

# INE5410 – Programação Concorrente

## Unidade II – Fundamentos de Programação Concorrente

Prof. Frank Siqueira  
[frank.siqueira@ufsc.br](mailto:frank.siqueira@ufsc.br)



# Conteúdo

- Processos
- Threads
- Exclusão Mútua
- Semáforos
- Deadlocks

# Deadlocks

- Os mecanismos de controle de concorrência são importantes para garantir a integridade dos recursos compartilhados
- No entanto, ao bloquear o acesso e colocar tarefas em espera, podemos comprometer a **vivacidade** (*liveness*, em inglês) das aplicações
- A vivacidade garante que a execução das aplicações concorrentes será concluída em algum momento, mesmo que elas tenham que esperar temporariamente por um recurso
- A vivacidade é comprometida se as tarefas entrarem em *deadlock* ('bloqueio mortal', em tradução literal, também chamado de *impasse*)

# Deadlocks

- Grande parte dos *deadlocks* envolve o acesso a recursos
  - Um recurso é algo que pode ser adquirido, usado e liberado por tarefas em execução
  - Exemplos:
    - Dispositivo de hardware: gravador de CD, scanner, impressora, câmera, microfone, etc.
    - Arquivos (se abertos com permissão de escrita, o que exige acesso exclusivo)
    - Uma variável compartilhada acessada concorrentemente
    - Registros de uma tabela do banco de dados

# Deadlocks

- O acesso a recursos pode ser feito de modo:
  - Compartilhado: quando várias tarefas podem acessá-lo simultaneamente sem que ocorram falhas no processamento  
Ex.: leitura de arquivo ou dado; saída de áudio.
  - Exclusivo: quando o recurso só pode ser acessado por uma tarefa de cada vez  
Ex.: escrita de um arquivo ou dado; scanner.
- O acesso a recursos em modo exclusivo deve ser controlado através de mecanismos de exclusão mútua

# Deadlocks

- Preempção de recursos
  - Um **recurso preemptível** é aquele que pode ser retirado da tarefa proprietária sem prejuízo para sua execução  
Ex.: CPU; memória.
  - Já um **recurso não preemptível** causará uma falha se for retirado do processo/thread  
Ex.: gravador de CD; impressora.
- Em geral, os *deadlocks* envolvem o acesso a recursos não preemptíveis

# Deadlocks

- *Deadlocks* podem ocorrer quando tarefas (*threads* ou processos) precisam de acesso exclusivo a mais de um recurso
- Suponha a seguinte situação:
  - Thread A bloqueia o acesso ao recurso X, mas precisa também do recurso Y para executar
  - Thread B bloqueia o acesso ao recurso Y, mas precisa também de X, que está com a Thread A
- No caso acima, ocorre um *deadlock* que leva a uma condição de inanição (*starvation*) e impede a aplicação de concluir sua execução

# Deadlocks

- Um *deadlock* também pode ser causado pela perda de mensagens enviadas pela rede
  - Exemplo:
    - Cliente envia uma requisição a um Servidor, e fica bloqueado aguardando uma resposta
    - Servidor envia resposta e fica bloqueado aguardando confirmação de recebimento
    - A resposta é perdida por uma falha na rede
    - Cliente e Servidor esperam indefinidamente
  - Soluções possíveis:
    - Uso de *timeouts* na comunicação + reenvio
    - Primitivas de comunicação não-bloqueantes

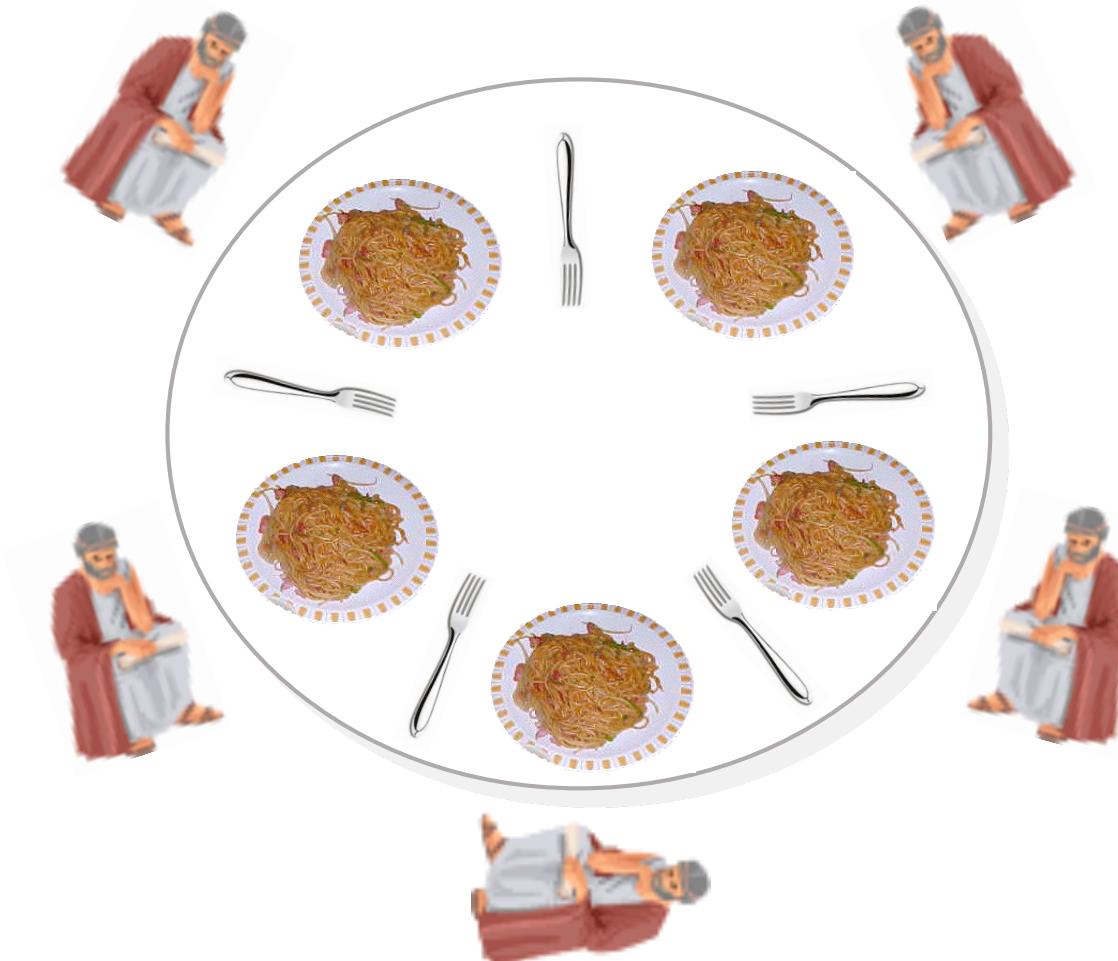
# Deadlocks

- *Livelock*

- Semelhante a um *deadlock*, mas não coloca as tarefas em espera
- Tarefas continuam a executar, mas não conseguem obter os recursos necessários para concluir sua execução → inanição
- Pode ocorrer com recursos preemptivos se a implementação não for feita com cuidado
- Pode ser ainda mais prejudicial que um *deadlock*, pois consome processamento e é muito difícil de detectar

# Deadlocks

- Exemplo: Jantar dos Filósofos [Dijkstra, 1965]



# Deadlocks

- Exemplo: Jantar dos Filósofos – Descrição
  - Os filósofos passam a vida pensando e comendo
  - Há 5 filósofos em volta de uma mesa redonda, com 5 pratos de macarrão e 5 garfos
  - O molho do macarrão é muito escorregadio, e os filósofos precisam pegar 2 garfos para comer
  - Se cada filósofo pegar um garfo, nenhum deles poderá comer → ocorre um *deadlock*!
  - Temos que encontrar uma forma de eles não morrerem de inanição

# Deadlocks

- Exemplo: Jantar dos Filósofos - Código

```
#define FILOSOFOS 5
#define TEMPO    5
#define COMIDA   50

void * filosofo (void *arg) {
    int comida = COMIDA;
    int id = *((int *)arg);
    int garfo_esq = id;
    int garfo_dir = (id + 1) % FILOSOFOS;

    while ( comida > 0 ) {
        printf ("Filósofo %d filosofando...\n", id);
        sleep(TEMPO);
        printf ("Filósofo %d cansou de filosofar e agora está com fome.\n", id);
        come(id, &comida, garfo_esq, garfo_dir);
    }
    printf ("Filósofo %d foi embora porque acabou a comida.\n", id);
    pthread_exit (0);
}
```

# Deadlocks

- Exemplo: Jantar dos Filósofos – Código (cont.)

```
pthread_mutex_t garfos[FILOSOFOS];  
  
void come (int id, int * comida, int garfo_esq, int garfo_dir)  
{  
    pthread_mutex_lock (&garfos[garfo_esq]);  
    printf ("Filósofo %d pegou garfo %d à sua esquerda.\n", id, garfo_esq);  
  
    pthread_mutex_lock (&garfos[garfo_dir]);  
    printf ("Filósofo %d pegou garfo %d à sua direita.\n", id, garfo_dir);  
  
    printf ("Filósofo %d comendo...\n", id);  
    sleep (TEMPO);  
    *comida -= TEMPO;  
    printf ("Filósofo %d terminou de comer.\n", id);  
  
    pthread_mutex_unlock (&garfos[garfo_esq]);  
    pthread_mutex_unlock (&garfos[garfo_dir]);  
    printf ("Filósofo %d soltou os garfos %d e %d.\n", id, garfo_esq, garfo_dir);  
}
```

# Deadlocks

- Exemplo: Jantar dos Filósofos – Análise
  - Caso os 5 filósofos peguem o garfo da esquerda, nenhum deles conseguirá comer
  - Essa situação caracteriza um impasse, que compromete a vivacidade e resulta em inanição
  - Isso pode levar algum tempo para acontecer, mas acabará ocorrendo em algum momento pois existe uma condição de corrida
  - O que pode ser feito nesse caso?

# Deadlocks

- Deadlock ocorre quando temos as seguintes condições:
  - Exclusão mútua: o recurso exige uso exclusivo
  - Posse e espera: um processo em posse de recurso(s) espera por outro recurso
  - Não preempção: o recurso não pode ser retirado do processo que o obteve
  - Espera circular: há um ciclo de espera entre os processos em posse de recursos



# Deadlocks

- Como evitar que recursos possibilitem que as condições de *deadlock* ocorram?
  - Evitar acesso concorrente a recursos que requerem exclusão mútua  
Ex.: criar um *spool* de impressão para acessar a impressora
  - Realizar a preempção do recurso  
Obs.: pode ser difícil fazê-lo sem causar falha na utilização do recurso

# Deadlocks

- Técnicas para contornar *deadlocks*:
  - Prevenção de *deadlocks*: busca evitar que o *deadlock* ocorra tomando certos cuidados ao escrever o código das tarefas
    - Problema: nem sempre se tem acesso ao código de todas as tarefas que utilizam os recursos
  - Detecção e resolução de *deadlocks*: consiste em deixar que ele ocorra, identificar quais tarefas estão nessa condição e abortar uma delas
    - Problemas: sacrificamos uma tarefa para que as demais possam executar; a tarefa abortada pode ser reexecutada, mas levará mais tempo para concluir.

# Deadlocks

- Prevenção de *Deadlocks*
  - As próprias threads/processos podem ser programadas de forma a evitar *deadlocks*
  - Devem ser adotados os seguintes cuidados:
    - Todas as tarefas devem acessar os dados/recursos na mesma ordem
    - Usar primitivas não-bloqueantes  
Ex.: `pthread_mutex_trylock();`
    - Se a tarefa não conseguir todos os recursos necessários, deve liberar os recursos alocados anteriormente

# Deadlocks

- Exemplo: Filósofo com prevenção de *deadlock*

```
pthread_mutex_t garfos[FILOSOFOS];  
  
void come_em_ordem (int id, int * comida, int garfo_esq, int garfo_dir) {  
    if (id == 0) {  
        // O filósofo 0 pegará os garfos na ordem inversa dos demais  
        pthread_mutex_lock (&garfos[garfo_dir]);  
        printf ("Filósofo %d pegou garfo %d à sua direita.\n", id, garfo_dir);  
        pthread_mutex_lock (&garfos[garfo_esq]);  
        printf ("Filósofo %d pegou garfo %d à sua esquerda.\n", id, garfo_esq);  
    } else {  
        pthread_mutex_lock (&garfos[garfo_esq]);  
        printf ("Filósofo %d pegou garfo %d à sua esquerda.\n", id, garfo_esq);  
        pthread_mutex_lock (&garfos[garfo_dir]);  
        printf ("Filósofo %d pegou garfo %d à sua direita.\n", id, garfo_dir);  
    }  
    // Come e libera os garfos; igual a come()  
}
```

# Deadlocks

- Exemplo: Filósofo com primitivas não-bloq.
  - Evita *deadlock*, mas pode causar *livelock/inanição*

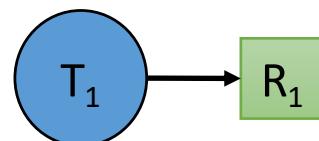
```
pthread_mutex_t garfos[FILOSOFOS];

void come_nao_bloq (int id, int * comida, int garfo_esq, int garfo_dir) {
    pthread_mutex_lock (&garfos[garfo_esq]);
    printf ("Filósofo %d pegou garfo %d à sua esquerda.\n", id, garfo_esq);

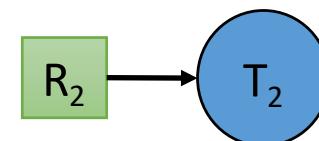
    while (pthread_mutex_trylock (&garfos[garfo_dir]) != 0) {
        // Solta o garfo da esquerda para que outro possa comer
        pthread_mutex_unlock (&garfos[garfo_esq]);
        printf ("Filósofo %d soltou garfo %d à sua esquerda.\n", id, garfo_esq);
        sleep(TEMPO); // Espera um tempo para que o garfo seja liberado
        pthread_mutex_lock (&garfos[garfo_esq]);
        printf ("Filósofo %d pegou garfo %d à sua esquerda.\n", id, garfo_esq);
    }
    printf ("Filósofo %d pegou garfo %d à sua esquerda.\n", id, garfo_esq);
    // Come e libera os garfos; igual a come()
}
```

# Deadlocks

- Detecção de *Deadlocks*
  - Verificar o estado do sistema periodicamente para determinar se ocorreu *deadlock*
  - Precisa saber que bloqueios estão ativos
  - *Deadlocks* detectados criando um **grafo de espera**
  - Arestas indicam espera ou posse de recursos



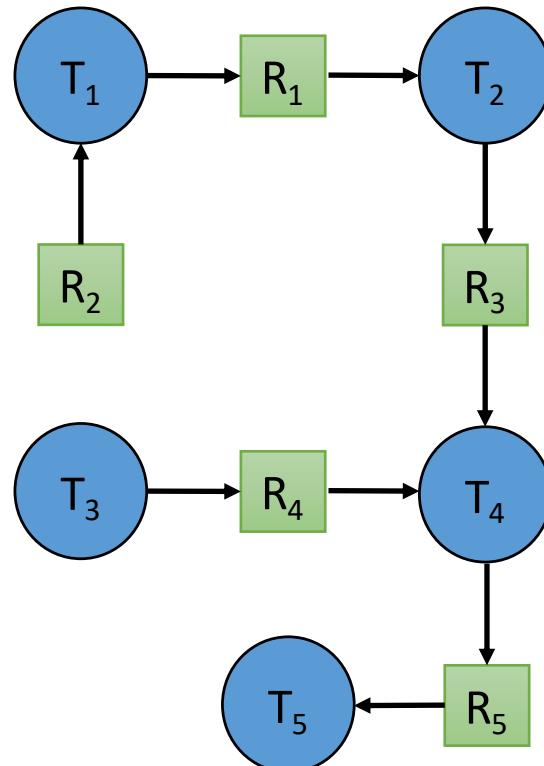
$T_1$  espera por  $R_1$



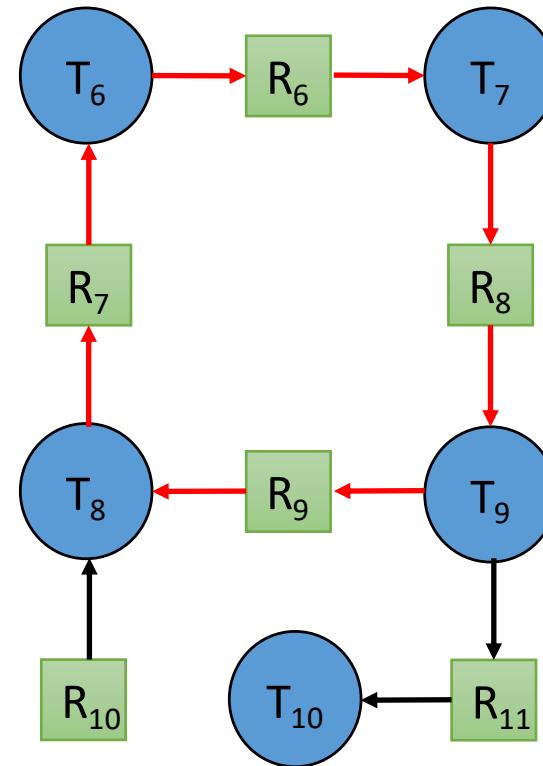
$T_2$  faz uso exclusivo de  $R_2$

# Deadlocks

- Grafo sem ciclo
  - Não há *deadlock*



- Grafo com ciclo
  - Ocorreu *deadlock*



# Deadlocks

- Recuperação de *Deadlocks*
  - Algumas estratégias podem ser adotadas pelo sistema para solucionar um impasse detectado
  - Recuperação por **retrocesso**: fazer com que o programa reverta a um estado anterior e libere recursos que estavam sendo usados
    - Exige que o programa mantenha pontos de restauração (*checkpoints*)
    - Problemas: custo e complexidade
  - Recuperação por **preempção do recurso**: retira o recurso temporariamente da tarefa
    - Só é possível com recursos preemptivos

# Deadlocks

- Recuperação de *Deadlocks* (cont.)
  - Recuperação por **eliminação**: consiste em abortar uma tarefa de modo a interromper o ciclo de espera
    - Critérios possíveis de escolha da vítima:
      - Tempo de execução da tarefa
      - Tempo necessário para conclusão
      - Operações de I/O já efetuadas ou a efetuar
      - Número de abortos sofridos (para evitar que a vítima seja sempre a mesma)
      - etc.
    - A tarefa abortada pode ser reexecutada
    - Problema: desperdício de processamento