CSE 332 Winter 2026
Lecture 21: Concurrency 3 and Amdahls Law

Nathan Brunelle

http://www.cs.uw.edu/332

Deadlock

- Occurs when two or more threads are mutually blocking each other
- T1 is blocked by T2, which is blocked by T3, …, Tn is blocked by T1
  - A cycle of blocking

- Three requirements for deadlock:
  - Multiple threads each need to acquire **multiple locks**
  - The locks need to be **held at the same time** by the threads
  - The locks may be **acquired in multiple orders**

Bank Account

```
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);
        }
}
```

Deadlock Example

Thread 1:

```
x.transferTo(1,y);
```

Thread 2:

```
y.transferTo(1,x);
```

acquire lock for account x b/c transferTo is synchronized **acquire lock for account y** b/c deposit is synchronized **release lock for account y** after depost **release lock for account x** at end of transferTo	**acquire lock for account y** b/c transferTo is synchronized **acquire lock for account x** b/c deposit is synchronized **release lock for account x** after deposit **release lock for account y** at end of transferTo

Deadlock Example

Thread 1:

```
x.transferTo(1,y);
```

Thread 2:

```
y.transferTo(1,x);
```

acquire lock for account x b/c transferTo is synchronized **acquire lock for account y** b/c deposit is synchronized **release lock for account y** after depost **release lock for account x** at end of transferTo	**acquire lock for account y** b/c transferTo is synchronized **acquire lock for account x** b/c deposit is synchronized **release lock for account x** after deposit **release lock for account y** at end of transferTo

Resolving Deadlocks

- Option 1: Address the “multiple locks” requirement
  - Merge critical sections so you only need one lock
  - E.g. one lock for ALL bank accounts
- Option 2: Address the “held at the same time” requirement
  - Break up critical sections so that only one lock is needed at a time
  - E.g. instead of a synchronized transferTo, have the withdraw and deposit steps locked separately
- Option 3: Address the “acquired in multiple orders” requirement
  - Force all threads to always acquire the locks in the same order
  - E.g. make transferTo acquire both locks before doing either the withdraw or deposit, make sure both threads agree on the order to acquire

Option 1: Coarser Locking

```
static final Object BANK = new Object();
class BankAccount {
        …
        synchronized void withdraw(int amt) {…}
        synchronized void deposit(int amt) {…}
        void transferTo(int amt, BankAccount a) {
                synchronized(BANK){
                        this.withdraw(amt);
                        a.deposit(amt);
                }
        }
}
```

Option 2: Finer Critical Section

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

Option 3: Get All Locks In A Fixed Order

```
class BankAccount {
        …
        synchronized void withdraw(int amt) {…}
        synchronized void deposit(int amt) {…}
        void transferTo(int amt, BankAccount a) {
                if (this.acctNum < a.acctNum){
                        synchronized(this){
                                synchronized(a){
                                        this.withdraw(amt);
                                        a.deposit(amt);
                } } }
                else {
                        synchronized(a){
                                synchronized(this){
                                        this.withdraw(amt);
                                        a.deposit(amt);
                } } }
        }
}
```

Parallel Algorithm Analysis

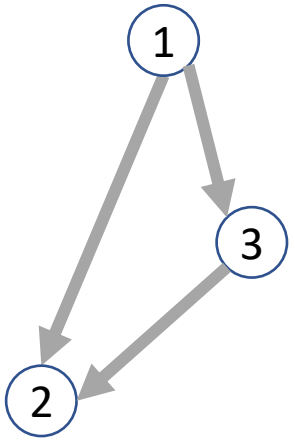
- How to define efficiency
  - Want asymptotic bounds
  - Want to analyze the algorithm without regard to a specific number of processors

Work and Span

- Let $T_P(n)$ be the running time if there are P processors available
- Two key measures of run time:
  - Work: How long it would take 1 processor, so $T_1(n)$
    - Just suppose all forks are done sequentially
    - Cumulative work all processors must complete
    - For array sum: $\Theta(n)$
  - Span: How long it would take an infinite number of processors, so $T_\infty(n)$
    - Theoretical ideal for parallelization
    - Longest “dependence chain” in the algorithm
    - Also called “critical path length” or “computation depth”
    - For array sum: $\Theta(\log n)$

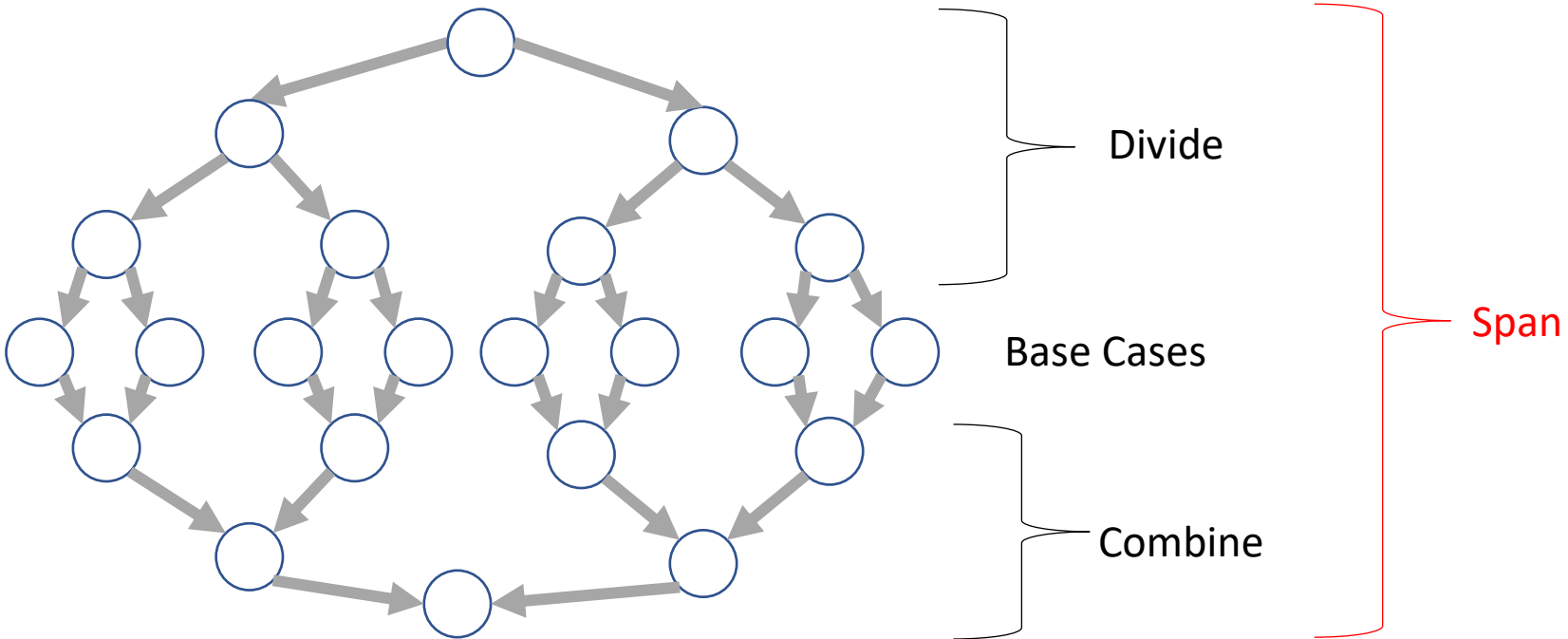
Directed Acyclic Graph (DAG)

- A directed graph that has no cycles
- Often used to depict dependencies
  - E.g. software dependencies, Java inheritance, dependencies among threads!



ForkJoin DAG

- "Sketches" what parts of the algorithm may be done in parallel vs. must be done in-order
  - Each node is a "step" of the algorithm that may depend on other steps (draw an edge) or not
- Fork/Compute each create a new node
  - When calling fork/compute
    - Algorithm creates new threads, there is a dependency from the creating code to the code done by these threads
  - When calling join
    - There is a dependency from the code done by the other thread to the code after join



Work Law

- States that $P \cdot T_P(n) \geq T_1(n)$
  - P processors can do at most P things in parallel
    - Work must match the sum of the operations done by all processors, so if this does not hold then the parallel algorithm somehow skipped steps that sequential version would have done.
  - If the “division of labor” across processors is uneven then it might be that $P \cdot T_P(n) > T_1(n)$

More Vocab

- Speedup:
  - How much faster (than one processor) do we get for more processors
    - Identifies how well the algorithm scales as processors increases
    - May be different for different algorithms
  - $T_1(n)/T_P(n)$
- Perfect linear Speedup
  - The “ideal” speedup
  - $\frac{T_1}{T_P} = P$
- Parallelism
  - Maximum possible speedup
  - T_1/T_∞
  - At some point more processors won’t be more helpful, when that point is depends on the span
- Writing parallel algorithms is about increasing span without substantially increasing work

Asymptotically Optimal T_P

- T_P cannot be better than $\frac{T_1}{P}$
  - Because of the Work Law
- T_P cannot be better than T_∞
  - A finite number of processors can't outperform an infinite number ("Span Law")
- Considering both of these, we can characterize the best-case scenario for T_P
  - $T_P(n) \in \Omega\left(\frac{T_1(n)}{P} + T_\infty(n)\right)$
  - $T_1(n)/P$ dominates for small P, $T_\infty(n)$ dominates for large P
- ForkJoin Framework gives an expected time guarantee of asymptotically optimal!

And now for some bad news…

- In practice it's common for your program to have:
  - Parts that parallelize well
    - Maps/reduces/filters over arrays and other data structures
    - Anything ForkJoin!
    - Tasks that don't need to access a shared data structure
  - Parts that don't parallelize at all
    - Reading a linked list, getting input, computations where each step needs the results of previous step, concurrency concerns forcing mutual exclusion
- These unparallelizable parts can turn out to be a big bottleneck

Amdahl's Law (mostly bad news)

- Suppose $T_1 = 1$
  - Work for the entire program is 1
- Let S be the proportion of the program that cannot be parallelized
  - $T_1 = S + (1 - S) = 1$
- Suppose we get perfect linear speedup on the parallel portion
  - $T_P = S + \frac{1-S}{P}$
- For the entire program, the speedup is:
  - $\frac{T_1}{T_P} = \frac{1}{S+\frac{1-S}{P}}$
- The parallelism (infinite processors) is:
  - $\frac{T_1}{T_\infty} = \frac{1}{S}$

Amdahl's Law Example

- Suppose 2/3 of your program is parallelizable, but 1/3 is not.
  - $S = \frac{1}{3}$
  - $T_1 = \frac{2}{3} + \frac{1}{3} = 1$
- $T_P = S + \frac{1-S}{P} = \frac{1}{3} + \frac{2/3}{P}$
- If T_1 is 100 seconds:
  - $T_P = 33 + \frac{67}{P}$
  - $T_2 = 33 + \frac{67}{2} \approx 67$
  - $T_3 = 33 + \frac{67}{3} \approx 55$
  - $T_6 = 33 + \frac{67}{6} \approx 44$
  - $T_{12} = 33 + \frac{67}{12} \approx 39$

Notice:
- We got a lot of speedup with the second processor (23%)
- Adding a third processor only gave half as much speedup (12%)
- After going up to 6 processors, we still only got 11% speedup
- No matter how many processors we add, we will never get more 11% additional improvement

Conclusion:
When a portion of our code is sequential, the improvement gained from more and more processors diminishes very quickly.

Conclusion

- Even with many *many* processors the sequential part of your program becomes a bottleneck
- Parallelizable code requires skill and insight from the developer to recognize where parallelism is possible, and how to do it well.

Parallel Code Conventional Wisdom

Reasons to Use Parallelism

- Code Responsiveness:
  - While doing an expensive computation, you don’t what your interface to freeze
- Processor Utilization:
  - If one thread is waiting on a deep-hierarchy memory access you can still use that processor time
- Failure Isolation:
  - If one portion of your code fails, it will only crash that one portion.

Memory Categories

All memory must fit one of three categories:

1. Thread Local: Each thread has its own copy
2. Shared and Immutable: There is just one copy, but nothing will ever write to it
3. Shared and Mutable: There is just one copy, it may change
   - Requires Synchronization!

Thread Local Memory

- Whenever possible, avoid sharing resources
- Dodges all race conditions, since no other threads can touch it!
  - No synchronization necessary!
- Use whenever threads do not need to communicate using the resource
  - E.g., each thread should have its own Random object
- In most cases, most objects should be in this category

Immutable Objects

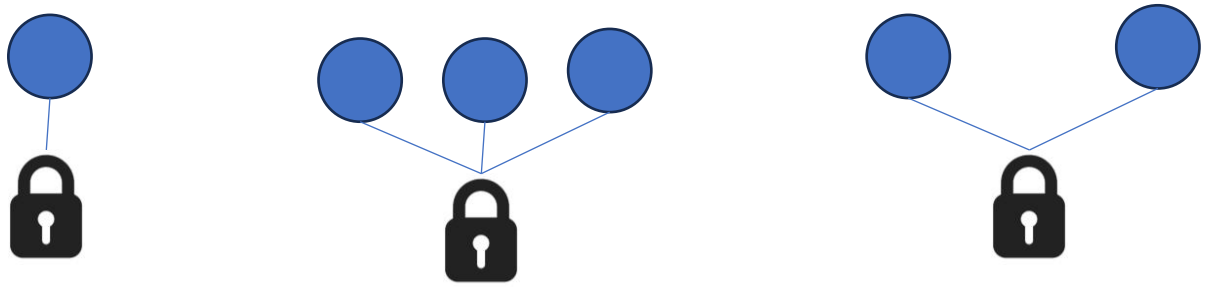
- Whenever possible, avoid changing objects
  - Make new objects instead
- Parallel reads are not data races
  - If an object is never written to, no synchronization necessary!
- Many programmers over-use mutation, minimize it

Shared and Mutable Objects

- For everything else, use locks
- Avoid all data races
  - Every read and write should be projected with a lock, even if it “seems safe”
  - Almost every Java/C program with a data race is wrong
- Even without data races, it still may be incorrect
  - Watch for bad interleavings as well!

Consistent Locking

- For each location needing synchronization, have a lock that is always held when reading or writing the location
- The same lock can (and often should) “guard” multiple fields/objects
  - Clearly document what each lock guards!
  - In Java, the lock should usually be the object itself (i.e. “this”)
- Have a mapping between memory locations and lock objects and stick to it!



Lock Granularity

- Coarse Grained: Fewer locks guarding more things each
  - One lock for an entire data structure
  - One lock shared by multiple objects (e.g. one lock for all bank accounts)
- Fine Grained: More locks guarding fewer things each
  - One lock per data structure location (e.g. array index)
  - One lock per object or per field in one object (e.g. one lock for each account)
- Note: there’s really a continuum between them…

Example: Separate Chaining Hashtable

- Maximally Coarse-grained: One lock for the entire hashtable
- Maximally Fine-grained: One lock for each bucket
- Which supports more parallelism in insert and find?
- Which makes rehashing easier?
- What happens if you want to have a size field?

Example: Separate Chaining Hashtable

- Maximally Coarse-grained: One lock for the entire hashtable
- Maximally Fine-grained: One lock for each bucket
- Which supports more parallelism in insert and find?
  - Fine-grained, since inserts and finds can happen in parallel when there is no collision
- Which makes rehashing easier?
  - Course-grained, as with fine-grained you would need to acquire ALL of the locks
- What happens if you want to have a size field?
  - For fine-grained, every insert would need to update the size, so there's risk of a data race

Tradeoffs

- Coarse-Grained Locking:
  - Simpler to implement and avoid race conditions
  - 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 Locking:
  - More simultaneous access (performance when coarse grained would lead to unnecessary blocking)
  - Can make multi-location operations more difficult: say, rotations in an AVL tree
- Guideline:
  - Start with coarse-grained, make finer only as necessary to improve performance

Similar But Separate Issue: Critical Section Granularity

- Coarse-grained
  - For every method that needs a lock, put the entire method body in a lock
- Fine-grained
  - Keep the lock only for the sections of code where it's necessary
- Guideline:
  - Try to structure code so that expensive operations (like I/O) can be done outside of your critical section
  - E.g., if you're trying to print all the values in a tree, maybe copy items into an array inside your critical section, then print the array's contents outside.

Atomicity

- Atomic: indivisible
- Atomic operation: one that should be thought of as a single step
- Some sequences of operations should behave as if they are one unit
  - Between two operations you may need to avoid exposing an intermediate state
  - Usually ADT operations should be atomic
    - You don’t want another thread trying to do an insert while another thread is rotating the AVL tree
- Think first in terms of what operations need to be atomic
  - Design critical sections and locking granularity based on these decisions

Use Pre-Tested Code

- Whenever possible, use built-in libraries!
- Other people have already invested tons of effort into making things both efficient and correct, use their work when you can!
  - Especially true for concurrent data structures
  - Use thread-safe data structures when available
    - E.g. Java as ConcurrentHashMap