Resolução dos Exercícios do Capítulo IX

1. Se a programação concorrente traz dificuldades para a programação, quais vantagens se têm com a sua utilização?

Nos dias atuais a programação concorrente está muito presente, seja quando imprimimos um documento e ao mesmo tempo o editamos, seja quando há duas instâncias do mesmo programa em execução. A programação concorrente permite que atividades sejam feitas em menor espaço de tempo e produz uma melhor interatividade entre o sistema e os usuários por conseqüência da multiprogramação.

2. Quais são as principais diferenças entre *threads* e processos? Cite as respectivas vantagens e desvantagens de sua utilização.

Processos são programas em execução, enquanto *threads* são fluxos de execução em um determinado processo. Processos apresentam estados (novo, executável, em espera, em execução e encerrado), enquanto o estado do *thread* é definido pelo estado do processo em que ele se encontra.
Utilizar apenas processos concorrentes pode levar a um uso exagerado da memória, visto que o estado atual dos registradores e demais atributos devem ser persistidos quando um processo sai do estado "em execução" e passa para o estado "em espera". Além disso, o controle dos processos produz um grande overhead no sistema, reduzindo a performance geral. *Threads*, por outro lado, possibilitam uma melhor performance e economia de memória. Note que não existem *threads* sem processos.

3. Quais são as principais diferenças entre memória compartilhada e de troca de mensagens? Cite vantagens e desvantagens.

Como o nome já diz, "memória compartilhada" permite um compartilhamento da memória física entre sistemas concorrentes. "Troca de mensagens" faz uso de passagem de mensagem para comunicação entre os sistemas.
Trabalhar com programação concorrente fazendo uso de memória compartilhada é mais eficiente que utilizar troca de mensagens. Entretanto, para utilizar memória compartilhada se deve implementar mecanismos que garantam a exclusão mútua no acesso à memória, o que não é trivial. Uma outra vantagem de utilizar troca de mensagens é a possibilidade de utilizar memórias físicas em locais diferentes, ou seja, não é necessário que a memória utilizada fique em apenas um computador.

4. Mostre como é possível utilizar semáforos junto aos laços *while* dos códigos do produtor e do consumidor no problema mostrado no exemplo 9.2, reduzindo assim o *overhead* do sistema.

Inicialização:

```
1     fim = 0;
2     ini = 0;
3     n = 0;
4
5     semaforo Cheio;
```

```
6       Cheio.valor = 1;
7       semaforo Vazio;
8       Vazio.valor = 1;
```

Código do produtor:

```
1       for (i=0; i<1000; i++) {
2          // while (n == capacidade) ;
3          while (n == capacidade) P(Cheio)
4          buf[fim] = produzir(i);
5          fim = (fim + 1) % capacidade;
6          n++;
7          V(Vazio)
8       }
```

Código do consumidor:

```
1       for (i=0; i<1000; i++) {
2          // while (n == 0) ;
3          while (n == 0) P(Vazio)
4          consumir(buf[ini]);
5          ini = (ini + 1) % capacidade;
6          n--;
7          V(Cheio)
8       }
```

Com a inclusão de dois semáforos ("Cheio" e "Vazio") é possível reduzir o overhead do sistema causado pelo uso exclusivo dos laços *while* nos códigos do produtor e do consumidor. Na linha 3 do código do produtor, caso o *buffer* esteja cheio, o processo produtor é colocado em espera até que seja consumido algo. Caso contrário, um novo elemento é produzido e o semáforo "Vazio" é liberado (linha 7). Analogamente, no código do consumidor, o processo é colocado em espera caso o *buffer* esteja vazio. Em caso contrário, um elemento é consumido e o semáforo "Cheio" é liberado. Observe que os laços *while* substituídos estão comentados na linha 2, tanto do código do produtor quanto do consumidor.

5. Faça uma classe *Semaforo* em JAVA que implemente as operações *P* e *V* de um semáforo. Utilize para isso os métodos *wait()* e *notify()*. A classe deve possuir métodos *P* e *V* em exclusão mútua (*synchronized*).

```
class Semaforo {
  private static int valor = 1;
  public synchronized void P() throws InterruptedException {
     valor -= 1;
     if (valor < 0) {
        wait();
     }
  }
```

```
  public synchronized void V() throws InterruptedException {
    valor += 1;
    if (valor <= 0) {
      notify();
    }
  }
}
```

6. Implemente uma tarefa *Semaforo* em ADA utilizando entradas *P* e *V*.

```
task Semaforo is
  entry P;
  entry V;
end Semaforo;

task body Semaforo is
begin
  loop
    accept P;
    accept V;
  end loop;
end;
```

7. Suponha que sejam retiradas as chamadas às entradas iniciar de *carro1* e *carro2* no exemplo 9.13. Indique a opção abaixo com o resultado correto da execução.

a) Aparecerá na tela as seguintes mensagens:
```
O carro 1 esta na posicao 0
O carro 2 esta na posicao 0
```

b) Aparecerá na tela as seguintes mensagens:
```
O carro 1 esta na posicao 0
O carro 1 esta na posicao 1
O carro 2 esta na posicao 0
O carro 2 esta na posicao -1
```

c) Não aparecerá nada na tela.
d) O programa dará erro em tempo de compilação.

Letra C. Não aparecerá nada na tela, pois a tarefa *Carro* espera pela chamada à entrada "*iniciar*" para prosseguir a execução. Somente após a chamada à entrada "*iniciar*" é feita a impressão de algo na tela.

8. Implemente o programa do exemplo 9.9 retirando o semáforo, descreva o que acontece e justifique.

Ao retirar todos os semáforos começam a aparecer algumas inconsistências. Um exemplo é que o algoritmo pode fazer com que os ponteiros de início e fim do *buffer* fiquem com indicação incorreta. Isso pode fazer com que as impressões na tela não reflitam as operações reais, podendo indicar que um elemento está sendo produzido mais de uma vez, quando na realidade não está.

9. Quais são as principais características das linguagens C, JAVA e ADA relacionadas à programação concorrente?

A linguagem C não oferece mecanismos próprios para programação concorrente. É necessário utilizar bibliotecas de funções ou utilizar recurso de chamada de sistema para trabalhar com processos concorrentes.
A linguagem JAVA disponibiliza recursos para implementação de *threads* e oferece mecanismos de sincronização de métodos.
A linguagem ADA utiliza-se da definição de módulos concorrentes permitindo a construção de sistemas concorrentes. ADA oferece ainda recursos para sincronização de tarefas através de objetos protegidos ou troca de mensagens.