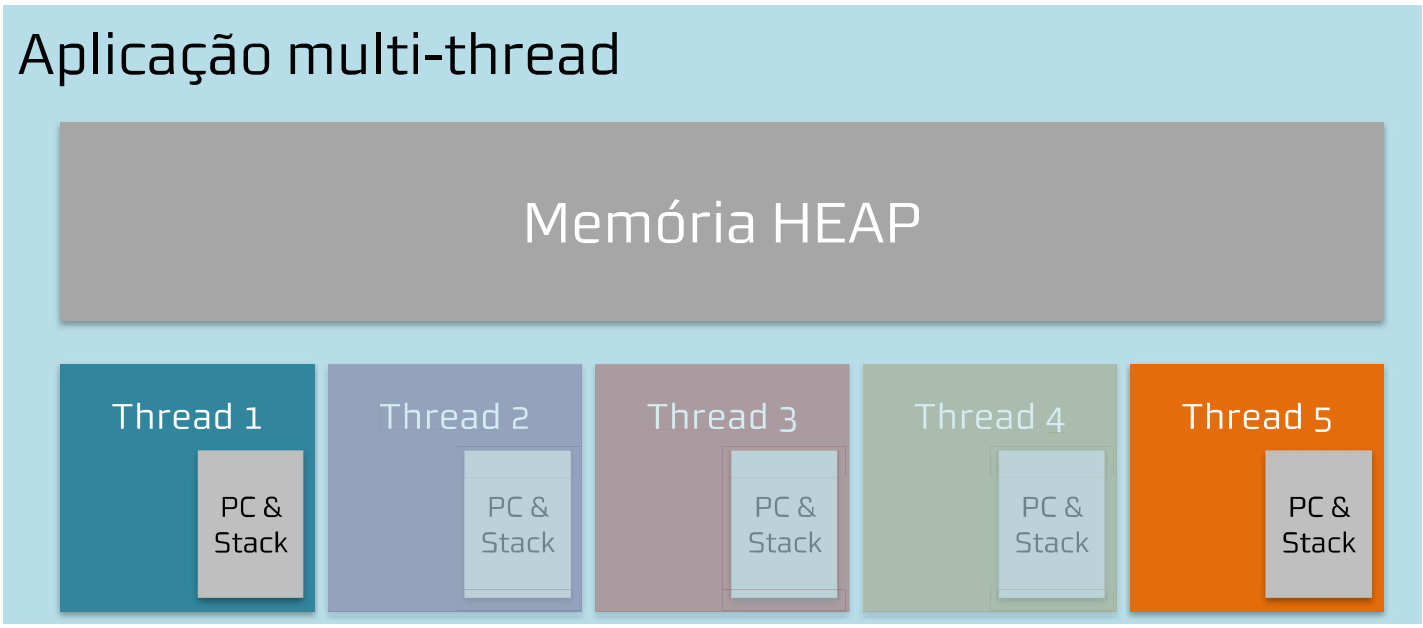
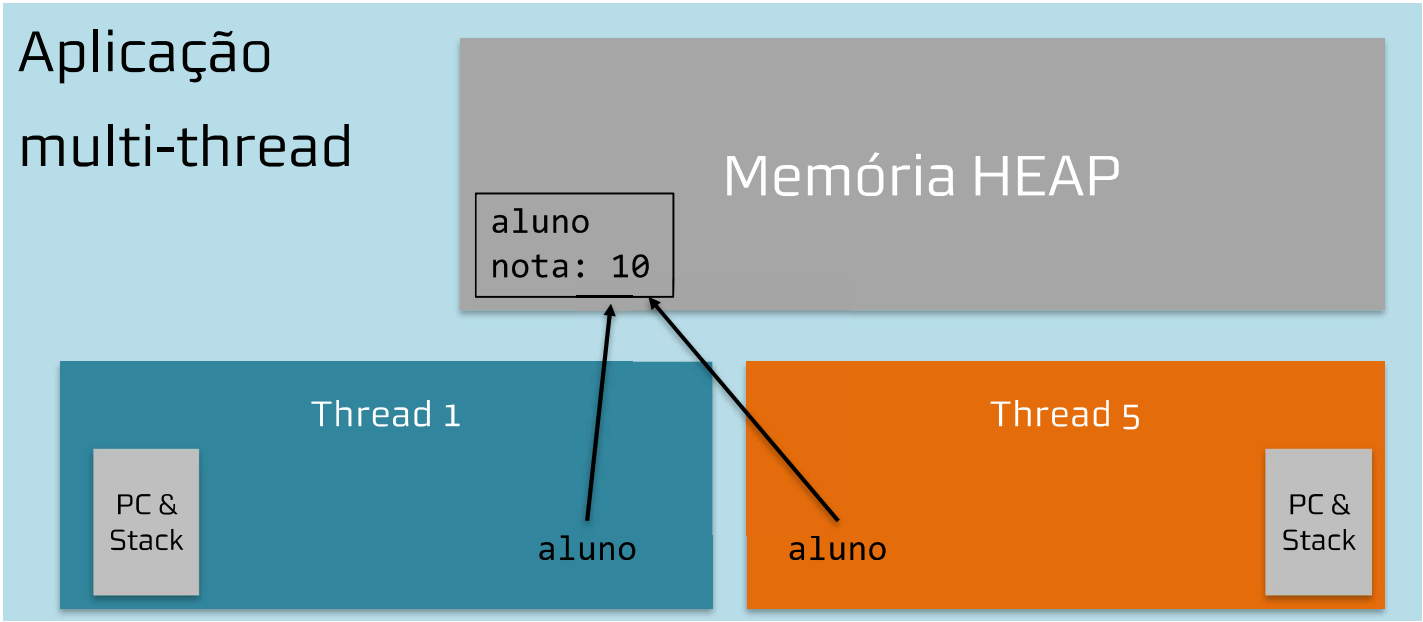
Sumário

- Race condition
- Interferência entre threads
- Inconsistência no acesso à memória
- Sincronização
- Cadeados intrínsecos
- Métodos e blocos sincronizados

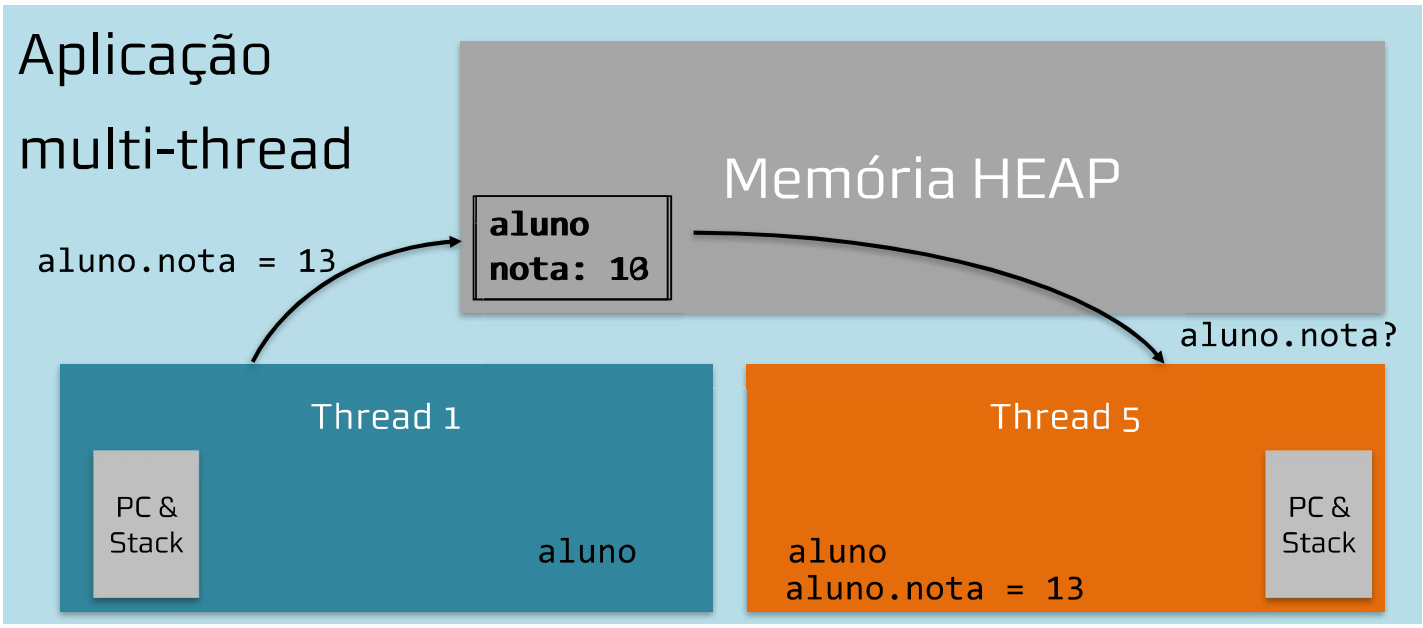
Partilha de Memória



Partilha de Memória



Partilha de Memória



Partilha de referências entre threads

- Se existirem 2 ou mais threads a ordem por que são executadas as diversas ações das threads varia de execução para execução.
- Podem ser gerados resultados diferentes cada vez que é executado o programa devido à ordem porque são executadas as instruções

Race Condition

- Quando o valor de uma variável, ou o resultado do programa, varia devido à alteração da ordem de execução das ações das diferentes threads temos um problema chamado “race condition”.

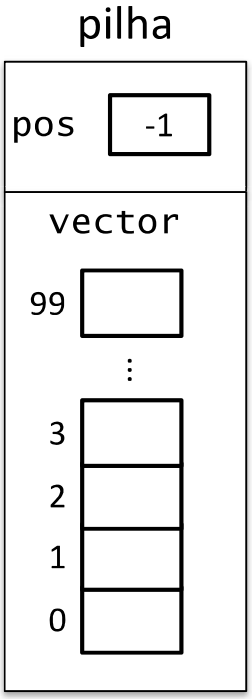
Pilha de Inteiros

```
class Pilha {
	private int pos = -1;
	private int vector[]= new int[100];

	public void push(int i) {
		pos++;
		vector[pos]=i;
	}

	public int pop() {
		if (pos>=0){
			return(vector[pos--]);
		} else
			throw new IllegalStateException() ;
	}

}
```

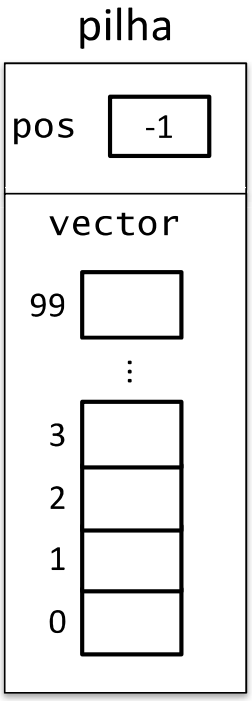


Pilha de Inteiros

Thread A
```
pilha.push(5);
```

Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

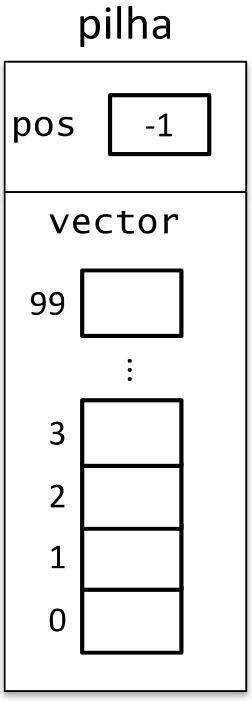


Pilha de Inteiros

Thread A
```
pilha.push(5);
```
STOP

Thread B
```
pilha.push(7);
```
STOP

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

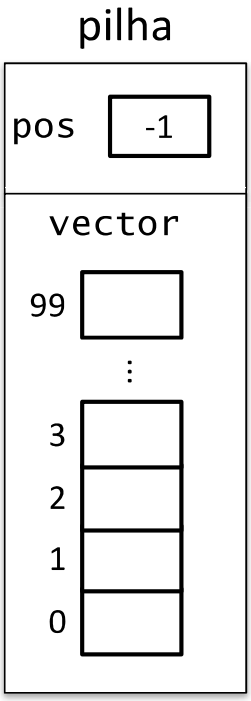


Pilha de Inteiros

Thread A
```
pilha.push(5);
```

Thread B
```
pilha.push(7);
```


```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```



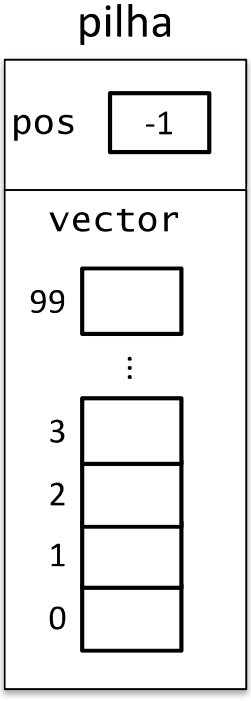
ISCTE INSTITUTO UNIVERSITÁRIO DE LISBOA SINCRONIZAÇÃO

Pilha de Inteiros

Thread A
```
pilha.push(5);
```

Thread B
```
pilha.push(7);
```


```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

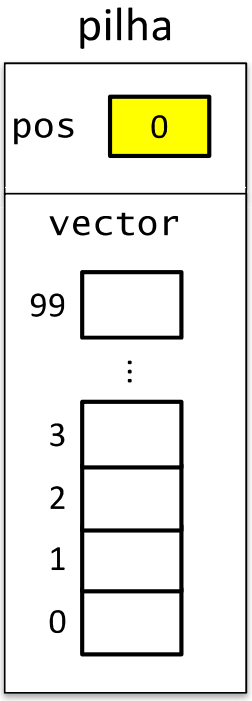


Pilha de Inteiros

Thread A
```
pilha.push(5);
```

Thread B
```
pilha.push(7);
```
STOP

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

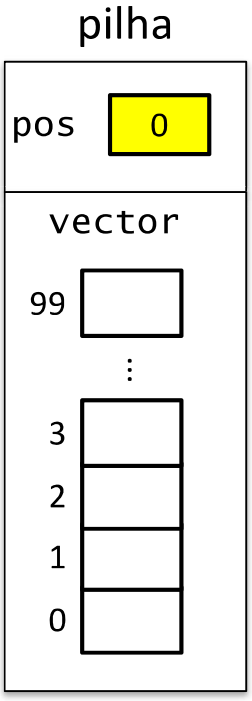


Pilha de Inteiros

Thread A
```
pilha.push(5);
```
STOP

Thread B
```
pilha.push(7);
```
STOP

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

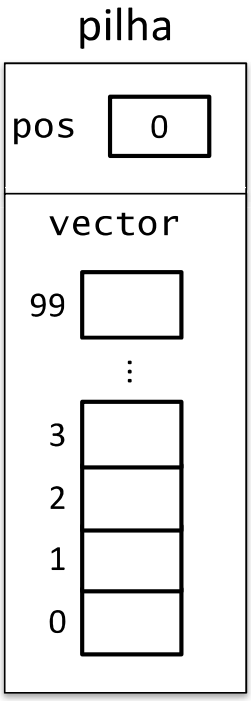


Pilha de Inteiros

Thread A
```
pilha.push(5);
```


Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

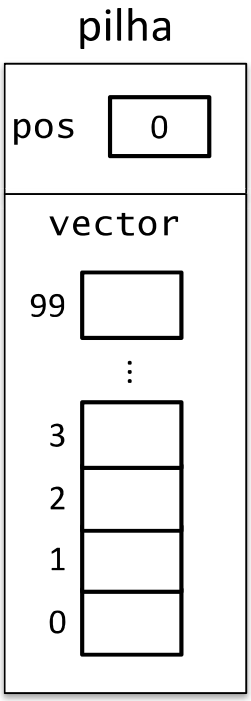


Pilha de Inteiros

Thread A
```
pilha.push(5);
```


Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

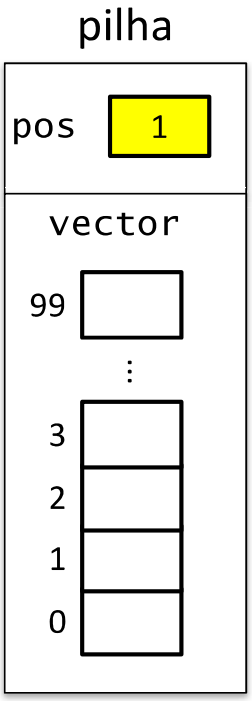


Pilha de Inteiros

Thread A
```
pilha.push(5);
```
STOP

Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

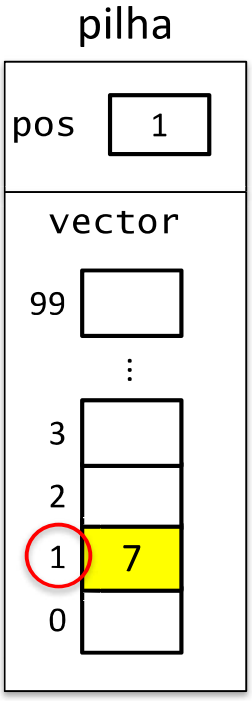


Pilha de Inteiros

Thread A
```
pilha.push(5);
```
STOP

Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

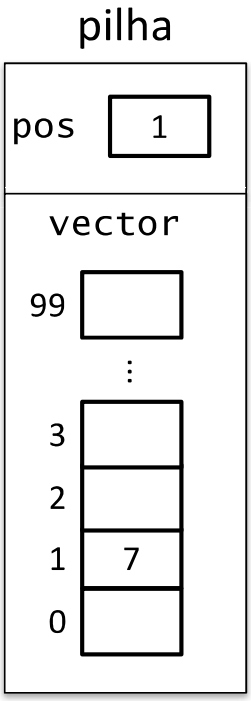


Pilha de Inteiros

Thread A
```
pilha.push(5);
```


Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

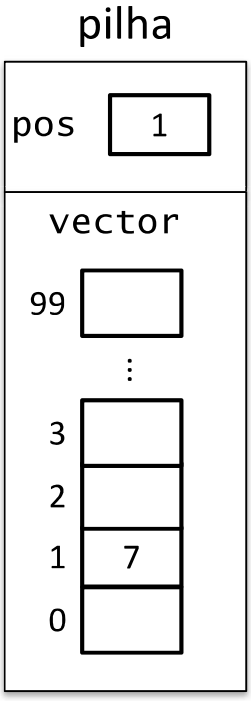


Pilha de Inteiros

Thread A
```
pilha.push(5);
```

Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

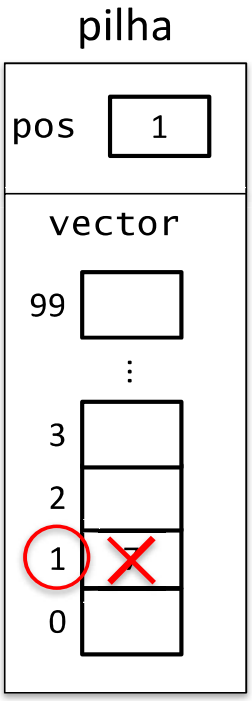


Pilha de Inteiros

Thread A
```
pilha.push(5);
```

Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```



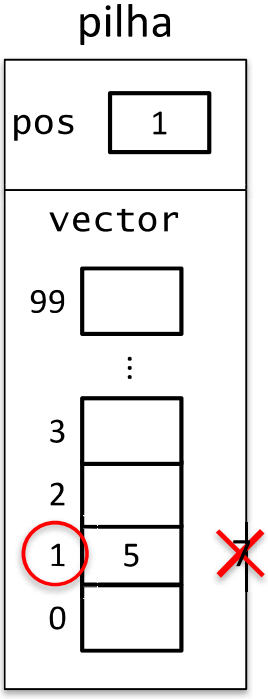
ISCTE INSTITUTO UNIVERSITÁRIO DE LISBOA SINCRONIZAÇÃO

Pilha de Inteiros

Thread A
```
pilha.push(5);
```

Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

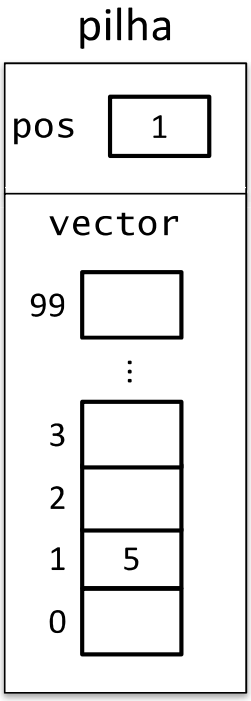


Pilha de Inteiros

Thread A
```
pilha.push(5);
```

Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```



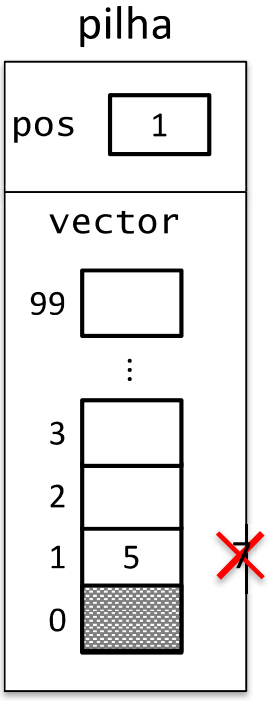
Pilha de Inteiros

Thread A
```
pilha.push(5);
```

Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

Resultado:
- a posição 0 da pilha contém lixo
- perde-se o valor 7



Interferência entre Threads

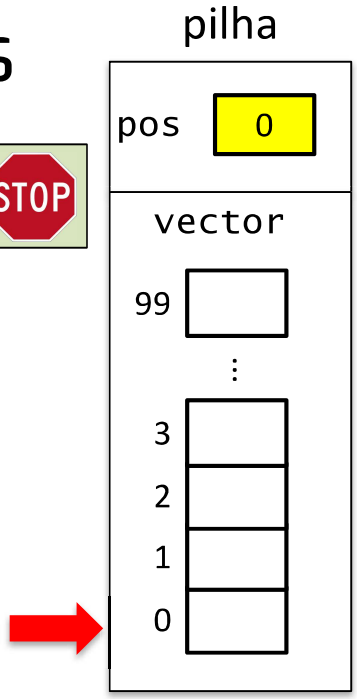
- Dizemos que temos interferência entre duas threads quando as suas ações sobre determinados dados não são atómicas.
- Ações não atómicas são divididas em ações atómicas. Assim uma thread pode ser intercalada ou executada em paralelo com outra thread e aceder aos dados num estado intermédio ou incompleto.

Pilha de Inteiros

Thread A
```
pilha.push(5);
```
STOP

Thread B
```
pilha.push(7);
```
STOP

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

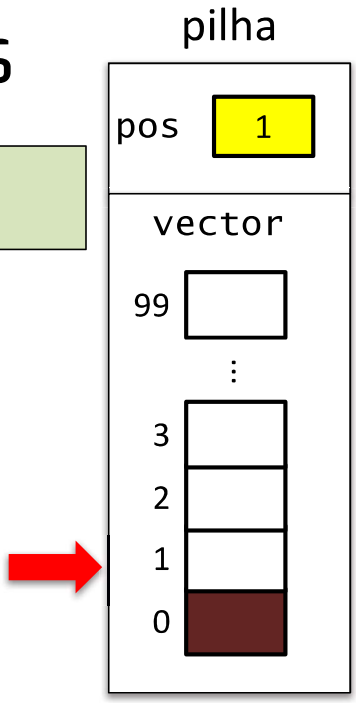


Pilha de Inteiros

Thread A
```
pilha.push(5);
```
STOP

Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

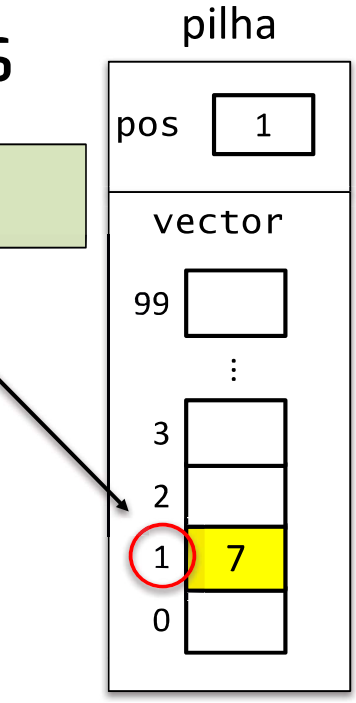


Pilha de Inteiros

Thread A
```
pilha.push(5);
```
STOP

Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```

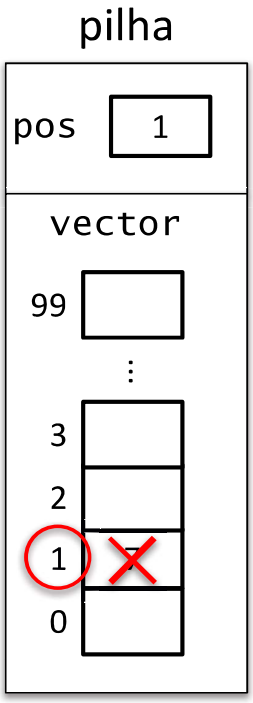


Pilha de Inteiros

Thread A
```
pilha.push(5);
```

Thread B
```
pilha.push(7);
```

```
public void push(int i) {
    pos++;
    vector[pos]=i;
}
```



Inconsistência

- Acontece quando duas threads estão a trabalhar com valores diferentes para os mesmos dados

Thread A)	Thread B)
pos++	a=pos;

- Quais são os possíveis valores de a?
- A ordem neste caso é muito relevante.

Secção Crítica

- Secção critica é um conjunto de ações cujo a execução não pode ser interrompida.

```
pos++;
- - - - - - - - - - - - -
pilha[pos]=i;
```

Secção Crítica

Exclusão Mútua

- Cadeado ou Mutex é o mecanismo base da programação concorrente.

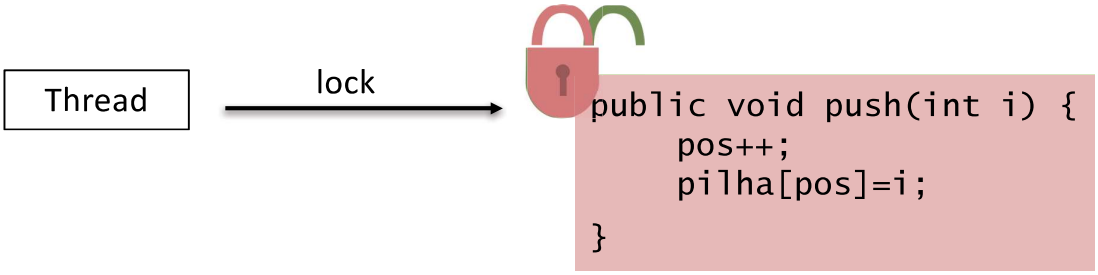
```
public void push(int i) {
    pos++;
    pilha[pos]=i;
}
```

Mutex

- Tem dois estados:
  - Livre 
  - Ocupado 
- Tem duas operações:
  - Lock 
  - Unlock 

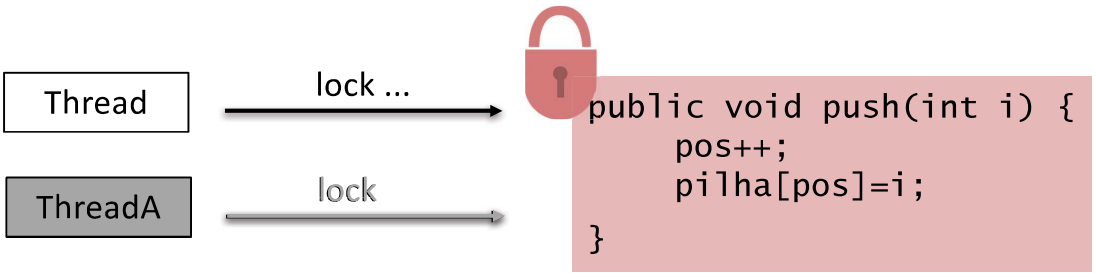
Exclusão Mútua

- O lock só pode ser feito por uma thread de cada vez.



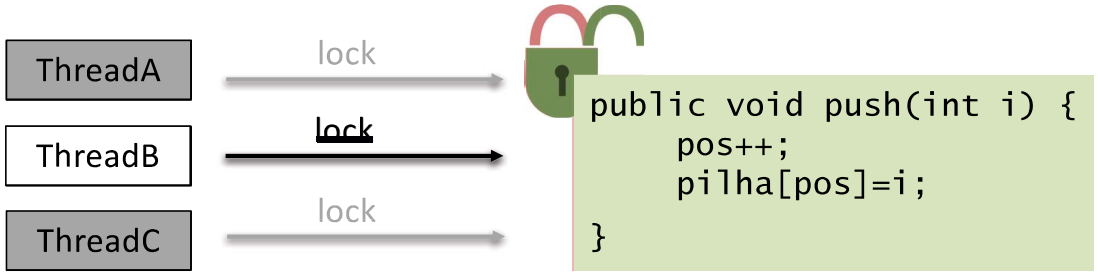
Exclusão Mútua

- Se uma thread tentar fazer lock a um mutex que está ocupado (locked por outra thread) terá de esperar até que seja feito o unlock.



Exclusão Mútua

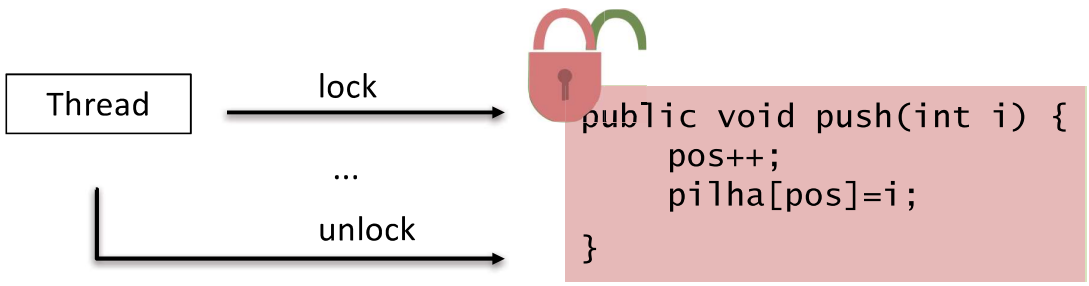
- Se várias threads estiverem à espera para fazer lock ao mutex, quando for feito o unlock, apenas uma, escolhida aleatoriamente prosseguirá.



Sincronização

- O mutex é usado para controlar o acesso às secções críticas.
- A thread deve fazer lock ao mutex que protege a secção critica antes de a executar.
- Quando terminar deve libertar o mutex fazendo o unlock.
- Assim não é possível que duas threads possam estar a executar a secção critica.

Sincronização



Sincronização em Java

- Em java todos os objetos possuem um mutex.
- Para definir uma secção critica usa-se a palavra reservada "synchronized"
- Em Java podem sincronizar-se:
  - métodos
  - bloco
- É da responsabilidade do programador definir as secções criticas.

Métodos Sincronizados

- Nos métodos sincronizados a sincronização é feita com o mutex do próprio objeto.

```
public synchronized void push(int i) {
    pos++;
    pilha[pos]=i;
}
```

Lock (this)

UnLock (this)

Métodos Sincronizados

```
class Pilha {
      private int pos = -1;
      private int pilha[]= new int[100];

      public synchronized void push(int i) {
            pos++;
            pilha[pos]=i;
      }

      public synchronized int pop() {
            if (pos>=0){
                  return(pilha[pos--]);
            } else
                  throw new IllegalStateException() ;
      }

}
```


Blocos Sincronizados

- Nos blocos sincronizados é necessário especificar o objeto cujo o mutex será usado para sincronizar.

```
private LinkedList<String> employees= …
public void addEmploye(String name){
    synchronized(employees) {
        employees.add(name);
    }
    …
}
```

Lock (employees)

UnLock (employees)

Blocos Sincronizados

- Nos blocos sincronizados é necessário especificar o objeto cujo o mutex será usado para sincronizar.

```
public void addName(String name) {
    synchronized(client) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Lock (client)

UnLock (client)

Interface do Mutex

void lock()	Acquires the lock.
boolean tryLock()	Acquires the lock only if it is free at the time of invocation.
void unlock()	Releases the lock.

```
Lock l = ...;
l.lock();
try {
// access the resource protected by this lock
} finally {
    l.unlock();
}
```

Sumário

- Partilha de memória
- Race Condition
- Interferência entre Threads
- Sincronização
- Sincronização emJava