6.1800 Spring 2024

Lecture #17: Atomicity via Logging

superior to shadow copies in almost every way

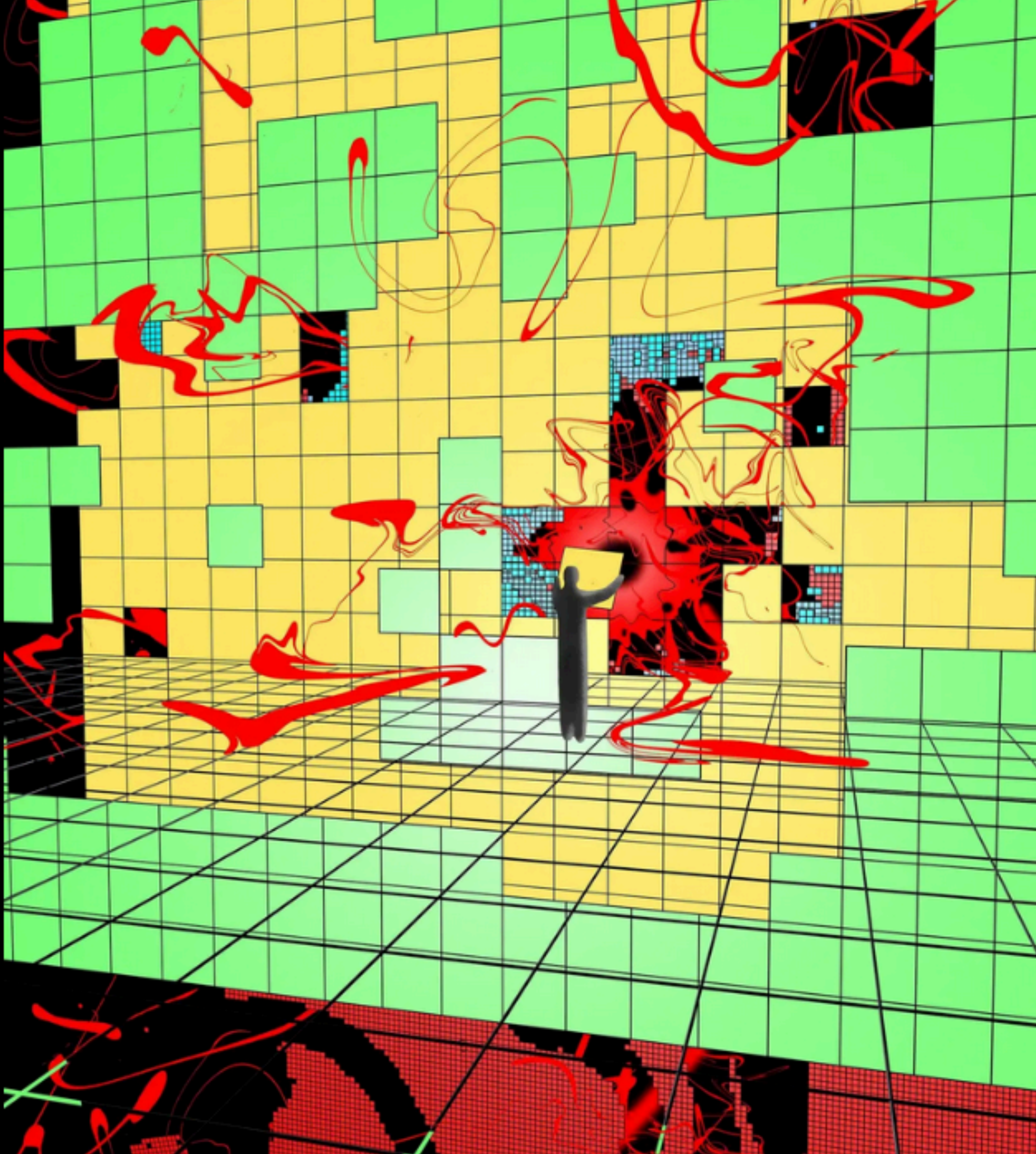
Katrina LaCurts | lacurts@mit.edu | 6.1800 2024

6.1800 in the news

THE SHIFT

Did One Guy Just Stop a Huge Cyberattack?

A Microsoft engineer noticed something was off on a piece of software he worked on. He soon discovered someone was probably trying to gain access to computers all over the world.

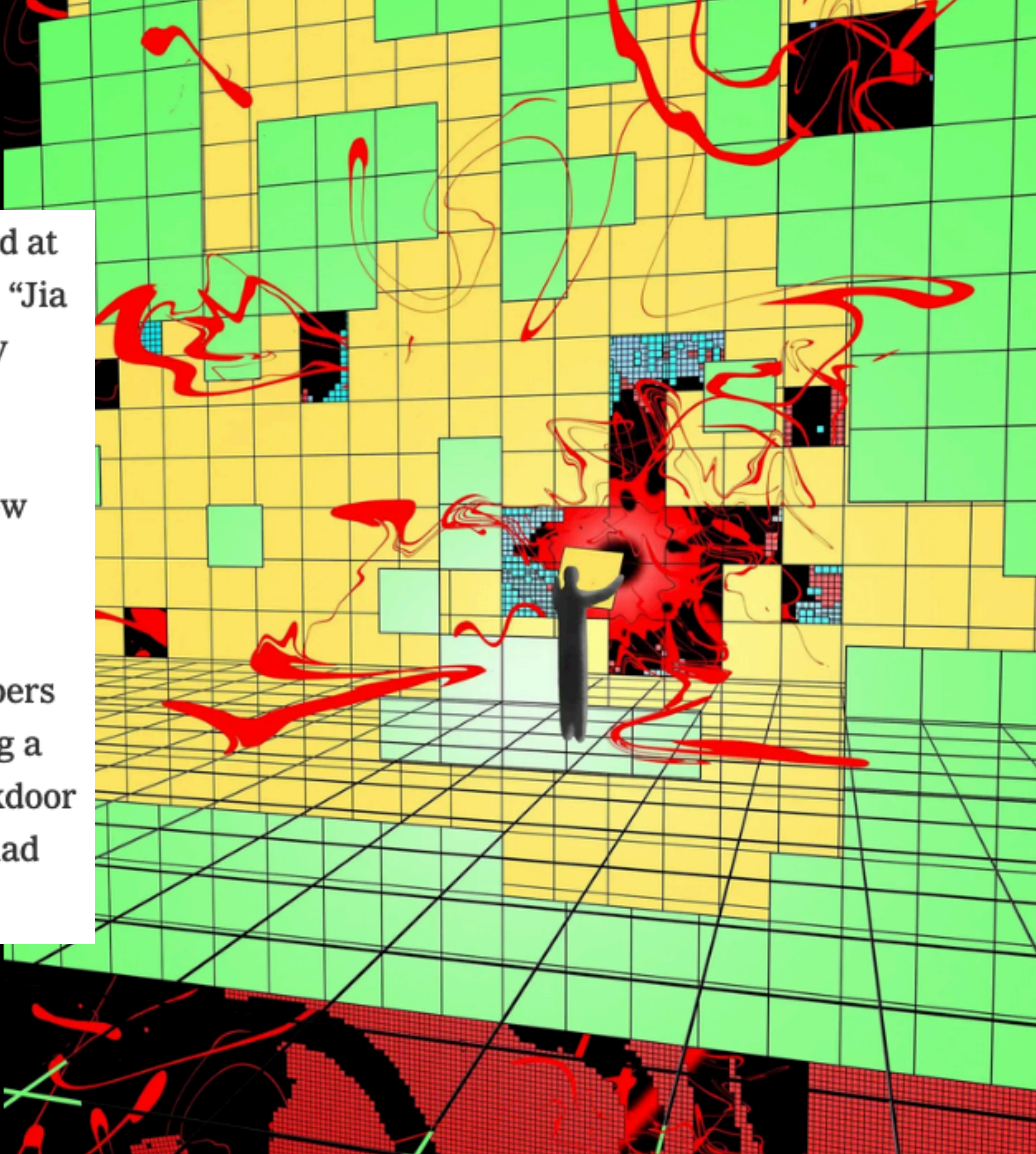


https://www.nytimes.com/2024/04/03/technology/prevent-cyberattack-linux.html

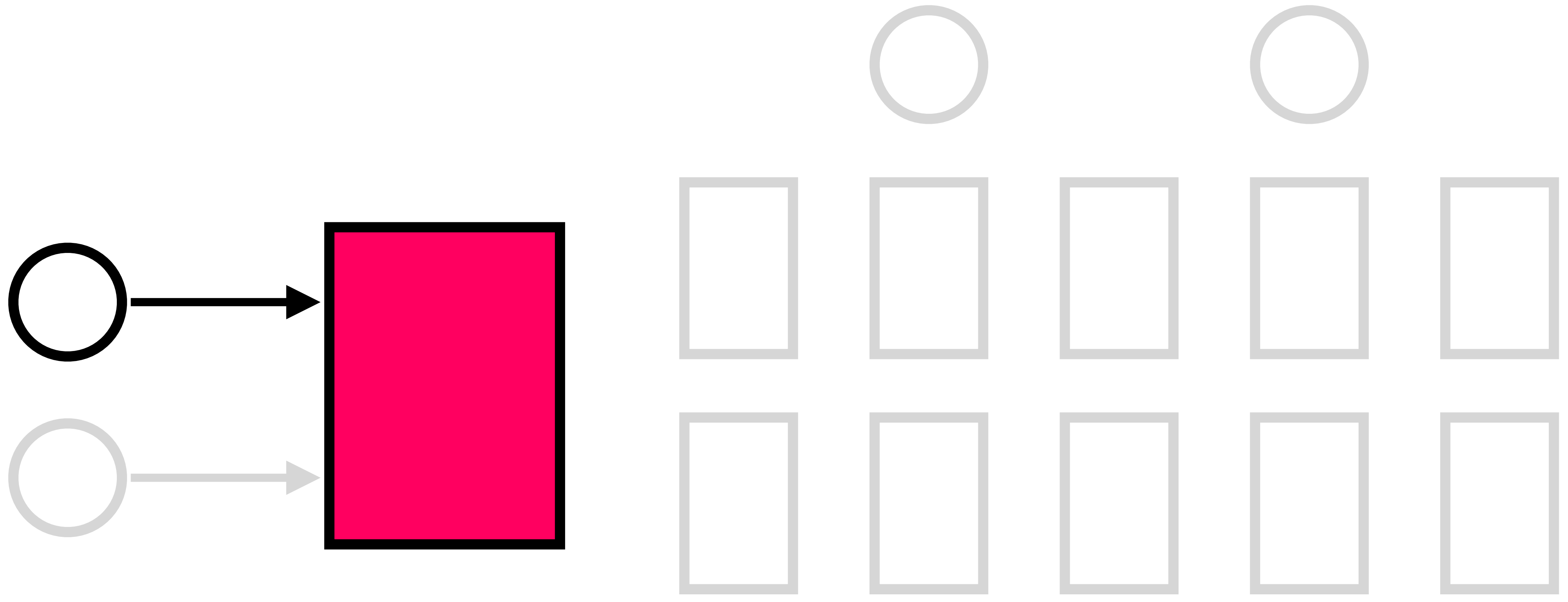
6.1800 in the news

According to some researchers who have gone back and looked at the evidence, the attacker appears to have used a pseudonym, "Jia Tan," to suggest changes to xz Utils as far back as 2022. (Many open-source software projects are governed via hierarchy; developers suggest changes to a program's code, then more experienced developers known as "maintainers" have to review and approve the changes.)

The attacker, using the Jia Tan name, appears to have spent several years slowly gaining the trust of other xz Utils developers and getting more control over the project, eventually becoming a maintainer, and finally inserting the code with the hidden backdoor earlier this year. (The new, compromised version of the code had been released, but was not yet in widespread use.)



https://www.nytimes.com/2024/04/03/technology/prevent-cyberattack-linux.html our goal is to build **reliable systems from unreliable components.** we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high



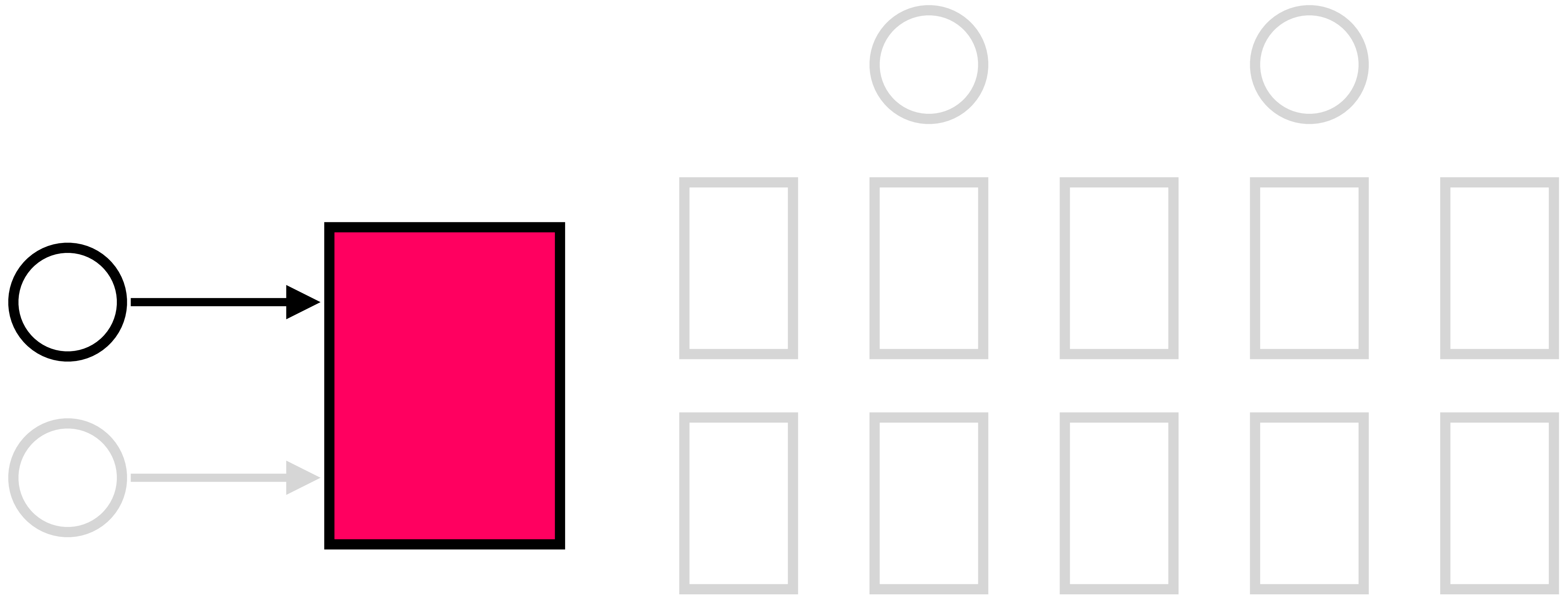
transactions — which provide **atomicity** and **isolation** — make it easier for us to reason about failures

our job in lecture is to understand how a system *implements* these two abstractions. how do our systems guarantee atomicity? how do they guarantee isolation?

atomicity: we have this working for one user and one file via *shadow copies*, but they perform poorly

isolation: we don't really have this yet
(coarse-grained locks perform poorly; fine-grained locks are difficult to reason about)

our goal is to build **reliable systems from unreliable components.** we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high



transactions — which provide **atomicity** and **isolation** — make it easier for us to reason about failures

our job in lecture is to understand how a system *implements* these two abstractions. how do our systems guarantee atomicity? how do they guarantee isolation?

atomicity: logging, which is going to provide us with much better performance at the cost of some added complexity

isolation: we don't really have this yet
(coarse-grained locks perform poorly; fine-grained locks are difficult to reason about)

```
transfer (bank, account_a, account_b, amount):
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
```

this was our starting bank transfer code from last week. let's put it into **transaction** syntax.

```
begin
A = read(A)
B = read(B)
write(A, A-amount)
write(B, B+amount)
commit
```

```
begin
write(A, read(A)-amount)
write(B, read(B)+amount)
commit
```

we can even be more succinct and get rid of the local variables

to broaden our horizons, we're going to move away from bank transfers and think about generic reads/writes

our goal today is to make sure that each transaction in a series (such as the one below) is atomic *and* that our system has good performance

remember that one problem with shadow copies is that they rewrite an entire file even for small changes

```
begin     // T1
write(A, 100)
write(B, 50)
commit    // A=100; B=50

begin     // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit    // A=80; B=70

begin     // T3
write(A, read(A)+30)
```

crash! 💥

our goal today is to make sure that each transaction in a series (such as the one below) is atomic *and* that our system has good performance

remember that one problem with shadow copies is that they rewrite an entire file even for small changes

```
begin     // T1
write(A, 100)
write(B, 50)
commit    // A=100; B=50

begin     // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit    // A=80; B=70

begin     // T3
write(A, read(A)+30)
```

crash! 💥

problem: after crash, `A=110`, but `T3` never committed

we need a way to revert to A's previous committed value

```
    +--------+--------+--------+--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

```
begin     // T1
write(A, 100)
write(B, 50)
commit    // A=100; B=50

begin     // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit    // A=80; B=70

begin     // T3
write(A, read(A)+30)
```

```
    +--------+--------+--------+
TID |   T1   |   T1   |   T1   |
    | UPDATE | UPDATE | COMMIT |
OLD | A=0    | B=0    |        |
NEW | A=100  | B=50   |        |
    +--------+--------+--------+
```

```
begin     // T1
write(A, 100)
write(B, 50)
commit    // A=100; B=50

begin     // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit    // A=80; B=70

begin     // T3
write(A, read(A)+30)
```

let's try to read the value of A from this log

```
read(log, var):
  commits = []
  // scan backwards
  for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == COMMIT:
      commits.add(r.tid)
    // find var's last committed value
    elif r.type == UPDATE and
      r.tid in commits and r.var == var:
        return r.new_value
```

```
commits = []
```

```
     +--------+--------+--------+
TID  |   T1   |   T1   |   T1   |
     | UPDATE | UPDATE | COMMIT |
OLD  | A=0    | B=0    |        |
NEW  | A=100  | B=50   |        |
     +--------+--------+--------+
```

```
begin      // T1
write(A, 100)
write(B, 50)
commit     // A=100; B=50

begin      // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit     // A=80; B=70

begin      // T3
write(A, read(A)+30)
```

let's try to read the value of A from this log

```
read(log, var):
  commits = []
  // scan backwards
  for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == COMMIT:
      commits.add(r.tid)
    // find var's last committed value
    elif r.type == UPDATE and
      r.tid in commits and r.var == var:
        return r.new_value
```

```
commits = [T1]
```

```
      +--------+--------+--------+--------+
TID   |   T1   |   T1   |   T1   |   T2   |
      | UPDATE | UPDATE | COMMIT | UPDATE |
OLD   | A=0    | B=0    |        | A=100  |
NEW   | A=100  | B=50   |        | A=80   |
      +--------+--------+--------+--------+
```

```
begin      // T1
write(A, 100)
write(B, 50)
commit     // A=100; B=50

begin      // T2
write(A, read(A)-20)
write(A, read(A)-30)
commit     // A=50; B=50

begin      // T3
write(A, read(A)+30)
```

brief interlude: we're going to change this example slightly, to illustrate one additional point

let's try to read the value of A from this log

```
read(log, var):
  commits = []
  // scan backwards
  for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == COMMIT:
      commits.add(r.tid)
    // find var's last committed value
    elif r.type == UPDATE and
      (r.tid in commits or r.tid == current_tid)
      and r.var == var:
        return r.new_value
```

```
    +--------+--------+--------+--------+--------+--------+--------+
TID | T1     | T1     | T1     | T2     | T2     | T2     | T3     |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

let's try to read the value of A from this log

```
begin      // T1
write(A, 100)
write(B, 50)
commit     // A=100; B=50

begin      // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit     // A=80; B=70

begin      // T3
write(A, read(A)+30)
crash! 💥
```

now back to our original example

```
read(log, var):
  commits = []
  // scan backwards
  for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == COMMIT:
      commits.add(r.tid)
    // find var's last committed value
    elif r.type == UPDATE and
      (r.tid in commits or r.tid == current_tid)
      and r.var == var:
        return r.new_value
```

after a crash, the log is still correct; uncommitted updates will not be read

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

```
begin       // T1
write(A, 100)
write(B, 50)
commit      // A=100; B=50

begin       // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit      // A=80; B=70

begin       // T3
write(A, read(A)+30)
crash! 💥
```

writes contain the old and new value of a variable. each write is a small append to the end of the log

to **read** a variable *x*, the system scans backwards through the log to find x's last committed value

the **commit point** for a transaction is writing the **`COMMIT`** record

problem: reads can be very slow

```
    +--------+--------+--------+--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

```
begin       // T1
write(A, 100)
write(B, 50)
commit      // A=100; B=50

begin       // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit      // A=80; B=70

begin       // T3
write(A, read(A)+30)
```

crash! 💥

```
read(log, var):
  commits = []
  // scan backwards
  for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == COMMIT:
      commits.add(r.tid)
    // find var's last committed value
    elif r.type == UPDATE and
      (r.tid in commits or r.tid == current_tid)
      and r.var == var:
        return r.new_value
```

problem: reads can be very slow

```
    +--------+
TID |   T1   |
    | UPDATE |
OLD | A=0    |
NEW | A=100  |
    +--------+
```

cell storage (on disk)

A 100

```
begin     // T1
write(A, 100)
write(B, 50)
commit    // A=100; B=50

begin     // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit    // A=80; B=70

begin     // T3
write(A, read(A)+30)
```

```
read(var):
  return cell_read(var)


write(var, value):
  log.append(current_tid, "UPDATE", var,
             read(var), value)
  cell_write(var, value)
```

```
    +--------+--------+
TID |   T1   |   T1   |
    | UPDATE | UPDATE |
OLD | A=0    | B=0    |
NEW | A=100  | B=50   |
    +--------+--------+
```

cell storage (on disk)

A 100	B 50

```
begin      // T1
write(A, 100)
write(B, 50)
commit     // A=100; B=50

begin      // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit     // A=80; B=70

begin      // T3
write(A, read(A)+30)
```

```
read(var):
  return cell_read(var)


write(var, value):
  log.append(current_tid, "UPDATE", var,
             read(var), value)
  cell_write(var, value)
```

```
    +--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |
    | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  |
NEW | A=100  | B=50   |        | A=80   |
    +--------+--------+--------+--------+
```

cell storage (on disk)

A	80

B	50

```
begin        // T1
write(A, 100)
write(B, 50)
commit       // A=100; B=50

begin        // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit       // A=80; B=70

begin        // T3
write(A, read(A)+30)
```

```
read(var):
  return cell_read(var)


write(var, value):
  log.append(current_tid, "UPDATE", var,
             read(var), value)
  cell_write(var, value)
```

```
     +--------+--------+--------+--------+--------+
TID  |   T1   |   T1   |   T1   |   T2   |   T2   |
     | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE |
OLD  | A=0    | B=0    |        | A=100  | B=50   |
NEW  | A=100  | B=50   |        | A=80   | B=70   |
     +--------+--------+--------+--------+--------+
```

cell storage (on disk)

A 80	B 70

```
begin      // T1
write(A, 100)
write(B, 50)
commit     // A=100; B=50

begin      // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit     // A=80; B=70

begin      // T3
write(A, read(A)+30)
```

```
read(var):
  return cell_read(var)


write(var, value):
  log.append(current_tid, "UPDATE", var,
             read(var), value)
  cell_write(var, value)
```

```
    +--------+--------+--------+--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk)

A 110	B 70

```
begin      // T1
write(A, 100)
write(B, 50)
commit     // A=100; B=50

begin      // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit     // A=80; B=70

begin      // T3
write(A, read(A)+30)
```

crash! 💥

problem: the value of A in cell storage never committed (and so should not be read after recovery); we need to repair cell storage

```
read(var):
  return cell_read(var)


write(var, value):
  log.append(current_tid, "UPDATE", var,
             read(var), value)
  cell_write(var, value)
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk)

A	80

B	70

```
begin     // T1
write(A, 100)
write(B, 50)
commit    // A=100; B=50

begin     // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit    // A=80; B=70

begin     // T3
write(A, read(A)+30)
```

crash! 💥

```
read(var):
  return cell_read(var)

write(var, value):
  log.append(current_tid, "UPDATE", var,
             read(var), value)
  cell_write(var, value)

recover(log):
  commits = []
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo

commits = []
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk)

A 80	B 70

```
begin      // T1
write(A, 100)
write(B, 50)
commit     // A=100; B=50

begin      // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit     // A=80; B=70

begin      // T3
write(A, read(A)+30)
crash! 💥
```

```
read(var):
  return cell_read(var)

write(var, value):
  log.append(current_tid, "UPDATE", var,
             read(var), value)
  cell_write(var, value)

recover(log):
  commits = []
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

commits = [T2, T1]

```
    +--------+--------+--------+--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
    +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk): A 80 | B 70

```
begin      // T1
write(A, 100)
write(B, 50)
commit     // A=100; B=50

begin      // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit     // A=80; B=70

begin      // T3
write(A, read(A)+30)
crash! 💥
```

writes go to the log first and then cell storage.

to **read** a variable *x*, the system reads *x*'s value from cell storage

on **recovery**, the system must repair cell storage by undo-ing any uncommitted transactions

problem: read performance is now great, but writes got slower

(recovery also got slower; we'll come to that)

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk)

A	B
80	70

```
begin     // T1
write(A, 100)
write(B, 50)
commit    // A=100; B=50

begin     // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit    // A=80; B=70

begin     // T3
write(A, read(A)+30)
```

crash! 💥

```
read(var):
  return cell_read(var)


write(var, value):
  log.append(current_tid, "UPDATE", var,
             read(var), value)
  cell_write(var, value)


recover(log):
  commits = []
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

problem: read performance is now great, but writes got slower

(recovery also got slower; we'll come to that)

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk)	A 110	B 70	cache (memory)	A 110	B 70

```
begin      // T1
write(A, 100)
write(B, 50)
commit     // A=100; B=50

begin      // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit     // A=80; B=70

begin      // T3
write(A, read(A)+30)
```

```
     +--------+--------+--------+--------+--------+--------+--------+
TID  |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
     | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD  | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW  | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
     +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk) | A 110 | B 70 | cache (memory) | A 110 | B 70

```
begin       // T1
write(A, 100)
write(B, 50)
commit      // A=100; B=50

begin       // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit      // A=80; B=70

begin       // T3
write(A, read(A)+30)
```

```
read(var):
  if var in cache:
    return cache[var]
  else:
    // may evict others from cache to cell storage
    cache[var] = cell_read(var)
    return cache[var]

write(var, value):
  log.append(current_tid, update, var,
             read(var), value)
  cache[var] = value

flush(): // called "occasionally"
  cell_write(var, cache[var]) for each var
```

question: on a crash, could we have updates that should be in cell storage, but aren't? what about changes that shouldn't be in cell storage, but are?

```
   +--------+--------+--------+--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
   +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk): A 100, B 50

cache (memory): A 110, B 70

```
begin       // T1
write(A, 100)
write(B, 50)
commit      // A=100; B=50

begin       // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit      // A=80; B=70

begin       // T3
write(A, read(A)+30)
crash! 💥
```

suppose we flushed the cache after T1 committed, but have not flushed it since then

```
read(var):
  if var in cache:
    return cache[var]
  else:
    // may evict others from cache to cell storage
    cache[var] = cell_read(var)
    return cache[var]

write(var, value):
  log.append(current_tid, update, var,
             read(var), value)
  cache[var] = value

flush(): // called "occasionally"
  cell_write(var, cache[var]) for each var

recover(log):
  commits = []
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk): A 80 | B 50

cache (memory): (empty) | (empty)

```
begin        // T1
write(A, 100)
write(B, 50)
commit       // A=100; B=50

begin        // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit       // A=80; B=70

begin        // T3
write(A, read(A)+30)
crash! 💥
```

all other updates were committed; **B**'s value won't ever be changed

```
read(var):
  if var in cache:
    return cache[var]
  else:
    // may evict others from cache to cell storage
    cache[var] = cell_read(var)
    return cache[var]

write(var, value):
  log.append(current_tid, update, var,
             read(var), value)
  cache[var] = value

flush(): // called "occasionally"
  cell_write(var, cache[var]) for each var

recover(log):
  commits = []
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
```

commits = []

```
   +--------+--------+--------+--------+--------+--------+--------+
TID |   T1   |   T1   |   T1   |   T2   |   T2   |   T2   |   T3   |
    | UPDATE | UPDATE | COMMIT | UPDATE | UPDATE | COMMIT | UPDATE |
OLD | A=0    | B=0    |        | A=100  | B=50   |        | A=80   |
NEW | A=100  | B=50   |        | A=80   | B=70   |        | A=110  |
   +--------+--------+--------+--------+--------+--------+--------+
```

cell storage (on disk): A 80 | B 70

cache (memory): (empty) | (empty)

```
begin      // T1
write(A, 100)
write(B, 50)
commit     // A=100; B=50

begin      // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit     // A=80; B=70

begin      // T3
write(A, read(A)+30)
crash! 💥
```

problem: recovery is still slow

```
read(var):
  if var in cache:
    return cache[var]
  else:
    // may evict others from cache to cell storage
    cache[var] = cell_read(var)
    return cache[var]

write(var, value):
  log.append(current_tid, update, var,
             read(var), value)
  cache[var] = value

flush(): // called "occasionally"
  cell_write(var, cache[var]) for each var

recover(log):
  commits = []
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
  for record r in log[0] .. log[len(log)-1]:
    if r.type == UPDATE and r.tid in commits:
      cell_write(r.var, r.new_value) // redo
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk): A 80 | B 70

cache (memory): (empty) | (empty)

```
begin      // T1
write(A, 100)
write(B, 50)
commit     // A=100; B=50

begin      // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit     // A=80; B=70

begin      // T3
write(A, read(A)+30)
```

crash! 💥

solution: write checkpoints and truncate the log

```
read(var):
  if var in cache:
    return cache[var]
  else:
    // may evict others from cache to cell storage
    cache[var] = cell_read(var)
    return cache[var]

write(var, value):
  log.append(current_tid, update, var,
             read(var), value)
  cache[var] = value

flush(): // called "occasionally"
  cell_write(var, cache[var]) for each var

recover(log):
  commits = []
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
  for record r in log[0] .. log[len(log)-1]:
    if r.type == UPDATE and r.tid in commits:
      cell_write(r.var, r.new_value) // redo
```

TID	T1	T1	T1	T2	T2	T2	T3
	UPDATE	UPDATE	COMMIT	UPDATE	UPDATE	COMMIT	UPDATE
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50		A=80	B=70		A=110

cell storage (on disk)	A 80	B 70	cache (memory)		

cell storage and the cache make reads and writes faster, but make our recovery process more complex. in particular, because cell storage is permanent, recovery must make sure it is correct — **undo**-ing any un-committed updates and **redo**-ing any updates that didn't get flushed from the cache

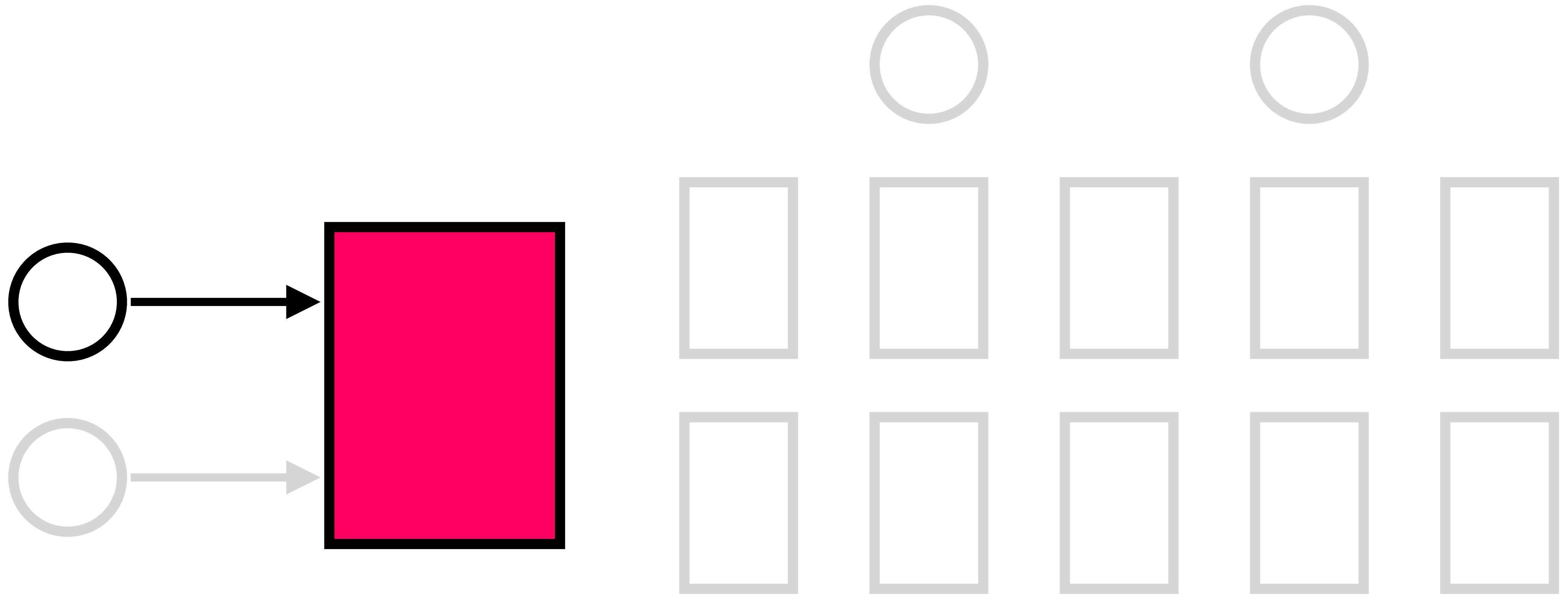
```
read(var):
  if var in cache:
    return cache[var]
  else:
    // may evict others from cache to cell storage
    cache[var] = cell_read(var)
    return cache[var]

write(var, value):
  log.append(current_tid, update, var,
             read(var), value)
  cache[var] = value

flush(): // called "occasionally"
  cell_write(var, cache[var]) for each var

recover(log):
  commits = []
  for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
      commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
      cell_write(r.var, r.old_val) // undo
  for record r in log[0] .. log[len(log)-1]:
    if r.type == UPDATE and r.tid in commits:
      cell_write(r.var, r.new_value) // redo
```

our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high



transactions — which provide **atomicity** and **isolation** — make it easier for us to reason about failures

our job in lecture is to understand how a system *implements* these two abstractions. how do our systems guarantee atomicity? how do they guarantee isolation?

atomicity: provided by **logging**, which gives better performance than shadow copies* at the cost of some added complexity

* shadow copies *are* used in some systems

isolation: we don't really have this yet

(coarse-grained locks perform poorly; fine-grained locks are difficult to reason about)

(write-ahead) logs provide **atomicity** with better performance than shadow copies. the primary benefit is making small appends for each update, rather than copy an entire file over for every change.

cell storage is used with the log to improve read performance, and **caches** and **truncation** can be used to improve write and recovery performance

the addition of these performance-improving techniques makes the system's **recovery** process more complex