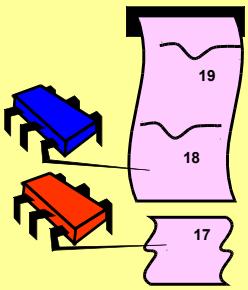
Alguns problemas de sistemas concorrentes

Balanceamento de carga



Contador compartilhado: cada thread pega um número

Fonte: Art of Multiprocessor Programming (rev. ed.), M. Herlihy & N. Shavit

Números primos

Desafio

Imprimir os números primos de 1 a 10^{10}

Dados

Multiprocessador com dez núcleos

Uma thread por processador

Tempo para imprimir desprezível

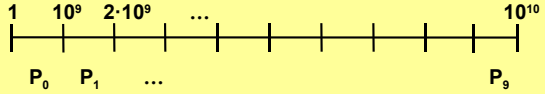
Objetivo

Speedup de 10 vezes

Um exemplo de solução

```
Counter counter = new Counter(1);
...
void primePrint {
  long j = 0;
  while (j < 10¹⁰) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Uma solução estática...



- dividir igualmente os intervalos

- **Problemas:**
  - Números primos ficam mais raros à medida que são maiores
  - Números grandes são mais difíceis
  - *Workload* irregular e difícil de prever

Um exemplo de solução

```
Counter counter = new Counter(1);//compartilhado!
...
void primePrint {
  long j = 0;
  while (j < 10¹⁰) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Implementação de `Counter`

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

Nomes aos bois

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    long temp = value;
    value = temp + 1;
    return temp;
  }
}
```

Região crítica (onde podem ocorrer **condições de corrida**)

Implementação de `Counter`

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

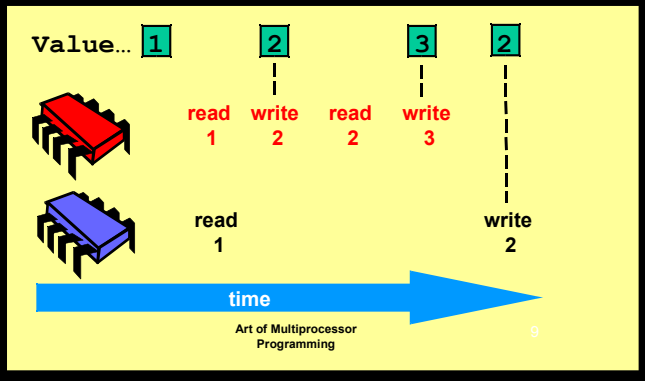
```
long temp = value;
value = temp + 1
return temp;
```

Incremento precisa ser atômico

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    long temp;
    synchronized(this) {
      temp = value;
      value = temp + 1;
    }
    return temp;
  }
}
```

O que pode acontecer...



Incremento precisa ser atômico

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    long temp;
    synchronized(this) {
      temp = value;
      value = temp + 1;
    }
    return temp;
  }
}
```

Garante **Exclusão Mútua** (só uma thread entra aí de cada vez)

Erros de consistência de memória

Divergências entre o que está na memória principal e na *cache*

- Ver a classe `Laco.java`

Algoritmo de Peterson generalizado

(também conhecido como o Algoritmo do Filtro)

```
level : array of N integers
last_to_enter : array of N-1 integers

for ℓ from 0 to N-1 exclusive
    level[i] ← ℓ
    last_to_enter[ℓ] ← i
    while last_to_enter[ℓ] = i
       and there exists k ≠ i,
           such that level[k] ≥ ℓ
        wait
```

Fonte: https://en.wikipedia.org/wiki/Peterson%27s_algorithm

Exercício

Construa uma classe que implementa uma fila segura para um número indeterminado de threads que funciona da seguinte maneira:

- Se apenas uma thread tentar inserir ou remover um elemento, ela consegue
- Se mais que uma estiver tentando ao mesmo tempo, uma consegue e as outras esperam. A próxima só conseguirá realizar a operação quando a anterior tiver terminado.

Sobre o algoritmo do filtro

Memória necessária cresce linearmente com número de threads

- Se houver muitas threads e travas, pode ser proibitivo

Complexidade de tempo de execução também

Necessário saber número máximo de threads a priori

Exercício

Construa uma classe que implementa uma fila segura para um número pré-estabelecido *N* de threads e que funciona da seguinte maneira:

- Se apenas uma thread tentar inserir ou remover um elemento, ela consegue
- Se mais que uma estiver tentando ao mesmo tempo, uma consegue e as outras esperam. A próxima só conseguirá realizar a operação quando a anterior tiver terminado.

Não é permitido usar blocos `synchronized` (e similares, como locks) para fazer esse exercício

E não falamos nada sobre desempenho...