L15: Future and Promises

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

March 3rd, 2020

Architecture Research Group

Software System Engineering

Thai-German Graduate School, KMUTNB

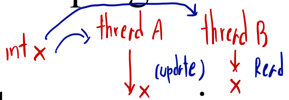
Concurrent+Asynchronous Programs

- What does it mean for a program to have concurrency?

↳ task A, task B - running at the same time (2 tasks in parrel)

- What does it mean for a program to be asynchronous?

↳ Don't have to wait to run



A → x → B

(B read before A update)

- Why does functional programming model fit well?
  - Recall dataflow paradigm?

Example

- Let’s count the number of the occurrence of a list of web URLs

$$\begin{tikzcd} \text{(Map)} \arrow[r, Rightarrow] & \textcircled{A} \arrow[d] \\ & \textcircled{B} \arrow[d] \\ & \textcircled{C} \end{tikzcd}$$

((Cannot do parrrel)

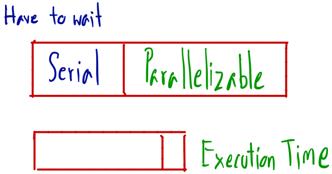
↳ Nothing is shared

- See code example on canvas

- Let’s break this down and see what is the blocking calls

- Blocking call: a part of the program that has to finish before the next part begins
  - Why is this bad for the performance of parallel programs?

Amdahl's Law

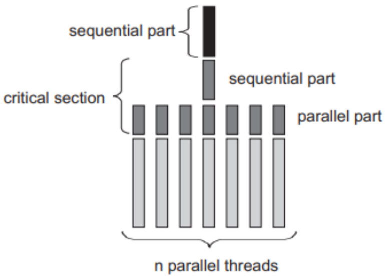


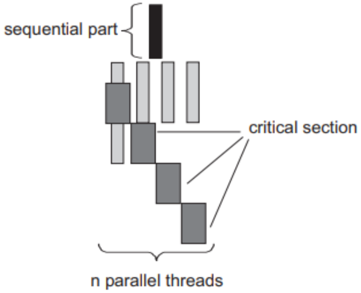
- Amdahl's Law:
  - The performance speedup in parallel programs depend on **both the serial parts and the parallel parts**
    - **Serial portion determined by the algorithm**
  - This can limit the benefit of parallelizing your program
  - Serial portion of the code becomes the critical performance bottleneck in parallel programs
  - Gene Amdahl, "Validity of the single processor approach to achieving large scale computing capabilitie", AFIPS 1967.

- Critical section Parts of program execution that require synchronization/need to resolve contentions
  - Amdahl's Law did not mention this part

Breaking Down Critical Sections

- Critical section can be broken down into two cases
  - With synchronization (What can cause this?)
  - Without synchronization





- Eyerman and Eeckhout, “Modeling Critical Sections in Amdahl’s Law and its Implications for Multicore Design”, ISCA 2010

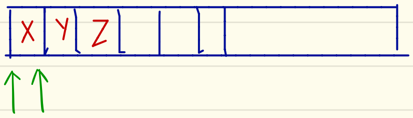
Synchronization

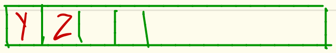
$$\underbrace{(M \times N)}_{\text{Multiply}}{}^{T} \leftarrow \text{Has to wait for multiply before transpose}$$

Without Synchronization

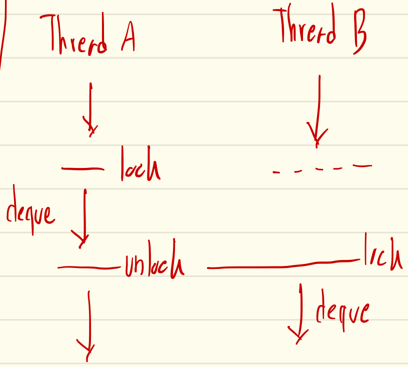
thread A	thread B	
CPU 1	CPU 2	Both share this Queue
dequeue	dequeue	

Queue

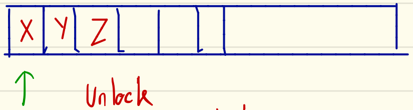




Solution

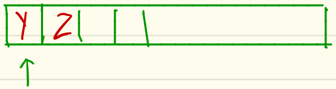


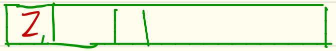
Queue Lock



Unlock

B → lock





How to Handle Blocking

- Synchronously - Wait before run second task
    - Wait for the serial portion/blocking call to complete
    - Run the next processing task afterward
    - What are the benefits? - Protect Buggy
    - What are the downsides? - Slow

- Asynchronously
    - While waiting for the blocking process $\rightarrow$ Switch to an independent task
    - What are the benefits? - Safe time, compute more thing per unit of time
    - What are the downsides?

The Notion of Time

- Traditionally, functional programming does not focus on the notion of time
  - Why?
  - This is along the same logic as why "state" does not exist

Status of certain things

- Scala supports this

This will become value of T at some point of time

- You can use Future[T] to encode the notion of time
  - This is a value that will eventually become available
  - This value can have multiple states depending on the time we request the value

The State of The Future

- In its simplest form, Future[T] can have two states
  - Completed/determined Computation is complete and the value is available - Got value
  - Incomplete/undetermined Computation is not complete

    ↳ Value incomplete

- Future[T] is a container/wrapper type
  - Represent a value "that will eventually" results in type T
  - If the computation go wrong (time out, die horribly, etc.), this Future[T] has an exception of sort
  - This is a write-once container
    - **Becomes immutable** once the computation is complete

Example using Fib

- import scala.concurrent.Future
- import scala.concurrent.ExecutionContext.Implicits.global

- OK-ish fib
  - def fib(n: Int): Long = if (n <= 2) 1 else fib(n-1) + fib(n-2)
- Better fib using Future
  - val f1 = Future { fib(45) } f1 will evaluate to fib(45)
- Now I can even run two of fib at the same time
  - val f2 = Future { fib(46) }
  - **They will be run completely in the background**

val f3 = Future(f1 + f1) Cannot - Wait for f1, f2

Accessing the Future

- Assume fib1 and fib2 are defined, you can do
- f1.onComplete { case **Success**(result) => println(result) }

(If f1 finish execution do ...)

- If not successful, you can do
- import scala.util.{Success, Failure}

```
val f = Future { fib(49) }
f.onComplete {
  case Success(v) => println("good ${v}")
  case Failure(e) => println("Error: " + e.getMessage)
}
```

Waiting for the Future (Time)

- Say if you want Scala to wait for the Future[T] to becomes ready
- import scala.concurrent.duration._
  Await.ready(f, 1 minutes)

↑ Wait for 1 minute before time out

Use Cases

- I/O calls □ wait for the user input (standard I/O)

  ↳ User type in something

- Long computation □ Use Future[T] to represent the completion of this task while running other tasks

  ↳ Compute prime after 1 million

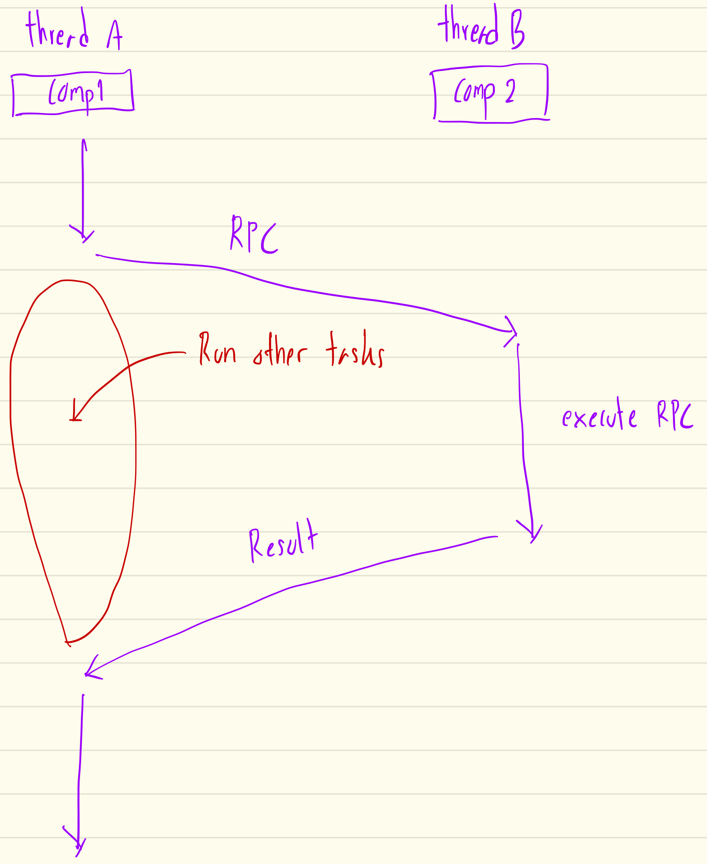
- Database queries □ Another instance of long processes

(Remote Proceder Call)

- RPC □ Network latency is long, use Future[T] to represent the result of the remote procedure calls

- Timeout □ Force a return of no result or empty result on Future[T] to represent a timeout

(Call from another machine)



Futures vs. Promises

- Scala also supports promises

- A promise is a single-assignment variable which the future refer to
  - You can get the future with a promise, but not vice-versa

- Think of this as how you handle real-life promise
  - If you promise something to someone, you must keep it
  - If someone promise something to you, they should honor it in the future

Realizing Futures vs. Promises

- Ok … all the things we discussed so far are good ideas

- But, hardware is plain and stupid
  - How do I actually implement this concept?

- Why you should care?
  - This **bridge the abstraction** between your program to the runtime and then to the hardware

Thread Pools and Event Loops

- What is a thread? - execute something
- A thread pool is a collection of idle threads that the runtime can assign work to
  - The actual thread pool implementation handles creating the workers, manage the workers, and schedule tasks (which thread run first)
    - Active area of research as new type of computing systems emerge
- An event loop is an underlying system layers that are specifically designed to handle certain tasks
  - File system (for I/Os)
  - Database system (for queries)
  - Web services
  - All of which relies on its implementation, libraries, frameworks

Utilizing Future/Promises

- How can we unblock things using future/promises?

- Using Future to pipeline program execution
  - What is pipelining?

- Let’s go back to our example earlier

In-Class Exercise 14

- Check the example from the exercise

- Your task:

1. Explain what the code does?
2. Explain what Future is used for?