INE 5645 – PROGRAMAÇÃO PARALELA E DISTRIBUIDA – PROVA 2 – 12/06/2017

ALUNO _____

1. **Sockets - Indicar (Verdade/Falso):**
   (a) (Verdade/Falso) *Sockets* são abstrações utilizadas nos protocolos de comunicação UDP e TCP, que implementam pontos de interação para comunicação entre processos.
   (b) (Verdade/Falso) *Sockets* só podem ser usados em Linux.
   Os sockets são originários do UNIX BSD,mas também estão presentes na maioria das versões UNIX, no Linux, no Windows e no Macintosh.
   (c) (Verdade/Falso) A comunicação entre processos consiste em transmitir uma mensagem entre o *socket* de um processo e o socket de outro processo.
   (d) (Verdade/Falso) Para que um processo receba mensagens, seu *socket* deve estar vinculado ao IP de um computador e a uma porta local desse computador em que é executado.
   (e) (Verdade/Falso) As mensagens enviadas para um endereço IP e uma porta específica, só podem ser recebidas por um processo cujo *socket* esteja associada a esse IP e a esse número de porta.
   (f) (Verdade/Falso) Processos podem usar o mesmo *socket* para enviar e receber mensagens.
   Basta ver os exemplos UDP, TCP e Multicast Socket.
   (g) (Verdade/Falso) Qualquer processo pode fazer uso de várias portas para receber mensagens.
   Uma mensagem da aplicação pode ser enviada em vários pacotes (a mensagem pode ser segmentada). O TCP cuida para que os pacotes recebidos sejam remontados no host de destino na ordem correta (caso algum pacote não tenha sido recebido, o TCP requisita novamente este pacote). Somente após a montagem de todos os pacotes é que as informações ficam disponíveis para nossas aplicações.
   (h) (Verdade/Falso) Um processo não pode compartilhar portas com outros processos no mesmo computador.
   Para não haver conflito entre aplicações. Cada aplicação tem sua porta estabelecida como um ponto de interação entre a aplicação e a camada de transporte. Por exemplo, a porta padrão para FTP é 21, a do SSH, porta 22, ...
   (i) (Verdade/Falso) Os processos que usam *IP Multicast* são uma exceção, pois esses compartilham portas.
   Um *DatagramSocket* pode ser usado para enviar e receber pacotes IP Multiccast. Exemplo, um *Multicast Server* que usa a porta 5000 para enviar e receber pacotes *IP Multicast*.
   (j) (Verdade/Falso) Qualquer número de processos podem enviar mensagens para a mesma porta.
   Ver o caso acima, da questão (i), no caso dos membros em um IP Multicast.

2. **(Formas de Escalonamento em Java)**

   No código seguinte,

```
1. ExecutorService executar_leitor Executors.newFixedThreadPool(4);
2. ScheduledExecutorService executar_escritor =
                                Executors.newScheduledThreadPool(1);
```

```
      ..............................
      try
      {
3.  executar_leitor(new Leitor( sharedLocation ));
4.  executar_escritor.scheduleAtFixedRate(new
                Escritor(sharedLocation),0,1,TimeUnit.MILLISECONDS)
      }
      ..................................
```

(a) Qual a diferença entre `ExecutorService e ScheduledExecutorService?`

O ExecutorService utiliza o tempo default do processador para escalonar. No ScheduledExecutorService, o programador pode estabelecer um tempo apropriado ...

(b) O que se pode afirmar sobre o **estado das threads** Leitor e Escritor, quando o trecho da linha 4 for executado ?

Ao serem criadas, as threads leitoras e escritoras passam do estado NEW THREAD para o estado de PRONTAS para executar.

(c) (Verdade/Falso) No código da linha 1, as threads no pool são escalonadas em paralelo.

O correto é o próximo item.

(d) (Verdade/Falso) No código da linha 2, as threads no pool são escalonadas concorrentemente.

Multithreading em Java trata de concorrência. A execução de uma thread é a execução de um programa sequencial.

3. **Controle de Concorrência – Semáforo**

Considere a seguinte a definição de semáforo binário S. Seja o pseudo-código:

```
S: Semaphore := 1;

Thread T1 is begin
loop
   Non_Critical_Section;
   wait(S);
   Critical_Section_1;
   signal(S);
   Non_Critical_Section;
end loop;
End T1;

Thread T2 is begin
loop
   Non_Critical_Section;
   wait(S);
   Critical_Section_2;
   signal(S);
   Non_Critical_Section;
end loop;
End T2;
```

Considerando níveis de prioridade de execução de 1 a 10, a Thread T1 é de menor prioridade e Thread T2, a de maior prioridade. Indique a resposta (E) errada e (C) correta.

(E) A Thread T1 entra no processador e executa até seu final. Em seguida a Thread T2 é executada.
(E) A Thread T1 entra no processador, ela é interrompida, e em seguida Thread T2 é executada e interrompida antes de chegar ao seu final.
(C) Se a Thread T1 entra no processador, ela é interrompida, e em seguida Thread T2
é escalonada e executada até ao seu final.
(C) O esquema de escalonamento fundamental para threads é preemptivo (o ato de forçar uma thread parar sua execução), ou seja, baseado em prioridade.
(E) *Starvation* ocorrerá neste cenário.

4. Componentes de processos e threads

(Verdade/Falso) Itens propriedade de **threads** são: (a) Espaço de endereçamento, (b) Variáveis globais, (c) Contador de programa lógico, (d) Registradores, (e) Pilha, (f) Estado, (g) Recursos.

(a) Variáveis globais, (b) Contador de programa lógico, (c) Registradores, (d) Pilha, (e) Estado.

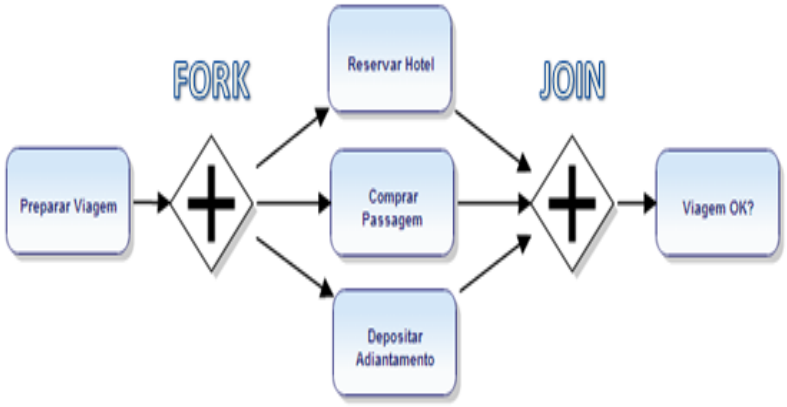
Threads usam os recursos do processo onde elas executam.

(Verdade/Falso) Itens propriedade de **processos** são: (a) Variáveis globais, (b) Contador de programa lógico, (c) Registradores, (d) Pilha, (e) Estado.

(a) Espaço de endereçamento, (b) Variáveis globais, (c) Contador de programa lógico, (d) Registradores, (e) Pilha, (f) Estado, (g) Recursos.

Processos tem espaço de endereçamento. Threads num processo usam o espaço de endereçamento do processo onde elas são executadas.

5. OpenMP (Modelo de Programação)



(a) No exemplo acima, como se denomina o modelo de programação do OpenMP ?

O modelo de programação do OpenMP chama-se FORK-JOIN.

(b) (Verdade/Falso) – A região paralela indicada representa uma região paralela contendo seções paralelas.

Neste caso, a região paralela conterá threads de códigos diferentes, implementadas neste caso, em seções paralelas dentro da região paralela.

6. OpenMp -

(a) Sobre o código seguinte pode afirmar (SIM/NÃO)

```
#define N 10000;
int i;
#pragma omp parallel
   #pragma omp for
        for (i=0 ; i < 10000 ; i++) {
            calculo();
        }
        printf("Terminado");
```

(1) As iterações são distribuídas entre as threads ? SIM.
(2) Tem uma barreira implícita de sincronização entre threads no final do loop ? SIM.
(3) *omp for* pode ser complementado pela schedule para especificar como fazer a distribuição da carga do *for* (i=0 ; i < 10000 ; i++) ? SIM.

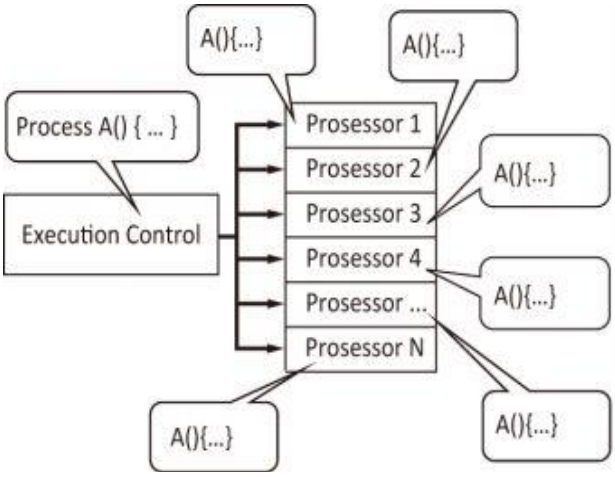
(b) Vimos que o OpenMP irá particionar automaticamente as iterações de um loop *for*. Como podemos otimizar a forma como as iterações de loop são divididas. No código abaixo, indique no pontilhado, qual tipo de schedule é mais apropriado ?

```
#define THREADS 4
#define N 16
int main ( ) {
 int i;
```

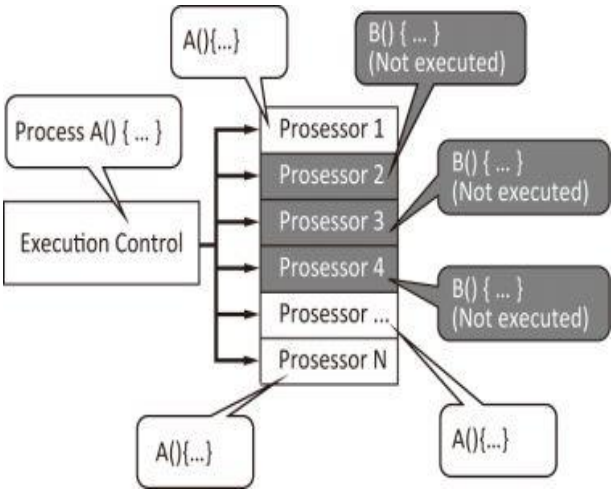
```
#pragma omp parallel for schedule(dynamic) num_threads(THREADS)
for (i = 0; i < N; i++) {
   /* wait for i seconds */
   sleep(i);

   printf("Thread %d has completed iteration %d.\n",
                                          omp_get_thread_num( ), i);
  }
  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

Veja o link que está na página da disciplina:
http://cs.umw.edu/~finlayson/class/fall16/cpsc425/notes/12-scheduling.html

7. OpenCL – Dadas as figuras abaixo, o que significam cada uma delas ?



Resposta:__Representa o uso eficiente de GPU para paralelismo de dados. Exemplifica o uso da arquitetura SIMD (Single Instruction Multiple Data). Quando vários processadores executam a mesma tarefa, o número de tarefas, igual ao número de processadores, pode ser executado ao mesmo tempo, de uma vez.



Resposta:__ Representa o uso ineficiente de GPU para paralelismo de dados. As GPUs funcionam muito bem para paralelismo de dados (códigos iguais) e são incapazes de

executar tarefas de códigos diferentes em paralelo, ao mesmo tempo. A figura mostra o caso em que tarefas A e B estão programadas para serem executadas em paralelo na GPU. Como processadores só podem processar o mesmo conjunto de instruções nos núcleos (códigos iguais, a unidade de controle envia uma única instrução), os processadores agendados para processar a Tarefa B (códigos diferentes de A) devem estar no modo inativo até que a Tarefa A esteja concluída (quando a unidade de controle poderá enviar um outra instrução para a tarefa B).

(E) (**Paralelismo de Tarefas**) Uma técnica de programação que divide uma quantidade de dados em partes menores que podem ser operadas em paralelo. A mesma operação é executada simultaneamente (isto é, em paralelo).

(C) (**Paralelismo de Dados**) Uma técnica de programação que divide uma grande quantidade de dados em partes menores que podem ser operadas em paralelo. A mesma operação é executada simultaneamente (isto é, em paralelo).

(E) As **arquiteturas** apropriadas para *Paralelismo de dados* é "SIMD" e para *Paralelismo de tarefas* é "MIMD".

(E) **SIMD** significa que as unidades paralelas têm instruções distintas, então cada uma delas pode fazer algo diferente em um dado momento.

(C) **SIMD** significa que todas as unidades paralelas compartilham a mesma instrução, mas a realizam em diferentes elementos de dados.

(C) No **modelo de paralelismo de tarefas**, para cada operação a ser executada, deve ser definido um kernel para executar uma determinada operação na arquitetura MIMD.

8. **OpenCL -** Dê um exemplo, envolvendo uma operação sobre números, que seja executada com paralelismo de dados.

   Seja a sequência de números inteiros:

   1, 2, 3, 4, 5, 6, 7, 8, 9

   Tome a operação da raiz quadrada de cada valor, A operação é a mesma (a raiz quadrada), mas o dado (o valor) é diferente.

9. **OpenCL**

   (a) No código abaixo, escrito em C, **quantos núcleos** são utilizados, se um processador quad-core for utilizado no processamento ?

```
void ArrayDiff(const int* a, const int* b, int* c, int n)
{
```

```
  for (int i = 0; i < n; ++i)
  {
    c[i] = a[i] - b[i];
  }
}
```

Resposta: _1 núcleo, apenas, pois o código é sequencial, e um programa sequencial utiliza apenas um núcleo._____

(b) Um objeto de programa encapsula o *código-fonte de um kernel*, sendo este identificados no código-fonte por meio da palavra-chave __kernel. O que significa, o seguinte código, executado em OpenCL ?

```
__kernel void ArrayDiff (
   __global const int* a,
   __global const int* b,
   __global int* c )
{
  int id = get_global_id(0);
  c[id] = a[id] - b[id];
}
```

Na verdade, no caso em tarefas estão executando em um único núcleo de uma CPU, essas não podem ser executados em paralelo. Neste caso, o aplicativo ou o sistema operacional (o schedule) devem alternar entre as tarefas, permitindo tanto tempo para executar no núcleo (pseudo-paralelismo), existindo o que se chama de concorrência.

O paralelismo diz respeito à execução de duas ou mais atividades em paralelo (paralelismo verdadeiro) com o objetivo explícito de aumentar o desempenho no processamento. É o que acontece com os kernels do OpenCL.

O código kernel em representa o código que é paralelizado.

Veja a explicação. A unidade de execução simultânea no OpenCL C é um item de trabalho (work-item). Tal como acontece com o exemplo acima, cada item de trabalho executa o corpo da função do kernel. Em vez de se executar o loop, muitas vezes, num work-item, mapeia-se uma única iteração do loop para um item de trabalho. Conceitualmente, isso é muito parecido com o paralelismo inerente a uma operação do paralelismo de dados para um loop em um modelo como o OpenMP. Quando um dispositivo OpenCL começa a executar um kernel, ele fornece funções intrínsecas que permitem que um item de trabalho se identifique. No código, a chamada para *get_global_id (0),* permite ao programador fazer uso da posição do work-item corrente, no caso para recuperar o contador do loop.