfo direito à mesa */
    }
}
```

- Isso funciona?





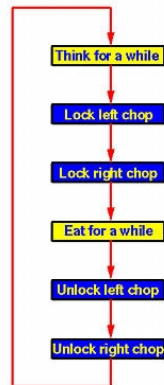
# Jantar dos Filósofos

```
#define N 5                                /* número de filósofos */

void philosopher(int i)                    /* i: número do filósofo, de 0 a 4 */
{
    while (TRUE) {
        think();                          /* o filósofo está pensando */
        take_fork(i);                     /* pega o garfo esquerdo */
        take_fork((i+1) % N);             /* pega o garfo direito; % é o operador módulo */
        eat();                            /* hummm! Espagete */
        put_fork(i);                      /* devolve o garfo esquerdo à mesa */
        put_fork((i+1) % N);             /* devolve o garfo direito à mesa */
    }
}
```

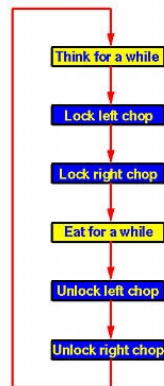
- Isso funciona?

- Em `take_fork()`, se todos os filósofos pegarem o hashi da esquerda, nenhum pegará o da direita – deadlock



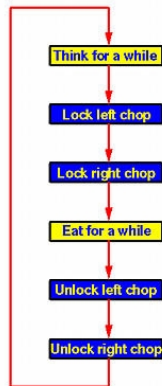
# Jantar dos Filósofos

- Como solucionar?
  - Após pegar o hashi da esquerda, o filósofo verifica se o da direita está livre.
  - Se não estiver, devolve o hashi que pegou, espera um pouco e tenta novamente



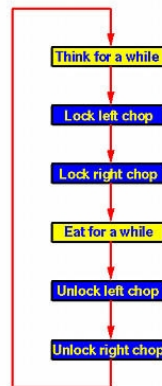
# Jantar dos Filósofos

- Isso funciona?



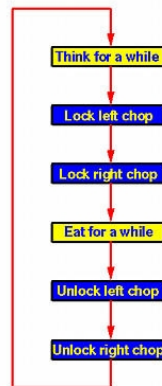
# Jantar dos Filósofos

- Isso funciona?
  - Se todos os filósofos pegarem o hashi da esquerda ao mesmo tempo:
    - Verão que o da direita não está livre
    - Largarão seu hashi e e esperarão
    - Pegarão novamente o hashi da esquerda
    - Verão que o da direita não está livre
    - ...



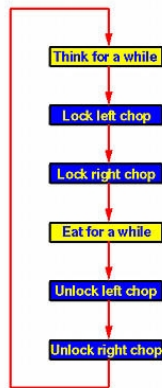
# Jantar dos Filósofos

- Isso funciona?
  - Se todos os filósofos pegarem o hashi da esquerda ao mesmo tempo:
    - Verão que o da direita não está livre
    - Largarão seu hashi e e esperarão
    - Pegarão novamente o hashi da esquerda
    - Verão que o da direita não está livre
    - ...
  - Starvation (inanição)



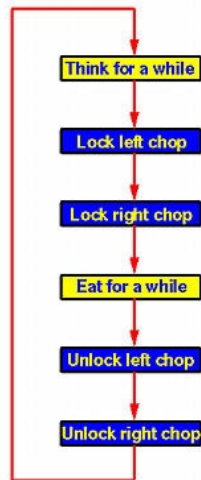
# Jantar dos Filósofos

- E agora?
  - Poderíamos fazer com que eles esperassem um tempo aleatório
    - Reduz a chance de starvation
  - Na maioria das aplicações, tentar novamente mais tarde não é problema
    - Via ethernet, é exatamente isso que é feito com envio de pacotes
    - Sistemas não críticos
    - E controle de segurança em usina nuclear? Será uma boa idéia?



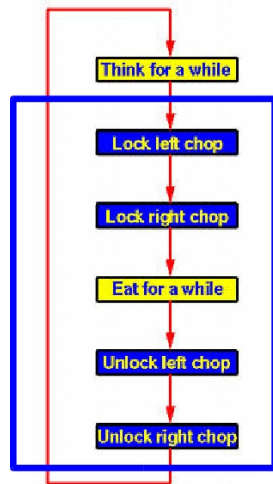
# Jantar dos Filósofos

- E agora, como evitar as múltiplas tentativas nos hashis?



# Jantar dos Filósofos

- E agora, como evitar as múltiplas tentativas nos hashis?
  - Proteger os passos após “pensar por um instante” com um semáforo binário – um mutex
  - down(mutex) ao entrar na área crítica
  - up(mutex) ao sair dela





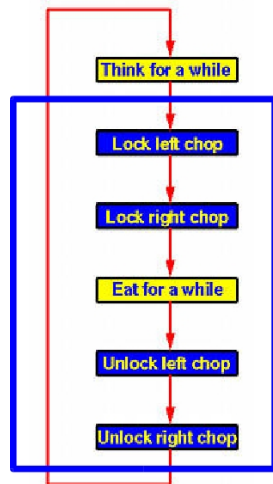
# Jantar dos Filósofos

```
#define N 5                                /* número de filósofos */
semaphore mutex = 1;
void philosopher(int i)                    /* i: número do filósofo, de 0 a 4 */
{
    while (TRUE) {
        think( );                          /* o filósofo está pensando */
        take_fork(i);                      /* pega o garfo esquerdo */
        take_fork((i+1) % N);              /* pega o garfo direito; % é o operador módulo */
        eat( );                            /* hummm! Espaguete */
        put_fork(i);                       /* devolve o garfo esquerdo à mesa */
        put_fork((i+1) % N);               /* devolve o garfo direito à mesa */
        up(&mutex);
    }
}
```

Somente um filósofo come!

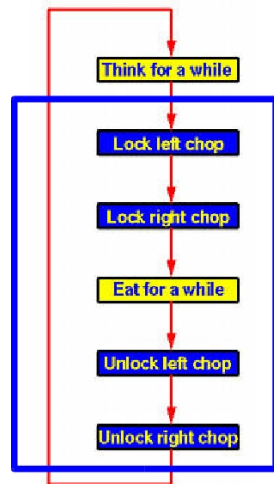
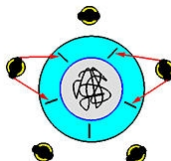
# Jantar dos Filósofos

- Há problemas?



# Jantar dos Filósofos

- Há problemas?
  - Teoricamente, é uma solução adequada
  - Na prática, contudo, tem um problema de desempenho:
    - Somente um filósofo pode comer em um dado momento
    - Com 5 hashis, deveríamos permitir que 2 filósofos comessem ao mesmo tempo



# Jantar dos Filósofos

- Como solucionar?
  - Sem deadlocks ou starvation
  - Com o máximo de paralelismo para um número arbitrário de filósofos

# Jantar dos Filósofos

- Como solucionar?
  - Sem deadlocks ou starvation
  - Com o máximo de paralelismo para um número arbitrário de filósofos
  - Usar um arranjo – estado – para identificar se um filósofo está comendo, pensando ou faminto (pensando em pegar os hashis)
    - Um filósofo só pode comer (estado) se nenhum dos vizinhos imediatos estiver comendo
  - Usar um arranjo de semáforos, um para cada filósofo
    - Filósofos famintos podem ser bloqueados se os hashis estiverem ocupados

# Jantar dos Filósofos

```
#define N 5                /* número de filósofos */
#define ESQ (i+N-1)%N      /* vizinho à esquerda de i */
#define DIR (i+1)%N        /* vizinho à direita de i */
#define PEN 0              /* filósofo pensando */
#define FAM 1              /* filósofo faminto */
#define COM 2              /* filósofo comendo */
typedef int semaforo;
int estado[N];             /* estado dos filósofos (deve iniciar com PEN) */
semaforo mutex = 1;        /* exclusão mútua para estado */
semaforo s[N];             /* para bloqueio dos filósofos (iniciados com 0) */

void filosofo(int i) {     /* i: número do filósofo (0 a N-1) */
    while (TRUE) {
        pensa();          /* o filósofo está pensando */
        pega_hashi(i);     /* pega dois hashis ou bloqueia */
        come();            /* manda o rango prá dentro */
        larga_hashi(i);    /* devolve os hashis à mesa */
    }
}
```

# Jantar dos Filósofos

```
void pega_hashi(int i) { /* i: número do filósofo (0 a N-1) */
    down(&mutex);        /* entra na região crítica */
    estado[i] = FAM;      /* registra que o filósofo está faminto */
    testa(i);             /* tenta pegar 2 hashis */
    up(&mutex);           /* sai da região crítica */
    down(&s[i]);          /* bloqueia se não pegou os hashis */
}

void larga_hashi(int i) { /* i: número do filósofo (0 a N-1) */
    down(&mutex);        /* entra na região crítica */
    estado[i] = PEN;      /* o filósofo acabou de comer */
    testa(ESQ);           /* vê se o vizinho da esquerda pode comer agora */
    testa(DIR);           /* vê se o vizinho da direita pode comer agora */
    up(&mutex);          /* sai da região crítica */
}

void testa(int i) {      /* i: número do filósofo (0 a N-1) */
    if (estado[i] == FAM && estado[ESQ] != COM && estado[DIR] != COM) {
        estado[i] = COM;
        up(&s[i]);
    }
}
```