Principles of Concurrent Programming

Oscar Palm

Jan 2023

Contents

Chapter no. 1

Introduction to Concurrent Programming

1.1 Practical Information

1.1.1 Labs

Create a zoom meeting without a password
Put a request on waglys
The demos/labs on fridays at 08:00 is online, all others are physical.

There's three main labs:

1. Trainspotting (Java)
2. CChat (Erlang)
3. A-mazed (Java)

1.1.2 Course overview

Three parts:

1. Classic shared-memory
2. Message-parsing
3. Parallellizing computation

1.1.3 Examination

It's an open-book exam, meaning you can bring:
up to 2 textbooks,
maximum of 4 two-sided A4 sheets (printed or handwritten),
and an English dictionary.

1.2 Technical Information

1.2.1 Motivation

Imagine a sequential counter, if you then count twice you'll always end up with 2. Now imagine the same counter implemented concurrently and we run the count function in separate threads, we now have no idea which will be executed first. More importantly we now have no way of checking that they don't run simultaneously, in a worst case they may even read the value before the other thread have saved their value, meaning they overwrite each others answers.

1.2.2 Amdahl's Law

If we have n processors that can run in parallel, how much speedup can we achieve?

$$\text{speedup} = \frac{\text{sequential execution time}}{\text{parallel execution time}}$$

$$\text{Maximum speedup} = \frac{1}{(1-p) + \frac{p}{n}}$$

This means that adding more processors running in parallel benefits the program less and less, the more you add.

1.2.3 Terminology

Processes

A process is an independent unit of execution.

- Identifier
- Program counter
- Memory space

Process states

The scheduler is the system unit in charge of setting process states:

- Ready

  Ready to be executed
- Blocked

  Waiting for event before execution
- Running

  Currently running on the CPU

Threads

A lightweight process, independent unit of execution in the same program space

- Identifier
- Program counter
- Memory

  local memory, each thread has its own

  global memory, shared throughout all threads

Shared memory vs. message sharing

Shared memory communicate by writing to the shared memory space while
Distributed memory communicate by sending messages between each others.

1.2.4 Java threads

Starts with the start() method, which in turn calls the method run(). If you need to wait for a thread to finish its course, run join().

There's two different ways of generating threads in Java, either through extending the class with Thread, or by implementing Runnable. More often than not you implement Runnable because you can have an unlimited amount of interfaces but only one superclass.

Chapter no. 2

Lecture no. 2

Chapter no. 3

Models of concurrency and synchronization algorithms

3.1 Analyzing concurrency

We can use state/transition diagrams to model elements of concurrent programs. States in a diagram capture possible program states, and transitions capture possible state changes. The following diagram models a simple program that reads a value from a shared variable, increments it, and writes it back to the variable.

3.1.1 States

A state captures the shared and local states of a concurrent program. When state is unambiguous, we simplify a state with only the value.

3.1.2 Transition

A transition is the execution leading to a state change.

3.1.3 Program properties

Mutual exclusion, Deadlock freedom, Starvation freedom, No race conditions.

3.2 Mutual exclusion with only atomic reads

Locks is a data structure used to control which thread has access to some value.
Can we implement lock using only atomic instructions - reading and writing shared variables?

It's possible, but quite tricky. The easiest way is to have some kind of a lock queue of threads waiting for the lock.

3.3 Peterson's Algorithm

A simple way of locking some memory from all but one thread without having to use a lock class.

Chapter no. 4

Lecture no. 3

Chapter no. 5

Synchronization problems with semaphores

5.1 The dining philosophers problem

If we have a table with five chairs and five philosophers, each of whom has a bowl of rice and a fork. The philosophers switch between thinking and eating, only that when they eat, they need two forks. Using semaphores we can ensure that the philosophers can eat and never end up in a stalemate.

5.1.1 Solution: limit the number of philosophers

One way of solving the problem is to only allow a certain number of philosophers at the table, as so the number of forks is always greater than the number of philosophers. This is not a very elegant solution, and it is not always possible to know how many philosophers will be at the table, however it does break the deadlock.

5.2 Producer-consumer

The producers and consumers exchange items with each others through a shared buffer (shop), both produce/consume these items asynchronously. Since both cannot access the buffer at the same time, how do we ensure that a consumer won't block an empty buffer or a producer won't block a full buffer? Our solution should also:

- Support an arbitrary number of producers and consumers
- Deadlock freedom
- Starvation freedom

5.2.1 Solution: semaphores

Using a lock and a semaphore, we can lock the buffer in a non-breaking way; we begin by locking the semaphore, then the lock, accessing the buffer, followed by unlocking the semaphore and the lock. Locking the semaphore before the lock not only ensures deadlock freedom, but also starvation freedom. The way we ensure starvation freedom is through letting the consumer and producer unlock each other, meaning the consumer has to wait for producer between purchase attempts.

5.3 Barriers

A barrier is a way of Synchronizing, where we select a point (the barrier) in the program execution which all threads in a group has to reach before any single thread is allowed through.

```
// ------ Header Begin -------
int nDone = 0;
Lock lock = new Lock();
Semaphore open = new Semaphore(0);
// ------ Header End ---------
public void myFunction(){
   // Code before barrier
   lock.lock()
   nDone = nDone + 1
   lock.unlock();
   if(nDone == nThreads)
   open.up();
   open.down();
   open.up();
   // Code after barrier
   }
```

Figure 5.1: A simple barrier implementation. (non-reusable)

In Java we can use the release(n) function to do this.

5.4 Readers-writers

Readers-writers concurrently access shared data, readers can access the data together, however many they may be, as long as no reader is there. A writer however, can only access the data when no other reader or writer is there.

5.4.1 Problem

Implement the Board data structure as so

- Multiple readers can access the board at the same time.
- Each writer has exclusive access.

Chapter no. 6

Erlang introduction

6.1 What is Erlang?

6.1.1 Erlang history

Erlang was first created in the mid 1980's at Ericsson. In 1998, they decide to ban the internal use of erlang, making it open-source. It is still used though in communication apps such as whatsapp.

6.1.2 What is a functional language?

Functional languages are based on elements different from imperative languages. In imperative programming you have states, such as variables, while in functional programming ,you have data, and side effectless functions. All you have is data, no variables, no side effects, you send in something and you always get the same result.

6.2 The erlang shell

```
$ erl
1> 1+3. % This is a comment
4
2> c(power). % Compile the module power (file power.erl)
{ok,power}
3> power:power(2,3). % Call the function power in the module power
8
```

6.3 Types

6.3.1 Primitive types

- Integer

  These are of an arbitrary size and therefore don't overflow

- Atoms

  Roughly equivalent to identifiers

- Floats
- References
- Binaries

- Pids

  process identifiers

- Ports

  For communication

- Funs

  function closures

Atoms

Atoms are used to denote distinguished values, they should always start with a lowercase letter. They are used to denote constants, such as true and false. They are also used to denote functions, such as the function name in the erlang shell.

There are no booleans, instead we use the atoms true and false as booleans out of convention.

6.3.2 Compound types

- Tuples
- Lists
- Maps
- Strings
- Records

Tuples

Tuples denote an ordered sequence with a fixed but arbitrary number of elements of arbitrary types.

Lists

Lists are a sequence of elements of arbitrary type with a variable size.

Strings

A sequence of characters enclosed between double quotation marks. Basically just a list of characters.

Order of types

number ¡ atom ¡ reference ¡ fun ¡ port ¡ pid ¡ tuple ¡ map ¡ list

6.4 Variables

Variables are variables in the mathematical sense, they are identifiers that can be bound to values (Similar to constants in imperative programming).

6.5 Recurrsion - Example

```
% A slow version of the factorial function using recursion
Factorial(N) when N = 1 -> 1;
Factorial(N) -> N * Factorial(N-1).

% A faster version of the function above using tail recursion
Factorial(N) ->
 FacHelper(N,i) when N = 1 -> i;
 FacHelper(N,i) -> FacHelper(N-1,i*N).
 FacHelper(N,1).
```