

# Concurrency: Threads, Processes, Race Conditions, Mutual Exclusion

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 13, 28 February 2023

## ■ 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

- Operated on same local variables
- Communicate via “shared memory”
- Context switches are easier

- Henceforth, we use **process** and **thread** interchangeably

- 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

# 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` — later

```
public class Parallel extends Thread{
    private int id;

    public Parallel(int i){ id = i; }
}

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);           // Sleep for 1000 ms
            }
            catch(InterruptedException e){}
        }
    }
}

public class TestParallel {
    public static void main(String[] args){
        Parallel p[] = new Parallel[5];
        for (int i = 0; i < 5; i++){
```

# Java threads ...

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

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

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

# 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

```
...
status =
    transfer(500.00,7,8);
...
```

Thread 2

```
...
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
  - Depends on how actions of `transfer` are interleaved with actions of `audit`
  - Can even report an error if `transfer` happens **atomically**

```
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
...	...
<code>m = n;</code>	<code>k = n;</code>
<code>m++;</code>	<code>k++;</code>
<code>n = m;</code>	<code>n = k;</code>
...	...

- 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
- 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;
}
```

- 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                                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
- 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;
...                                     ...
```

- 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
- If both try simultaneously, `turn` decides who goes through
- If only one is alive, `request` for that process is stuck at false and `turn` is irrelevant

# 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

- 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