Lecture 27: Parallelism I

CS 62

Spring 2018

Alexandra Papoutsaki & William Devanny

Some slides based on those fom Dan Grossman, U. of Washington

The story so far assumed

- *Sequential programming*: everything is part of one sequence and happens one thing at a time

- If we take this assumption away, complicates things
- In multi-threaded programming we need to rethink:
  - Programming: work is divided among threads of execution that need to be coordinated (*synchronized*)
  - Algorithms: parallelism increases the work done per unit time (*throughput*)
  - Data Structures: need to provide *concurrent* access if multiple threads access the same data

A simplified view of history

- Writing correct and efficient multithreaded code is often much more difficult than sequential code
  - Especially in common languages like Java and C
  - So typically stay sequential if possible
- From roughly 1980-2005, desktop computers got twice as fast every couple years at running sequential programs
- But nobody knows how to continue this
  - Increasing clock rate generates too much heat
  - Relative cost of memory access is too high
  - But we can keep making "wires exponentially smaller" (Moore's "Law"), so put multiple processors on the same chip ("multicore")

What can we do with multiple cores?

- Single-processor computers gone away.
- Run multiple totally different programs at the same time
  - Already doing that, but with *time-slicing*
- Do multiple things at once in one program
  - Our focus – more difficult
  - Requires rethinking everything from asymptotic complexity to how to implement data-structure operations

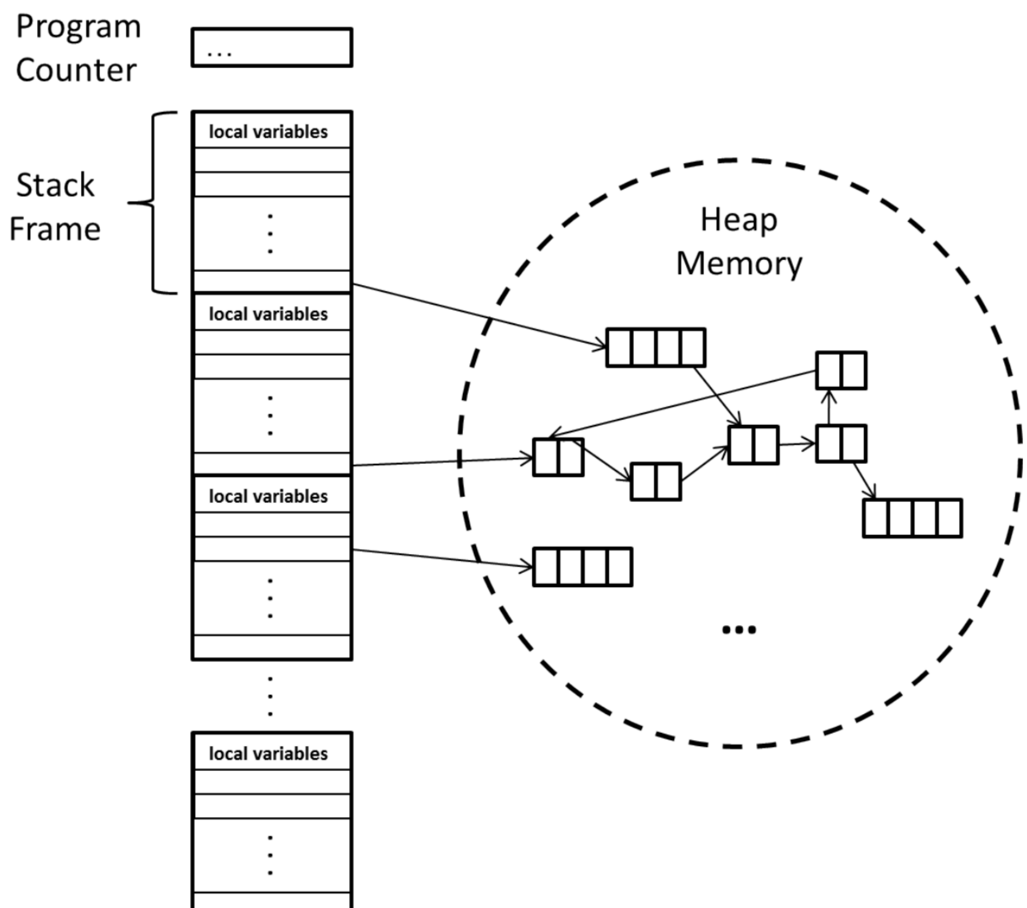
Parallelism vs Concurrency

- Separate terms
- **Parallelism**: Use extra resources to solve a problem faster
- **Concurrency**: Correctly and efficiently manage shared resources
- Common ground:
  - They both use threads
  - If parallel computations need access to shared resources, then the concurrency needs to be managed
- Analogy: a program is like a recipe for a cook
  - Parallelism: many helpers slice potatoes
  - Concurrency: only 4 burners

Models Change

- Model: Shared memory w/explicit threads
- Program on single processor:
  - One call stack:
    each stack frame holds local variables and refs to parameters
  - One program counter (current statement executing)
  - Static fields
  - Objects (created by new) in the heap (nothing to do with heap data structure)

Program state in sequential programming



Multiple Threads/Processors Model

- A set of threads, each with its own call stack & program counter
- No access to another thread's local variables
- Threads can (implicitly) share static fields / objects
- To communicate, write somewhere another thread reads

Shared memory

Threads, each with own unshared call stack & current statement

- (pc for "program counter")
- local variables are primitives, `null`, or heap references

Program state in parallel programming