CSE 332 Spring 2026
Lecture 22: Amdahl's Law and Mutual Exclusion

Nathan Brunelle

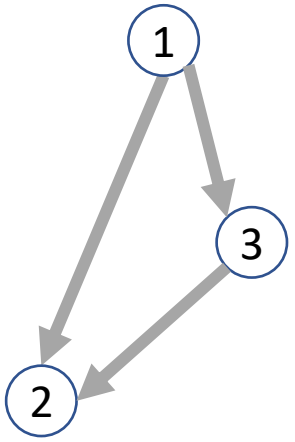
http://www.cs.uw.edu/332

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: $\Theta(\log n)$

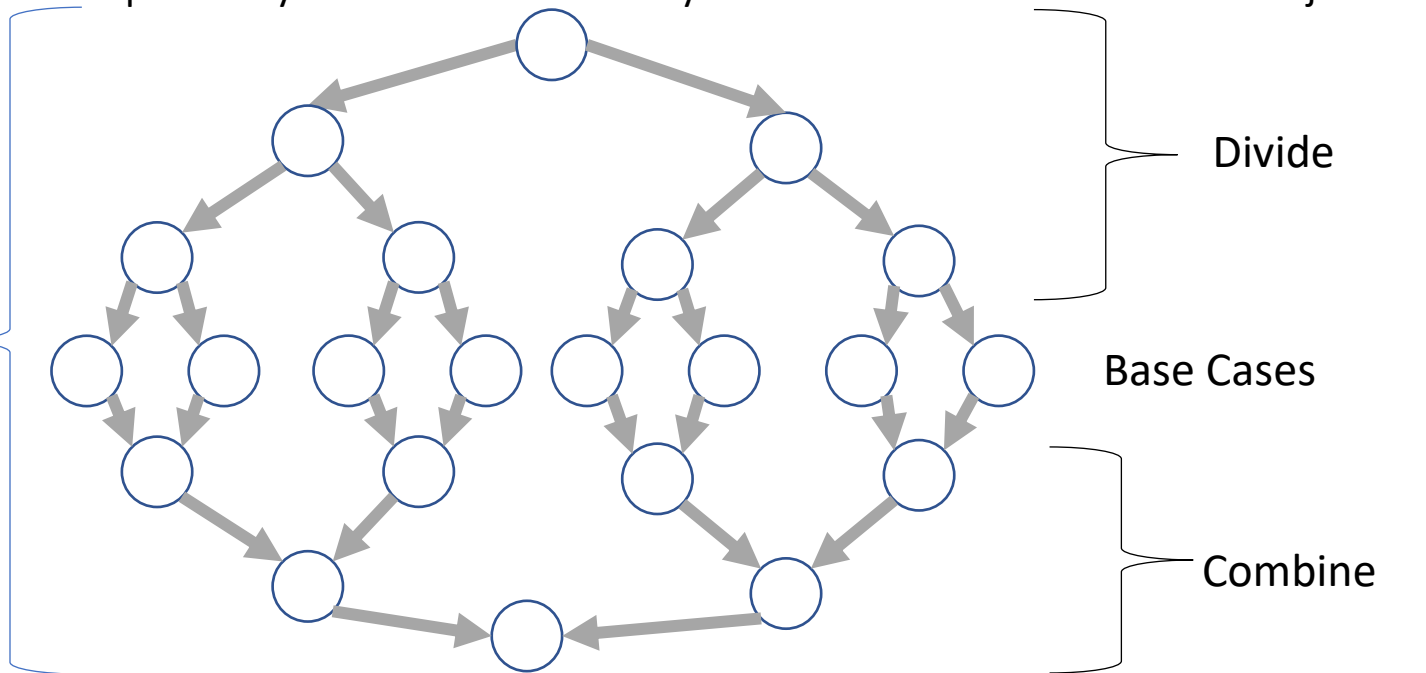
Directed Acyclic Graph (DAG)

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



Task Dependency Graph

- "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)
  - This graph depicts a forkjoin algorithm. Each step takes constant time.
- Fork and Join each create a new node
  - When calling fork/compute
    - Algorithm creates two 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



Σ gives work

Work = sum of the running times of all nodes

max path gives span

Span = longest path from start node to end node.

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 $P \cdot T_P(n) > T_1(n)$

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!

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

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.

Memory Sharing With ForkJoin

- Structure of ForkJoin:
  - All threads are executing the same algorithm
  - Each task is responsible for **its own portion** of the input/output
  - If one task needs another's result, use join() to ensure it uses the final answer

- This does not help when:
  - Memory accessed by threads is overlapping or unpredictable
  - Threads are doing independent tasks using same resources (rather than implementing the same algorithm)

Example: Shared Queue

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

Imagine two threads are both using the same linked list based queue.

What could go wrong?

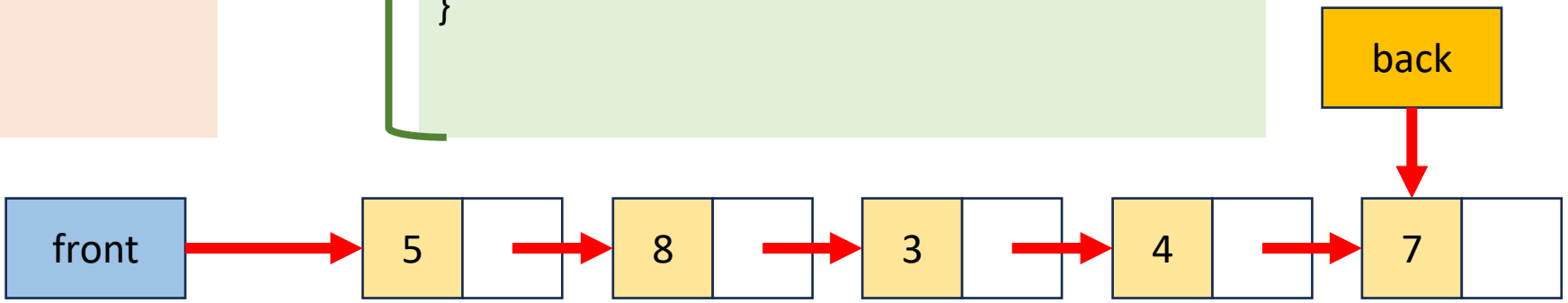
Shared Queue Interleaving

Thread 1

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

Thread 2

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```



Empty Shared Queue

Suppose the Queue is empty, and the threads execute in this order.
Assume Thread 1 enqueues 1, and Thread 2 enqueues 2.

Thread 1

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

Thread 2

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

Thread 1 – first two lines

Thread 1 has decided the queue is empty

Thread 1

```
~~enqueue(x){~~
~~        if ( back == null ){~~
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

Thread 2

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

front

back

Thread 2 – all lines

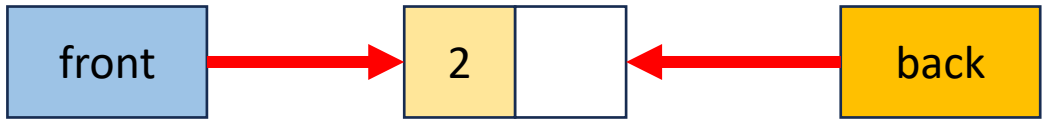
Thread 1 has decided the queue is empty
Meanwhile, thread 2 enqueues 2

Thread 1

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

Thread 2

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```



Thread 1 – remaining lines

Thread 1 has decided the queue is empty
Meanwhile, thread 2 enqueues 2
Thread 1 continues as if the queue is still empty, overwriting Thread 2's work

Thread 1

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

Thread 2

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

Interleaving

- Due to time slicing, a thread can be interrupted at any time
  - Between any two lines of code
  - Within a single line of code
- The sequence that operations occur across two threads is called an interleaving
- Without doing anything else, we have no control over how different threads might be interleaved

Concurrent Programming

- Concurrency:
  - Correctly and efficiently managing access to shared resources across multiple possibly-simultaneous tasks
- Requires synchronization to avoid incorrect simultaneous access
  - Use some way of “blocking” other tasks from using a resource when another modifies it or makes decisions based on its state
  - That blocking task will free up the resource when it’s done
- Warning:
  - Because we have no control over when threads are scheduled by the OS, even correct implementations are highly non-deterministic
  - Errors are hard to reproduce, which complicates debugging

Analogue Example – Data Race

1. Count to 10 aloud
2. Clap three times
3. Erase the contents of the box
4. Write your name in the box,
   pausing 2 seconds between letters
5. Wave your arms in the air

Analogue Example – With “Mutual Exclusion”

1. Count to 10 aloud
2. Clap three times
3. **Pick up the trident**
4. Erase the contents of the box
5. Write your name in the box,
   pausing 2 seconds between letters
6. **Put down the trident**
7. Wave your arms in the air

There is only one trident, so no two threads (both executing this same code) can execute lines 4 and 5 at the same time.

All threads after the first must wait at line 3 for the trident to become available.

The trident causes any one thread to “lock out” (exclude) all other threads.

Any thread being between lines 3 and 6 excludes all other threads, so this is called “mutual exclusion”.

The trident itself we call a “lock”

Bank Account Example

- The following code implements a bank account object correctly for a synchronized situation
- Assume the initial balance is 150

```
class BankAccount {
        private int balance = 0;
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                int b = getBalance();
                if (amount > b)
                        throw new WithdrawTooLargeException();
                setBalance(b – amount); }
        // other operations like deposit, etc.
}
```

What Happens here?

```
withdraw(100);
withdraw(75)
```

Bank Account Example - Parallel

- Assume the initial balance is 150

```
class BankAccount {
        private int balance = 0;
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                int b = getBalance();
                if (amount > b)
                        throw new WithdrawTooLargeException();
                setBalance(b – amount); }
        // other operations like deposit, etc.
}
```

Thread 1:

```
withdraw(100);
```

Thread 2:

```
withdraw(75);
```

An “OK” Interleaving

- Assume the initial balance is 150

Thread 1:

```
withdraw(100);
```

Thread 2:

```
withdraw(75);
```

int b = getBalance();
if (amount > b)
throw new Exception();
setBalance(b – amount);
	int b = getBalance();
	if (amount > b)
	throw new Exception();
	setBalance(b – amount);

A “Bad” Interleaving

- Assume the initial balance is 150

Thread 1:

```
withdraw(100);
```

Thread 2:

```
withdraw(75);
```

int b = getBalance();
 	
if (amount > b)
  throw new Exception();
setBalance(b – amount);
	int b = getBalance();
	if (amount > b)
	  throw new Exception();
	setBalance(b – amount);

A Bad Fix

- Assume the initial balance is 150

```
class BankAccount {
        private int balance = 0;
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                if (amount > getBalance())
                        throw new WithdrawTooLargeException();
                setBalance(getBalance() – amount); }
        // other operations like deposit, etc.
}
```

Avoiding Variables doesn't work

- Assume the initial balance is 150

Thread 1:

```
withdraw(100);
```

Thread 2:

```
withdraw(75);
```

if (amount > getBalance())
throw new Exception();
setBalance(getBalance() – amount);
 	
setBalance(getBalance() – amount);
	if (amount > getBalance())
	throw new Exception();
	
	setBalance(getBalance() – amount);

What we want – Mutual Exclusion

- While one thread is withdrawing from the account, we want to exclude all other threads from also withdrawing
- Called mutual exclusion:
  - One thread using a resource (here: a bank account) means another thread must wait
  - We call the area of code that we want to have mutual exclusion (only one thread can be there at a time) a **critical section**.
- The programmer must implement critical sections!
  - It requires programming language primitives to do correctly

A Bad attempt at Mutual Exclusion

```
class BankAccount {
        private int balance = 0;
        private Boolean busy = false;
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                while (busy) { /* wait until not busy */ }
                busy = true;
                int b = getBalance();
                if (amount > b)
                        throw new WithdrawTooLargeException();
                setBalance(b – amount);
                busy = false;}
        // other operations like deposit, etc.
}
```

The “Busy Signal” Doesn’t Work

- Assume the initial balance is 150

Thread 1:

```
withdraw(100);
```

Thread 2:

```
withdraw(75);
```

```
while (busy) { /* wait until not busy */ }

busy = true;

int b = getBalance();




if (amount > b)
        throw new Exception();
setBalance(b – amount);
busy = false;
```

```
while (busy) { /* wait until not busy */ }

busy = true;

int b = getBalance();
if (amount > b)
        throw new Exception();
setBalance(b – amount);
busy = false;
```

Solution

- We need a construct from Java to do this
- One Solution – A **Mutual Exclusion Lock** (called a Mutex or Lock)
- We define a **Lock** to be an ADT with operations:
  - New:
    - make a new lock, initially “not held”
  - Acquire:
    - If lock is not held, mark it as “held”
      - These two steps always done together in a way that cannot be interrupted!
    - If lock is held, pause until it is marked as “not held”
  - Release:
    - Mark the lock as “not held”

Almost Correct Bank Account Example

```
class BankAccount {
        private int balance = 0;
        private Lock lck = new Lock();
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                lk.acquire();
                int b = getBalance();
                if (amount > b)
                        throw new WithdrawTooLargeException();
                setBalance(b – amount);
                lk.release();}
        // other operations like deposit, etc.
}
```

Questions:

1. What is the critical section?
2. What is the Error?

Exceptions End the Method!

```
class BankAccount {
        private int balance = 0;
        private Lock lck = new Lock();
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                lk.acquire();
                int b = getBalance();
                if (amount > b)
                        throw new WithdrawTooLargeException();
                setBalance(b – amount);
                lk.release();}
        // other operations like deposit, etc.
}
```

If we throw an exception, we never finish the method!

The lock would never get released!

Try…Finally

- Try Block:
  - Body of code that will be run
- Finally Block:
  - Always runs once the program exits try block (whether due to a return, exception, anything!)

Correct (but not Java) Bank Account Example

```
class BankAccount {
        private int balance = 0;
        private Lock lck = new Lock();
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                try{
                        lk.acquire();
                        int b = getBalance();
                        if (amount > b)
                                throw new WithdrawTooLargeException();
                        setBalance(b – amount); }
                finally { lk.release(); } }
        // other operations like deposit, etc.
}
```

Questions:

1. Should deposit have its own lock object?
2. What about getBalance?
3. What about setBalance?
4. Should they share one?