Concurrent Programming

CS511

Teachers

Instructor: Eduardo Bonelli

E-mail: `ebonelli@stevens.edu`

Office hours: MTF 2-3PM/3-4PM

CA Office hours: By appointment

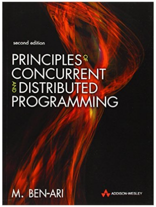
Office: North Building 318

Ask questions!

- Feel free to interrupt and ask questions at any time
  - Your questions also help me better understand the topics
  - It also helps classmates who might have similar doubts
- Contact me by email
- Come see me during office hours

Bibliography

- Slides, above all
- The book we use



Credits

This course has benefitted from material from the following sources:

- `https://sites.google.com/site/pconctpiunq/` (Daniel Ciolek and Hernán Melgratti)
- Slides by Dan Duchamp
- Slides from the course on Concurrency at Chalmers (TDA382/DIT390)

General Structure of the Course

- Lectures
- Assignments:
  - Compulsory
- Exercise booklets
  - Crucial
- Quizzes
- Exams:
  - Midterm and Endterm
  - Additional Makeup

Read syllabus for full details

Concurrency

- The study of systems of interacting computer programs which share resources and run concurrently, i.e. at the same time
- Parallelism vs Concurrency
  - Parallelism: Occurring physically at the same time
  - Concurrency: Occurring logically at the same time, but could be implemented without real parallelism
- We focus on Concurrency:
  - It suffices to restrict attention to a unique execution model (all programs are executed in a unique processor)

Interaction Models

Concurrency

Abstract model of computation that allows us to understand the behavior of sets of programs that share resources

- Shared Memory
  - Centrally shared memory
  - Distributed shared
- Message passing
  - Send/receive
  - Multi-cast
  - Broadcast

Course Objectives

- Understand classic problems in Concurrent Programming (CP) such as synchronization
- Understand the primary primitives used in CP
- Develop skills to be able to use these primitives in solving synchronization problems
- Get to know modern CP techniques
- Understand the fundamentals of model-checking for checking properties of concurrent systems

Concurrency is Hard!

- Harder than sequential programming:
  - Huge number of possible executions
  - Inherently non-deterministic
  - Parallelism conceptually harder
- Consequences:
  - Programs are harder to write
  - Programs are harder to debug
  - Errors are not always reproducible
  - New kinds of errors possible: Deadlock, starvation, etc.

PLs used in this Course

- Java
- Hydra (Examples only)
  - Based on BeanShell (scripting language for Java)
  - Allows succinct representation of examples
- Erlang
  - Functional language
  - Used for distributed programming
- Promela
  - Used for model-checking, one of the topics we shall cover later

Before proceeding, a word on terminology

Terminology

- Processes: Sequential program that runs in its own address space managed by the OS
- Threads: Runs inside the address space of a unique process
  - Terminology popularized by pthreads (POSIX threads) of UNIX
  - Differences between processes and threads irrelevant for study of synchronization and algorithms
- State: Data + Instruction Pointer/Program Counter
- Scheduler: A component of an operating system that decides which process is to be executed in the next time interval

Example 1 of Thread in Java – Extending `Thread`

```java
public class HelloThread extends Thread{
  public void run()
  {
    while (true){
      System.out.println("Hello");
      try {
        Thread.sleep(3000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }

  public static void main(String args[]) {
    new HelloThread().start();
  }
}
```

Example 2 of a Thread in Java – Anonymous Functions

```java
public class HelloThreadAnonymous {

  public static void main(String args[]) {
    Thread hello = new Thread(){
                 public void run()
                 {
                   while (true){
                     System.out.println("Hello");
                     try {
                       Thread.sleep(3000);
                     } catch (InterruptedException e) {
                       e.printStackTrace();
                     }
                   }
                 }
    };
    hello.start();
  }
}
```

Example 2 of a Thread in Java – Implementing `Runnable`

```java
public class HelloThreadRunnable implements Runnable{

  public void run()
  {
    while (true){
      System.out.println("Hello");
      try {
        Thread.sleep(3000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }}

  public static void main(String args[]) {
    new Thread(new HelloThreadRunnable()).start();
  }
}
```

Same Example in Hydra

```
thread P: {
  while (true) {
    sleep(3000);
    print("Hello");
  }
}
```

Example of Concurrent Threads in Java

```java
public class Example1 implements Runnable{
  private String s;
  private int wait;
  public Example1(String s, int wait) {
    this.s=s;
    this.wait=wait;
  }

  public void run() {
    while (true) {
    try {
      Thread.sleep(wait);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println(s);
    }
  }

  public static void main(String args[]) {
    (new Thread(new Example1("Thread 1 here!",200))).start();
    (new Thread(new Example1("Thread 2 here!",200))).start();
  }
}
```

Example in Hydra

```
thread P: {
  while (true) {
    sleep(200);
    print("Thread 1 is here!");
  }
}

thread Q: {
  while (true) {
    sleep(200);
    print("Thread 2 is here!");
  }
}
```

Example in Hydra - Equivalent to previous one

```
for (i=0; i<2; i++) {
  {
    int local = i; // local copy of i required
    thread  {
      while (true) {
        sleep(200);
        print("Thread " + local + " is here!");
      }
    }
  }
}
```

Example in Hydra (cont.)

What's wrong with this?

```
for (i=0; i<2; i++) {
  {
    thread  {
      while (true) {
        sleep(200);
        print("Thread " + i + " is here!");
      }
    }
  }
}
```

Example in Erlang

```
-module(first).
-export([hello/0,init/0]).

hello() ->
    timer:sleep(2000),
    io:format("Hello World!~n"),
    hello().

init() ->
    spawn(first,hello,[]),
    spawn(first,hello,[]).
```

Process Scheduling

- In a standard Von Neumann machine, threads appear to be running at the same time, but in fact their execution is interleaved

Scheduling

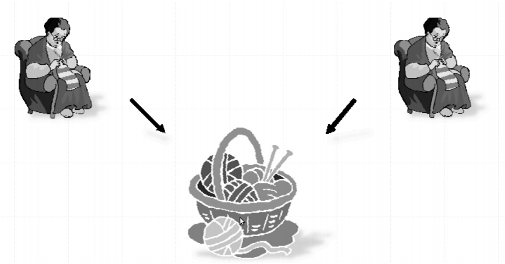
Task of alternating between the execution of multiple threads

- Cooperative: A thread executes until it voluntarily frees the processor (eg. it finished, it sleeps, it executes I/O operations, etc).
- Pre-emptive: Its execution is interrupted so that another thread can run (eg. time-slicing)

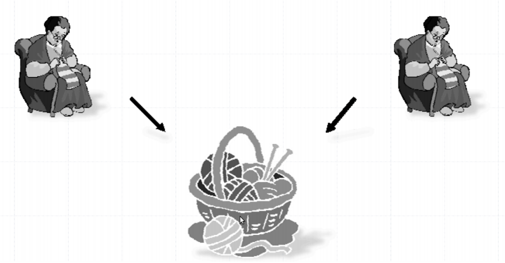
Independent Processes

There are no shared resources nor communication and hence no cooperation problems

Competitive Processes

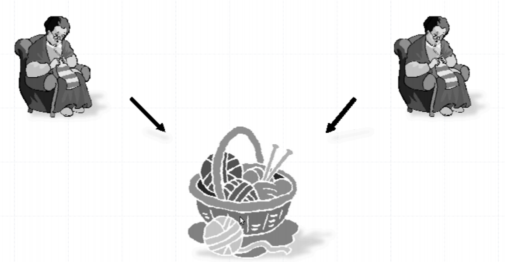


Competitive Processes



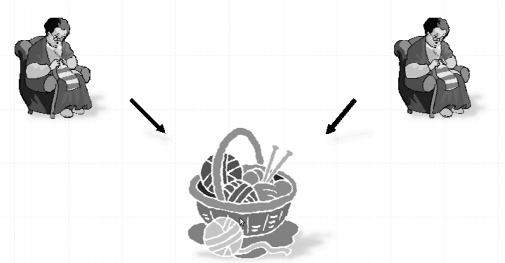
- Deadlock: each granny takes a needle and waits indefinitely until the other one has freed the one she has.

Competitive Processes



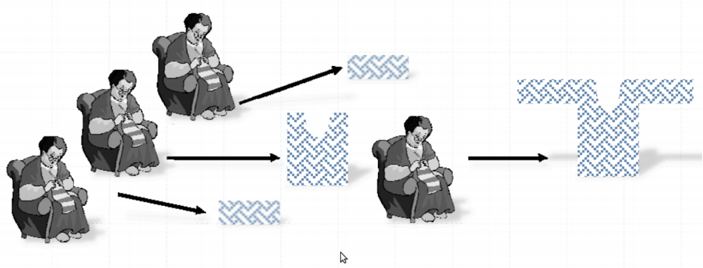
- Deadlock: each granny takes a needle and waits indefinitely until the other one has freed the one she has.
- Livelock: each granny takes a needle, sees that the other granny has the other needle and returns it (this repeats indefinitely).

Competitive Processes

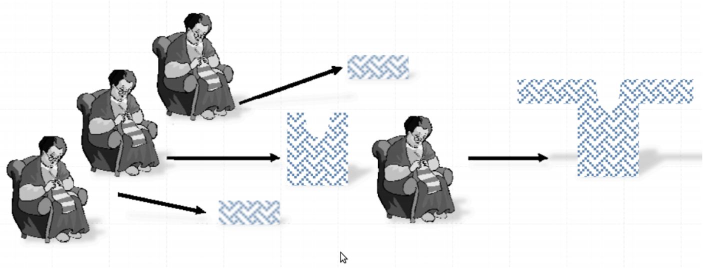


- Deadlock: each granny takes a needle and waits indefinitely until the other one has freed the one she has.
- Livelock: each granny takes a needle, sees that the other granny has the other needle and returns it (this repeats indefinitely).
- Starvation: one of the grannies always takes the needles before the other one.

Cooperative Processes



Cooperative Processes



- Communication mechanisms are necessary for cooperation to be possible

Modelling Program Execution

- What is the value of `x` after executing this program?

```
global int x=0;

thread P: {
  x = 1;
  x = x + 3;
}
```

- Consider this program

```
global int x=0;

thread P:
  x = 1;

thread Q:
  x = 2;
```

  - Value of `x` after execution of just P?

Modelling Program Execution

- What is the value of `x` after executing this program?

```
global int x=0;

thread P: {
  x = 1;
  x = x + 3;
}
```

- Consider this program

```
global int x=0;

thread P:
  x = 1;

thread Q:
  x = 2;
```

  - Value of `x` after execution of just P?
  - Value of `x` after execution of just Q?

Modelling Program Execution

- What is the value of `x` after executing this program?

```
global int x=0;

thread P: {
  x = 1;
  x = x + 3;
}
```

- Consider this program

```
global int x=0;

thread P:
  x = 1;

thread Q:
  x = 2;
```

  - Value of `x` after execution of just P?
  - Value of `x` after execution of just Q?
  - Value of `x` after execution of P $\|$ Q?

Modelling Program Execution

- What is the value of `x` after executing this program?

```
global int x=0;

thread P: {
  x = 1;
  x = x + 3;
}
```

- Consider this program

```
global int x=0;

thread P:
  x = 1;

thread Q:
  x = 2;
```

  - Value of `x` after execution of just P?
  - Value of `x` after execution of just Q?
  - Value of `x` after execution of P $\parallel$ Q? $\{x = 0, x = 1\}$ More than one result is possible!

Modelling Program Execution

A Transition System $\mathcal{A}$ is a tuple

$$(S, Act, \rightarrow, I, AP, L)$$

where

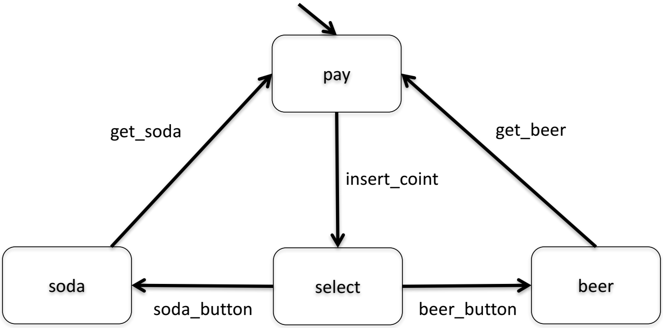
- S is a set of states
- Act is a set of actions
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation
- $I \subseteq S$ set of initial states
- AP is a set of atomic propositions
- $L : S \longrightarrow 2^{AP}$ is a labeling function

Note:

- $\mathcal{A}$ is said to be finite if S, Act and AP are finite.
- We write $s \stackrel{\alpha}{\rightarrow} s'$ for $(s, \alpha, s') \in \rightarrow$.

Example 1 – Beverage Machine

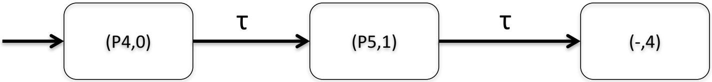
$$S = \{pay, select, soda, beer\}$$
$$I = \{pay\}$$
$$Act = \{insert\_coin, beer\_button, soda\_button, get\_soda, get\_beer\}$$
$$AP = \{paid, drink\}$$
$$L(pay) = \emptyset,\ L(select) = L(soda) = L(beer) = \{paid, drink\}$$



Example 2 – Sequential Program

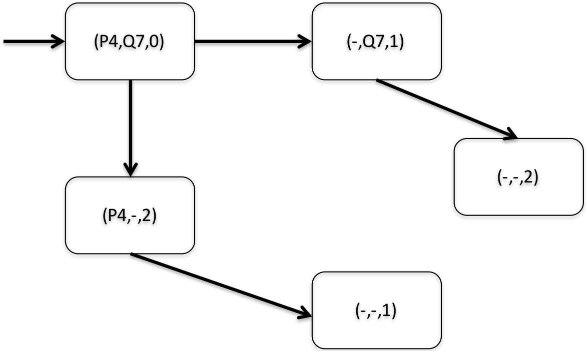
```
global int x=0;

thread P: {
  x = 1;
  x = x + 3;
}
```

- S: tuples that include
  1. the value of the variables and
  2. the program pointer of each thread at a given point in time
- $s \rightarrow t$ if executing a statement in s results in the state t
- $Act = \{\tau\}$ (τ represents an internal, non observable action)
- AP and L will be ignored for now



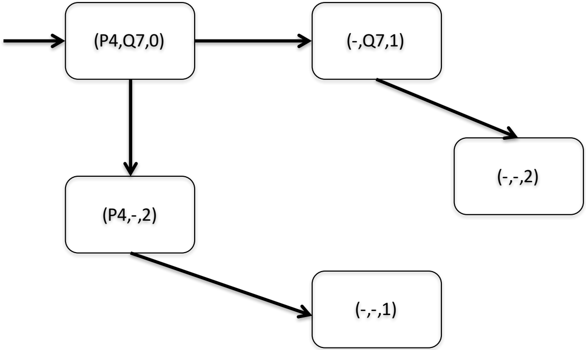
Example 3 – Concurrent Processes

```
global int x=0;

thread P:
  x = 1;

thread Q:
  x = 2;
```



Note: We omit τ for readability

Example 3 – Concurrent Processes



Examples of paths in textual notation:

- $(P4, Q7, 0) \rightarrow (-, Q7, 1) \rightarrow (-, -, 2)$
- $(P4, Q7, 0) \rightarrow (P4, -, 2) \rightarrow (-, -, 1)$
- $(P4, Q7, 0) \rightarrow (P4, -, 2)$

Execution Speed as a Synchronization Mechanism?

- No
- Eg. The following still has two possible results

```
global int x=0;

thread P: {
  sleep(1000);
  x = 1;
}

thread Q: {
  x = 2;
}
```

Summary

- We need concurrency to exploit the processor
- Concurrent programs are non-deterministic
- In this course we will study different synchronization mechanisms that will allow us to control the behavior of concurrent programs
- In particular, we will use synchronization mechanisms to ensure that our programs satisfy desirable properties to be introduced later