Lecture 10

Transactional Memory

Shared Memory Concurrency

- Lock-based programming is difficult
- There are many potential problems:
  - Deadlock
  - Starvation
  - Priority inversion
  - Convoying
  - Non-compositionality
- Is there some way to eliminate at least some of these problems?

Convoying

- Convoying occurs when a process has taken a mutex and is then preemted by the scheduler
- It has the effect that other processes may not be allowed to enter the mutex
- This inhibits concurrency

Non-compositionality

- Lock-based programming doesn't compose
- Example:
  - Suppose you have two thread safe buffers and you want to atomically take an element from one of them and put it in the other

```
class Buffer<A> {
    A get();
    void put(A);
}
```

Non-compositionality

- A not so nice solution:
  - Expose the the locks of the buffers
  - Lock both buffers before moving the element
  - This brakes the abstraction!

```
class Buffer<A> {
    void aquireLock();
    void releaseLock();
    A get();
   void put(A);
}
```

Non-compositionality

- Another not so nice solution
  - Create a new lock which must be taken each time any of the two buffers are accessed
- The number of locks grows as we compose algorithms
  - Takes time
  - Increases the risk of programming errors

Optimistic Concurrency

- Lock-based synchronization can be seen as Pessimistic Concurrency: ”We always assume that we need mutual exclusion”
- Another option would be Optimistic Concurrency
  - Assume we have mutual exclusion
  - Perform our critical section
  - Check if everything was OK
  - Revert our actions if it wasn't
  - Otherwise proceed

Lock-free synchronization

- It is possible to write algorithms without locks, called *lock-free* synchronization
- Example:
  - Peterson's algorithm from lecture 1
- Typically uses complex instructions
  - Compare & Swap
  - Test & Set
- Is often faster than lock-based sync. because it allows for more concurrency
- Very difficult to do in general

Transactional Memory

- A concept to allow easy lock-free programming
- Although the programming model is lock-free implementations uses locks
- Can either be implemented in
    - Hardware
    - Software

Transactional Memory

- Used to be considered by many of the big companies to be the ”enabler” of concurrent programming
    - As computers get more cores programmers will need to write concurrent programs to make them faster
    - Transactional memory makes this sufficiently easy to be usable to a majority of the programmers
- Experience has shown that it is hard to add Transactional Memory to most existing programming language

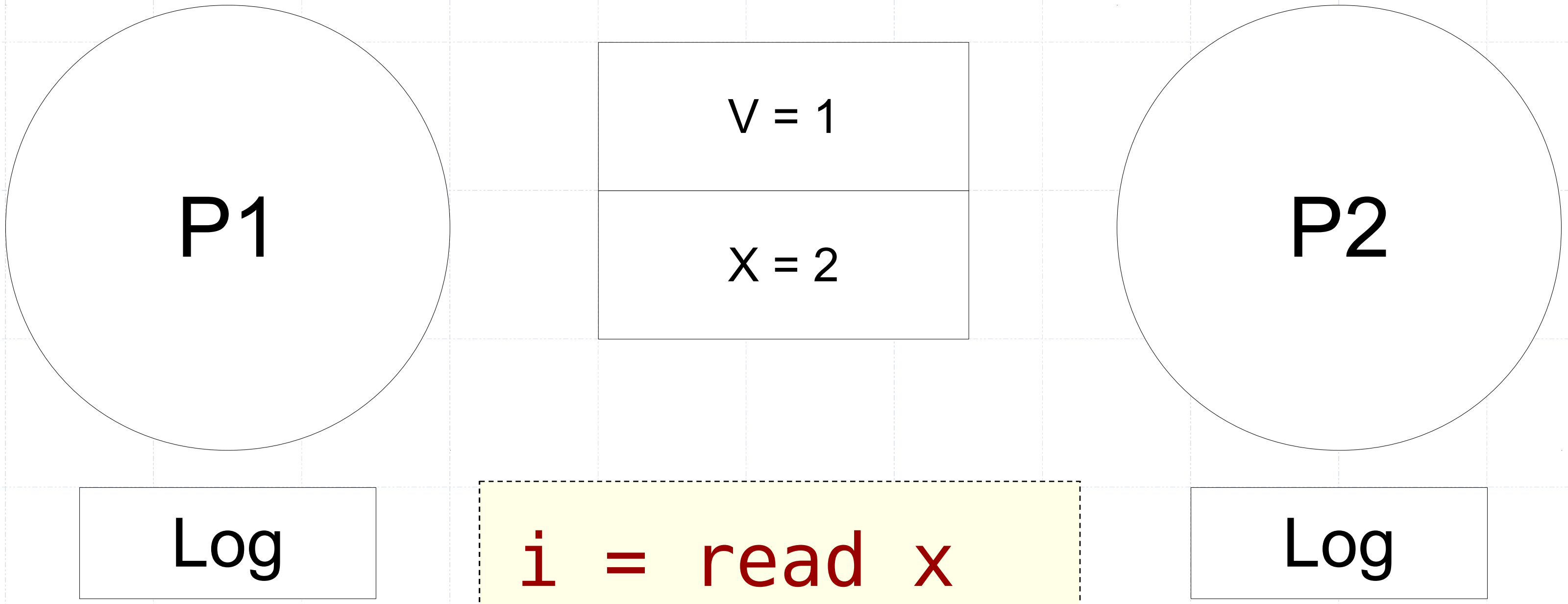
Transactions

- A standard database concept
    - A group of operations should execute atomically,
    - Or not at all
- Transactional Memory takes this idea to operations on memory and shared variables

Transactions

- One possible implementation of Transactions
    - When writing to variables, don't actually modify them, instead:
    - Keep a log over all the reads and writes that are made
    - When the transaction is done:
        - Validate: check that any read variables still have the same value
        - Commit: make the changes permanent
        - If the validation failed rerun the transaction

Transactions



P1

V = 1

X = 2

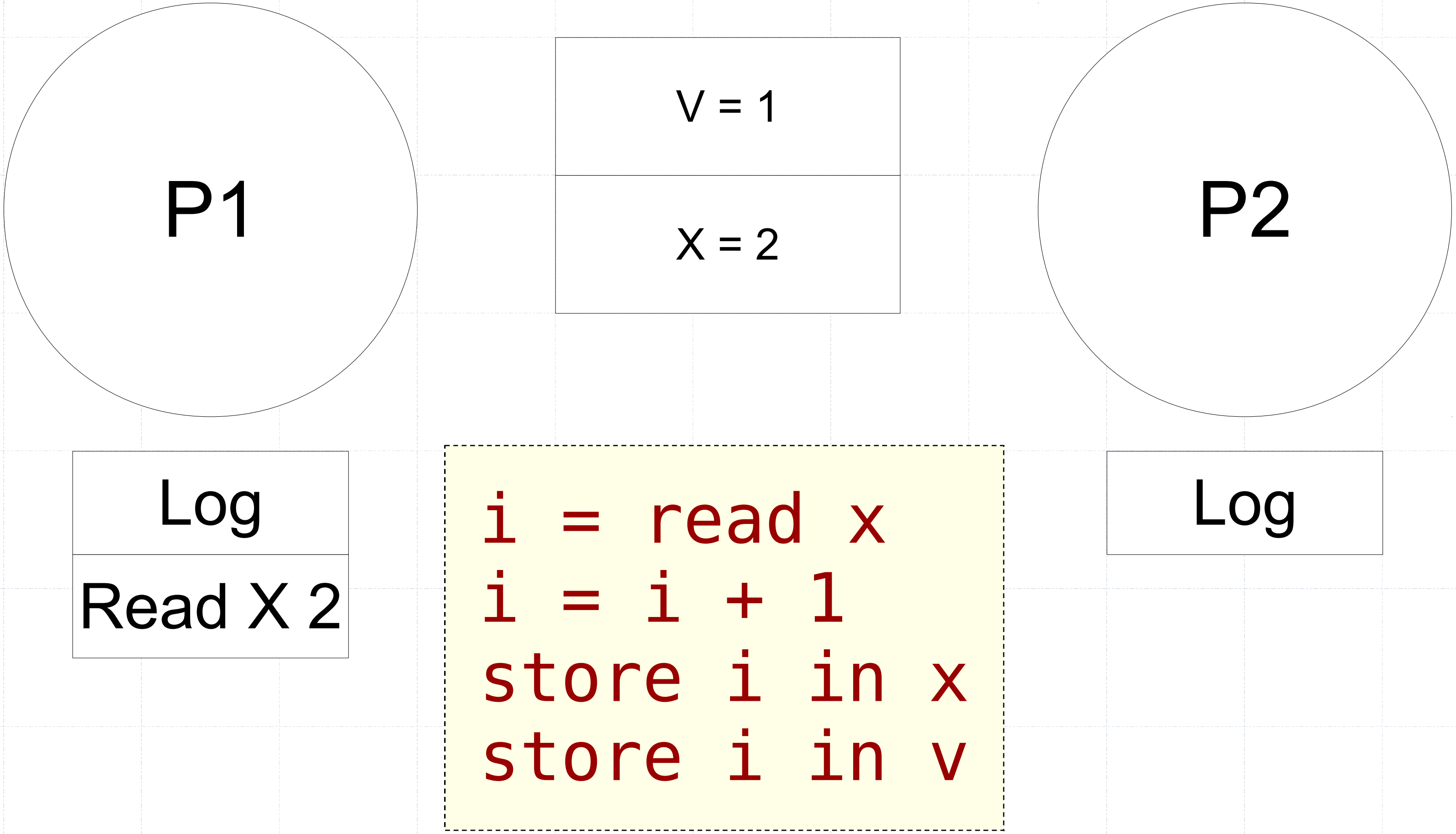
P2

Log

Log

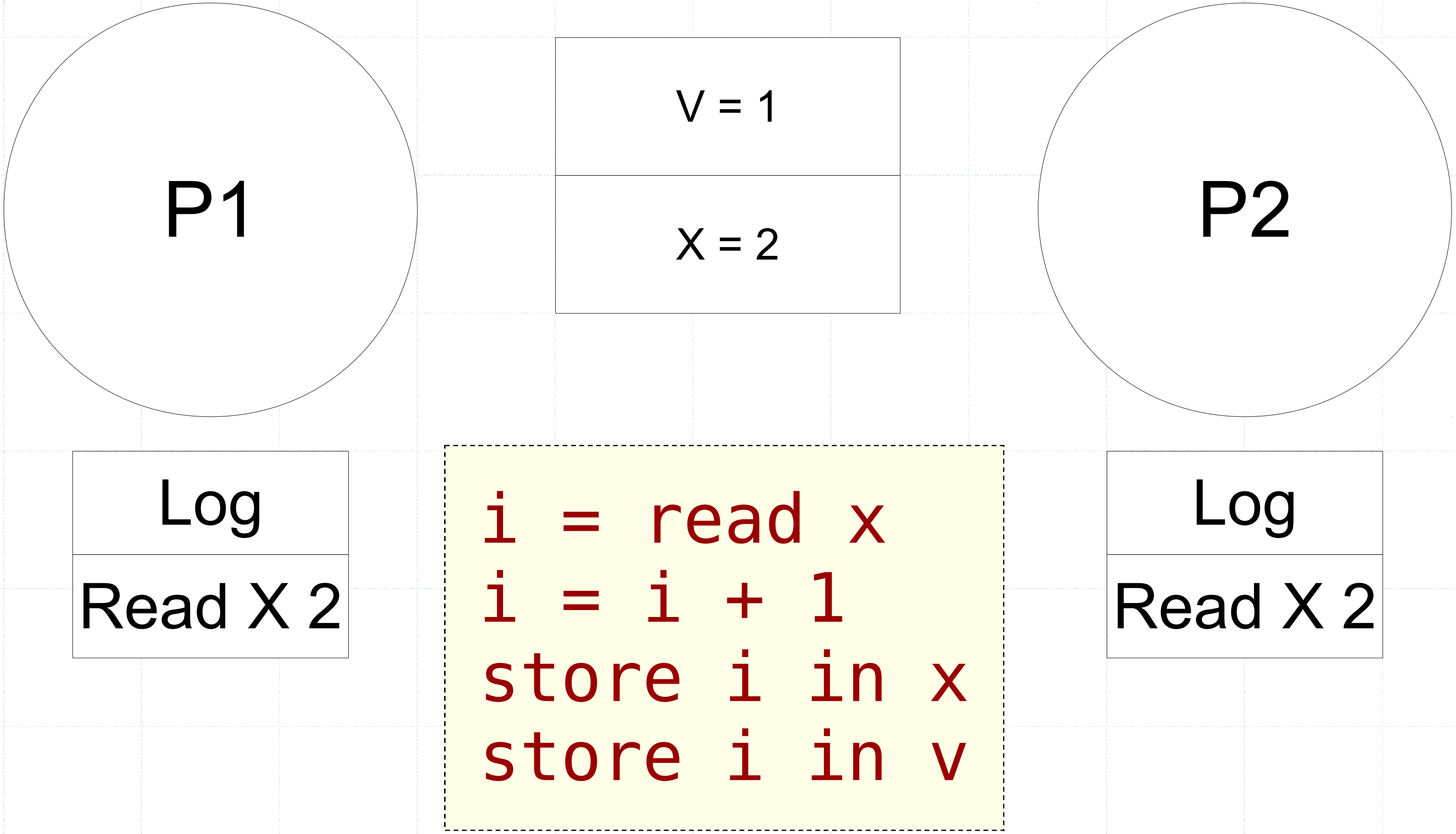
```
i = read x
i = i + 1
store i in x
store i in v
```

Transactions



```
i = read x
i = i + 1
store i in x
store i in v
```

Transactions



P1

V = 1

X = 2

P2

Log

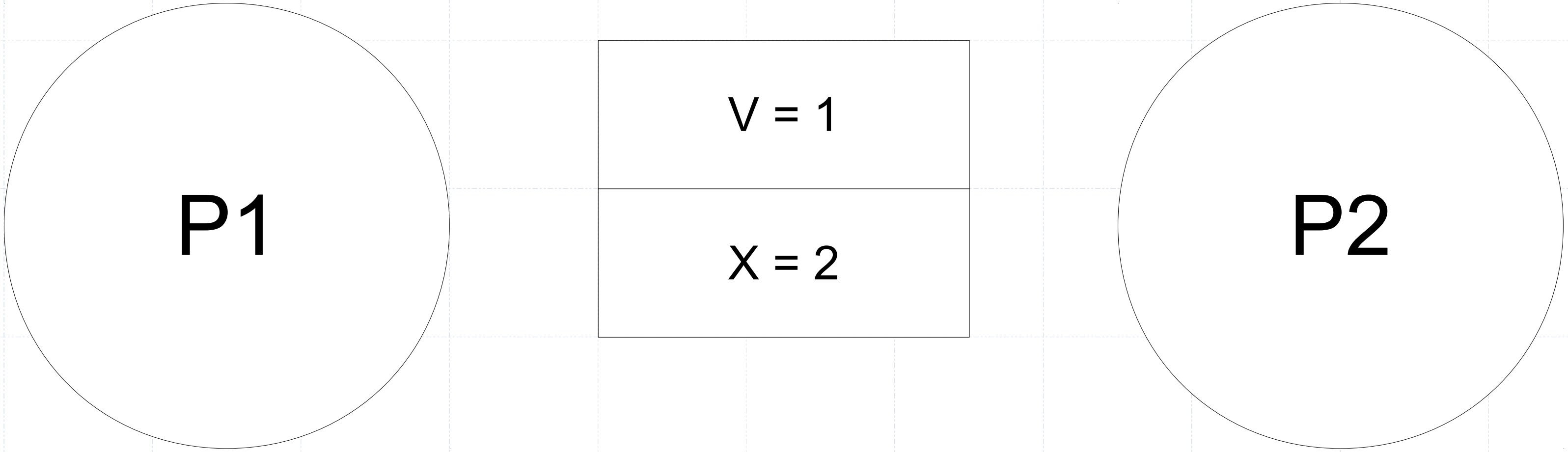
Read X 2

```
i = read x
i = i + 1
store i in x
store i in v
```

Log

Read X 2

Transactions

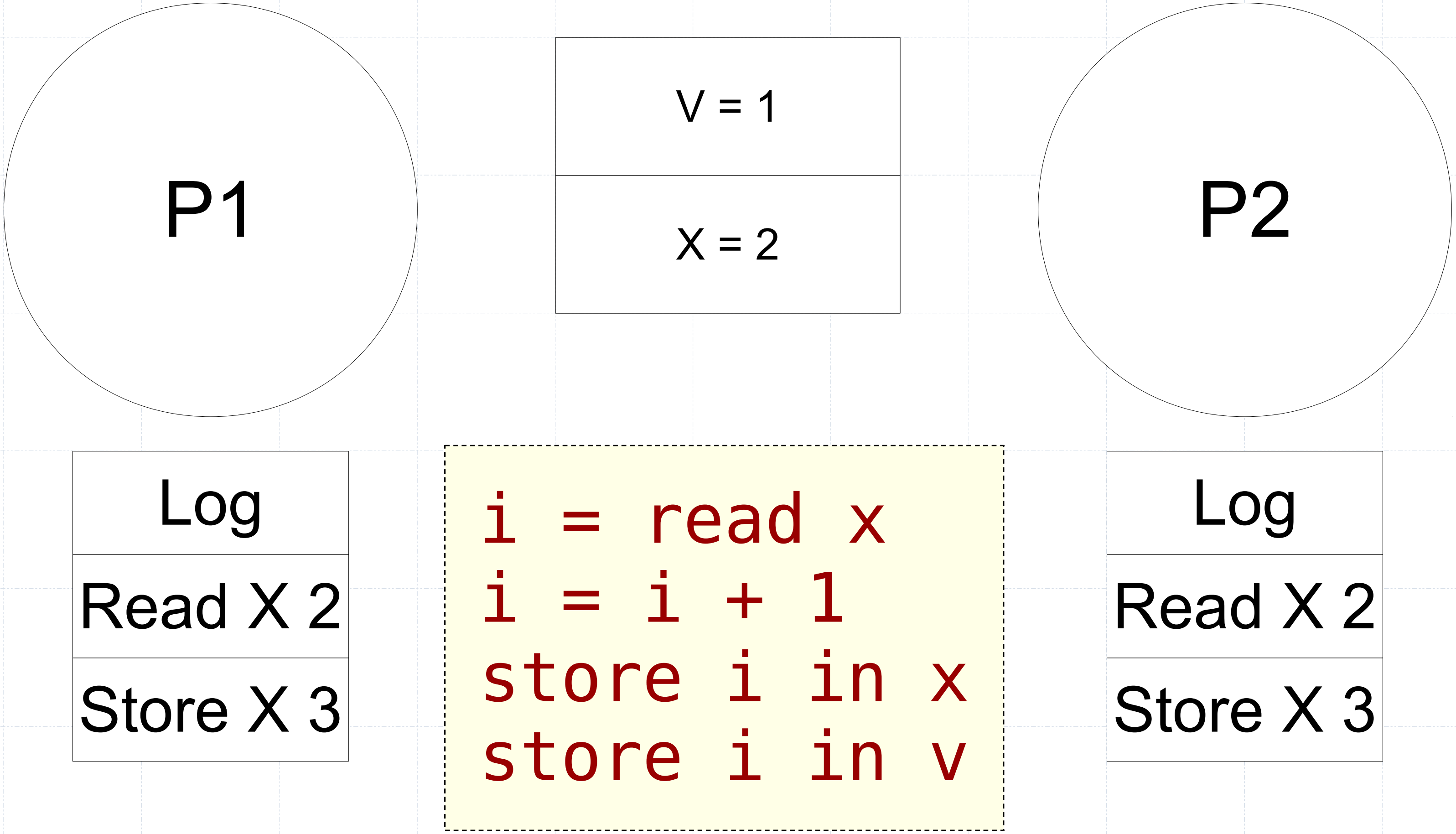


Log
Read X 2
Store X 3

```
i = read x
i = i + 1
store i in x
store i in v
```

Log
Read X 2

Transactions



P1

V = 1
X = 2

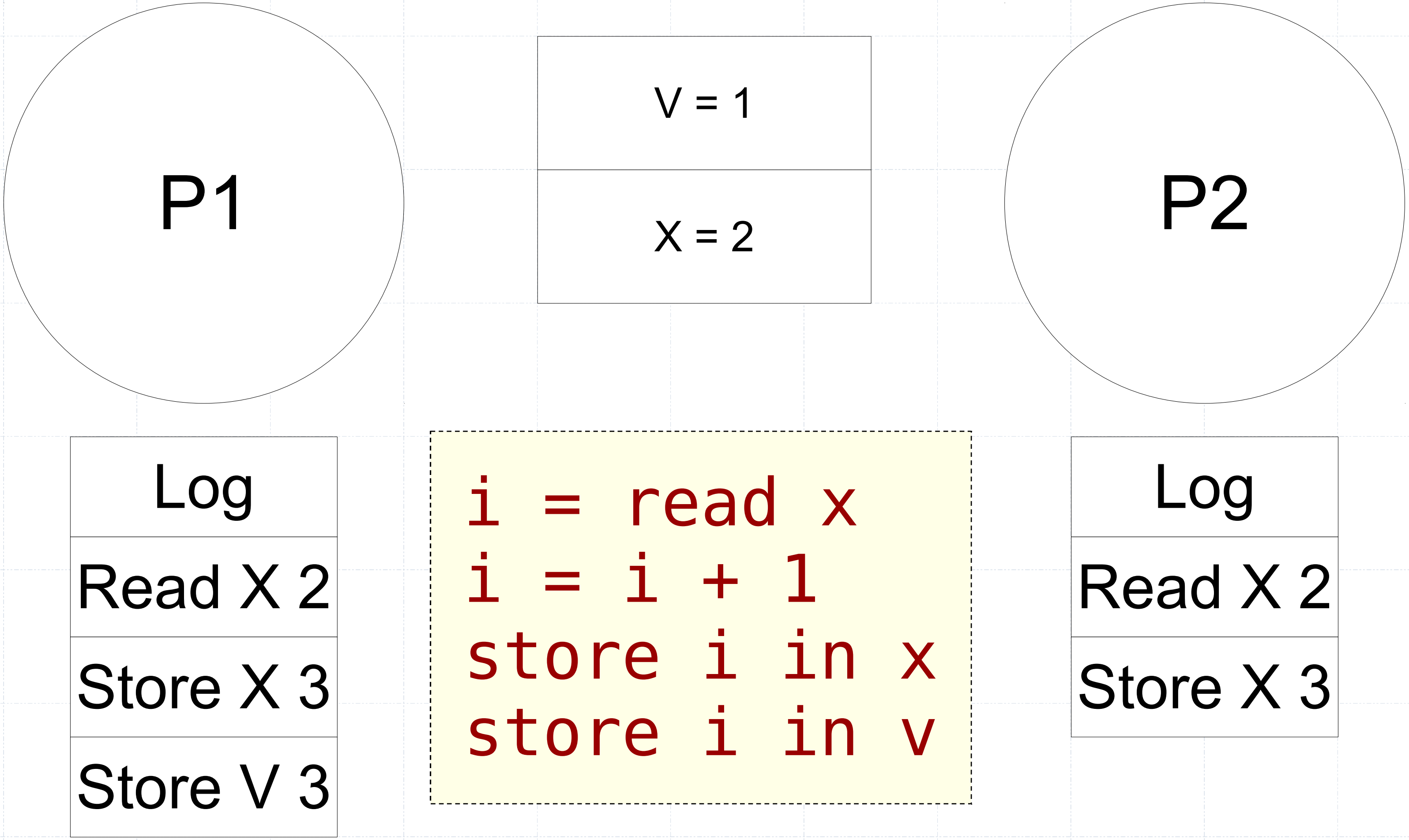
P2

Log
Read X 2
Store X 3

```
i = read x
i = i + 1
store i in x
store i in v
```

Log
Read X 2
Store X 3

Transactions



P1

V = 1

X = 2

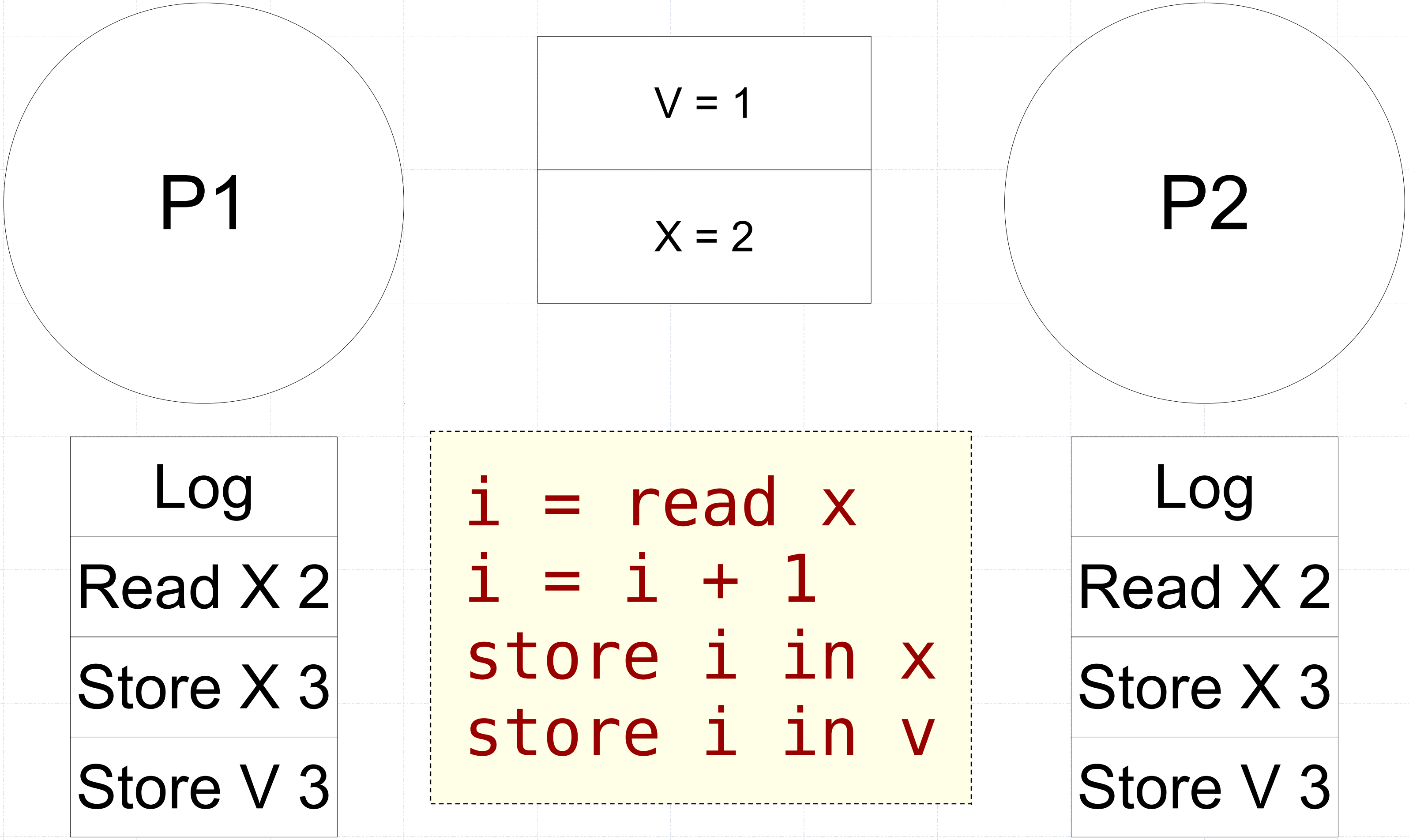
P2

Log
Read X 2
Store X 3
Store V 3

```
i = read x
i = i + 1
store i in x
store i in v
```

Log
Read X 2
Store X 3

Transactions



V = 1

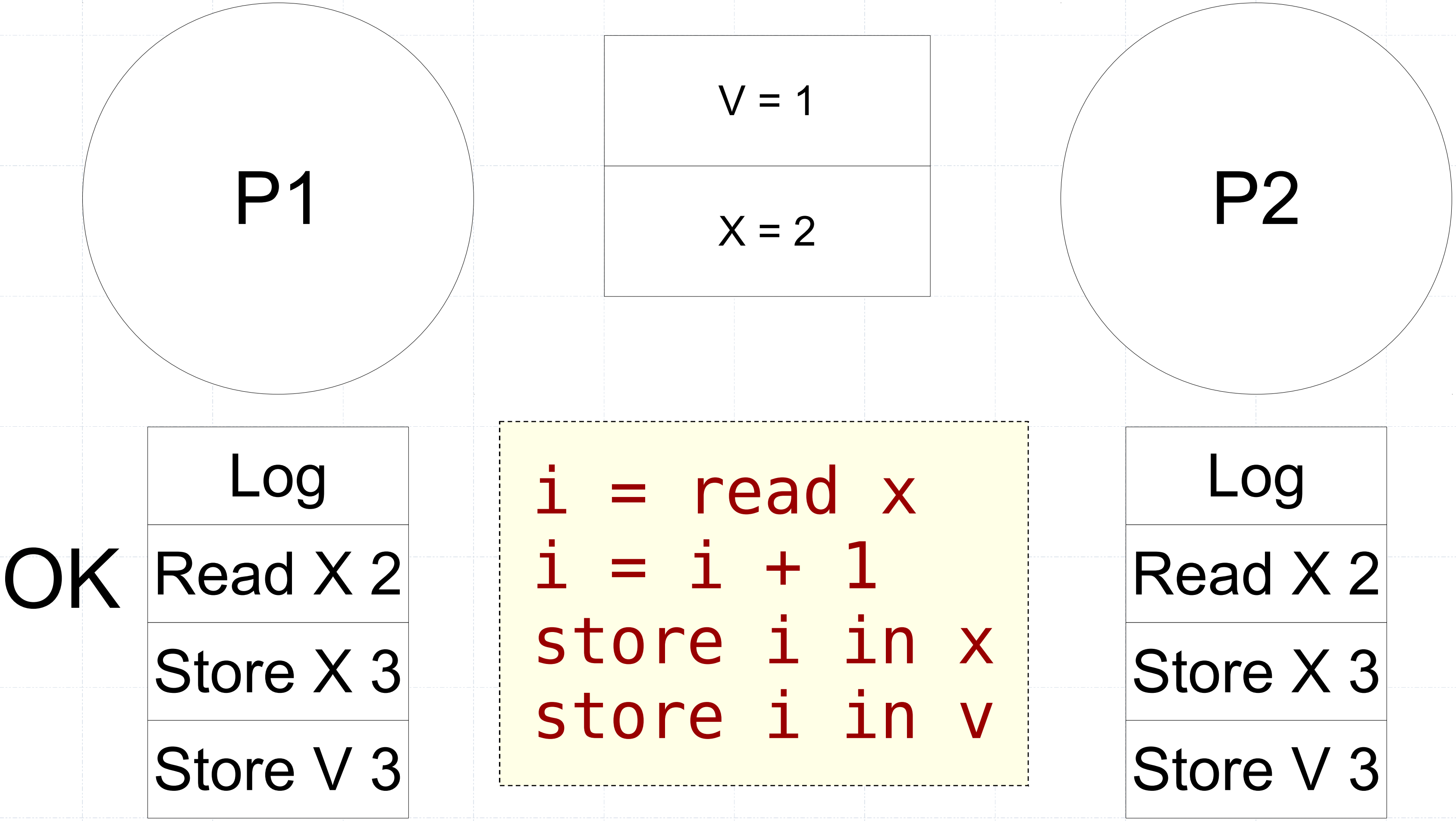
X = 2

Log
Read X 2
Store X 3
Store V 3

```
i = read x
i = i + 1
store i in x
store i in v
```

Log
Read X 2
Store X 3
Store V 3

Transactions



P1

V = 1
X = 2

P2

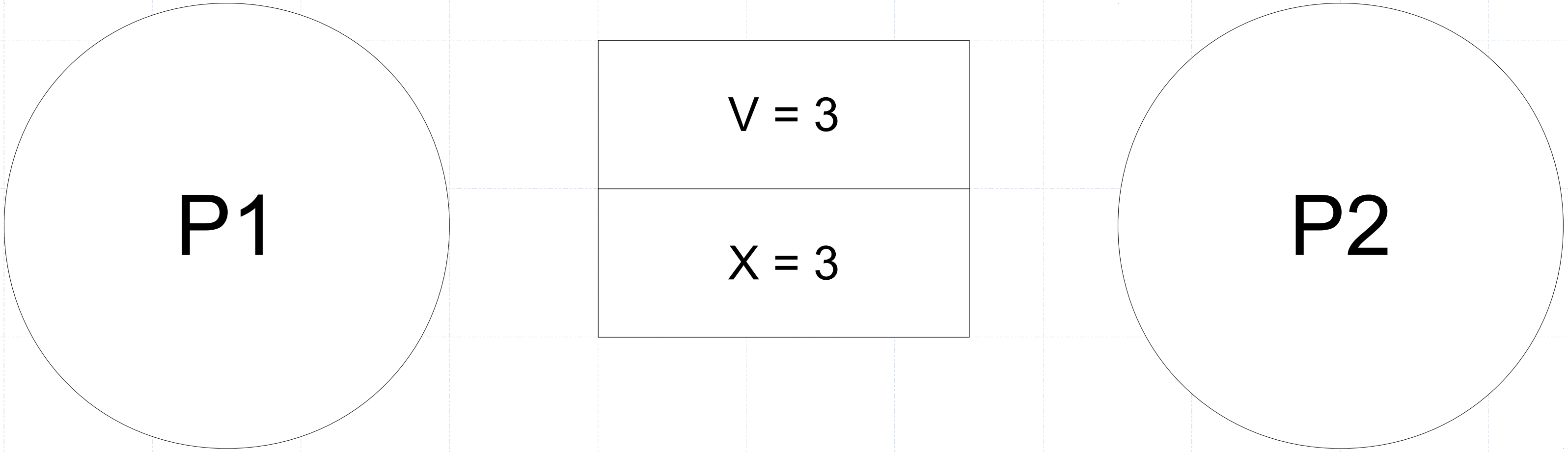
OK

Log
Read X 2
Store X 3
Store V 3

```
i = read x
i = i + 1
store i in x
store i in v
```

Log
Read X 2
Store X 3
Store V 3

Transactions



P1

V = 3

X = 3

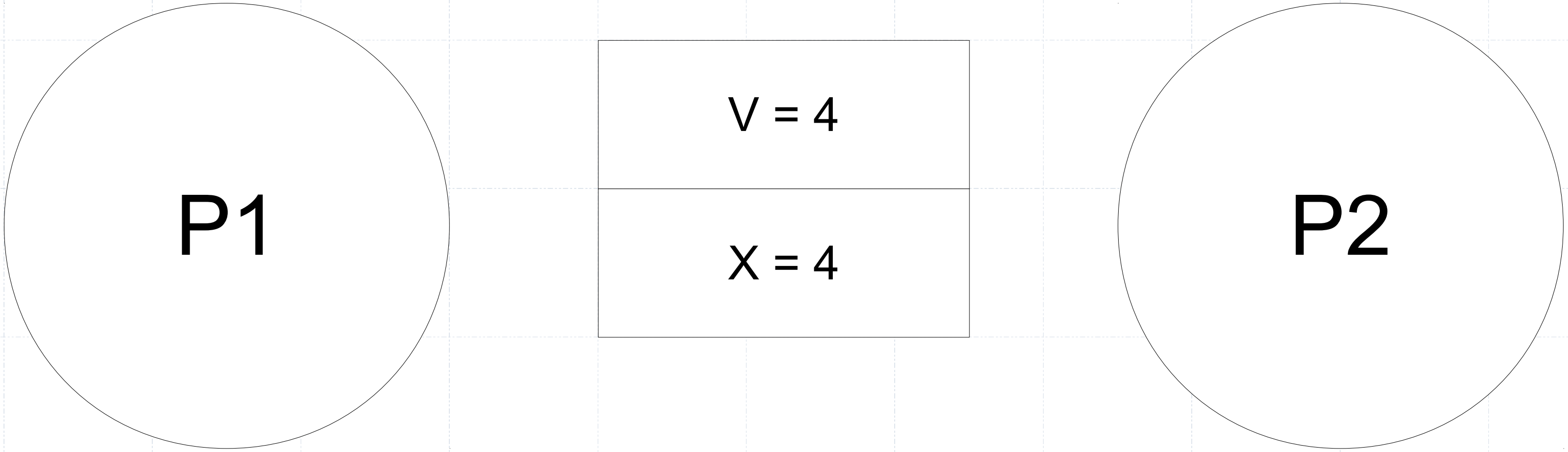
P2

```
i = read x
i = i + 1
store i in x
store i in v
```

Log
Read X 2
Store X 3
Store V 3

Wrong Redo!

Transactions



P1

V = 4

X = 4

P2

```
i = read x
i = i + 1
store i in x
store i in v
```

Transactions

- There are variations on how to implement transactions
- Previous slides only show one example implementation
- Still a research topic

Transactions

- Benefits of transactions:
  - Many processes can be in the critical section at the same time
    - More parallelism
    - They only need to rerun if there is an actual runtime conflict
  - Deadlocks cannot occur
  - Easy to compose
    - Commit only after the second transaction is done

Transactions

- Drawbacks of transactions:
    - Cannot guarantee fairness
        - A large transaction can be starved by many small ones
    - All the book keeping can be expensive

Hardware TM

- The initial proposal for Transactional Memory envisioned implementing it in Hardware
- Not a huge success in practice
  - Only one or two chips has ever had that feature
  - More chips planned but abandoned

Software Transactional Memory

- Software Transactional Memory (STM) can be used in various ways:
  - As a library
  - As a language construct

STM Libraries

- There exist several libraries for STM
    - Java: jvstm, JSTM (XSTM), DSTM2, Deuce
    - C/C++: TinySTM, LibLTX, LibCTM, RSTM
- Exists for C#, Python, Lisp, Ocaml ...

Language Support for STM

- Haskell
    - Glasgow Haskell Compiler has STM support in the runtime system
    - No new language construct, functionality exposed as a library
- Clojure
    - A descendant of lisp which uses STM for all mutable variables
- Perl 6
    - PUGS uses Haskell's support for STM

Language Support for STM

- Java
  - Proposed langage extension: Conditional Critical Regions
  - No implementation yet

Conditional Critical Regions

- Introduced by the `atomic` keyword
- Reminiscent of the `synchronized` keyword in Java
- Introduces a transaction, guarded by a condition

```
atomic (condition) {
    statements
}
```

Conditional Critical Regions

- (Part of) a shared buffer in Java

```
public synchronized int get() {
    int result;
    while (items == 0) wait();
    items--;
    result = buffer[items];
    notifyAll();
    return result;
}
```

Conditional Critical Regions

- A shared buffer using CCR

```
public int get() {
    atomic (items != 0) {
        items--;
        return buffer[items];
    }
}
```

Conditional Critical Regions

- Recognize this?

```
public int get() {
     atomic (items != 0) {
          items--;
          return buffer[items];
     }
}
```

```
public int get() {
     <await (items != 0)
          items--;
          return buffer[items];>
}
```

Conditional Critical Regions

- Conditional Critical Regions implements the await statement
- Clearly a powerful and convenient language construct

Side effects

- How many missiles will be launched?
- When will they be launched?

```
atomic {
    ...
    launchMissile();
    ...
}
```

Side effects

- How many times will we be promted to input something?

```
atomic {
    ...
    inp = inputFromKeyboard();
    ...
}
```

Side effects

- Side effects such as I/O don't mix very well with transactional memory
- Programs raise a runtime exception if I/O is performed during a transaction
- Issues like these make it difficult to implement and program with transactional memory in most languages

Haskell

- Functional
- Pure: side effects cannot occur everywhere
- Ideally suited for supporting STM
- GHC, a Haskell compiler, has support for STM

Haskell

- Pure and side effecting computations are separated by the type system

```
”a string” :: String
readLine   :: IO String
putStrLn   :: String -> IO ()
```

Haskell

- Pure and side effecting computations are separated by the type system

```
”a string” :: String
readLine   :: IO String
putStrLn   :: String -> IO ()
```

The type constructor IO indicates that this function can perform side effects

Haskell

- I/O is isolated using the type system
- STM can therefore easily be isolated from I/O
- But Haskell does not allow variables to be updated everywhere
- Solution: Add a new separate type constructor STM which allows separation

Haskell STM

```
module Control.Concurrent.STM

data STM a
data TVar a

newTVarIO  :: a -> IO (TVar a)
newTVar    :: a -> STM (TVar s)
readTVar   :: TVar a -> STM a
writeTVar  :: TVar a -> a -> STM ()
atomically :: STM a -> IO a
retry      :: STM a
orelse     :: STM a -> STM a -> STM a
instance Monad STM
```

Updating a counter

- Updating a counter in Haskell STM

```
update :: TVar Int -> STM ()
update counter =
    do v <- readTVar counter
       writeTVar counter (v+1)

updateIO :: TVar Int -> IO ()
updateIO counter =
    do putStrLn ”Before update”
       atomically (update counter)
       putStrLn ”After update”
```

Updating a counter

- Updating a counter in Haskell STM

STM means: part of a transaction

```
update :: TVar Int -> STM ()
update counter =
    do v <- readTVar counter
       writeTVar counter (v+1)

updateIO :: TVar Int
updateIO counter =
    do putStrLn ”Before update”
       atomically (update counter)
       putStrLn ”After update”
```

Performing the transaction

Conditional Synchronization

- The `retry` function is used for conditional synchronization
- Whenever a condition is not met simply call the `retry` function
- The transaction is then aborted and rerun at a later time
- When should a transaction rerun?

Conditional Synchronization

- Remember that transactions keep a log of which variables it accesses
- A transaction should not be rerun until any variables that it read has been modified

Semaphores in Haskell STM

```
type Sem = TVar Int

newSem :: Int -> IO Sem
newSem n = newTVarIO n

p :: Sem -> STM ()
p sem = do n <- readTVar sem
            if n > 0
               then writeTVar sem (n-1)
               else retry

v :: Sem -> STM ()
v sem = do n <- readTVar sem
           writeTVar sem (n+1)
```

Semaphores in Haskell STM

- Using semaphores

```
process n mutex = do
    ...
    atomically (p mutex)
    putStrLn (”Process ” ++ show n)
    atomically (v mutex)
    ...
```

Resource Allocation – Multiple

- Clients requiring multiple resources should not ask for resources one at a time
  - Why would this be bad?
- A controller controls access to copies of some resource
- Clients make requests to take or return *any* number of the resources
  - A request should only succeed if there are sufficiently many resources available,
  - Otherwise the request must block

Resource Allocation

```
type Resource = TVar Int

resource n = newTVarIO n

aquire res nr = do
    n <- readTVar res
    if n < nr
      then retry
      else writeTVar res (n-nr)

release res nr = do
    n <- readTVar res
    writeTVar res (n+nr)
```

Resource Allocation

- Contolling fairness
  - We've previously seen examples of how to explicitly controlling the fairness of resource allocation by waking up processes in the order we want
  - This doesn't apply to the transactional setting
  - ALL processes that blocks on a particular variable are woken up when a variable is modified.
  - It is up to the scheduler to ensure fairness

Unbounded Buffer

```
newBuffer = newTVarIO []

put buffer item = do
    ls <- readTVar buffer
    writeTVar buffer (ls ++ [item])

get buffer = do
    ls <- readTVar buffer
    case ls of
      [] -> retry
      (item:rest) -> do
        writeTVar buffer rest
        return item
```

Compositionality

- Composing transactions is embarrassingly simple

```
transfer buffer1 buffer2 = do
      item <- get buffer1
      put buffer2 item
```

Composing Alternatives

- It is useful to be able to compose transactions as *alternatives*
- Example: reading from one of several buffers
- Enters `orelse`

Composing Alternatives

- The workings of `orelse`:
  - It takes two transactions
  - Execute the first one and if it succeed, commit
  - If the first one retries, execute the second one
- Can be used to listen to several channels at once, like JR's input statement

```
getEither buffer1 buffer2 =
    get buffer1 `orelse` get buffer2
```

Dining Philosophers

```
simulation n = do
    forks <- replicateM n (newSem True)
    outputBuffer <- newBuffer
    for [0..n-1] $ \i ->
      forkIO (philosopher i outputBuffer
                 (forks!!i)
                 (forks!!((i+1)`mod`n)))
     output outputBuffer

output buf = do
    str <- atomically (get buf)
    putStrLn str
    output buf
```

Dining Philosophers

```
philosopher n buf fork1 fork2 = do
  atomically $ put buf
    (show n ++ ” thinking”)
  randomDelay
  atomically $ do
    p fork1
    p fork2
  atomically $ put buf
    (show n ++ ” eating”)
  randomDelay
  atomically $ do
    v fork1
    v fork2
  philosopher n buf fork1 fork2
```

Transactional Memory

- Provides a way to write concurrent programs without locks
- Advantages:
  - No deadlocks
  - Compositionality
- Disadvantages:
  - Fairness