Lecture 33: Concurrency III & Graphs

CS 62

Fall 2018

Alexandra Papoutsaki & William Devanny

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

Volatile

- *Atomic* action: effectively happens all at once
  - `x++` is not an atomic action!
- Java contains volatile keyword
- Changes to a volatile variable are always visible to other threads
- Accesses don't count as data races
- Implementation forces memory consistency
  - though slower!
- Really for experts -- better to use locks.

Lock granularity

- Coarse-grained: Fewer locks, i.e., more objects per lock
  - Example: One lock for entire data structure (e.g., array)
  - Example: One lock for all bank accounts
- Fine-grained: More locks, i.e., fewer objects per lock
  - Example: One lock per data element (e.g., array index)
  - Example: One lock per bank account
- "Coarse-grained vs. fine-grained" is really a continuum.

Granularity trade-offs

- Coarse-grained advantages:
  - Simpler to implement
  - Faster/easier to implement operations that access multiple locations (because all guarded by the same lock)
  - Much easier for operations that modify data-structure shape
- Fine-grained advantages:
  - More simultaneous access (performance when coarse-grained would lead to unnecessary blocking)
- *Guideline*: Start with coarse-grained (simpler) and move to fine-grained (performance) only if contention on the coarser locks becomes an issue. Alas, often leads to bugs.

Critical-section granularity

- A second, orthogonal granularity issue is critical section size
  - How much work to do while holding lock(s)
- If critical sections run for too long:
  - Performance loss because other threads are blocked (contending)
- If critical sections are too short:
  - Bugs because you broke up something where other threads should not be able to see intermediate state

- *Guideline*: Don't do expensive computations or I/O in critical sections, but also don't introduce race conditions

Don't roll your own

- Most data structures provided in standard libraries
  - Point of lectures is to understand the key trade-offs and abstractions
- Especially true for concurrent data structures
  - Far too difficult to provide fine-grained synchronization without race conditions
  - Standard thread-safe libraries like `ConcurrentHashMap` written by world experts

- *Guideline*: Use built-in libraries whenever they meet your needs
  - e.g., `Vector` vs `ArrayList`. `Vector` is synchronized, `ArrayList` assumes program is thread-safe

Deadlock


IMPERIAL COLLEGE OF
ERING NEW
Explain Deadlock, and we'll hire you.
Hire me, and I'll explain it to you.
let's discuss the salary.

Deadlock

```
class BankAccount {
   ...
   synchronized void withdraw(int amt) { ... }
   synchronized void deposit(int amt) { ... }
   synchronized void transferTo(int amt, BankAccount a) {
     this.withdraw(amt);
     a.deposit(amt);
   }
}
```

The deadlock

Suppose x and y are static fields holding accounts

```
Thread 1: x.transferTo(1,y)        Thread 2: y.transferTo(1,x)
--------------------------         ---------------------------
acquire lock for x
withdraw 1 from x
                                   acquire lock for y
                                   withdraw 1 from y
                                   block on lock for x
block on lock for y
```

Deadlock in general

- A deadlock occurs when there are threads $T_1, \ldots, T_n$ such that:
  - For $i = 1, .., n-1$, T_i is waiting for a resource held by T_{i+1}
  - T_n is waiting for a resource held by T_1
- In other words, there is a cycle of waiting
  - Can formalize as a graph of dependencies with cycles bad
- Deadlock avoidance in programming amounts to techniques to ensure a cycle can never arise

Back to our example

- Options for deadlock-proof transfer:

1. Make a smaller critical section: `transferTo` not synchronized
   - Exposes intermediate state after `withdraw` before `deposit`
   - May be okay here, but exposes wrong total amount in bank
2. Coarsen lock granularity: one lock for all accounts allowing transfers between them
   - Works, but sacrifices concurrent deposits/withdrawals
3. Give every bank-account a unique number and always acquire locks in the same order
   - Entire program should obey this order to avoid cycles
   - Code acquiring only one lock can ignore the order

Concurrency summary

- Concurrent programming allows multiple threads to access shared resources (e.g., hash table, work queue)
- Introduces new kinds of bugs:
  - Data races and Bad Interleavings
  - Deadlocks
- Requires synchronization
  - Locks for mutual exclusion
  - Other Synchronization Primitives
- Guidelines for correct use help avoid common pitfalls
- Shared Memory model is not only approach, but other approaches (e.g., message passing) are not painless either

Graphs

- Represent relationships that exist between pairs of objects
- Nothing to do with charts and function plots!
- Extremely versatile, can be used to represent many problems

The Graph ADT

A graph $G = (V, E)$

- V is a finite, non-empty set of vertices (or nodes)
- E is a binary relation on V
  (that is, E is a collection of edges that connects pairs of vertices)
- Edges are either *directed* or *undirected*

Applications

- transportation networks (flights, roads, etc.)
  - flights and flight patterns.
  - what sort of questions might we ask? What sort of application might we be interested in having a graph?
  - booking flights, picking shortest time? shortest distance?
  - airlines save fuel, number of people who use the route
- Google maps
  - driving directions, mapping out sightseeing

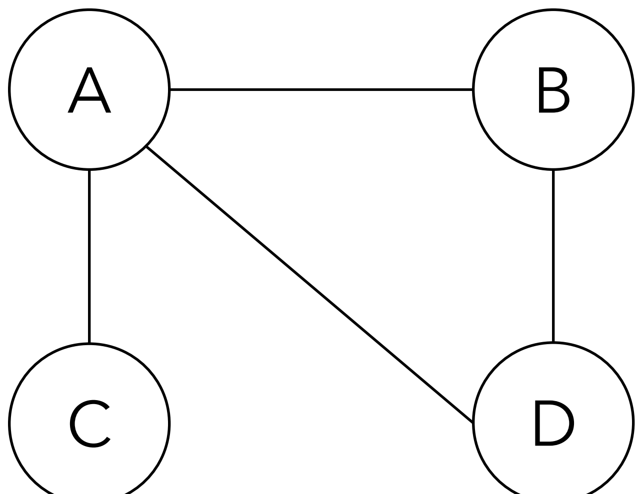
More Applications

- communications networks/utility networks
  - electrical grid, phone networks, computer networks
  - minimize cost for building infrastructure
  - minimize losses, route packets faster
- social networks
  - Does this person know that person.
  - Can this person introduce me to that person – job opportunities

Undirected Graphs

Example: $G = (V, E)$, where

- $V = \{A, B, C, D\}$
- $E = \{\{A, C\}, \{A, B\}, \{A, D\}, \{B, D\}\}$



Definitions for Undirected Graphs

- **subgraph**: is a subset of a graph's edges (and associated vertices) that constitutes a graph.
- **path**: a sequence of connected vertices.
  - **simple path** – a path where all vertices occur only once.
- **path length**: number of edges in the path.
  - Example: path C-A-D-B has length 3.
- **cycle**: path of length ≥ 1 that begins and ends with the same vertex.
  - Example: path A-D-B-A is a cycle.
- **simple cycle**: a simple path that begins and ends with the same vertex.

More Definitions for Undirected Graphs

- **self loop**: Cycle consisting of one edge and one vertex.
- **adjacent vertices**: when connected by an edge.
- **incident edge**: the edge that is incident on two adjacent vertices
    - Edge (A,B) above is incident on adjacent vertices A and B
- **degree**: number of incident edges on a vertex.
- **simple graph**: a graph with no self loops.
- **acyclic graph**: a graph with no cycles.

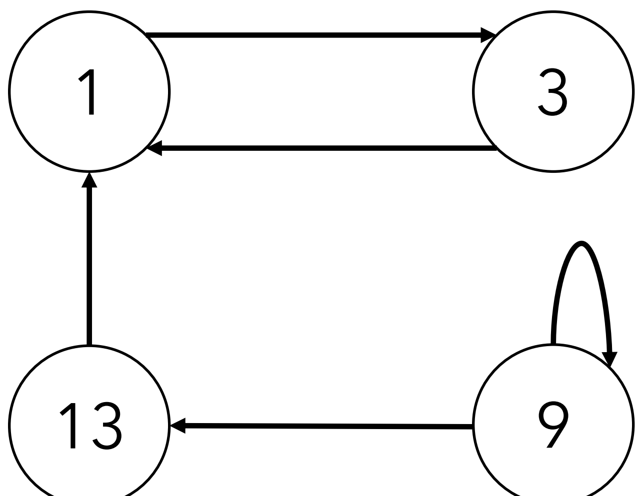
Even More Definitions for Undirected Graphs

- **connected vertices**: if path that connects them exists
- **connected graph**: a graph where every pair of vertices is connected by a path.
- **tree**: acyclic connected graph
- **forest**: disjoint set of trees

Directed Graphs (Digraphs)

Example: $G = (V, E)$, where

- $V = \{1, 3, 9, 13\}$
- $E = \{(1,3), (3, 1), (13,1), (9,9)(9,13)\}$



Definitions for Digraphs

- **subgraph**: subset of a digraph's edges (and associated vertices) that constitutes a digraph.
- **path**: sequence of vertices with a (directed) edge pointing from each vertex to its successor
  - **simple path** – a path where all vertices occur only once.
- **length**: number of edges in the path.
- **cycle**: directed path of length ≥ 1 that begins and ends with the same vertex.
  - **simple cycle**: a simple path that begins and ends with the same vertex.

More Definitions for Digraphs

- **self loop**: Cycle consisting of one edge and one vertex.
  - Example: 9
- **outdegree**: number of edges pointing from it.
- **indegree**: number of edges pointing to it.
- **directed acyclic graph (DAG)**: a digraph with no directed cycles.

Even More Definitions for Digraphs

- **reachable vertices**: when there is a directed path from one to another.
- **strongly connected vertices**: if mutually reachable
- **strongly connected digraph**: directed path from every vertex to every other vertex
- **weakly connected graph**: a digraph that would be connected if all of its directed edges were replaced by undirected edges.