Concurrent programming

Madhavan Mukund, **S P Suresh**

Programming Language Concepts
Lecture 11, 18 February 2025

Concurrent programming

- Multiprocessing
  - Single processor executes several computations “in parallel”
  - Time-slicing to share access

Concurrent programming

- Multiprocessing
  - Single processor executes several computations "in parallel"
  - Time-slicing to share access
- Logically parallel actions within a single application
  - Clicking `Stop` terminates a download in a browser
  - User-interface is running in parallel with network access

Concurrent programming

- Multiprocessing
    - Single processor executes several computations "in parallel"
    - Time-slicing to share access
- Logically parallel actions within a single application
    - Clicking `Stop` terminates a download in a browser
    - User-interface is running in parallel with network access

- **Process**
    - Private set of local variables
    - Time-slicing involves saving the state of one process and loading the suspended state of another

Concurrent programming

- Multiprocessing
  - Single processor executes several computations "in parallel"
  - Time-slicing to share access
- Logically parallel actions within a single application
  - Clicking `Stop` terminates a download in a browser
  - User-interface is running in parallel with network access

- **Process**
  - Private set of local variables
  - Time-slicing involves saving the state of one process and loading the suspended state of another
- **Threads**
  - Operate on same local variables
  - Communicate via "shared memory"
  - Context switches are easier

Concurrent programming

- Multiprocessing
  - Single processor executes several computations "in parallel"
  - Time-slicing to share access
- Logically parallel actions within a single application
  - Clicking `Stop` terminates a download in a browser
  - User-interface is running in parallel with network access

- **Process**
  - Private set of local variables
  - Time-slicing involves saving the state of one process and loading the suspended state of another
- **Threads**
  - Operate on same local variables
  - Communicate via "shared memory"
  - Context switches are easier
- Henceforth, we use **process** and **thread** interchangeably

Creating threads in Java

- Have a class extend `Thread`

```
public class Parallel extends Thread {
    private int id;
    public Parallel(int i){ id = i; }
}
```

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where execution can begin in parallel

```
public class Parallel extends Thread {
    private int id;
    public Parallel(int i){ id = i; }
    public void run() {
        for (int j = 0; j < 100; j++) {
            System.out.println("My id is "+id);
            try { sleep(1000);
            } catch(InterruptedException e){}
        }
    }
}
```

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where execution can begin in parallel
- Invoking `p[i].start()` initiates `p[i].run()` in a separate thread

```
public class Parallel extends Thread {
    private int id;
    public Parallel(int i){ id = i; }
    public void run() {
        for (int j = 0; j < 100; j++) {
            System.out.println("My id is "+id);
            try { sleep(1000);
            } catch(InterruptedException e){}
        }
    }
}
```

```
public class TestParallel {
    public static void main(String[] args) {
        Parallel p[] = new Parallel[5];
        for (int i = 0; i < 5; i++) {
            p[i] = new Parallel(i);
            p[i].start();   // Start p[i].run()
        }                   // in concurrent thread
    }
}
```

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where execution can begin in parallel
- Invoking `p[i].start()` initiates `p[i].run()` in a separate thread
  - Directly calling `p[i].run()` does **not** execute in separate thread!

```java
public class Parallel extends Thread {
    private int id;
    public Parallel(int i){ id = i; }
    public void run() {
        for (int j = 0; j < 100; j++) {
            System.out.println("My id is "+id);
            try { sleep(1000);
            } catch(InterruptedException e){}
        }
    }
}
```

```java
public class TestParallel {
    public static void main(String[] args) {
        Parallel p[] = new Parallel[5];
        for (int i = 0; i < 5; i++) {
            p[i] = new Parallel(i);
            p[i].start();   // Start p[i].run()
        }                   // in concurrent thread
    }
}
```

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where execution can begin in parallel
- Invoking `p[i].start()` initiates `p[i].run()` in a separate thread
  - Directly calling `p[i].run()` does **not** execute in separate thread!
- `sleep(t)` suspends thread for `t` milliseconds
  - Static function — use `Thread.sleep()` if current class does not extend `Thread`
  - Throws `InterruptedException`

```
public class Parallel extends Thread {
    private int id;
    public Parallel(int i){ id = i; }
    public void run() {
        for (int j = 0; j < 100; j++) {
            System.out.println("My id is "+id);
            try { sleep(1000);
            } catch(InterruptedException e){}
        }
    }
}
```

```
public class TestParallel {
    public static void main(String[] args) {
        Parallel p[] = new Parallel[5];
        for (int i = 0; i < 5; i++) {
            p[i] = new Parallel(i);
            p[i].start();   // Start p[i].run()
        }                   // in concurrent thread
    }
}
```

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where execution can begin in parallel
- Invoking `p[i].start()` initiates `p[i].run()` in a separate thread
  - Directly calling `p[i].run()` does **not** execute in separate thread!
- `sleep(t)` suspends thread for `t` milliseconds
  - Static function — use `Thread.sleep()` if current class does not extend `Thread`
  - Throws `InterruptedException`

Typical output

```
My id is 0
My id is 3
My id is 2
My id is 1
My id is 4
My id is 0
My id is 2
My id is 3
My id is 4
My id is 1
My id is 0
My id is 3
My id is 1
My id is 2
My id is 4
...
```

Java threads …

- Cannot always extend `Thread`
  - Single inheritance

Java threads ...

- Cannot always extend `Thread`
  - Single inheritance
- Instead, implement `Runnable`

```
public class Parallel implements Runnable {
    // only the line above has changed
    private int id;
    public Parallel(int i){ ... } // Constructor
    public void run(){ ... }
}
```

Java threads ...

- Cannot always extend `Thread`
  - Single inheritance
- Instead, implement `Runnable`
- To use `Runnable` class, explicitly create a `Thread` and `start()` it

```java
public class Parallel implements Runnable {
    // only the line above has changed
    private int id;
    public Parallel(int i){ ... } // Constructor
    public void run(){ ... }
}
```

```java
public class TestParallel {
    public static void main(String[] args) {
        Parallel p[] = new Parallel[5];
        Thread t[]   = new Thread[5];

        for (int i = 0; i < 5; i++) {
            p[i] = new Parallel(i);
            t[i] = new Thread(p[i]);
            // Make a thread t[i] from p[i]
            t[i].start();   // Start off p[i].run()
                            // Note: t[i].start(),
                            //   not p[i].start()
        }
    }
}
```

Summary

- Common to have logically parallel actions with a single application
  - Download from one webpage while browsing another
- Threads are lightweight processes with shared variables that can run in parallel
- Use `Thread` class or `Runnable` interface to create parallel threads in Java

Threads and shared variables

- Threads are lightweight processes with shared variables that can run in parallel
- Browser example: download thread and user-interface thread run in parallel
  - Shared boolean variable `terminate` indicates whether download should be interrupted
  - `terminate` is initially false
  - Clicking `Stop` sets it to true
  - Download thread checks the value of this variable periodically and aborts if it is set to true
- Watch out for **race conditions**
  - Shared variables must be updated consistently

Shared variables

- Suppose we wish to compute a function `f` on all numbers from `1` to `10000` ...

Shared variables

- Suppose we wish to compute a function `f` on all numbers from `1` to `10000` ...
- ...and split the task across `10` threads

Shared variables

- Suppose we wish to compute a function `f` on all numbers from `1` to `10000` ...
- ...and split the task across `10` threads
- Time taken by `f` on different numbers is unpredictable

Shared variables

- Suppose we wish to compute a function `f` on all numbers from `1` to `10000` ...
- ...and split the task across `10` threads
- Time taken by `f` on different numbers is unpredictable
- **Robust division of labour**

Shared variables

- Suppose we wish to compute a function `f` on all numbers from `1` to `10000` ...
- ...and split the task across `10` threads
- Time taken by `f` on different numbers is unpredictable
- **Robust division of labour**
  - A shared `Counter` object `c` (initial value `0`) that stores the next number to be processed

Shared variables

- Suppose we wish to compute a function `f` on all numbers from `1` to `10000` ...
- ...and split the task across `10` threads
- Time taken by `f` on different numbers is unpredictable
- **Robust division of labour**
  - A shared `Counter` object `c` (initial value `0`) that stores the next number to be processed

- The `Counter` class:

```
class Counter {
    int value;
    public Counter(int c) {value = c;}
    int getAndIncrement {
        int ret = value;
        value += 1;
        return ret;
    }
}
```

Shared variables

- Suppose we wish to compute a function `f` on all numbers from `1` to `10000` ...
- ...and split the task across `10` threads
- Time taken by `f` on different numbers is unpredictable
- **Robust division of labour**
  - A shared `Counter` object `c` (initial value `0`) that stores the next number to be processed
  - Each thread invokes `c.getAndIncrement` every time it is free to run `f` again

- The `Counter` class:

```
class Counter {
    int value;
    public Counter(int c) {value = c;}
    int getAndIncrement {
        int ret = value;
        value += 1;
        return ret;
    }
}
```

Shared variables

- Suppose we wish to compute a function `f` on all numbers from `1` to `10000` ...
- ...and split the task across `10` threads
- Time taken by `f` on different numbers is unpredictable
- **Robust division of labour**
  - A shared `Counter` object `c` (initial value `0`) that stores the next number to be processed
  - Each thread invokes `c.getAndIncrement` every time it is free to run `f` again

- The `Counter` class:

```
class Counter {
    int value;
    public Counter(int c) {value = c;}
    int getAndIncrement {
        int ret = value;
        value += 1;
        return ret;
    }
}
```

- Code for each thread

```
val = 0;
while (val < 10000) {
    val = c.getAndIncrement;
    f(val);
}
```

Shared variables

- Suppose we wish to compute a function `f` on all numbers from `1` to `10000` ...
- ...and split the task across `10` threads
- Time taken by `f` on different numbers is unpredictable
- **Robust division of labour**
  - A shared `Counter` object `c` (initial value `0`) that stores the next number to be processed
  - Each thread invokes `c.getAndIncrement` every time it is free to run `f` again
- Watch out for **race conditions** again

- The `Counter` class:

```
class Counter {
    int value;
    public Counter(int c) {value = c;}
    int getAndIncrement {
        int ret = value;
        value += 1;
        return ret;
    }
}
```

- Code for each thread

```
val = 0;
while (val < 10000) {
    val = c.getAndIncrement;
    f(val);
}
```

Maintaining data consistency

- `double accounts[100]` describes 100 bank accounts

Maintaining data consistency

- `double accounts[100]` describes 100 bank accounts
- Two functions that operate on `accounts`: `transfer()` and `audit()`

```
boolean transfer (double amount,
                  int source, int target) {
    if (accounts[source] < amount) {
        return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}
double audit() {
    // total balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++) {
        balance += accounts[i];
    }
    return balance;
}
```

Maintaining data consistency

- `double accounts[100]` describes 100 bank accounts
- Two functions that operate on `accounts`: `transfer()` and `audit()`
- What are the possibilities when we execute the following?

Thread 1	Thread 2
`...`	`...`
`status = transfer(500.00,7,8);`	`System.out.print(audit());`
`...`	`...`

```
boolean transfer (double amount,
                  int source, int target) {
    if (accounts[source] < amount) {
        return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}
double audit() {
    // total balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++) {
        balance += accounts[i];
    }
    return balance;
}
```

Maintaining data consistency ...

- What are the possibilities when we execute the following?

```
Thread 1                        Thread 2
...                             ...
status =                        System.out.
transfer(500.00,7,8);          print(audit())
    ;
...                             ...
```

- `audit()` can report an overall total that is `500` more or less than the actual assets

```
boolean transfer (double amount,
                  int source, int target)
                  {
    if (accounts[source] < amount) {
        return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}
double audit() {
    // total balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++) {
        balance += accounts[i];
    }
    return balance;
}
```

Maintaining data consistency ...

- What are the possibilities when we execute the following?

```
Thread 1                     Thread 2
...                          ...
status =                     System.out.
transfer(500.00,7,8);       print(audit())
    ;
...                          ...
```

- `audit()` can report an overall total that is `500` more or less than the actual assets
  - Depends on how actions of `transfer` are interleaved with actions of `audit`

```
boolean transfer (double amount,
                  int source, int target)
                     {
    if (accounts[source] < amount) {
        return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}
double audit() {
    // total balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++) {
        balance += accounts[i];
    }
    return balance;
}
```

Atomicity of updates

- Two threads increment a shared variable n

```
Thread 1          Thread 2
 ...               ...
m = n;            k = n;
m++;              k++;
n = m;            n = k;
 ...               ...
```

Atomicity of updates

- Two threads increment a shared variable n

```
Thread 1          Thread 2
 ...               ...
m = n;            k = n;
m++;              k++;
n = m;            n = k;
 ...               ...
```

- Expect n to increase by 2 …but time-slicing may order execution as follows

Atomicity of updates

- Two threads increment a shared variable n

```
Thread 1                Thread 2
 ...                     ...
m = n;                  k = n;
m++;                    k++;
n = m;                  n = k;
 ...                     ...
```

- Expect n to increase by 2 …but time-slicing may order execution as follows

```
Thread 1: m = n;
Thread 1: m++;
Thread 2: k = n;   // k gets the original value of n
Thread 2: k++;
Thread 1: n = m;
Thread 2: n = k;   // Same value as that set by Thread 1
```

Race conditions and mutual exclusion

- **Race condition** — concurrent update of shared variables, unpredictable outcome
  - Executing `transfer()` and `audit()` concurrently can cause `audit()` to report more or less than the actual assets

```
boolean transfer (double amount,
                  int source, int target) {
    if (accounts[source] < amount) {
        return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}
double audit() {
    // total balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++) {
        balance += accounts[i];
    }
    return balance;
}
```

Race conditions and mutual exclusion

- **Race condition** — concurrent update of shared variables, unpredictable outcome
  - Executing `transfer()` and `audit()` concurrently can cause `audit()` to report more or less than the actual assets
- Avoid this by insisting that `transfer()` and `audit()` do not interleave

```
boolean transfer (double amount,
                  int source, int target) {
    if (accounts[source] < amount) {
        return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}
double audit() {
    // total balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++) {
        balance += accounts[i];
    }
    return balance;
}
```

Race conditions and mutual exclusion

- **Race condition** — concurrent update of shared variables, unpredictable outcome
  - Executing `transfer()` and `audit()` concurrently can cause `audit()` to report more or less than the actual assets
- Avoid this by insisting that `transfer()` and `audit()` do not interleave
- Never simultaneously have current control point of one thread within `transfer()` and another thread within `audit()`

```
boolean transfer (double amount,
                  int source, int target) {
    if (accounts[source] < amount) {
        return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}
double audit() {
    // total balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++) {
        balance += accounts[i];
    }
    return balance;
}
```

Race conditions and mutual exclusion

- **Race condition** — concurrent update of shared variables, unpredictable outcome
  - Executing `transfer()` and `audit()` concurrently can cause `audit()` to report more or less than the actual assets
- Avoid this by insisting that `transfer()` and `audit()` do not interleave
- Never simultaneously have current control point of one thread within `transfer()` and another thread within `audit()`
- **Mutually exclusive** access to **critical regions** of code

```
boolean transfer (double amount,
                  int source, int target) {
    if (accounts[source] < amount) {
        return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}
double audit() {
    // total balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++) {
        balance += accounts[i];
    }
    return balance;
}
```

Mutual exclusion

- Concurrent update of a shared variable can lead to data inconsistency
  - **Race condition**
- Control behaviour of threads to regulate concurrent updates
  - **Critical sections** — sections of code where shared variables are updated
  - **Mutual exclusion** — at most one thread at a time can be in a critical section

Mutual exclusion for two processes

- First attempt

```
Thread 1
 ...
while (turn ≠ 1) {
    // "Busy" wait
}
// Enter critical section
 ...
// Leave critical section
turn = 2;
 ...
```

```
Thread 2
 ...
 while (turn ≠ 2) {
    // "Busy" wait
}
// Enter critical section
 ...
// Leave critical section
turn = 1;
 ...
```

Mutual exclusion for two processes

- First attempt

```
Thread 1                          Thread 2
...                               ...
while (turn ≠ 1) {                 while (turn ≠ 2) {
    // "Busy" wait                    // "Busy" wait
}                                 }
// Enter critical section         // Enter critical section
...                               ...
// Leave critical section         // Leave critical section
turn = 2;                         turn = 1;
...                               ...
```

- Shared variable `turn` — no assumption about initial value, atomic update

Mutual exclusion for two processes

- First attempt

```
Thread 1
...
while (turn ≠ 1) {
    // "Busy" wait
}
// Enter critical section
...
// Leave critical section
turn = 2;
...
```

```
Thread 2
...
while (turn ≠ 2) {
    // "Busy" wait
}
// Enter critical section
...
// Leave critical section
turn = 1;
...
```

- Shared variable `turn` — no assumption about initial value, atomic update
- Mutually exclusive access is guaranteed …

Mutual exclusion for two processes

- First attempt

Thread 1	Thread 2

```
Thread 1
...
while (turn ≠ 1) {
    // "Busy" wait
}
// Enter critical section
...
// Leave critical section
turn = 2;
...
```

```
Thread 2
...
while (turn ≠ 2) {
    // "Busy" wait
}
// Enter critical section
...
// Leave critical section
turn = 1;
...
```

- Shared variable `turn` — no assumption about initial value, atomic update
- Mutually exclusive access is guaranteed ...
- ...but one thread is locked out permanently if other thread shuts down – **starvation**!

Mutual exclusion for two processes ...

- Second attempt

```
Thread 1                              Thread 2
...                                   ...
request_1 = true;                     request_2 = true;
while (request_2) {                   while (request_1) {
    // "Busy" wait                        // "Busy" wait
}
// Enter critical section             // Enter critical section
...                                   ...
// Leave critical section             // Leave critical section
request_1 = false;                    request_2 = false;
...                                   ...
```

Mutual exclusion for two processes …

- Second attempt

```
Thread 1
...
request_1 = true;
while (request_2) {
    // "Busy" wait
}
// Enter critical section
...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
while (request_1) {
    // "Busy" wait

// Enter critical section
...
// Leave critical section
request_2 = false;
...
```

- Mutually exclusive access is guaranteed …

Mutual exclusion for two processes ...

- Second attempt

```
Thread 1                              Thread 2
...                                   ...
request_1 = true;                     request_2 = true;
while (request_2) {                   while (request_1) {
    // "Busy" wait                        // "Busy" wait
}
// Enter critical section             // Enter critical section
...                                   ...
// Leave critical section             // Leave critical section
request_1 = false;                    request_2 = false;
...                                   ...
```

- Mutually exclusive access is guaranteed ...
- ...but if both threads try simultaneously, they block each other
  - **Deadlock**!

Peterson's algorithm

Thread 1
```
...
request_1 = true;
turn = 2;
while (request_2 &&
        turn ≠ 1) {
    // "Busy" wait
}
// Enter critical section
...
// Leave critical section
request_1 = false;
...
```

Thread 2
```
...
request_2 = true;
turn = 1;
while (request_1 &&
        turn ≠ 2) {
    // "Busy" wait
}
// Enter critical section
...
// Leave critical section
request_2 = false;
...
```

- Combines the previous two approaches

Peterson's algorithm

```
Thread 1                              Thread 2
...                                   ...
request_1 = true;                     request_2 = true;
turn = 2;                             turn = 1;
while (request_2 &&                   while (request_1 &&
        turn ≠ 1) {                           turn ≠ 2) {
    // "Busy" wait                        // "Busy" wait
}                                     }
// Enter critical section             // Enter critical section
...                                   ...
// Leave critical section             // Leave critical section
request_1 = false;                    request_2 = false;
...                                   ...
```

- Combines the previous two approaches
- We need to argue that mutual exclusion is guaranteed and no process starves!

Correctness of Peterson's algorithm – Mutual exclusion

- Suppose both threads are in their critical sections at time t_0
- Let $t_i < t_0$ be the last time at which thread i sets the value of `turn`
- Let the value of `turn` at time t_0 be `1`, w.l.o.g.
- Then $t_1 < t_2 < t_0$ and the value of `request_1` is **`true`** throughout the interval of time from t_1 to t_0
- Thread 2 enters its busy wait loop after time t_2 but then it cannot exit the loop before t_0
- **Contradiction! So mutual exclusion is guaranteed!**

Correctness of Peterson's algorithm – Freedom from starvation

- If both threads are in their busy wait loops and value of `turn` is `i`, thread i will exit its loop!
- W.l.o.g. suppose thread 1 sets `request_1` to **`true`** at time t_0 and never enters its c.s. after that
- It sets `turn` to 2 at time $t_1 > t_0$ and then gets stuck in its busy wait loop forever
- This means that `request_2` has value **`true`** whenever thread 1 checks
- If thread 2 is already in or about to enter its busy wait loop at t_1, it will eventually exit (because `turn` has value 2)!
- It then enters and exits its c.s. and sets `request_2` to **`false`** at time $t_2 > t_1$
- Since thread 1 sees the value of `request_2` to be **`true`** after t_2, it has to be that thread 2 set its value to **`true`** at time $t_3 > t_2$
- It will then set `turn` to 1 at time $t_4 > t_3$ and get stuck in its busy wait loop!
- When thread 1 subsequently checks the value of `turn`, it will exit its busy wait loop!
- **Contradiction! So no thread starves!**

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist
- Lamport's **Bakery Algorithm**
  - Each new process picks up a token (increments a counter) that is larger than all waiting processes
  - Lowest token number gets served next
  - Still need to break ties — token counter is not atomic

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist
- Lamport's **Bakery Algorithm**
  - Each new process picks up a token (increments a counter) that is larger than all waiting processes
  - Lowest token number gets served next
  - Still need to break ties — token counter is not atomic
- Need specific clever solutions for different situations

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist
- Lamport's **Bakery Algorithm**
  - Each new process picks up a token (increments a counter) that is larger than all waiting processes
  - Lowest token number gets served next
  - Still need to break ties — token counter is not atomic
- Need specific clever solutions for different situations
- Need to argue correctness in each case

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist
- Lamport's **Bakery Algorithm**
  - Each new process picks up a token (increments a counter) that is larger than all waiting processes
  - Lowest token number gets served next
  - Still need to break ties — token counter is not atomic
- Need specific clever solutions for different situations
- Need to argue correctness in each case
- Instead, provide higher level support in programming language for synchronization

Summary

- We can construct protocols that guarantee mutual exclusion to critical sections
  - Watch out for **starvation** and **deadlock**
- These protocols cleverly use regular variables
  - No assumptions about initial values, atomicity of updates
- Difficult to generalize such protocols to arbitrary situations
- Look to programming language for features that control synchronization