CS 361: Concurrent Theory

Mark Boady

Department of Computer Science
Drexel University

September 9, 2022

This lecture is adapted from Chapter 2
Principles of Concurrent and Distributed Programming
M. Ben-Ari
Second Edition
Available online from library:
https://drexel.primo.exlibrisgroup.com/permalink/01DRXU_INST/7nf6pv/cdi_askewsholts_vlebooks_9781292122588

Role of Abstraction

- Abstraction is used in almost every science
- Computer Science uses lots of abstraction
  - Systems and Libraries: abstract OS commands from the programmer
  - Programming Languages: abstract Assembly level instructions
  - Instructions Sets: abstract assembly from the specific CPU hardware
- What semi-conductors cause $x=y+z$ work?

Encapsulation

- Divides a software module into a public and private implementation
- Public specification describes the available operations on a data structure
- Private implementation explains how the structure actually works
- Private implementation is hidden from the public
- Private implementation can be changed without effecting the public specification

Concurrency

- **Concurrency** is an abstraction that allows us to reason about dynamic programs
- We cannot truly examine every possible execution path of a large program
- We need to focus on specific problems related to the **critical section** of a program
- We focus on **atomic operations** that will always complete once started

Process

- A **process** is a set of sequential steps that will be executed on the processor
- A **process** is *sequential*, all steps will execute it order
- A **process** is built of **atomic statements**
- A **atomic statement** will always complete execute without a **context switch**
- A **context switch** is when the OS switches the process running on the processor

Concurrent Program

- A **Concurrent Program** is a finite set of **processes**
- The **concurrent program** executes the **processes** using **arbitrary interleaving**
- **Arbitrary interleaving** means the order in which the atomic statements are switched between is unknown.
- A **computation** (or scenario) is a specific interleaving of the **processes**
- A **Concurrent Program** may have many different possible **computations**

Control Pointer

- Each **process** has a **control pointer**
- The **control pointer** indicates what the *next* atomic statement to be executed is
- These are called **instruction pointers** if working at the assembly level

Interleaving

- Imagine we have 2 processes name p and q
- The process p has 2 atomic statements $p1$ and $p2$.
- The process q has 2 atomic statements $q1$ and $q2$.
- Each process must execute in sequential order
- How many possible **computations** exist?

Computations

$$p1 \rightarrow q1 \rightarrow p2 \rightarrow q2$$
$$p1 \rightarrow q1 \rightarrow q2 \rightarrow p2$$
$$p1 \rightarrow p2 \rightarrow q1 \rightarrow q2$$
$$q1 \rightarrow p1 \rightarrow q2 \rightarrow p2$$
$$q1 \rightarrow p1 \rightarrow p2 \rightarrow q2$$
$$q1 \rightarrow q2 \rightarrow p1 \rightarrow p2$$

Total Computations

- There are 6 possible **computations**
- There are other **permutations** of these steps, but they are not valid computations
- Each **process** must still execute in sequential order
- $p2 \rightarrow p1 \rightarrow q1 \rightarrow q2$ is not valid
- The statements of p must execute in order

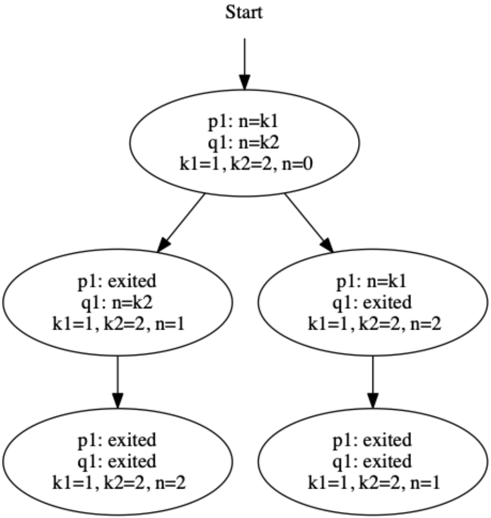
Analysis Pseudocode

- These concepts are language agnostic
- Language details can get in the way of understanding concepts
- We will use a language-independent pseudocode to look at concepts
- These ideas can be easily translated to C++
- **Assume**: each line is always assumed to be a atomic operation
- If an operation is not atomic it will be broken into multiple lines

Algorithm Example

Algorithm 1: Trivial Example	
Setup: $n = 0, k1 = 1, k2 = 2$	
Process P	**Process Q**
p1: n = k1	q1: n = k2

State Diagram



States

- The **state** of a concurrent program is a tuple
- Each Process has a **control pointer**
- A variable currently in memory are shown
- There is a **transition** between two states if execution of one of the **control pointers** moves between the states
- A **state diagram** is a graph that shows every **reachable state** in the concurrent program
- A **computation** is a path in the **state diagram**

Scenario Tables

- State Diagrams get big *fast*
- We can show a specific scenario (path) using a table
- The line being executed is in bold
- Each row shows a state

Scenario 1				
Process P	Process Q	n	k1	k2
p1: n=k1	q1: n=k2	0	1	2
exited	**q1: n=k2**	1	1	2
exited	exited	2	1	2

- This concurrent program has two scenarios

Scenario 2				
Process P	Process Q	n	k1	k2
p1: n=k1	**q1: n=k2**	0	1	2
p1: n=k1	exited	2	1	2
exited	exited	1	1	2

Multitasking

- On one processor, execution is truly interleaved
- On multiple processors/cores, each program has its own local memory and CPU
- Global memory access is still interleaved
- On distributed systems, all processing is done independently but **communication** between systems is interleaved

Interleaving

- Atomic statements may execute at exactly the same moment
- Modifications to shared memory is interleaved
- For practical purposes, we only look at interleaved actions
- We trust the operating system to disallow perfectly overlapped memory writes
- We also assume atomic steps will never be context switched

Debugging

- Debugging concurrent programs becomes exceptionally difficult
- Running a debugger with breakpoints will change the context switches
- Running programs multiple times will change context switches
- Different hardware/OS will also context switch at different times
- We need to be able to analyze all possible cases

Correctness

- There are two important properties we look for in concurrent programs
- Safety Properties
  - A safety property is **always** true regardless of interleaving
  - Example: The mouse is displayed on the screen
- Liveness Properties
  - A liveness property **will eventually** hold true regardless of interleaving
  - Example: mouse changes shape after clicking on object to drag-and-drop
- These two properties are **duals**
- If a property is not always true, then it becomes false at some point

Algorithm Example

- Assume each line is an atomic statement

Algorithm 2: Assignment Statements	
Setup: $n = 0$	
Process P	**Process Q**
p1: n = n + 1	q1: n = n + 1

- This concurrent program has two scenarios

Scenario 1		
Process P	Process Q	n
p1: n = n + 1	q1: n = n + 1	0
exited	**q1: n = n + 1**	1
exited	exited	2

Scenario 2		
Process P	Process Q	n
p1: n = n + 1	**q1: n = n + 1**	0
p1: n = n + 1	exited	1
exited	exited	2

Postcondition

- This program has a **postcondition**
- **Postcondition**: something that is always true when execution of the processes ends
- This is a **liveness** condition, it becomes true at some point
- Postcondition: The variable n will always end with value 2

Multi-Step Add

- What if n=n+1 is not an atomic action?
- We need to break it up into two lines
- Remember: We treat each line as an atomic action

Algorithm Example

- Break n=n+1 into two statements
- Does this algorithm still always end with n=2?

Algorithm 3: Add With Temp	
Setup: $n = 0$	
Process P	**Process Q**
p1: int temp = n p2: n = temp + 1	q1: int temp = n q2: n = temp + 1

- This scenario works correctly

Scenario 1				
Process P	Process Q	n	p::temp	q::temp
p1: int temp = n	q1: int temp=n	0	?	?
p2: n = temp+1	q1: int temp=n	0	0	?
exited	**q1: int temp=n**	1		?
exited	**q2: n=temp+1**	1		1
exited	exited	2		

Incorrect Scenario

- This scenario works fails to reach $n = 2$

Scenario 1				
Process P	Process Q	n	p::temp	q::temp
p1: int temp = n	q1:int temp=n	0	?	?
p2: n=temp+1	**q1:int temp=n**	0	0	?
p2: n=temp+1	q2:n=temp+1	0	0	0
exited	**q2:n=temp+1**	1		0
exited	exited	1		

Incorrectness

- Algorithm 3 is incorrect if we want $n = 2$ to always happen
- A scenario exists where the code interleaves in away that never reaches our post-condition
- We need to think about **every** interleaving of code statements

Amdahl's Law

- There is a limit to the speedup parallel algorithms can give up
- Let $S_{latency}$ but the theoretical speedup of a task
- Let s be the improvement caused by the parellel execution
- Let p be the precent of the code that can be parallelized

$$S_{latency} = \frac{1}{1 - p + \frac{p}{s}}$$

Wikipedia contributors. (2022, September 2). Amdahl's law. In Wikipedia, The Free Encyclopedia. Retrieved 20:49, September 9, 2022, from

`https://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=1108054797`

Amdahl's Law

- Assume that 30% of a program can be converted to parallel (then $p = 0.3$)
- Assume that this parallel improvement will be twice as fast as the original
- Then the total speedup will be 118% of the serial version of the code

$$
\begin{aligned}
S_{latency} &= \frac{1}{1 - p + \frac{p}{s}} \\
&= \frac{1}{1 - 0.3 + \frac{0.3}{2}} \\
&= 1.18
\end{aligned}
$$

Wikipedia contributors. (2022, September 2). Amdahl's law. In Wikipedia, The Free Encyclopedia. Retrieved 20:49, September 9, 2022, from

https://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=1108054797

Gustafson's Law

- Amdahl assumed a fixed speedup from parallelism
- Gustafson lets us keep throwing more processors at the problem
- S is the speedup of the program
- N is the number of processors
- s is the precent of the program that must be serial
- p is the precent of the program that can be parallel

$$S = s + p * N$$

Wikipedia contributors. (2022, August 30). Gustafson's law. In Wikipedia, The Free Encyclopedia. Retrieved 21:18, September 9, 2022, from https://en.wikipedia.org/w/index.php?title=Gustafson%27s_law&oldid=1107600217

Gustafson's Law

- Assume 30% of a project can be made parallel
- More processors will improve the speedup
- Problem: Not all problems can be split up endlessly like this

$$
\begin{aligned}
S =& s + p * N \\
=& 0.7 + 0.3 * 2 = 1.3 \text{ (for 2 Processors)} \\
=& 0.7 + 0.3 * 4 = 1.9 \text{ (for 4 Processors)} \\
=& 0.7 + 0.3 * 8 = 3.1 \text{ (for 8 Processors)} \\
=& 0.7 + 0.3 * 32 = 10.3 \text{ (for 32 Processors)}
\end{aligned}
$$

Wikipedia contributors. (2022, August 30). Gustafson's law. In Wikipedia, The Free Encyclopedia. Retrieved 21:18, September 9, 2022, from

`https://en.wikipedia.org/w/index.php?title=Gustafson%27s_law&oldid=1107600217`