1) Explique resumidamente no que consistem os dois tipos de sincronização: cooperação e competição.

A sincronização de cooperação é necessária entre as tarefas A e B quando a tarefa A deve esperar que a tarefa B complete alguma atividade específica, antes de continuar sua execução.
A sincronização de competição é necessária entre duas tarefas quando ambas requerem o uso de algum recurso que não pode ser usado simultaneamente.

2) Suponha que duas tarefas A e B devem usar uma variável compartilhada BUF_SIZE. A tarefa A adiciona 2 a BUF_SIZE e a tarefa B subtrai 1 de BUF_SIZE. Assuma que essas operações aritméticas são feitas pelo processo de três passos de buscar o valor atual de BUF_SIZE, efetuar a operação e armazenar de volta o resultado. A figura abaixo mostra uma das possíveis sequências de eventos e o valor resultante, assumindo que o valor inicial de BUF_SIZE é 6. Não havendo sincronização de competição, quais as outras sequências de eventos que são possíveis e quais são os valores resultantes dessas operações, para o mesmo valor inicial?

Resultado final: BUF_SIZE = 7

Tarefa A	Tarefa B
Busca BUF_SIZE	
BUF_SIZE = BUFSIZE+2	
Armazena BUF_SIZE	
	Busca BUF_SIZE
	BUF_SIZE = BUFSIZE-1
	Armazena BUF_SIZE

Resultado final: BUF_SIZE = 8

Tarefa A	Tarefa B
Busca BUF_SIZE	
	Busca BUF_SIZE
BUF_SIZE = BUFSIZE+2	
	BUF_SIZE = BUFSIZE-1
	Armazena BUF_SIZE
Armazena BUF_SIZE	

Resultado final: BUF_SIZE = 5

Tarefa A	Tarefa B
Busca BUF_SIZE	
	Busca BUF_SIZE
BUF_SIZE = BUFSIZE+2	
	BUF_SIZE = BUFSIZE-1
Armazena BUF_SIZE	
	Armazena BUF_SIZE

3) Descreva as ações de wait (P, esperar) e release (V, liberar) dos semáforos.

Um semáforo é uma estrutura de dados consistindo de um inteiro e uma fila que armazena descritores de tarefas (estrutura que armazena todas as informações relativas ao estado de execução da tarefa). Possui duas operações: P e V.

O campo inteiro de um semáforo pode ser um contador, quando o semáforo é usado para sincronização de cooperação, ou um valor binário, quando o semáforo é usado para sincronização de competição.
Na sincronização de cooperação, a operação P sobre um semáforo S testa o contador desse semáforo. Se o contador é maior que zero, a tarefa que chamou P pode executar sua operação e o valor do contador é decrementado. Se o valor do contador é zero, a tarefa que chamou P é colocada na fila de S. A operação V sobre o semáforo S é usada para permitir que algum outro processo utilize o recurso que esse semáforo está contando. Se a fila de S está vazia, a operação V incrementa o contador de S para indicar que mais um dos recursos que ele está contando se tornou disponível. Se uma ou mais tarefas estão esperando na fila do semáforo, a operação V move uma delas da fila do semáforo para a fila de tarefas prontas.
Na sincronização de competição, a operação P sobre um semáforo S testa se o processo que executou P(S) pode ou não entrar na região crítica, isto é, se já existir outro processo na região crítica, ele não poderá entrar e será colocado na fila de espera daquele semáforo. A operação V sobre um semáforo S sinaliza ao semáforo que o processo não está mais na região crítica e retira outro processo da fila de espera, colocando-o novamente em execução.

4) Considere o código a seguir, escrito em uma pseudo-linguagem, definindo tarefas de produtor e consumidor. Caso esse código fosse executado sequencialmente usando um tamanho de buffer maior ou igual a 1000, o produtor iria primeiro produzir todos os dados para que, na seqüência, o consumidor consumisse todos os dados. Supondo que essas duas tarefas serão executadas por processo concorrentes, pode haver inconsistências devido ao compartilhamento da variável n. Utilize as operações de semáforo para modificar esse código e evitar esse problema, garantindo acesso mutuamente exclusivo à região crítica.

```
// inicializacao
    fim = 0;
    ini = 0;
    n = 0;
    // codigo do produtor
    for (i=0; i<1000; i++) {
      while (n == capacidade);
      buf[fim] = produzir(i);
      fim = (fim + 1) % capacidade;
      n++;
    }
    // codigo do consumidor
    for (i=0; i<1000; i++) {
      while (n == 0) ;
      consumir(buf[ini]);
      ini = (ini + 1) % capacidade;
      n--;
    }
```

RESPOSTA:

```
P(S)
    S.valor -= 1;
    Se (S.valor < 0)
            // bloqueia o processo e insere em S.fila

V(S)
   S.valor += 1;
   Se (S.valor <= 0)
          //retira algum processo de S.fila e o coloca em execução

// inicializacao
    fim = 0;
    ini = 0;
    n = 0;
    semáforo S;
    S.valor = 1;

    // codigo do produtor
    for (i=0; i<1000; i++) {
```

```
    while (n == capacidade);
    buf[fim] = produzir(i);
    fim = (fim + 1) % capacidade;
    P(S);
    n++;
    V(S);
  }
  // codigo do consumidor
  for (i=0; i<1000; i++) {
    while (n == 0) ;
    consumir(buf[ini]);
    ini = (ini + 1) % capacidade;
    P(S);
    n--;
    V(S);
  }
```

5) Considerando o mesmo código do exemplo anterior, inclua dois semáforos nos laços while para permitir a sincronização de cooperação entre esses processos.

```
// inicializacao
    fim = 0;
    ini = 0;
    n = 0;
    semaforo Cheio;
    Cheio.valor = 1;
    semaforo Vazio;
     Vazio.valor = 1;

    // codigo do produtor
    for (i=0; i<1000; i++) {
      //  while (n == capacidade);
      while (n == capacidade) P(Cheio) ;
      buf[fim] = produzir(i);
      fim = (fim + 1) % capacidade;
      n++;
      V(Vazio)
    }
    // codigo do consumidor
    for (i=0; i<1000; i++) {
      // while (n == 0) ;
      while (n == 0) P(Vazio)  ;
      consumir(buf[ini]);
      ini = (ini + 1) % capacidade;
      n--;
      V(Cheio)
    }
```

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. No 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. 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, tanto do código do produtor quanto do consumidor.

6) Considere o código do exemplo produtor-consumidor dado em aula. Suponha que o comando release (access) na tarefa consumer foi incorretamente substituído por wait(access). Qual seria o resultado provocado por esse erro na execução do sistema?

```
semaphore access, fullspots, emptyspots;
access.count := 1;
fullspots.count := 0;
emptyspots.count := BUFLEN;

task producer;
  loop
  /* produzir VALOR */
  wait (emptyspots);   /* esperar por espaço vazio
  wait( access);       /* esperar por acesso
  DEPOSIT (VALOR);
  release (access);    /* liberar acesso
  release (fullspots); /* incrementar espaços vazios
  end loop;
end producer;
task consumer;
  loop
  wait (fullspots);    /* confirmar que não está vazio
  wait( access);       /* esperar por acesso
  FETCH (VALOR);
  release (access);    /* liberar acesso
  release (emptyspots); /* incrementar espaços vazios
  /* consumir VALOR */
end loop;
end consumer;
```

A tarefa consumer, que deveria nesse ponto liberar o acesso ao dado compartilhado, vai requisitar novamente o acesso a esse dado. Como o acesso está restrito a essa mesma tarefa e nenhuma outra tarefa poderá emitir uma chamada a release, ocorre um deadlock, todos as tarefas ficam suspensas esperando um evento que não vai ocorrer.

7) Construa uma classe Semáforo 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(); }
  } }
```

8) Construa uma classe Banco em JAVA que simula operações de transferências entre contas bancárias. A classe deve ter campos de dados que representam as contas bancárias, que podem ser em número fixo. O método que faz a transferência deve ser sincronizado e recebe como parâmetros a conta de origem, a conta destino e o valor a ser transferido. A transferência consiste em reduzir o valor a ser transferido da conta de origem e depositar na conta destino. Usar os métodos wait() e notify() para garantir que há saldo suficiente na conta de origem e para avisar que a operação de transferência foi concluída.

```
class Banco  {
    double conta[] = new double[5];
     void set_conta(int i, double d)  {
          conta[i] = d;
     }
   public synchronized void transferencia(int origem, int destino, double valor)  {
          while (conta[origem] < valor) {
            try {
                wait ();
                } catch (InteruptedException e) {
                  }
          }
          conta[origem] -= valor;
          conta[destino] += valor;
          notify();
   }
}
```

9) Construa um programa em JAVA que simule uma corrida de barreiras individuais de 200 metros com 4 barreiras, usando linhas múltiplas de execução. Os trechos de corrida têm o mesmo tamanho, as barreiras estão distribuídas a cada 40 metros e têm a mesma altura. Caso a barreira seja derrubada na hora do salto, o corredor pode continuar correndo, mas perde pontos. As linhas de execução devem executar um comando de repetição que inclui os trechos de corrida e as barreiras que devem ser puladas. A corrida em cada trecho pode ser representada por outro comando de iteração e pela impressão de caracteres que indicam a distância percorrida a cada cinco ou dez metros. Os saltos de barreiras também devem ser impressos. Para realizar um salto, deve ser gerado um valor aleatório que indica se a barreira foi pulada corretamente ou se o corredor a derrubou. Pode ser definida uma "taxa de salto" para cada corredor, por exemplo, 90%. Se o valor aleatório gerado for menor que 0,9, o corredor conseguiu pular a barreira sem derrubá-la. Definir nomes para as linhas de execução e fazer impressões que indiquem a evolução da corrida, os saltos de barreiras e a conclusão da corrida para cada corredor, bem como o número de pontos perdidos.

10) Em uma conta bancária simplificada, o correntista pode fazer apenas as operações de saque e consulta de saldo. Caso não haja saldo suficiente, a operação não é realizada. Uma determinada instituição pode fazer pagamentos ao correntista, realizando depósitos mensais. Crie uma classe em JAVA para representar a conta corrente e defina as operações como métodos sincronizados.

```
class Conta
{   double conta;
    public Conta (int saldoInicial)
    {   conta = saldoInicial;
     }
    public synchronized void deposito (double valorDeposito)
    {   conta = conta + valorDeposito;
     }
    public synchronized void saque (double valorSaque)
    {   while (conta < valorSaque)  {
            try {
                wait( ) ;
            } catch (InterruptedException e) {}
      }
       conta = conta - valorSaque;
       notify ( );
     }
    public synchronized double consultaSaldo ( )
    {
            notify();
```

```
        return conta;
    }
```

11) Estenda o exercício anterior criando uma classe Banco que contém algumas contas bancárias e altere os métodos definidos para a classe conta para que sejam da classe Banco. Escreva uma classe para definir o método run de uma linha de execução que represente a instituição, que deve fazer um depósito em cada uma das contas. Escreva uma classe para definir o método run de uma linha de execução para cada correntista, que deve repetir um número de operações de saque ou consulta de saldo, dependendo de um valor aleatório. O número de operações pode ser fixo.

```
class Conta {
  private double saldo;
    void set_saldo(double saldo) {
     this.saldo = saldo;
  }

  double get_saldo() {
    return saldo;
  }
}

class Banco
{   private Conta ccorrente[ ];
    private double s;
     public Banco (int n_conta, int saldoInicial)
     {   ccorrente[ ] = new Conta[n_conta];
         for (int i = 0; i<= n_conta, i++) {
               ccorrente[i].set_saldo(saldoInicial);
            }
      }

     public synchronized void deposito (int qc, double valorDeposito)
     {   s = ccorrente[qc].get_saldo() += valorDeposito;
        ccorrente[qc].set_saldo(s);
      }
     public synchronized void saque (int qc, double valorSaque)
     {   while (ccorrente[qc].get_saldo() < valorSaque)  {
              try {
                  wait( ) ;
              } catch (InterruptedException e) {}
        }
         s = ccorrente[qc].get_saldo -= valorSaque;
         ccorrente[qc].set_saldo(s);
         notify ( );
      }
     public synchronized double consultaSaldo (int qc )
     {
           s = ccorrente[qc].get_saldo();
           notify();
           return s;
      }
class UFSCarThread extends Thread  {
   private Banco bb;
   private int n;
   private double valor;
```

```
  UFSCarThread(Banco bb, int n, double valor)  {
    this.bb = bb;
    this.n = n;
    this.valor = valor;
  }
    public void run () {
      for (int i = 0, i < n, i++) {
        try {
            bb.deposito(i,valor);
          sleep((int) (500 * Math.random()) );
          } catch (InterruptedException e) {
         }
      }
   }
}

class CorrentistaThread extends Thread  {
  private Banco bb;
  private int i_conta;
  private double valor;
  CorrentistaThread(Banco bb, int i_conta, int n, double valor)  {
    this.bb = bb;
    this.i_conta= i_conta;
    this.valor = valor;
  }
    public void run () {
      for (int i = 0, i < n, i++) {
        try {
          double d_ou_s = Math.random();
           if(d_ou_s  < 0.5)
               {bb.saque(i_conta,valor);}
           else {bb.consultaSaldo(i_conta)}
          sleep((int) (500 * Math.random()) );
          } catch (InterruptedException e) {
         }
      }
   }
}
```

12) Escreva um programa que utiliza as classes do exercício anterior para instanciar uma linha de execução da instituição e pelo menos cinco linhas para correntistas.