UNIVERSIDADE FEDERAL DE VIÇOSA
INF310 – PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA
Lista de Exercícios 1

1. Explique porque se consegue maior eficiência na utilização dos recursos quando se dá prioridade aos processos IO-bound, e não aos CPU-bound?

2. Faça uma comparação entre programação concorrente, programação paralela e sistemas distribuídos.

3. Desenhe o grafo de precedência que representa o resultado do código a seguir. Pode-se dizer que o grafo é propriamente aninhado? Demonstre.

```
void func(int id) {
    cout<<"p"<<id<<" ";
}
int main() {
    thread t1(func,1);
    thread t2(func,2);
    func(3);
    t2.join();
    thread t4(func,4);
    t1.join();
    t4.join();
    func(5);
    cout<<endl;
}
```

4. Em relação ao programa mostrado na questão anterior, mostre quais das saídas apresentadas abaixo são possíveis. Justifique.
   a) p1 p3 p2 p4 p5
   b) p2 p3 p4 p1 p5
   c) p1 p2 p4 p3 p5
   d) p3 p2 p1 p5 p4
   e) ppp123   p4 p5

5. Quais das alternativas a seguir representam uma saída válida para o código abaixo? Justifique.

```
int main() {
    if (fork() == 0){
        cout<<"1";
        if (fork() == 0)
            cout<<"2";
        else
            cout<<"3";
    }
    else
        cout<<"4";
    if (fork() == 0)
        cout<<"5";
    else
        cout<<"6";
    return 0;
}
```

a) 123456
b) 14235656
c) 1234555666
d) 4321565656
e) 12233445566
f) 1251264546135136

6. O que é condição de corrida? Ela pode acontecer tanto em arquiteturas monoprocessadas quanto multiprocessadas? Justifique.

7. O que caracteriza um mecanismo de exclusão mútua com espera ocupada? Qual o problema com essa abordagem?

8. Analise os algoritmos de exclusão mútua para 2 threads propostas a seguir e responda se são válidos. Se não for válido, mostre quais princípios são violados e justifique. Em todos os casos, “vez” e “quer” são variáveis globais onde “quer” é um array com todas as posições iniciadas como false e vez é iniciada como 0.

a) Código 1

```
...
quer[eu]=true;
while(vez==outro) {
  while (quer[outro]);
  vez=eu;
}
//REGIÃO CRÍTICA
vez=outro;
quer[eu]=false;
...
```

b) Código 2

```
...
quer[eu]= true;
while(quer[outro])
  if(vez==outro) {
    quer[eu]=false;
    while (vez!=eu);
    quer[eu]=true;
  }
//REGIÃO CRÍTICA
vez=outro;
quer[eu]=false;
...
```

9. Mostre um caso de uso para cada um dos 3 tipos de sincronização entre threads. DICA: Não é necessário mostrar o código, apenas identifique e explique uma situação real em que cada sincronização poderia ser utilizada.

10. Escreva um programa multithread em C++ onde as tarefas executadas sigam a especificação de concorrência dada através de funções S e P a seguir. A sincronização entre as threads deve ser realizada **apenas através de join**, ou seja, neste exercício as operações mutexbegin, mutexend, block e wakeup **não** são permitidas. Por simplicidade, considere que cada tarefa consiste apenas na impressão do número correspondente. Quando possível, você pode fazer com que uma mesma thread execute mais de uma tarefa, desde que a sincronização siga a especificação abaixo.

```
S(P(S(P(t1,t2),t3),t4),t5)
```

11. A relação de precedência entre um conjunto de 5 threads é mostrada abaixo através de funções P e S. Dado que Tn é o conjunto de instruções específicas da thread n, mostre um esboço (em alto nível) do código de cada thread utilizando os comandos block/wakeup(n) para sincronização.

```
S(P(S(T1, P(T2,T3)),T4),T5)
```

12. Considerando que a função **f**, a seguir, implementa o código a ser executado por várias threads de forma paralela, analise sua implementação e aponte os erros encontrados. Para cada erro, você deverá inserir estruturas e operações de sincronização de threads para propor soluções eficientes para os erros, **sem modificar** as linhas já implementadas. Em outras palavras, indique quais operações de sincronização e em quais linhas elas devem ser inseridas (utilize como referência os números de linha já disponíveis na figura). Para cada operação, explique como ela poderá resolver o erro apontado.

```
 1| void f(int tid, int num_threads, int *dados, int tam, int *res, int &n) {
 2|     int ini = tid * tam / num_threads;
 3|     int fim = (tid+1) * tam / num_threads;
 4|     int soma= 0;
 5|
 6|     for (int i=ini; i<fim; i++) {
 7|         soma += dados[i];
 8|     }
 9|
10|     res[n++]=soma; //soma de cada partição (não necessariamente ordenado)
11|
12|     if (tid == 0) {
13|         n=0;
14|         for (int i=0; i<num_threads; i++) {
15|             n += res[i];
16|         }
17|     }
18|
19|     for (int i=ini; i<fim; i++) {
20|         dados[i] = n-dados[i]; //soma final dos dados menos o valor original
21|     }
22| }
23|
24| int main (){
25|     //... (inicialização de variáveis tam_dados, num_threads, etc)
26|     int *dados = new int[tam_dados];
27|     int *res = new int[num_threads];
28|     int n=0;
29|     inicializa_dados(dados, tam_dados); //código não exibido
30|
31|     vector<thread> threads;
32|     for (int i=0; i<num_threads; ++i)
33|        threads.push_back(thread(f,i,num_threads,dados,tam_dados,res,ref(n)));
34|
35|     //... (finalização das threads, deslocamento de arrays, etc.)
36| }
```

13. Considerando que `thread1` e `thread2` a seguir são threads concorrentes, atualize o código das mesmas de modo a satisfazer as seguintes restrições:
    - A qualquer momento, $0 \leq x \leq 10$
    - Qualquer uma das threads pode atualizar **`x`** quando seu valor estiver no intervalo [1,9]
    - Não deve existir condição de corrida na atualização de x
    - O uso de espera ocupada não é permitido

    Considere que as operações **`block()`** e **`wakeup(id)`** já estão implementadas e disponíveis para uso, sendo que **`id`** é o identificador da thread a ser acordada. O **`mutex`** também está disponível para uso. Considere que o **`id`** das threads são 1 e 2, respectivamente.

```
int x = 0; //variável global

void thread1() {
    while(true) {
        x = x+1;
    }
}

void thread2() {
    while(true) {
        x = x-1;
    }
}
```