Concurrency and Streams

It’s all flowing, all the time

Creating a timetable

As a study technique

You can’t manage time

you can only manage your priorities

- For some of you, you may not be used to studying under current conditions— even if you were homeschooled
- The pressure of several college courses simultaneously, while maintaining a home life, may be a bit much
- I encourage you to *schedule* your time; don’t approach it as one big block, but to parcel out the time you have available for the tasks which need to be done
- I personally use paper for this when scheduling vacations, for example


Concurrency
Time is of the essence

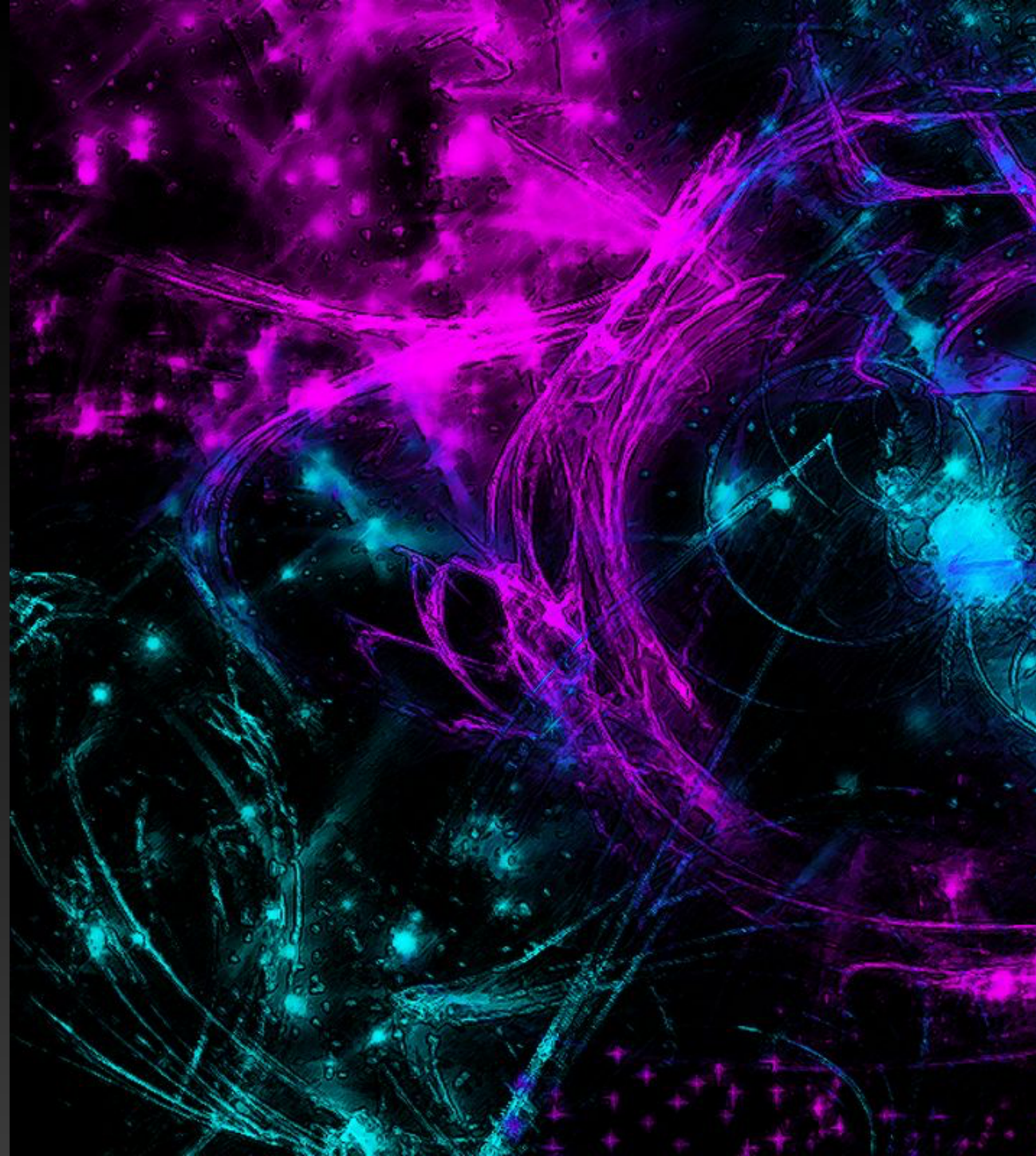
State, Sameness, and Change

- State, sameness and change all introduce a single problem into our programs; the concept of *time*
  - Absent time, there is no possibility of change
  - Absent time, sameness is trivial
  - Absent time, state is meaningless

- Evaluating expressions have different values based upon when the evaluation takes place

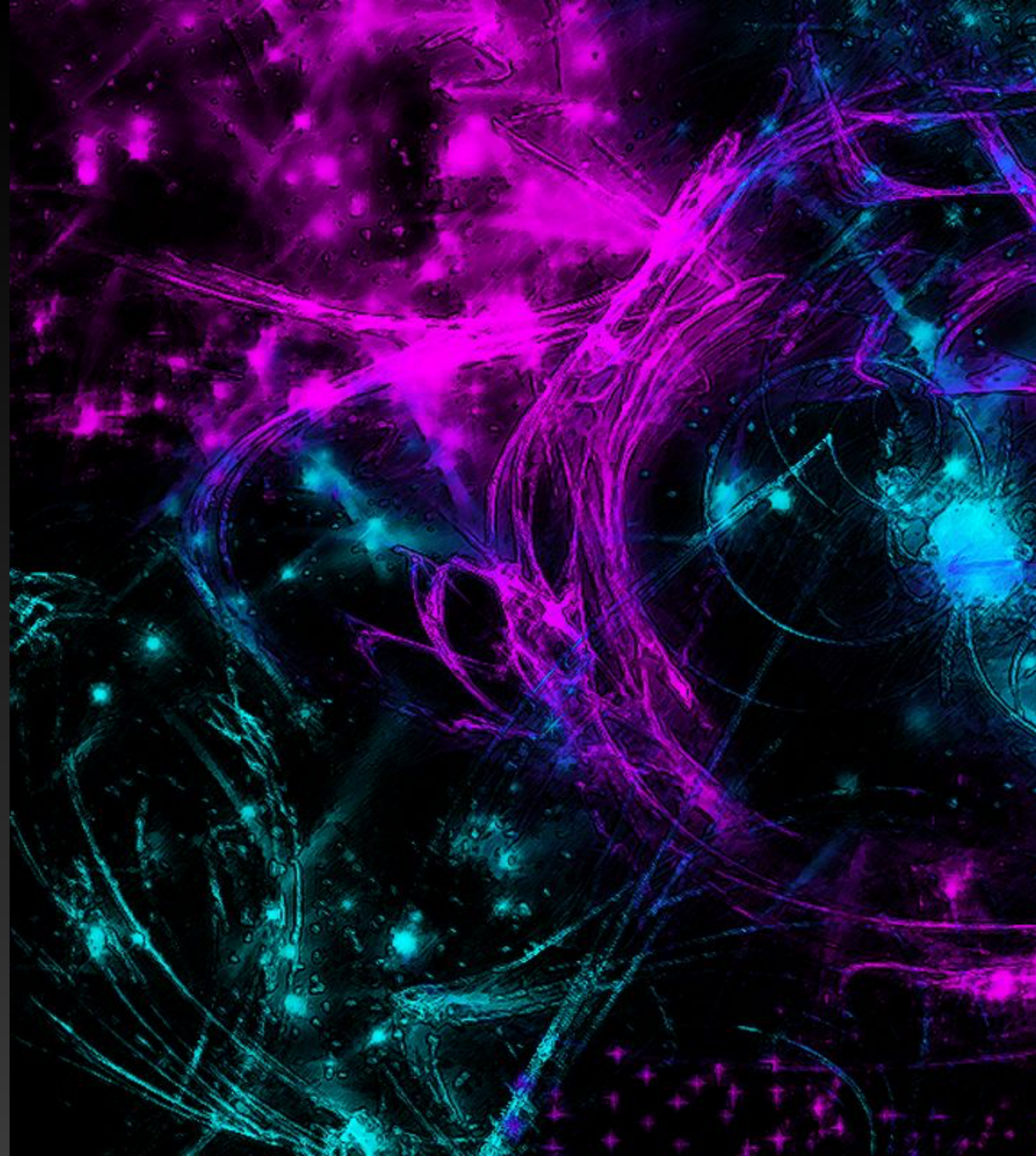
The real world
(whatever that means)

- We generally do not view objects in the real world as changing one at a time in sequence
- We perceive the world as a set of changes all occurring simultaneously
- We should therefore build our models to approximate this concurrency



Concurrency
and modularity

- By building systems which support concurrency, we will find that our systems are in fact more modular
- Even if our programs are executed sequentially, we will benefit
- Decomposition into parallelizable tasks is key



- Modern computers are all highly parallelized, even though we may not think of them as such (other than when viewing technical specifications)
- It’s nearly impossible to buy a computer that does not have multiple cores on a single die
- Even a "lowly" raspberry pi 3b has 4 CPU cores (and don’t get me started on phones)
- To truly get a program we write to "cook" a CPU and get as much from it as possible, we have to design our programs to be multithreaded— another way of saying parallelized

You may have studied this before

So patience, please

As we've done before, we're taking a little look behind the curtain at this kind of processing and programming technique

You may have learned how to use semaphores, mutexes, threads, and so on— that's awesome. I'm not asking you to disregard what you've previously learned

Just be patient, because we're going to cover what these mean in terms of programming languages, so you can really apply what you already know

The Nature of Time in concurrent systems

What is time?
(Check out Forky Asks A Question)

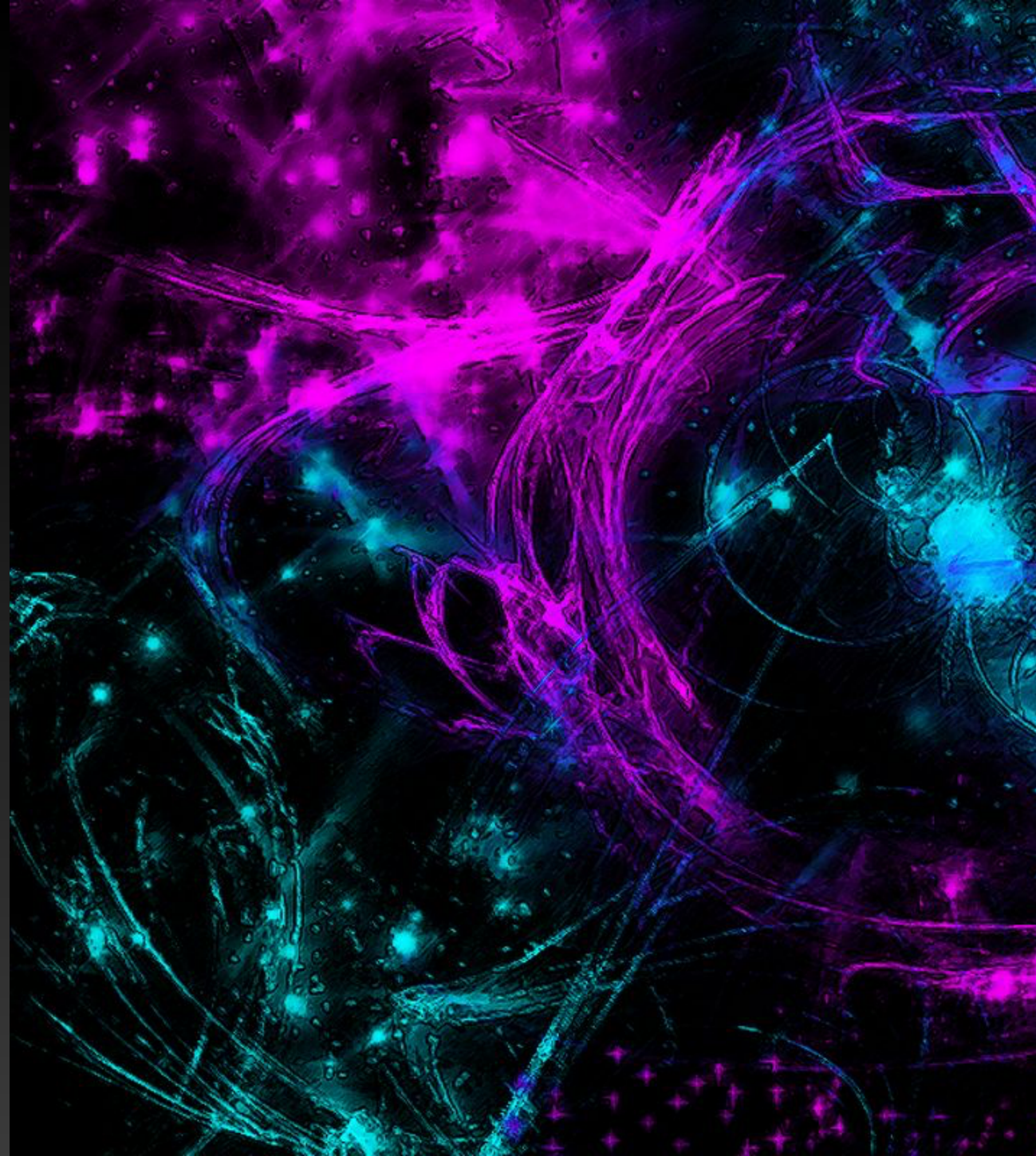
- It’s an ordering imposed on a series of events
  - It keeps everything from happening all at once
- For any two events X and Y, there are three possible timings:
  - X happens before Y
  - Y happens before X
  - X and Y happen at the same time

Withdrawing money

a simple example

- Peter draws $10 from the account
- Paul draws $25 from the account

- $100 - $10 = $90, $90 - $25 = $65
- $100 - $25 = 75, $75 - $10 = $65



- The devil is always in the details
  - What if the initial balance were $30, and not $100?
  - Paul might check the balance, and Peter might check the balance, and both transactions are "approved"
  - Peter withdraws his $10
- When Paul withdraws his $25, suddenly the account is overdrawn by $5
- But all of the rules were followed!

The rules
sample code

```
(define (withdraw amount)
  (if (>= balance amount)
      (begin
        (set! balance
              (- balance amount))
        balance)
      "Insufficient funds"))
```

It gets worse!

Even in one line!

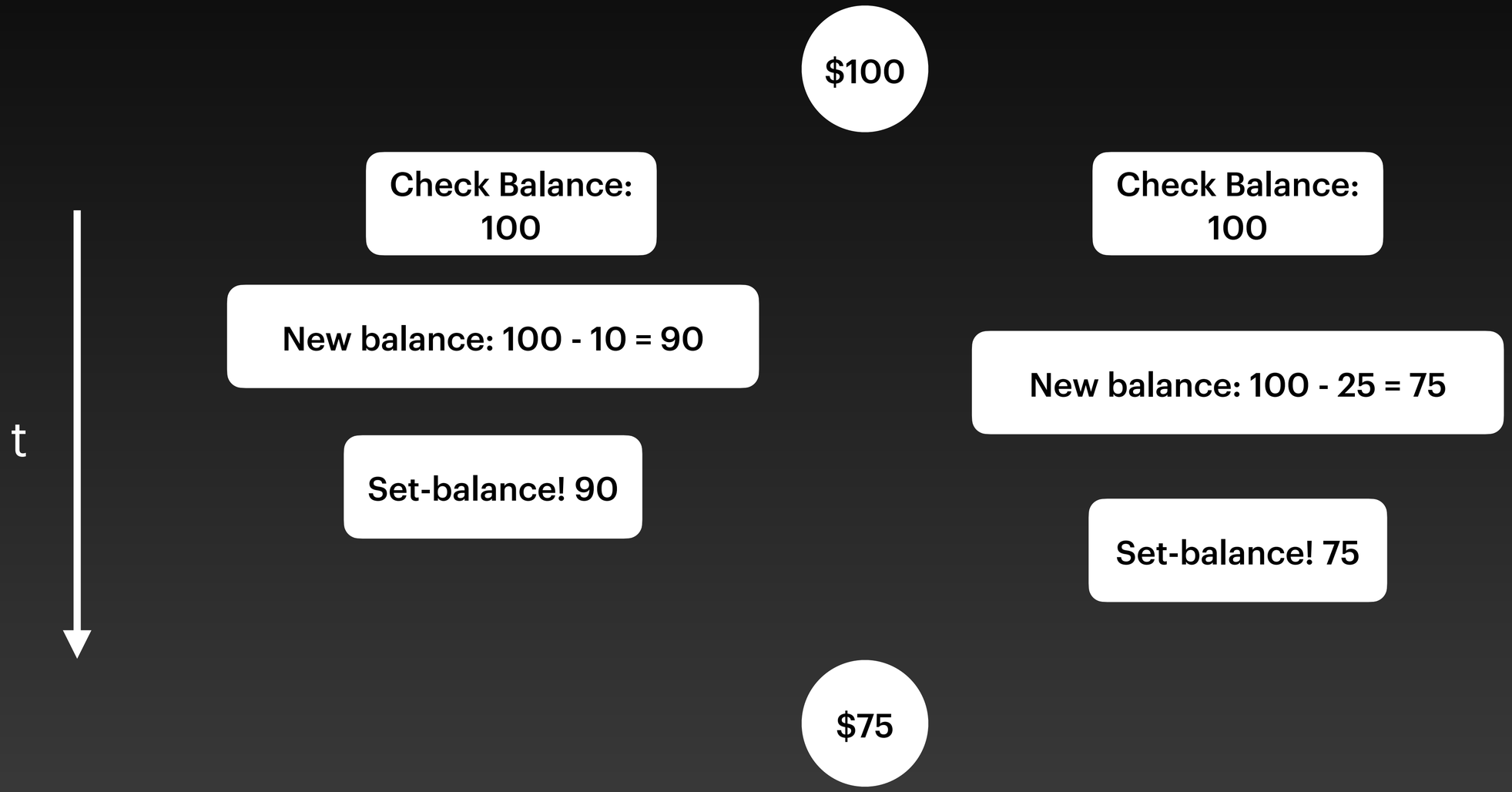
```
(set! balance (- balance amount))
```

This method gets the balance, subtracts amount from it

AT THIS MOMENT BALANCE COULD CHANGE

And then sets the balance to the new value

In diagram form

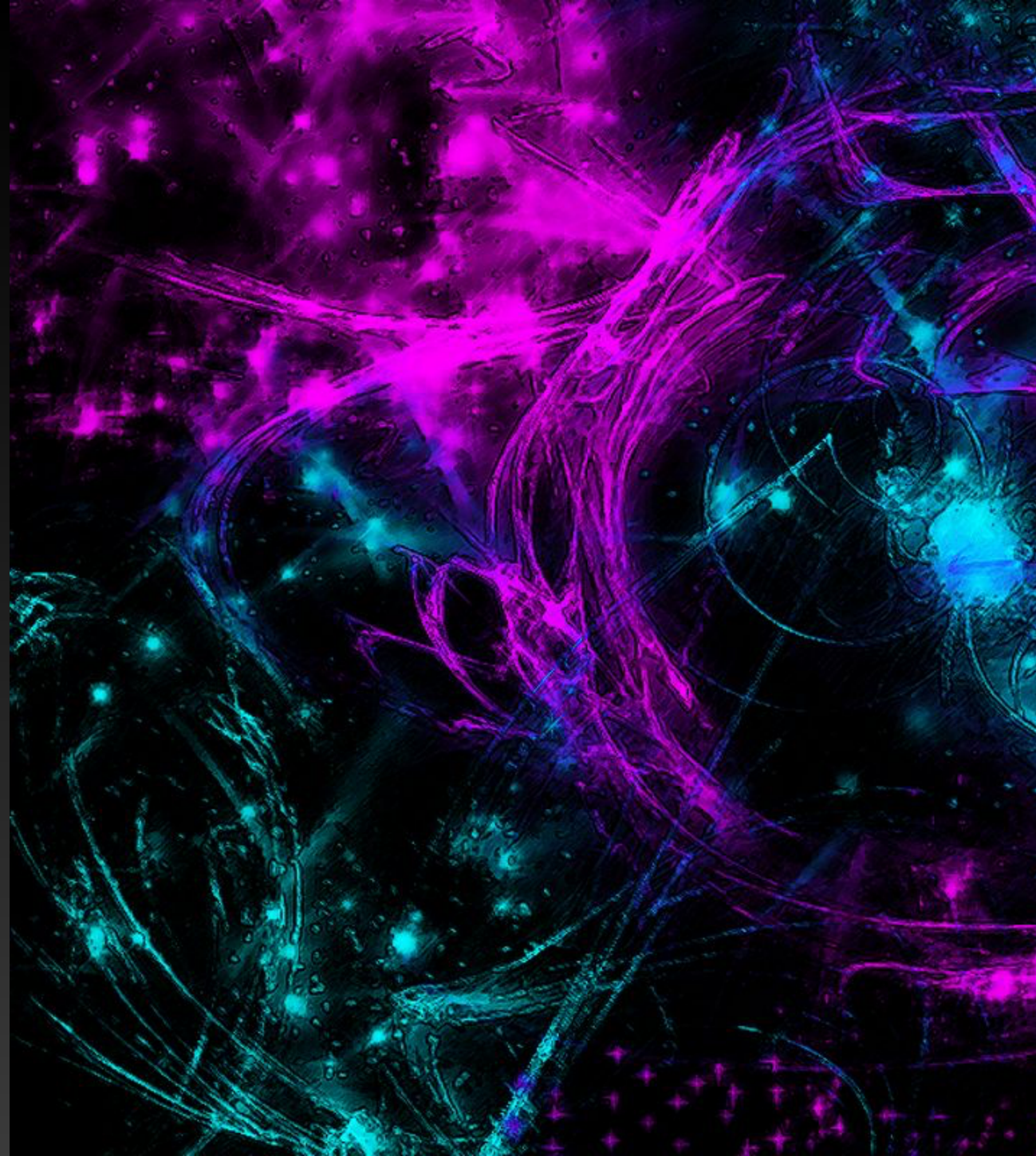


Core issue: State

- Multiple processes are sharing the same state variable
- Each is manipulating that state variable (mutating) correctly, but because state is shared, anomalies occur
- Each of these processes *should* be able to act as though no other process exists
- In the example above, each process MUST be able to assume that the balance is what it previously thought it was

Correct Behavior

- There are no global rules for correct behavior in concurrency, much to our chagrin as programmers
- Rules must be determined on a per-application basis
- We cannot be overly prohibitive, or we will not be able to gain any performance benefits from parallel processing



Prohibitive Example

- No two operations that change any shared state variable can execute at the same time
  - This is incredibly stringent, but may not be stringent enough
  - What about reading?
  - Please remember *ownership* in Rust is largely based around this rule

Less prohibitive rule

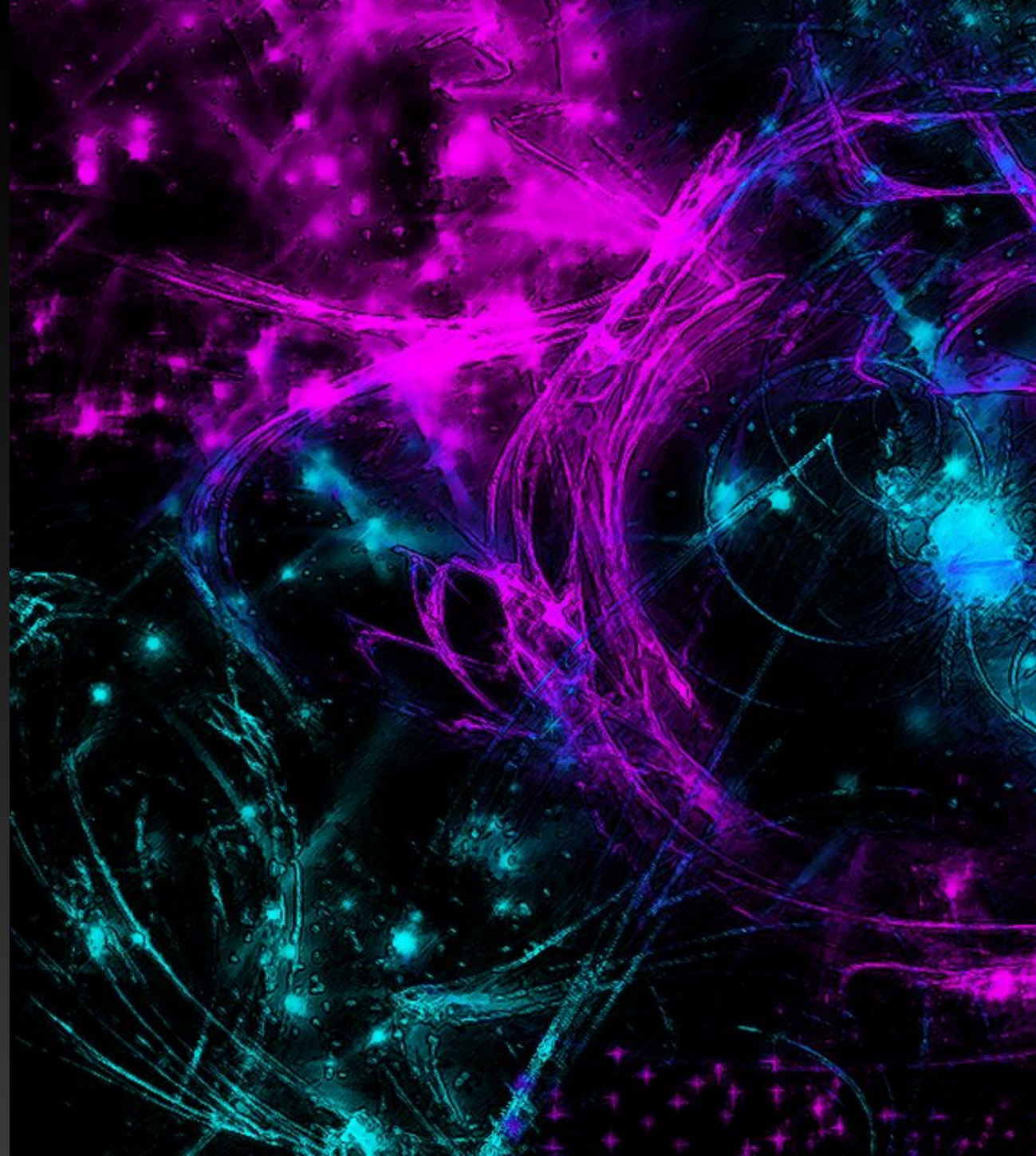
- A concurrent process must provide a result that is the same as *one* of the possible values of sequential processing
  - Example: what if the above withdrawals were "withdraw $10" and "withdraw half" (starting value of $100)?
  - One possible answer is $45, the other is $40.

Still less prohibitive

- Sometimes, there’s no interaction at all!
  - Parallelize image reduction
  - Each 4x4 cell averages down to a 1x1 cell
  - Answers do not impact one another

How to decide?

- Making decisions like this is part of what we do as developers
- No one answer is correct all the time
- Work at *business analysis* to determine the correct answer for a given situation



Mechanisms for controlling concurrency

The interleaving problem

(a b c) (x y z)

(a,b,c,x,y,z) (a,x,b,y,c,z) (x,a,b,c,y,z)

(x,a,y,z,b,c) (a,b,x,c,y,z) (a,x,b,y,z,c)

(x,a,b,y,c,z) (x,y,a,b,c,z) (a,b,x,y,c,z)

(a,x,y,b,c,z) (x,a,b,y,z,c) (x,y,a,b,z,c)

(a,b,x,y,z,c) (a,x,y,b,z,c) (x,a,y,b,c,z)

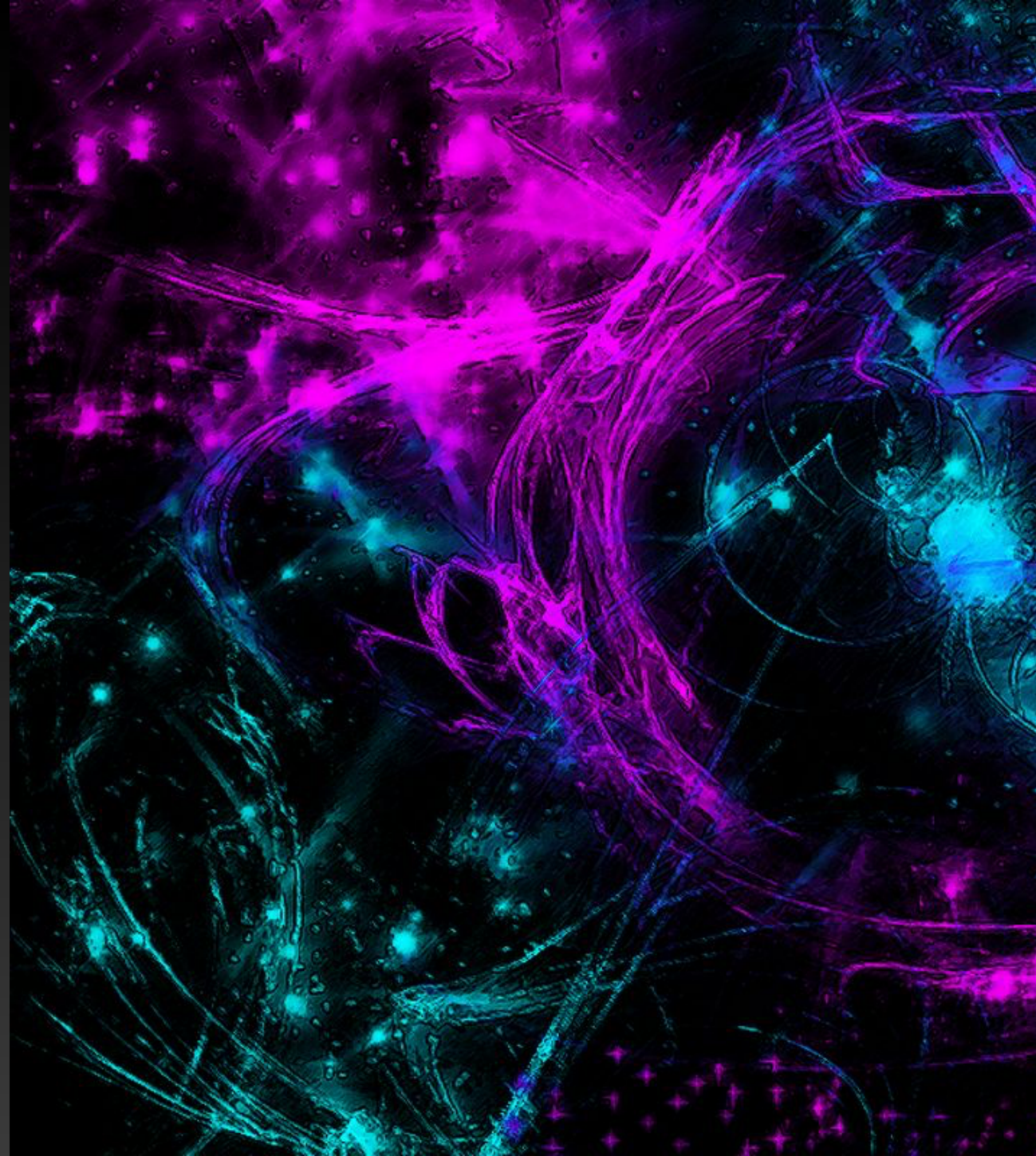
(x,y,a,z,b,c) (a,x,b,c,y,z) (a,x,y,z,b,c)

(x,a,y,b,z,c) (x,y,z,a,b,c)

- With 20 possible combinations, we would have to consider each of them to determine the correct way to design such a system
- This is impractical at best
- We need a more general approach
- Serialization

Serializing access
to shared state

- Processes occur concurrently
- Certain collections of procedures cannot be executed concurrently
- Only one execution in each serialized set is allowed at any given time
- If one process in that set is being executed, all others must wait



Shared access to shared variables

- If we wish to update a shared variable based upon its previous value, we can put access to that shared variable and the assignment in the same procedure
- We can ensure that no other assignment to that shared variable will occur concurrently *if* we ensure that all assignments go through that one procedure
- This is why we focus on *mutators* and procedures that *assign values*, instead of directly modifying a variable— it lends a layer of protection!

An example in scheme

parallel-execute as a serializer

(parallel-execute <p^1> <p^2> ... <p^n>

Let’s presume that this will execute all its arguments in parallel

using parallel-execute

```
(define x 10)
(parallel-execute (lambda () (set! x (* x x)))
                  (lambda () (set! x (+ x 1))))
```

How many different combinations are there?

There are 5 combinations!

- 101: P1 sets X to 100, and then P2 sets X to 101
- 121: P2 sets X to 11, and then P1 sets X to 121 (* x x)
- 110: P1 gets a value for X, P2 increments X, and P1 multiplies 10 by 11
- 11: P2 gets a value for X, P1 sets a value of 100 for X, and then P2 increments X
- 100: P1 gets a value for X, P1 gets a value for X, P2 sets a value for X, then P1 sets X to its values of X*X

- Some of these cases are truly degenerate, but demonstrate the very real problems with concurrency!

Providing a serializer

assume make-serializer creates exclusivity

```
(define x 10)

(define s (make-serializer))

(parallel-execute
 (s (lambda () (set! x (* x x))))
 (s (lambda () (set! x (+ x 1)))))
```

Now there are only two possibilities, 101 and 121!

Serializers

- With such a serializer, our problems with withdrawals from previous examples would not occur
- Each account could have its own serializer, which would ensure that each individual account could be accessed securely while allowing operations on different accounts to be parallelized

"Thank goodness that’s sorted, then"

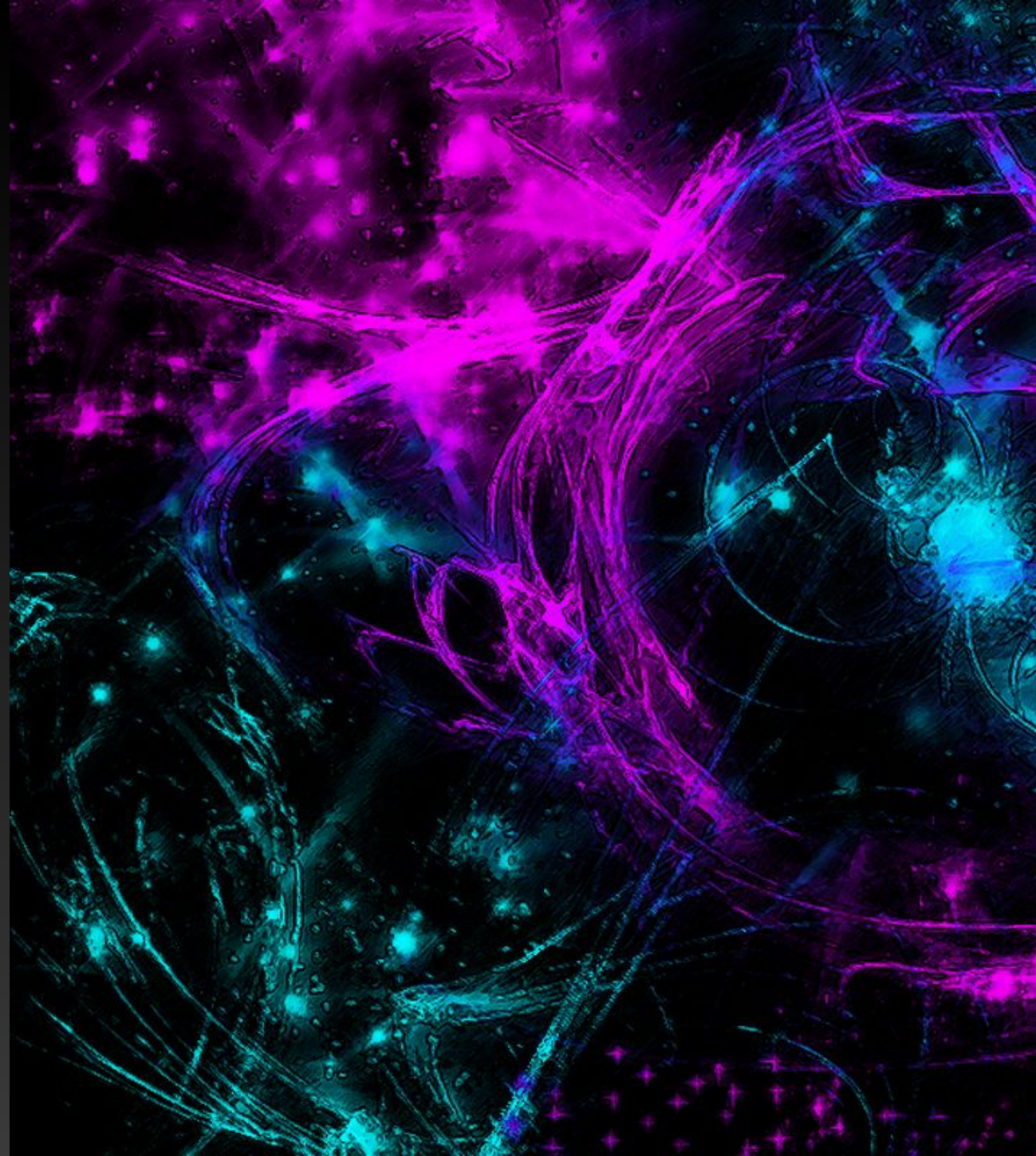
-absolutely nobody, because you know the other shoe will drop...

More complexity
shared resources

- Let’s say we want to swap the balances in two accounts.

- Account 1 is locked while setting Account 2, and Account 2 is locked while setting Account 1

- This almost works!



A, B, C

Now let’s swap between A, B, and C

- You could easily end up with a problem if you try to swap between 3 accounts (Peter has access to A and B, Paul has access to A and C)

- Pause the video to think about how this could be problemematic!

A, B, C

Now let’s swap between A, B, and C

- You could easily end up with a problem if you try to swap between 3 accounts (Peter has access to A and B, Paul has access to A and C)
- Peter could exchange A with B, and then before B has been set to A, Paul could change C to A.

- We need to make this entire exchange process lockable for safety
  - Under serialization, we’d need access to each account’s serializer, which would break our modularity

Synchronization Primitives

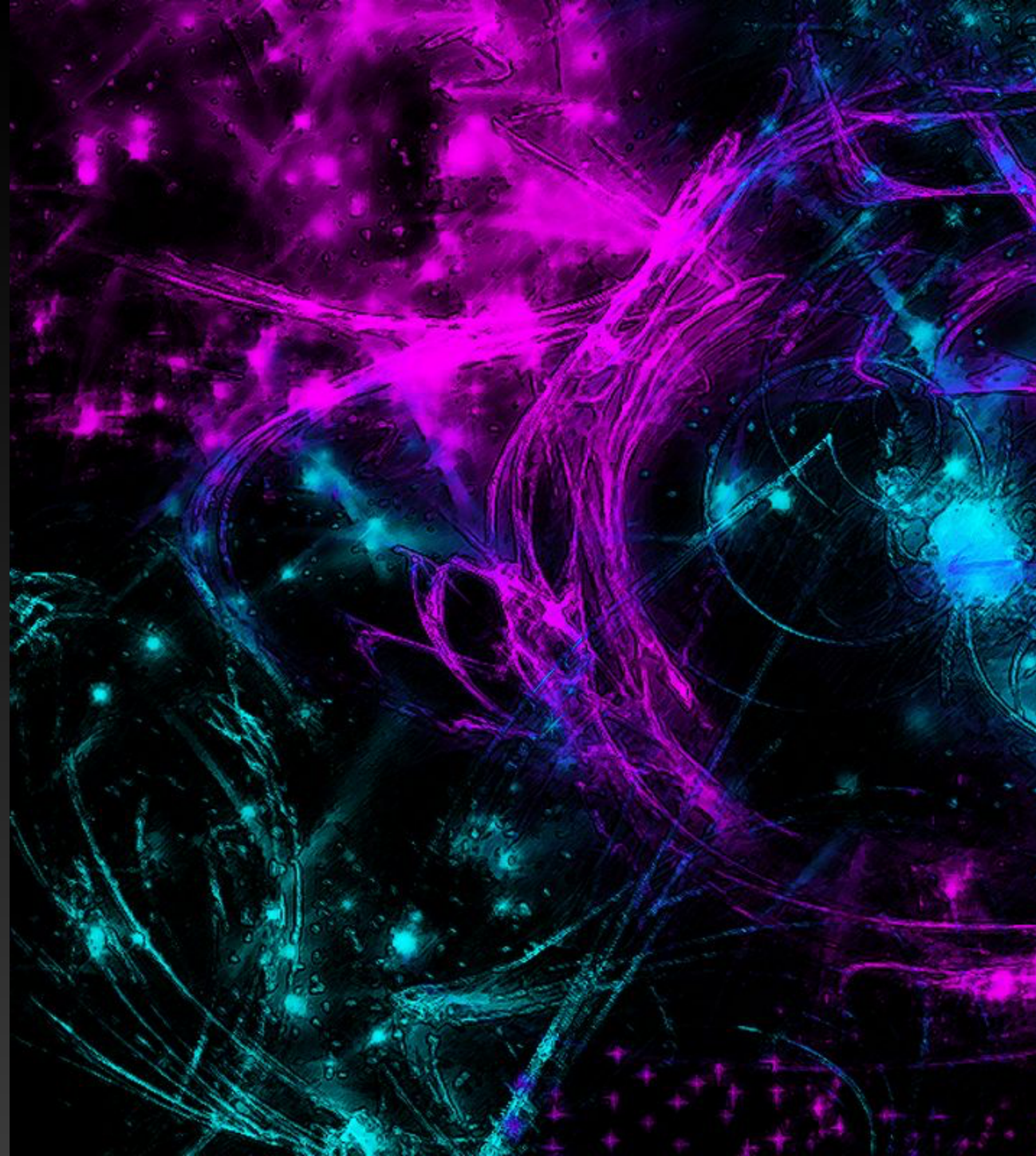
low-level processes for synchronization

- Standing outside the world of Scheme for a moment, one of the key low level mechanisms for synchronization is the Mutex
- You’ve probably heard of these before
- Let’s reconsider what a mutex actually means

Mutexes

- A mutex is an object with two operations
  - A mutex can be *acquired*
  - A mutex can be *released*

- Once a mutex has been *acquired*, no other operation on that mutex can be performed until it has been *released*



Implementing mutexes

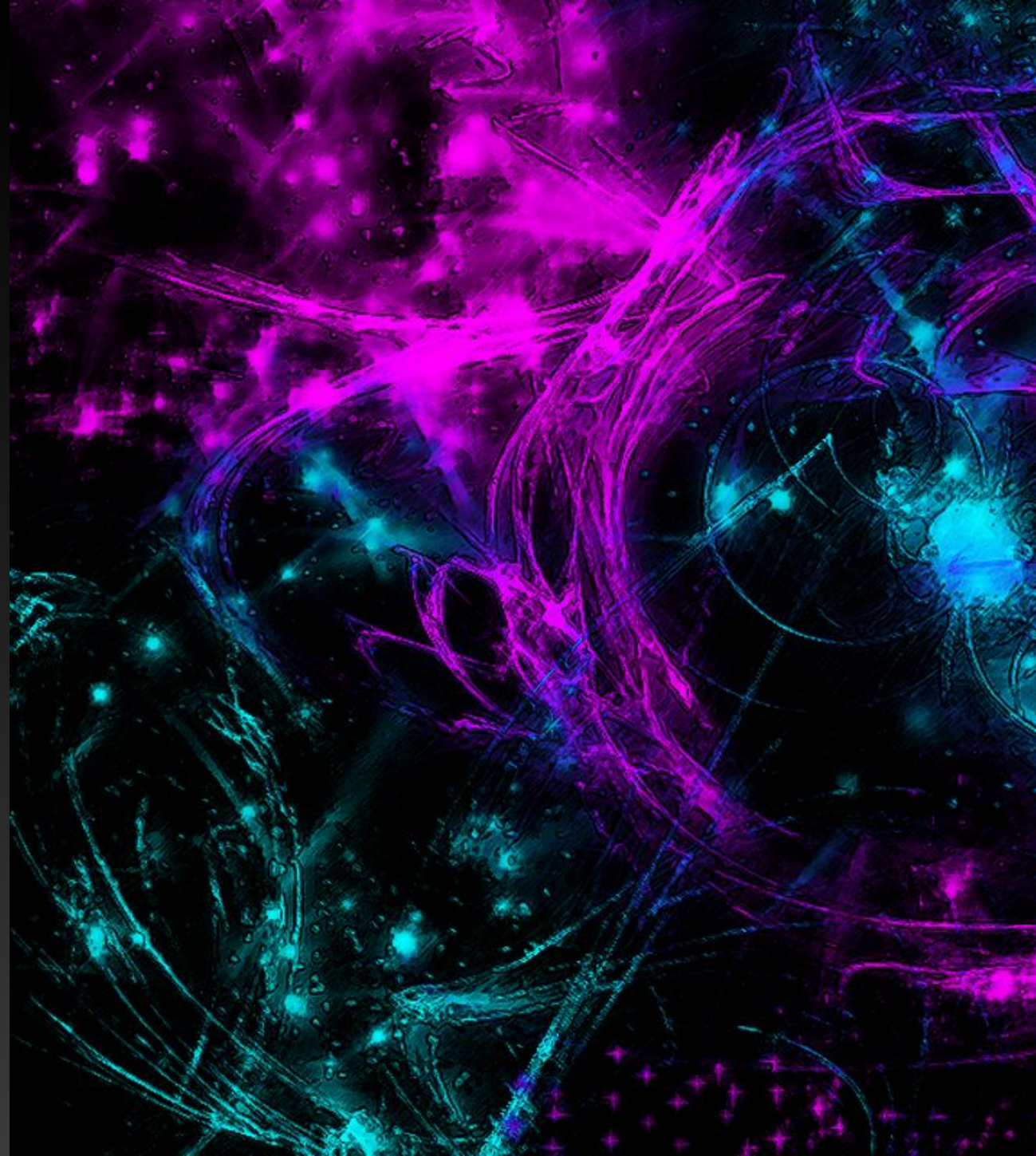
- Mutexes are generally provided by our languages, and implemented at the hardware level
- We usually *check* a mutex and then *lock* it if it's available
- This check and lock must be *atomic*
  - Atomic operations must all be performed at the same time, and fail if they cannot all be performed at once

Deadlock

- Deadlock occurs when there are multiple shared resources, each of which has access restricted
  - Think of resources E and F
  - Process A wants to swap E and F
  - Process B wants to swap F and E
  - Process A locks E
  - Process B locks F
  - Process A tries to lock F, and fails; Process B tries to lock E, and fails

Solving deadlock

- Deadlock is another instance of a problem set where there's no universal solution
  - You could use process IDs to determine who goes first
  - Sometimes it cannot be avoided at all!
  - If a procedure is failable, it could give up and retry after a wait



Shared state and communication

- While not a new problem, concurrency issues are more prevalent today than ever
- We are more often providing services which travel around the world
- Establishing rules around when things occur is vital, and is often attributed directly at the communication layer of our systems

The problems of addressing time in our programs are a reflection of the complexity of our physical world!

Streams

Embracing the unknown

- It’s very easy for me to sit here on the outside and suggest that you should embrace our current unknown status
- On the other hand, do you have choices?

- Identify the worst possible outcomes
- Identify the most likely outcomes
- Embrace the risk
- Find sources of positivity
- Find avenues to focus on the present

Streams

- Things have gotten frighteningly complex lately
- Our substitution model that we used for so long no longer works
  - Assignment
  - State
  - Change
  - Identity
  - Objects
  - Sharing
- All of these things came about as a consequence of allowing *assignment*
- Can we remove assignment but still have a meaningful representation of the world?

What is a stream?

- Streams provide an alternative to modeling state
- What if, rather than maintaining state, we built our system to incorporate time but retained our function-oriented view of programming
- We often like to think of time as a continuous variable; the arc of a thrown baseball, for example
- Rather than think of a system instant-by-instant, if we think of it as continuously changing, we can avoid statefulness
- Streams let us model systems with state without ever introducing mutable data

Streams as delayed lists

Map / Filter / Accumulate

- Map, Filter, and Accumulate are powerful tools, as we've seen
- However, if we represent our data as a list, this power can become expensive
- Our procedures copy and construct lists at every step of the process

Degenerate example

Find the second prime

```
(car (cdr
       (filter
        prime?
        (enumerate-interval 10000 1000000))))
```

In order to find the second prime, we first construct an enormous list of integers, and then check each value's primacy

Enter the stream

- A stream allows us to use sequences without incurring the cost of a list
- We construct a stream partially, and then hand that partial construction to a program that consumes the stream
- If more of the stream is needed, the consumer requests it

We interleave the construction of the stream with its use!

Program changes

- Our programs can continue to be written as though they have a complete set of data
- The program can wait until the next elements are ready to be operated upon

- Ideally, our program that consumes the stream does not know anything about the implementation of the stream, only its use. One could even write it to initially be list-aware, and then add streameyness as needed

- We can think of stream access in terms of *stream-cons* and *stream-cdr*, which behave the same as their otherwise list-like counterparts
- Note there is no *stream-car;* it is assumed that *car* is available instantly.

- *Cdr* is evaluated when its selector is used

- This can all be routed through a *delay* mechanism that makes a promise to fulfill a requirement in the future

Example of cons-stream

```
(cons-stream a b)

is equivalent to

(cons a (delay b))
```

Forcing a stream

- As a companion to *delay*, we also introduce the concept of a *force*
- A *force* would require a stream to produce its next value
  - (Alternatively, the promise must be fulfilled)

Fixing the prime-finder

```
(car (cdr
      (filter
       prime?
       (enumerate-interval 10000 1000000))))
```

Fixing the prime-finder

```
(stream-car
 (stream-cdr
  (stream-filter
   prime? (stream-enumerate-interval
           10000 1000000))))
```

Delayed Streams

- The delayed evaluation of streams can be thought of as demand-driven programming
- We write procedures as if they happen all at once, whereas in reality they happen a little at a time
- This is broadly speaking a difficult programming concept

Infinite streams

Infinity

- If we treat a stream as though it were there in its entirety, but in reality only provide as much as needed at one moment, we have a way to model infinity!

- (note not actual infinity, but rather a never-ending stream of input)

An infinite sequence of integers

```
(define (integers-starting-from n)
  (cons-stream
   n (integers-starting-from (+ n 1))))
(define integers (integers-starting-from 1))
```

An infinite sequence of ones

```
(define ones (cons-stream 1 ones))
```

An alternate stream of integers

```
(define ones (cons-stream 1 ones))


(define integers
  (cons-stream (stream-map + ones integers)))
```

Merging streams

- Taking two sequences and merging them together is often called *zipping* the two sequences
- This is a very common technique for managing sequences
- Especially useful for delayed sequences, where zip will wait to fulfill its promise until both streams' promises are fulfilled

Homework

I have to squeeze 3 more assignments in

Homework 13

Due Saturday May 2

- Read SICP chapter 3.4 & 3.5
- Answer question 3.38, both sections 1 & 2
- If you draw a diagram, use as low-fi a technique as you feel comfortable with. Take a picture of a piece of paper, that’s fine!