Capítulo 2

Processos e Threads

2.1 Processos
2.2 **Threads**
2.3 Comunicação interprocesso
2.4 Problemas clássicos de IPC
2.5 Escalonamento

Sequencial vs Concorrente

- Programa Sequencial
  - Programa executado por apenas um processo
- Programa Concorrente
  - Programa executado por diversas threads que cooperam entre si para a realização de uma tarefa.
    - Necessidade de interação para troca de informações

Paralelismo

- Paralelismo Real
  - Execução simultânea de dois ou mais processos (threads)
  - Só ocorre em CPU multithread
  - Multiprocessamento físico (exemplo: tecnologia Hyper-Threading (HT) da Intel)

- Paralelismo Aparente
  - Utiliza vários recursos, como compartilhamento de tempo de CPU entre vários processos, para simular simultaneidade
    - Dá impressão ao usuário de que os programas são executados ao mesmo tempo

Programação Concorrente

- Composta por um conjunto de processos sequenciais que executam concorrentemente (threads)
- Disputam recursos comuns
  - Variáveis, periféricos, etc.
- Processos (Threads) cooperantes
  - Capaz de afetar, ou ser afetado, pela execução de outro processo (thread)

Programação Concorrente

Motivação

- Aumento de desempenho
  - Permite exploração do paralelismo real disponível em CPUs multicore/multithreading
  - Sobreposição do operações de E/S com processamento
- Facilidade de desenvolvimento de aplicações que possuem um paralelismo intrínseco
  - Ex.: Jogos, Sistema Operacional, etc.

Programação Concorrente

Desvantagens

- Programação complexa
- Além dos erros da programação convencional, adicionam-se os erros próprios do modelo
  - Erros acossiados às interações entre as threads (Erro de concorrência)
- Difícil depuração

Relação Produtor-Consumidor

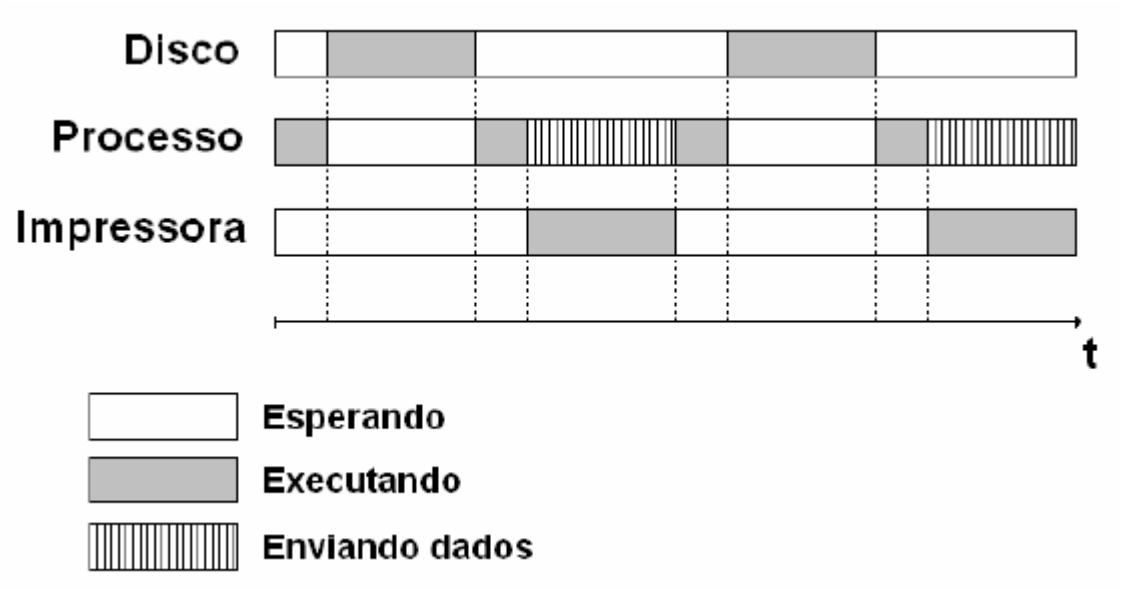
- Processo produtor produz um fluxo de dados consumido pelo processo consumidor
- Exemplo: Programa lê um arquivo, formata os dados e envia para uma impressor
  - Programa sequencial
  - Programa concorrente

Programa Sequencial : Exemplo

1. Processo envia um comando para a leitura do arquivo e fica bloqueado
2. Disco é acionado para realizar operação de leitura
3. Concluída a leitura, processo realiza formatação e inicia a transferência dos dados para a impressora
4. Processo executa um laço no qual os dados são enviados para a impressora
   - Processo fica preso até o final da impressão

Programa Sequencial : Exemplo

- Disco e impressora nunca trabalham simultaneamente, apesar de não existir nenhuma limitação de natureza eletrônica
  - Programa sequencial não consegue ocupar ambos

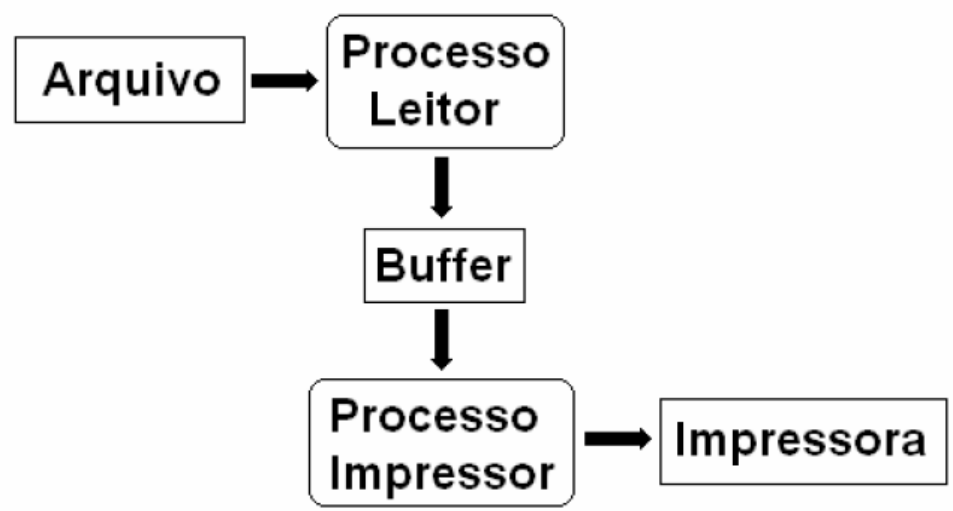


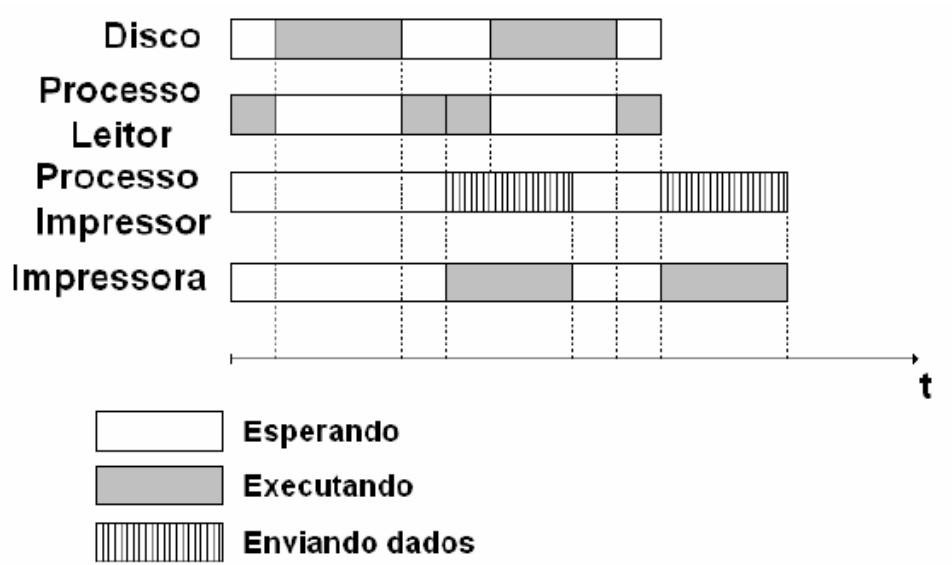
Programa Concorrente : Exemplo

- Dois processos
  - Leitor
    - Lê arquivo, formata e coloca em um buffer na memória
  - Impressor
    - Retira dados do buffer e envia para a impressora

Obs.: Buffer é uma região de memória temporária utilizada para escrita e leitura de dados (software ou hardware)

Programa Concorrente : Exemplo





Programa Concorrente : Exemplo

- Disco e impressora trabalham simultaneamente

- Limitações
  - Se o processor leitor for sempre mais rápido, o buffer ficará cheio e então o processo leitor terá que esperar até que o processo impressor retire algo do buffer (buffer overflow)
  - Se o processor impressor for sempre mais rápido, o buffer ficará vazio e terá que esperar até pelo processo leitor (buffer underrun)

Especificação do Paralelismo

- Para construir um programa concorrente, antes de mais nada, é necessário ter a capacidade de especificar o paralelismo dentro do programa

- E necessário especificar
  - Quantas threads farão parte do programa
  - Quais rotinas cada uma executará

Especificação em Pseudocódigo

- Delimitam todos os comandos que serão executados em paralelo
- Parbegin (Parallel begin)
  - Comando indicando que a execução sequencial passa a ser dividida em várias sequências de execução em paralelo
  - Marca o início da execução paralela
- Parend (Parallel end)
  - Comando indicando que certas sequências de execução paralela devem se juntar para a execução sequencial continuar
  - Marca o fim da execução paralela

Exemplo:

```
Processo A:
Início
    ...
    Parbegin
        comando_1;
        comando_2;
        ...
        comando_n;
    Parend
    ...
Fim
```

Especificação do Paralelismo

- Processos paralelos podem ser executados em qualquer ordem
  - Duas execuções consecutivas do mesmo programa, com os mesmos dados de entrada podem gerar resultados diferentes
    - Não é, necessariamente, um erro
- Possibilidade de forçar a execução em uma determinada ordem

Compartilhamento de Recursos

- Programação concorrente implica em compartilhamento de recursos
- Threads podem compartilhar todos seu espaço de memória ou apenas uma parte
- Variáveis compartilhadas
  - Processo escreve em uma variável que será lida por outro
  - É necessário controlar o acesso a essas variáveis

Seção Crítica

- Parte do código de um processo que realiza a alteração de um recurso compartilhado
- Exemplo:

```
Parbegin
    x = x + 1;
    x = x * x;
Parend
```

- Deve-se garantir que nenhum outro processo acesse a seção crítica enquanto uma thread o faz

Seção Crítica

- Uma solução para o problema de seção crítica estará correta quando apresentar as seguintes propriedades:
  - Existe exclusividade mútua entre os processos com referência a execução das respectivas seções críticas
  - Quando uma thread T deseja entrar na seção crítica e nenhuma outra thread está executando sua seção crítica, a thread T não é impedida de entrar
  - Nenhum processo pode ter seu ingresso na seção crítica postergado indefinidamente, ou seja, ficar esperando para sempre
  - A solução não depende das velocidades relativas das threads

Seção Crítica

- Soluções erradas para o problema da seção crítica, normalmente, apresentam a possibilidade de postergação indefinida ou a de deadlock

- Postergação indefinida
  - Processo está preso tentando entrar na seção crítica e nunca consegue por ser sempre preterido em benefício de outros processos

- Deadlock
  - Dois ou mais processos estão à espera de um evento que nunca vai ocorrer

Seção Crítica

```java
package semaforos;

public class EstaticoThread extends Thread{

    static float x;
    int idThread;

    public EstaticoThread(int idThread){
        this.idThread = idThread;
    }

    public void run(){
        conta();
    }

    public void conta(){
        if (idThread % 2 == 0){
            for (int i = 1 ; i < 10 ; i++){
                x = (float) (x + Math.pow(idThread, i));
                System.out.println(x);
            }
        } else {
            for (float i = 1 ; i < 10 ; i++){
                x = x + (1 / i);
                System.out.println(x);
            }
        }
    }

}
```

```java
package semaforos;

public class Estatico {

    public static void main(String[] args) {
        for (int i = 2 ; i < 4 ; i++){
            Thread t = new EstaticoThread(i);
            t.start();
        }
    }


}
```

```
1.0
3.5
3.8333333
4.083333
4.283333
4.4499993
4.5928564
4.7178564
4.8289676
3.0
8.828968
16.828968
32.828968
64.828964
128.82896
256.82898
512.829
1024.829
```

Seção Crítica : Soluções

- Desabilitar Interrupções
  - Thread desabilita interrupções antes de acessar variáveis compartilhadas
  - Ao final da seção crítica thread reabilita as interrupções
  - Utilizado em Sistemas Operacionais pequenos e dedicados a uma única aplicação, sistemas embarcados
    - Ex.: Celulares

- Desvantagens
  - Vai contra os mecanismos de proteção do S.O.
  - Poder demais para processos de usuários
  - Diminuição da eficiência do sistema
  - Não funcionam em máquinas paralelas

Seção Crítica : Soluções

- Semáforos
  - Tipo abstrato de dados composto por um valor inteiro e uma fila de processos
  - Somente duas operações permitidas no semáforo
    - P (Testar)
    - V (Incrementar)

- Todo semáforo deve possuir dois métodos: P e V, que têm sua origem das palavras Parsen (passar) e Vrygeren (liberar). Esta definição de semáforo foi proposta por Dijkstra para evitar o tão temido DeadLock

Seção Crítica : Soluções

- Semáforos
  - Quando um processo executa a operação P em um semáforo, o seu valor inteiro é decrementado
  - Caso o novo valor do semáforo seja negativo, a thread é bloqueada e vai para o fim da fila do semáforo
  - Quando uma thread executa a operação V sobre um semáforo, o seu valor inteiro é incrementado
  - Caso exista alguma thread bloqueada na fila deste semáforo, a primeira thread da fila é liberada

- P(S):
  - S.valor = S.valor – 1;
  - Se valor < 0
    - Então bloqueia a thread, insere em S.fila
- V(S):
  - S.valor = S.valor + 1;
  - Se S.fila está vazia
    - Então retira thread T de S.fila, acorda T

Seção Crítica : Soluções

- Semáforos
  - Para cada estrutura de dados compartilhada, deve ser criado um semáforo S inicializado com o valor 1
  - Toda thread, antes de acessar essa estrutura, deve executar a operação P sobre o semáforo S associado à estrutura de dados em questão
  - Ao sair da seção crítica, o processo executa a operação V sobre o semáforo
  - Se o valor do semáforo é negativo, significa que existem processos na fila de espera do semáforo
    - Valor absoluto do semáforo é igual ao número de processos na fila

Seção Crítica : Soluções

- Semáforos binário (mutex)
  - Semáforo capaz de assumir apenas os valores 0 e 1
  - Assume apenas os valores livre e ocupado
  - P e V são chamados de lock e unlock, respectivamente
    - No Java, `acquire()` (lock) e `release()` (unlock), são os métodos da classe `Semaphore`

Semáforos em Java

```java
package semaforo;

import java.util.concurrent.Semaphore;

public class ProcessadorBasico{

    private static Semaphore semaforo;

    private static void processar(int idThread) {
        try {
            System.out.println("Thread #" + idThread + " processando");
            int tempoDormir = (int)(Math.random() * 10000);
            Thread.sleep(tempoDormir);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void entrarRegiaoNaoCritica(int idThread) {
        System.out.println("Thread #" + idThread + " em região não crítica");
        processar(idThread);
    }

    private static void entrarRegiaoCritica(int idThread) {
        System.out.println("Thread #" + idThread
                + " entrando em região crítica");
        processar(idThread);
        System.out.println("Thread #" + idThread + " saindo da região crítica");
    }

    public static void processamento(int idThread){
      entrarRegiaoNaoCritica(idThread);
      try {
          semaforo.acquire();
          entrarRegiaoCritica(idThread);
      } catch (InterruptedException e) {
          e.printStackTrace();
      } finally {
          semaforo.release();
      }
    }
```

```java
    static Thread t1 = new Thread(){
        public void run() {
            processamento(1);
        }
    };

    static Thread t2 = new Thread(){
        public void run() {
            processamento(2);
        }
    };

    static Thread t3 = new Thread(){
        public void run() {
            processamento(3);
        }
    };

    static Thread t4 = new Thread(){
        public void run() {
            processamento(4);
        }
    };

    public static void main(String[] args) {
        int numeroDePermicoes = 1;
        semaforo = new Semaphore(numeroDePermicoes);
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }

}
```

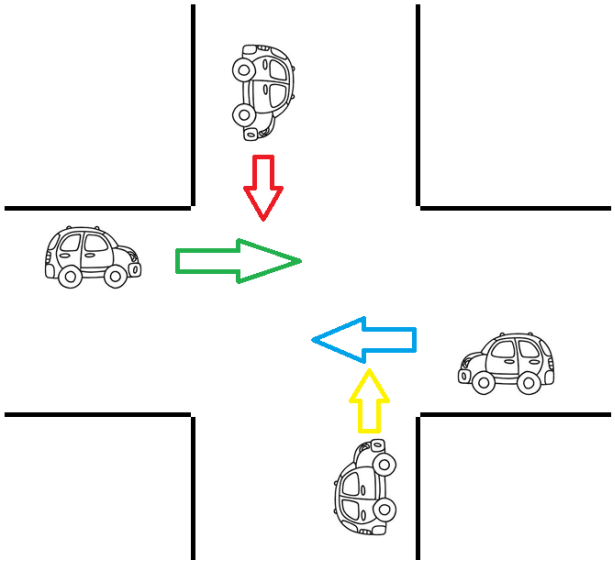
Exemplo

Considere que 10 automóveis se deslocam de lugares distintos, distantes de 1Km a 2km de um estacionamento. Todos pretendem estacionar nesse estacionamento que tem 3 vagas. Os carros se deslocam em velocidade de 100 m a cada 100 milissegundos e, quando estacionam, pretendem ficar de 1 a 3 segundos estacionados. Ao chegar e ao sair do estacionamento, deve-se fazer uma notificação. Ao estacionar, também deve-se fazer uma notificação. Os carros que chegam e não conseguem estacionar, devem aguardar em fila por ordem de chegada.

Exercício

1) Fazer uma aplicação, console, que gerencie a figura abaixo:

Para tal, usar uma variável sentido, que será alterado pela Thread que controla cada carro com a movimentação do carro. Quando a Thread tiver a possibilidade de ser executada, ela deve imprimir em console o sentido que o carro está passando. Só pode passar um carro por vez no cruzamento.



Exercício

2) 4 pessoas caminham, cada uma em um corredor diferente. Os 4 corredores terminam em uma única porta. Apenas 1 pessoa pode cruzar a porta, por vez. Considere que cada corredor tem 200m. e cada pessoa anda de 4 a 6 m/s. Cada pessoa leva de 1 a 2 segundos para abrir e cruzar a porta. Faça uma aplicação em java que simule essa situação.

Exercício

3) Um banco deve controlar Saques e Depósitos.

O sistema pode permitir um Saque e um Depósito Simultâneos, mas nunca 2 Saques ou 2 Depósitos Simultâneos. Para calcular a transação (Saque ou Depósito), o método deve receber o código da conta, o saldo da conta e o valor a ser transacionado. Deve-se montar um sistema que considera 20 transações simultâneas enviadas ao sistema (aleatoriamente, essas transações podem ser qualquer uma das opções) e tratar todas as transações.

Exercício

4) Você foi contratado para automatizar um treino de Fórmula 1. As regras estabelecidas pela direção da provas são simples:

"No máximo 5 carros das 7 escuderias (Cada escuderia tem 2 carros diferentes, portanto, 14 carros no total) presentes podem entrar na pista simultaneamente, mas apenas um carro de cada equipe. O segundo carro deve ficar à espera, caso um companheiro de equipe já esteja na pista. Cada piloto deve dar 3 voltas na pista. O tempo de cada volta deverá ser exibido e a volta mais rápida de cada piloto deve ser armazenada para, ao final, exibir o grid de largada, ordenado do menor tempo para o maior."