CSE 332

AUGUST 4TH – SYNCHRONIZATION AND INTRO TO THE GRAPH

ADMINISTRIVIA

- **P3 checkpoint today**

ADMINISTRIVIA

- P3 checkpoint today
- P2 back next week

ADMINISTRIVIA

- P3 checkpoint today
- P2 back next week
- Moving onto Graphs by end of lecture today

REVIEW

- **Concurrency**
  - Dealing many things at once, the OS does this automatically

REVIEW

- **Concurrency**
  - Dealing with many things at once, the OS does this automatically
- **Parallelism**
  - Breaking a problem down so that multiple pieces can be done at once

REVIEW

- **Concurrency**
  - Dealing with many things at once, the OS does this automatically
- **Parallelism**
  - Breaking a problem down so that multiple pieces can be done at once
- **Synchronization**
  - Making sure that threads don’t interfere with each other while they’re running in parallel

CONCURRENCY

- Not covered in this class

CONCURRENCY

- **Not covered in this class**
  - Largely moderated by the OS

CONCURRENCY

- **Not covered in this class**
  - Largely moderated by the OS
  - Ensures that valuable computer resources are being used as effectively as possible

CONCURRENCY

- **Not covered in this class**
  - Largely moderated by the OS
  - Ensures that valuable computer resources are being used as effectively as possible
  - This has consequences in parallelism and synchronization, but on it's own, it is just about running objects at the same time

CONCURRENCY

- **Not covered in this class**
  - Largely moderated by the OS
  - Ensures that valuable computer resources are being used as effectively as possible
  - This has consequences in parallelism and synchronization, but on it's own, it is just about running objects at the same time
  - See CSE 333 for more!

PARALLEL REVIEW

- **Parallelism in Java works around the ForkJoin framework**

PARALLEL REVIEW

- **Parallelism in Java works around the ForkJoin framework**
  - You create a pool and parallelize using invoke()

PARALLEL REVIEW

- **Parallelism in Java works around the ForkJoin framework**
  - You create a pool and parallelize using invoke()
  - You must invoke an object of type recursive task, which must have a constructor and a compute() function

PARALLEL REVIEW

- **Parallelism in Java works around the ForkJoin framework**
  - You create a pool and parallelize using invoke()
  - You must invoke an object of type recursive task, which must have a constructor and a compute() function
  - Whenever you call fork() on a RT object, it begins a new thread and begins work with the data it's been initialized

PARALLEL REVIEW

- **Limitations of ForkJoin**

PARALLEL REVIEW

- **Limitations of ForkJoin**
  - Threads only communicate at creation and death

PARALLEL REVIEW

- **Limitations of ForkJoin**
  - Threads only communicate at creation and death
  - All input data should be immutable

PARALLEL REVIEW

- **Limitations of ForkJoin**
  - Threads only communicate at creation and death
  - All input data should be immutable

PARALLEL REVIEW

- **Limitations of ForkJoin**
  - Threads only communicate at creation and death
  - All input data should be immutable
  - The threads must all operate on the same code

PARALLEL REVIEW

- **Limitations of ForkJoin**
  - Threads only communicate at creation and death
  - All input data should be immutable
  - The threads must all operate on the same code
  - Need to be smart about how threads are created and destroyed in order to maximize the benefit

PARALLEL REVIEW

- **Limitations of ForkJoin**
  - Threads only communicate at creation and death
  - All input data should be immutable
  - The threads must all operate on the same code
  - Need to be smart about how threads are created and destroyed in order to maximize the benefit
  - Each thread needs to do some work, there should never be a “moderating” thread

PARALLEL REVIEW

- **Limitations of ForkJoin**
  - Threads only communicate at creation and death
  - All input data should be immutable
  - The threads must all operate on the same code
  - Need to be smart about how threads are created and destroyed in order to maximize the benefit
  - Each thread needs to do some work, there should never be a “moderating” thread
  - fork(); compute(); join();

PARALLEL REVIEW

- **Parallelism takes advantage of having more than one thing being able to execute at a time**

PARALLEL REVIEW

- **Parallelism takes advantage of having more than one thing being able to execute at a time**
  - We analyze asymptotically this using work and span
    - Work – The input-dependent runtime for sequential computation
    - Span – the runtime of a parallelized algorithm given infinite processors – this will not always be O(1)!

PARALLEL REVIEW

- **Parallelism takes advantage of having more than one thing being able to execute at a time**
  - We analyze asymptotically this using work and span
    - Work – The input-dependent runtime for sequential computation
    - Span – the runtime of a parallelized algorithm given infinite processors – this will not always be O(1)!
    - Speed up is the amount of time we save given P processors:
      - We can lower bound with: $T_p = T_1/P + T_{inf}$
      - If we have 4 procesors, and we have speed up of 4, then we have perfect linear speed up

PARALLEL REVIEW

- **Parallelism takes advantage of having more than one thing being able to execute at a time**
  - We analyze asymptotically this using work and span
    - Work – The input-dependent runtime for sequential computation
    - Span – the runtime of a parallelized algorithm given infinite processors – this will not always be O(1)!
    - Speed up is the amount of time we save given P processors:
      - We can lower bound with: $T_p = T_1/P + T_{inf}$
      - If we have 4 procesors, and we have speed up of 4, then we have perfect linear speed up

PARALLEL REVIEW

- **Parallelism takes advantage of having more than one thing being able to execute at a time**
  - We analyze asymptotically this using work and span
    - Work – The input-dependent runtime for sequential computation
    - Span – the runtime of a parallelized algorithm given infinite processors – this will not always be O(1)!
    - Speed up is the amount of time we save given P processors:
      - We can lower bound with: $T_p = T_1/P + T_{inf}$
      - If we have 4 procesors, and we have speed up of 4, then we have perfect linear speed up

PARALLEL REVIEW

- **Analyzing work and span**

PARALLEL REVIEW

- **Analyzing work and span**
  - This follows along a similar recurrence. Except, when we consider 2T(N/2) elements, we can reduce them to T(N/2) because we can run both of those smaller tasks in parallel

PARALLEL REVIEW

- **Analyzing work and span**
  - This follows along a similar recurrence. Except, when we consider 2T(N/2) elements, we can reduce them to T(N/2) because we can run both of those smaller tasks in parallel
  - Some tasks can be parallelized beyond this using the parallel primitives, so base-cases and constants can be changed too

PARALLEL REVIEW

- **Analyzing work and span**
  - This follows along a similar recurrence. Except, when we consider 2T(N/2) elements, we can reduce them to T(N/2) because we can run both of those smaller tasks in parallel
  - Some tasks can be parallelized beyond this using the parallel primitives, so base-cases and constants can be changed too
  - For example, quicksort's recurrence is:
    - $T(n) = O(n) + 2T(N/2)$

PARALLEL REVIEW

- **Analyzing work and span**
  - This follows along a similar recurrence. Except, when we consider 2T(N/2) elements, we can reduce them to T(N/2) because we can run both of those smaller tasks in parallel
  - Some tasks can be parallelized beyond this using the parallel primitives, so base-cases and constants can be changed too
  - For example, quicksort's recurrence is:
    - $T(n) = O(n) + 2T(N/2)$
  - But the parallel recurrence is:
    - $T(n) = O(\log n) + T(N/2)$ – we can use a parallel pack!

PARALLEL REVIEW

- **Analyzing work and span**
  - This follows along a similar recurrence. Except, when we consider 2T(N/2) elements, we can reduce them to T(N/2) because we can run both of those smaller tasks in parallel
  - Some tasks can be parallelized beyond this using the parallel primitives, so base-cases and constants can be changed too
  - For example, quicksort's recurrence is:
    - $T(n) = O(n) + 2T(N/2)$
  - But the parallel recurrence is:
    - $T(n) = O(\log n) + T(N/2)$
  - This is $\log^2 n$, which is very fast

PARALLEL REVIEW

- **We discussed four primary parallel primitives**

PARALLEL REVIEW

- **We discussed four primary parallel primitives**
  - Reduce – getting a single value from an input array

PARALLEL REVIEW

- **We discussed four primary parallel primitives**
  - Reduce – getting a single value from an input array
  - Map – creating a new array that where elements from the original array have been mutated by a constant function (that does not require input from other elements)

PARALLEL REVIEW

- **We discussed four primary parallel primitives**
  - Reduce – getting a single value from an input array
  - Map – creating a new array that where elements from the original array have been mutated by a constant function (that does not require input from other elements)
  - Scan – Creates a modified array where the result depends on elements that came before it, (partial sum example)

PARALLEL REVIEW

- **We discussed four primary parallel primitives**
  - Reduce – getting a single value from an input array
  - Map – creating a new array that where elements from the original array have been mutated by a constant function (that does not require input from other elements)
  - Scan – Creates a modified array where the result depends on elements that came before it, (partial sum example)
  - Pack – Filters an array to produce only elements subject to a certain condition

PARALLEL REVIEW

- **Parallel Primitives**

PARALLEL REVIEW

- **Parallel Primitives**
  - Easier ways to break down more common problems into reasonable pieces

PARALLEL REVIEW

- **Parallel Primitives**
  - Easier ways to break down more common problems into reasonable pieces
  - These approaches are seen all the time, so be prepared to use them to solve and parallelize problem types that you've never seen before

PARALLEL REVIEW

- **Parallel Primitives**
  - Easier ways to break down more common problems into reasonable pieces
  - These approaches are seen all the time, so be prepared to use them to solve and parallelize problem types that you've never seen before
- **Cutoffs**

PARALLEL REVIEW

- **Parallel Primitives**
  - Easier ways to break down more common problems into reasonable pieces
  - These approaches are seen all the time, so be prepared to use them to solve and parallelize problem types that you've never seen before
- **Cutoffs**
  - Creating new threads takes a lot of overhead, at a certain point, it is faster to do the work sequentially

PARALLEL REVIEW

- **Parallel Primitives**
  - Easier ways to break down more common problems into reasonable pieces
  - These approaches are seen all the time, so be prepared to use them to solve and parallelize problem types that you've never seen before
- **Cutoffs**
  - Creating new threads takes a lot of overhead, at a certain point, it is faster to do the work sequentially
  - Don't make unnecessary threads and use divide and conquer to create new ones.

PARALLEL REVIEW

- **Parallel Primitives**
  - Easier ways to break down more common problems into reasonable pieces
  - These approaches are seen all the time, so be prepared to use them to solve and parallelize problem types that you've never seen before
- **Cutoffs**
  - Creating new threads takes a lot of overhead, at a certain point, it is faster to do the work sequentially
  - Don't make unnecessary threads and use divide and conquer to create new ones.
- **Memory hierarchy**

PARALLEL REVIEW

- **Parallel Primitives**
  - Easier ways to break down more common problems into reasonable pieces
  - These approaches are seen all the time, so be prepared to use them to solve and parallelize problem types that you've never seen before
- **Cutoffs**
  - Creating new threads takes a lot of overhead, at a certain point, it is faster to do the work sequentially
  - Don't make unnecessary threads and use divide and conquer to create new ones.
- **Memory hierarchy**
  - We didn't explicitly talk about this, but it does make a difference

CONCURRENCY

- **Not all parallelism falls under the constraint of the ForkJoin**

CONCURRENCY

- **Not all parallelism falls under the constraint of the ForkJoin**
  - Shared memory can be problematic

CONCURRENCY

- **Not all parallelism falls under the constraint of the ForkJoin**
  - Shared memory can be problematic
  - Race conditions can occur if our output can be made incorrect by OS scheduling

CONCURRENCY

- **Not all parallelism falls under the constraint of the ForkJoin**
  - Shared memory can be problematic
  - Race conditions can occur if our output can be made incorrect by OS scheduling
  - Need protection from a lock or mutex to ensure that *critical sections* are able to ensure *mutual exclusion* (where mutex comes from)

CONCURRENCY

- **Locking**

CONCURRENCY

- **Locking**
  - Need to lock resources so that they can be safely accessed by multiple threads

CONCURRENCY

- **Locking**
  - Need to lock resources so that they can be safely accessed by multiple threads
  - Easy to provide a mutex to lock organized code

CONCURRENCY

- **Locking**
  - Need to lock resources so that they can be safely accessed by multiple threads
  - Easy to provide a mutex to lock organized code
  - Java also provides the *synchronized* framework that allows you to restrict access to code based on ownership of an object.

CONCURRENCY

- **Locking**
  - Need to lock resources so that they can be safely accessed by multiple threads
  - Easy to provide a mutex to lock organized code
  - Java also provides the *synchronized* framework that allows you to restrict access to code based on ownership of an object.
  - Mutex supports two primary functions:

CONCURRENCY

- **Locking**
  - Need to lock resources so that they can be safely accessed by multiple threads
  - Easy to provide a mutex to lock organized code
  - Java also provides the *synchronized* framework that allows you to restrict access to code based on ownership of an object.
  - Mutex supports two primary functions:
    - lock() attempt to monopolize the resource and if it's unavailable, stall until it is

CONCURRENCY

- **Locking**
  - Need to lock resources so that they can be safely accessed by multiple threads
  - Easy to provide a mutex to lock organized code
  - Java also provides the *synchronized* framework that allows you to restrict access to code based on ownership of an object.
  - Mutex supports two primary functions:
    - lock() attempt to monopolize the resource and if it's unavailable, stall until it is
    - unlock() signal that your critical section is complete and that other threads may use the resource

CONCURRENCY

- **Deadlock**

CONCURRENCY

- **Deadlock**
  - Deadlock occurs when threads need access to multiple resources to continue

CONCURRENCY

- **Deadlock**
  - Deadlock occurs when threads need access to multiple resources to continue
  - Multiple strategies for solving

CONCURRENCY

- **Deadlock**
  - Deadlock occurs when threads need access to multiple resources to continue
  - Multiple strategies for solving
    - Random drop and try again

CONCURRENCY

- **Deadlock**
  - Deadlock occurs when threads need access to multiple resources to continue
  - Multiple strategies for solving
    - Random drop and try again
    - Meta-locks to grab common combinations at once

CONCURRENCY

- **Deadlock**
  - Deadlock occurs when threads need access to multiple resources to continue
  - Multiple strategies for solving
    - Random drop and try again
    - Meta-locks to grab common combinations at once
    - Provide a computation ordering—know which threads are more important to complete

CONCURRENCY

- **Deadlock**
  - Deadlock occurs when threads need access to multiple resources to continue
  - Multiple strategies for solving
    - Random drop and try again
    - Meta-locks to grab common combinations at once
    - Provide a computation ordering—know which threads are more important to complete
  - Even with this, we can have process starvation if high priority processes keep reexecuting

CONCURRENCY

- **Deadlock**
  - Deadlock occurs when threads need access to multiple resources to continue
  - Multiple strategies for solving
    - Random drop and try again
    - Meta-locks to grab common combinations at once
    - Provide a computation ordering—know which threads are more important to complete
  - Even with this, we can have process starvation if high priority processes keep reexecuting
    - The OS can usually prevent total starvation, but instituting a thread hierarchy can be difficult if threads are starting over with frequency.

CONCURRENCY

- **Concurrent design**

CONCURRENCY

- **Concurrent design**
  - Avoid data races using the mutex (must recognize when multiple threads can interleave and interrupt each other)

CONCURRENCY

- **Concurrent design**
  - Avoid data races using the mutex (must recognize when multiple threads can interleave and interrupt each other)
  - Use locks consistently and clearly indicate (in code comments) what the lock is protecting

CONCURRENCY

- **Concurrent design**
  - Avoid data races using the mutex (must recognize when multiple threads can interleave and interrupt each other)
  - Use locks consistently and clearly indicate (in code comments) what the lock is protecting
  - In general, fewer locks are better, moving to thread specific or immutable memory may be a way to reduce the need

CONCURRENCY

- **Concurrent design**
  - Avoid data races using the mutex (must recognize when multiple threads can interleave and interrupt each other)
  - Use locks consistently and clearly indicate (in code comments) what the lock is protecting
  - In general, fewer locks are better, moving to thread specific or immutable memory may be a way to reduce the need
  - Have a granularity in mind

CONCURRENCY

- **Granularity**

CONCURRENCY

- **Granularity**
  - Fine-grained granularity means that there are more locks that protect smaller resources

CONCURRENCY

- **Granularity**
  - Fine-grained granularity means that there are more locks that protect smaller resources
    - Allows for more simultaneous access

CONCURRENCY

- **Granularity**
  - Fine-grained granularity means that there are more locks that protect smaller resources
    - Allows for more simultaneous access
    - Increases the likelihood that threads need more than one resource – deadlock

CONCURRENCY

- **Granularity**
  - Fine-grained granularity means that there are more locks that protect smaller resources
    - Allows for more simultaneous access
    - Increases the likelihood that threads need more than one resource – deadlock
  - Course-grained granularity
    - Simpler implementation

CONCURRENCY

- **Granularity**
  - Fine-grained granularity means that there are more locks that protect smaller resources
    - Allows for more simultaneous access
    - Increases the likelihood that threads need more than one resource – deadlock
  - Course-grained granularity
    - Simpler implementation
    - Less concurrency, lots of threads are always waiting for the large resource locks, even if they just need a little piece

CONCURRENCY

- **Hash table example**

CONCURRENCY

- **Hash table example**
  - Do we lock the whole HT or do we only lock the individual boxes?

CONCURRENCY

- **Hash table example**
  - Do we lock the whole HT or do we only lock the individual boxes?
    - Individual boxes:

CONCURRENCY

- **Hash table example**
  - Do we lock the whole HT or do we only lock the individual boxes?
    - Individual boxes:
      - Boxes don’t interfere, we can allow more concurrency

CONCURRENCY

- **Hash table example**
  - Do we lock the whole HT or do we only lock the individual boxes?
    - Individual boxes:
      - Boxes don’t interfere, we can allow more concurrency
      - We have more boxes, and more overhead, difficult to have critical sections that rely on multiple things in the same HT

CONCURRENCY

- **Hash table example**
  - Do we lock the whole HT or do we only lock the individual boxes?
    - Individual boxes:
      - Boxes don’t interfere, we can allow more concurrency
      - We have more boxes, and more overhead, difficult to have critical sections that rely on multiple things in the same HT
    - Whole HT

CONCURRENCY

- **Hash table example**
  - Do we lock the whole HT or do we only lock the individual boxes?
    - Individual boxes:
      - Boxes don’t interfere, we can allow more concurrency
      - We have more boxes, and more overhead, difficult to have critical sections that rely on multiple things in the same HT
    - Whole HT
      - Operations are fast, so we may not need much concurrent access

CONCURRENCY

- **Hash table example**
  - Do we lock the whole HT or do we only lock the individual boxes?
    - Individual boxes:
      - Boxes don’t interfere, we can allow more concurrency
      - We have more boxes, and more overhead, difficult to have critical sections that rely on multiple things in the same HT
    - Whole HT
      - Operations are fast, so we may not need much concurrent access
      - We can allow multiple operations with one lock

CONCURRENCY

- **Hash table example**
  - Do we lock the whole HT or do we only lock the individual boxes?
    - Individual boxes:
      - Boxes don't interfere, we can allow more concurrency
      - We have more boxes, and more overhead, difficult to have critical sections that rely on multiple things in the same HT
    - Whole HT
      - Operations are fast, so we may not need much concurrent access
      - We can allow multiple operations with one lock
      - Resizing!

CONCURRENCY

- **Hash table example**
  - Do we lock the whole HT or do we only lock the individual boxes?
    - Individual boxes:
      - Boxes don't interfere, we can allow more concurrency
      - We have more boxes, and more overhead, difficult to have critical sections that rely on multiple things in the same HT
    - Whole HT
      - Operations are fast, so we may not need much concurrent access
      - We can allow multiple operations with one lock
      - Resizing!
      - Keeping a count of the number of elements

SYNCHRONIZATION

- **Tips**

SYNCHRONIZATION

- **Tips**
  - Keep critical sections as small as possible

SYNCHRONIZATION

- **Tips**
  - Keep critical sections as small as possible
    - Too long and there's a big performance loss

SYNCHRONIZATION

- **Tips**
  - Keep critical sections as small as possible
    - Too long and there's a big performance loss
    - To short, however, and we can introduce race conditions

SYNCHRONIZATION

- **Tips**
  - Keep critical sections as small as possible
    - Too long and there's a big performance loss
    - To short, however, and we can introduce race conditions
  - Atomicity

SYNCHRONIZATION

- **Tips**
  - Keep critical sections as small as possible
    - Too long and there's a big performance loss
    - To short, however, and we can introduce race conditions
  - Atomicity
    - Which parts of code can't be interfered with: Identify them and make sure their resources are properly locked

SYNCHRONIZATION

- **Tips**
  - Keep critical sections as small as possible
    - Too long and there's a big performance loss
    - To short, however, and we can introduce race conditions
  - Atomicity
    - Which parts of code can't be interfered with: Identify them and make sure their resources are properly locked
  - Use libraries

SYNCHRONIZATION

- **Tips**
  - Keep critical sections as small as possible
    - Too long and there's a big performance loss
    - To short, however, and we can introduce race conditions
  - Atomicity
    - Which parts of code can't be interfered with: Identify them and make sure their resources are properly locked
  - Use libraries
    - ConcurrentHashMap solves a lot of these problems in ways that will not make immediate sense, but they do a very good job

WRAP UP

- That ends our discussion on parallelism

WRAP UP

- **That ends our discussion on parallelism**
  - You should be able to parallelize common programs

WRAP UP

- **That ends our discussion on parallelism**
  - You should be able to parallelize common programs
  - Analyze how quickly those programs will run using work, span and speed up

WRAP UP

- **That ends our discussion on parallelism**
  - You should be able to parallelize common programs
  - Analyze how quickly those programs will run using work, span and speed up
  - Know the 4 primitives and how to use them

WRAP UP

- **That ends our discussion on parallelism**
  - You should be able to parallelize common programs
  - Analyze how quickly those programs will run using work, span and speed up
  - Know the 4 primitives and how to use them
  - Understand the constraints and limitations of parallelism and synchronization

WRAP UP

- **That ends our discussion on parallelism**
  - You should be able to parallelize common programs
  - Analyze how quickly those programs will run using work, span and speed up
  - Know the 4 primitives and how to use them
  - Understand the constraints and limitations of parallelism and synchronization
  - Locks and mutexes protect critical sections

WRAP UP

- **That ends our discussion on parallelism**
  - You should be able to parallelize common programs
  - Analyze how quickly those programs will run using work, span and speed up
  - Know the 4 primitives and how to use them
  - Understand the constraints and limitations of parallelism and synchronization
  - Locks and mutexes protect critical sections
  - Mutex design is non-trivial. There are many good reasons to be fine-grained or coarse-grained in your protections. Think through them all

GRAPHS

- **Final big topic**

GRAPHS

- **Final big topic**
  - We've talked a lot about data structures and parallelism, that's the course title, but there is still a lot of introductory algorithm materia

FRIDAY

- Concurrency and locking
- Concurrent design
- Granularity
- P3 checkpoint