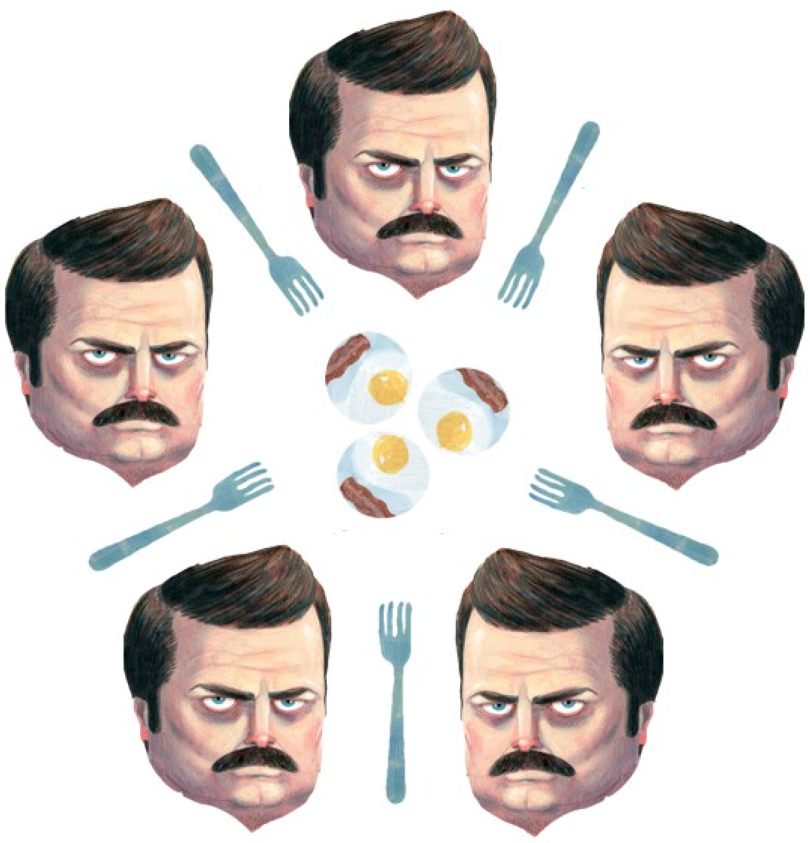
CS 430
Spring 2015

Mike Lam, Professor



http://adit.io/posts/2013-05-11-The-Dining-Philosophers-Problem-With-Ron-Swanson.html

Concurrency

Concurrency

- Instruction-level concurrency
  - "Hidden concurrency" - instruction pipelining and prefetching
  - Mostly an architecture and compiler design issue (**CS 456/480**)
- Statement-level concurrency
  - Often enabled by language features (**CS 430**)
- Unit (subprogram)-level concurrency
  - Sometimes enabled by language features (**CS 430**)
  - Often a distributed/parallel systems issue (**CS 470**)
- Program-level concurrency
  - Mostly an OS or batch scheduler issue (**CS 450/470**)

Scalability

- An algorithm is *scalable* if the speed of execution increases when more processors are available
  - Strong scaling: same data, less time
- Alternately: an algorithm is *scalable* if the availability of more processors allows for efficient processing of larger datasets
  - Weak scaling: same time, more data
- *Amdahl's Law*: scalability is limited by how much of the program is non-parallelizable
  - Modern systems sometimes experience anomalies that violate Amdahl's Law

History

- 1950s: special-purpose I/O or graphics processors
- 1960s: multiple complete processors
- 1970s: vector processors
- 1980s: computing clusters
- 1990s-2000s: rise of multicore consumer machines and graphical processing units (GPUs)
- 2010s: hybrid CPU/GPU architectures
- Future: low-cost, low-power

Categories

- Single-Instruction, Multiple-Data (SIMD)
  - Vector processors
  - GPUs
  - SSE/AVX instructions on x86
- Multiple-Instruction, Multiple-Data (MIMD)
  - Multicore processors
  - Distributed computing

Concepts

- Physical vs. logical concurrency
  - Is the concurrency actually happening on the hardware level, or are executions being interleaved?
  - Users and language designers don't care
  - Language implementers and OS designers do care

Concepts

- Single threaded vs. multi threaded
  - Thread: sequence of control flow points
  - Coroutines are single threaded (*quasi-concurrent*)
  - Multi-threaded programs may still be executed on a single CPU via *interleaving*
- Synchronous vs. asynchronous
  - Synchronous tasks must take turns and wait for each other
  - Asynchronous tasks may execute simultaneously

Concepts

- *Task/process/thread*: program unit that supports concurrent execution
  - Typically, a process may contain multiple threads
  - All threads in a process share a single address space
  - Textbook: heavyweight = process, lightweight = thread
  - Some OSes support lightweight processes

Scheduling

- *Scheduler*: a system program that manages the sharing of processors between tasks
  - Priority-based scheduling
  - Round-robin scheduling
  - Real-time scheduling
- Task states
  - New: created but execution has not yet begun
  - Ready: not currently executing, but may be started
    - Often stored in a ready queue
  - Running: currently executing
  - Blocked: running, but waiting on an event (often I/O)
  - Dead: no longer active

Concepts

- *Liveness*: a program executes to completion
- *Deadlock*: loss of liveness due to mutual waiting
  - E.g., dining philosophers!
- *Race condition*: concurrency outcome depends on interleaving order
  - Example: Two concurrent executions of `bump()`

```
def bump(x)
  tmp = $counter   (1)
  tmp += x         (2)
  $counter = tmp   (3)
end
```

```
def bump(x)
  tmp = $counter   (4)
  tmp += x         (5)
  $counter = tmp   (6)
end
```

OK:	**BAD:**
1	1
2	4
3	2
4	5
5	3
6	6

Concepts

- *Synchronization*: mechanism that controls task ordering
  - *Cooperative* synchronization: ordering based on inter-task dependencies
    - E.g., Task A is waiting on task B to finish an activity
    - Common issue: producer/consumer problem
  - *Competition*: ordering based on resource contention
    - E.g., Task A and Task B both need access to a resource
    - Common issue: file or CPU contention, dining philosopher problem

Synchronization

- *Semaphore*: guarding mechanism (1965)
  - Integer (n = “empty slots”) and a task queue
  - Produce
    - if n > 0: write, decrement n, notify consumers
    - else: wait in producer queue
  - Consume
    - if n < nSlots: read, increment n, notify producers
    - else: wait in consumer queue
  - Binary semaphore: single “slot” (*mutex*)
  - Issue: burden of correct use falls on the programmer

Synchronization

- *Monitors*: encapsulation mechanism (1974)
  - Abstract data types for concurrency
  - Handles locking and corresponding thread queue
  - Shifts responsibility to language implementer and runtime system designer
  - Generally considered safer
- Message passing (1978)
  - Fairness in communication
  - Synchronous vs. asynchronous
  - Can be difficult to program and expensive
  - Necessary in distributed computing

Theory

- Actor model (1973)
  - Actors respond to messages by sending messages and creating new actors
- Communicating sequential processes (CSP) (1978)
  - Events and processes, choice and interleaving
  - Message passing via channels
- π-calculus
  - Concurrency, communication (input/output), and replication, restriction
- Tuple spaces (JavaSpaces, Linda)
  - Data-centric coordination with shared memory
  - Operations: “in” (read and remove), “out” (write), “rd” (read), “eval” (new process)

CSP and π-calculus

$$
\begin{array}{lclc}
Proc & ::= & STOP & \\
 & | & SKIP & \\
 & | & e \rightarrow Proc & \text{(prefixing)} \\
 & | & Proc \,\Box\, Proc & \text{(external choice)} \\
 & | & Proc \,\sqcap\, Proc & \text{(nondeterministic choice)} \\
 & | & Proc \,|||\, Proc & \text{(interleaving)} \\
 & | & Proc \,|[\{X\}]|\, Proc & \text{(interface parallel)} \\
 & | & Proc \setminus X & \text{(hiding)} \\
 & | & Proc; Proc & \text{(sequential composition)} \\
 & | & \text{if } b \text{ then } Proc \text{ else } Proc & \text{(boolean conditional)} \\
 & | & Proc \,\triangleright\, Proc & \text{(timeout)} \\
 & | & Proc \,\triangle\, Proc & \text{(interrupt)}
\end{array}
$$

$$
\begin{array}{rlr}
P, Q, R ::= & x(y).P & \text{Receive on channel } x\text{, bind the result to } y\text{, then run } P \\
| & \overline{x}\langle y\rangle.P & \text{Send the value } y \text{ over channel } x\text{, then run } P \\
| & P|Q & \text{Run } P \text{ and } Q \text{ simultaneously} \\
| & (\nu x)P & \text{Create a new channel } x \text{ and run } P \\
| & !P & \text{Repeatedly spawn copies of } P \\
| & 0 & \text{Terminate the process}
\end{array}
$$

Language Support

- C/C++/Fortran
  - Pthreads, OpenMP, MPI
- Java
  - Threads, `synchronized` keyword and `wait/notify`
- Haskell
  - `Control.Parallel` and `Control.Concurrent`
- High-Performance Fortran (HPF)
  - `DISTRIBUTE` and `FORALL`
- Chapel
  - `coforall`, `cobegin`, and domains

High-Performance Fortran

- Motivation: higher abstractions for parallelism
  - Predefined data distributions and parallel loops
- Development
  - Proposed 1991 w/ intense design efforts in early 1990s
  - Standardized in 1993 and later in 1996
- Problems
  - Poor support for non-standard data distributions
  - Immature compilers and no reference implementation
  - Poor code performance, difficult to optimize and tune
  - Slow uptake among the HPC community
- Legacy
  - Profound influence on later efforts
  - Examples: OpenMP, X-10, Fortress and Chapel