Verification & Testing
Java Path Finder

Roderick Bloem

Java Path Finder

Developed at Nasa Ames to check space craft software

Used to find a concurrency bug in Deep Space 1 Software

Use to find bugs in a real-time operating system

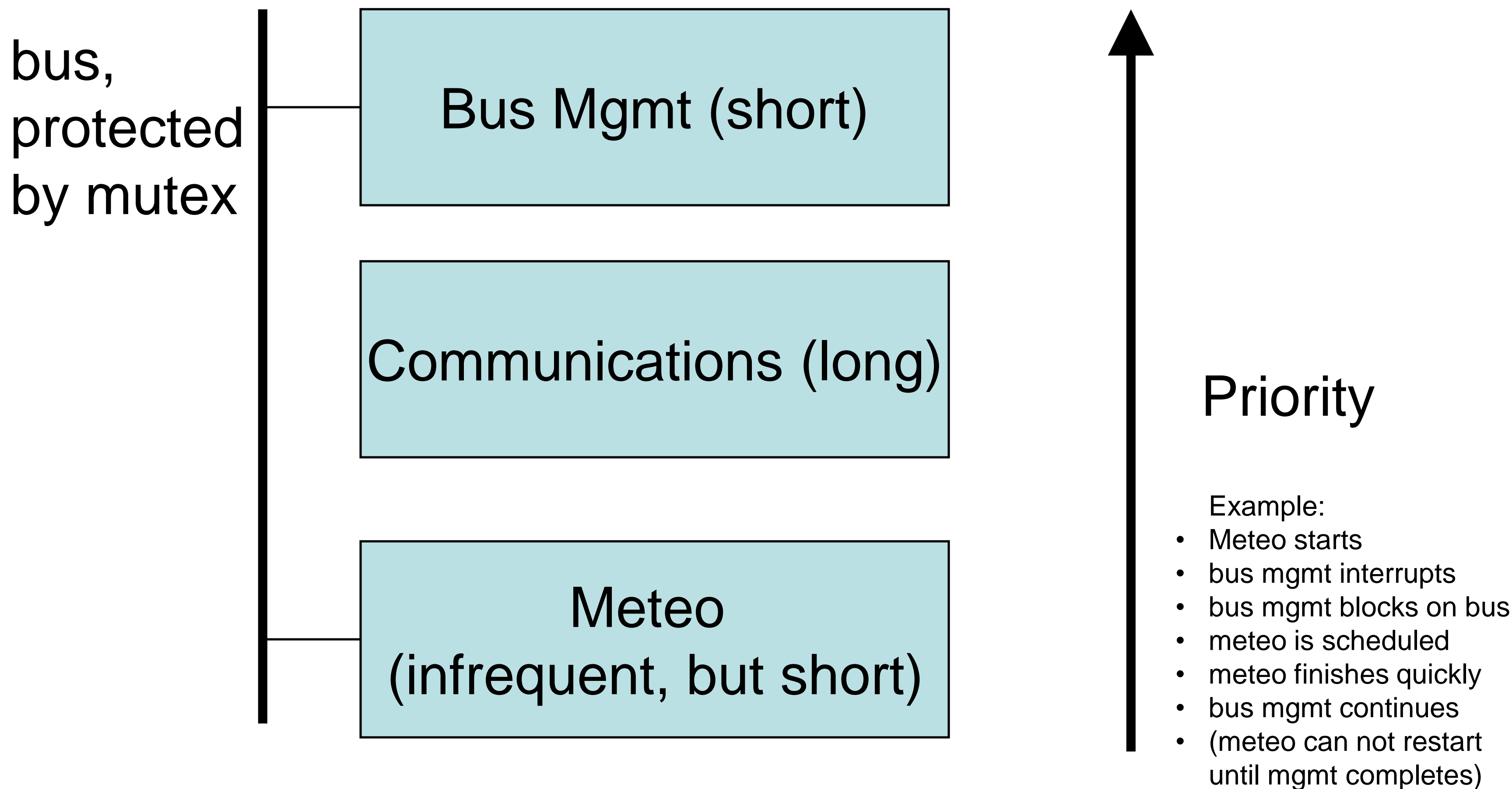
(Not quite automatically!)

Willem Visser, one of the JPF developers

This Week: Details of JPF

- Example of a concurrency bug
- Backtracking JVM
- Example
- Making the JVM efficient
  - Symmetry

Mars Rover Bug



Example:
- Meteo starts
- bus mgmt interrupts
- bus mgmt blocks on bus
- meteo is scheduled
- meteo finishes quickly
- bus mgmt continues
- (meteo can not restart until mgmt completes)

Mars Rover Bug

Meteo starts, locks bus

Management interrupts, blocks on locked bus

Communication starts, blocks out meteo because of priority

Management can not proceed until Communications done.
This takes too long, the system times out and diagnosis software reboots.

http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html

JPF: How It Is Done

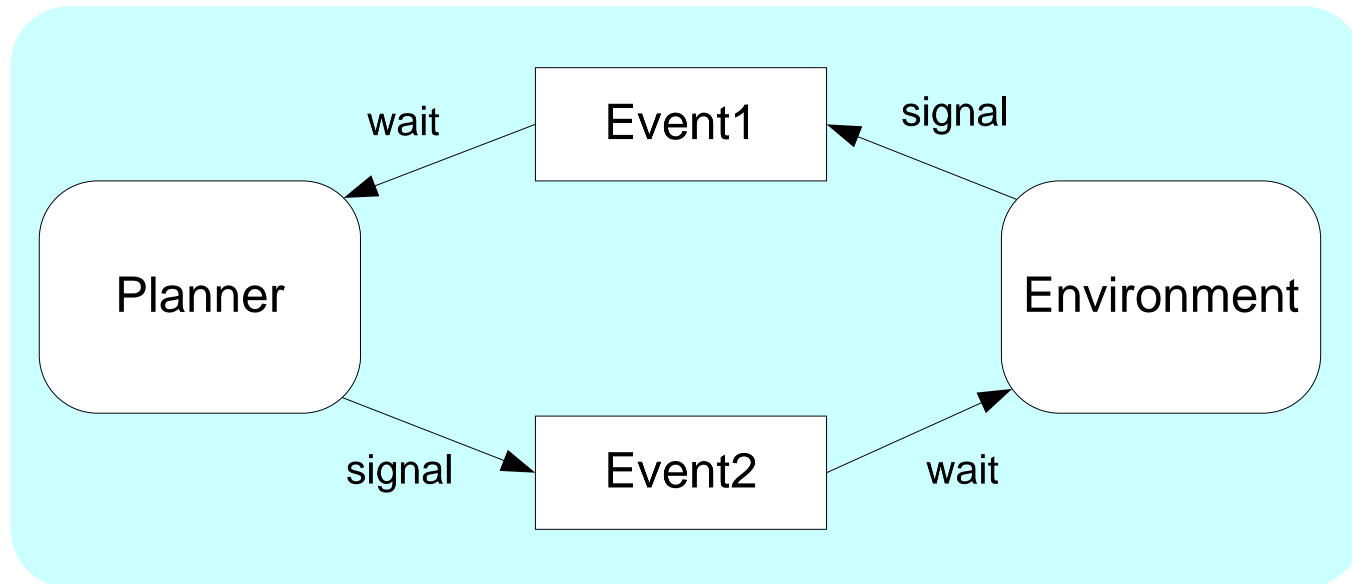
The meat:

- A backtracking Java Virtual Machine
- Supports all byte codes, including libraries

What makes it practical:

- Closed system assumption (you provide the inputs, JPF does thread scheduling)
- collapses states
- partial order reduction
- symmetry reduction
- program slicing
- Abstraction
- Runtime Analysis Techniques

JPF: Example



The events are not blocking for the sender. You can signal and then continue.

JPF: Example

```
class Event{
  int count = 0;

  public synchronized
  void wait_for_event(){
V1  wait();
  }

  public synchronized
  void signal_event(){
V2  count = count + 1;
V3  notifyAll();
  }
}
```

```
class Planner extends Thread{
   int count = 0;

  public void run(){
P1    while(true){
P2      if(count == e1.count)
P3        e1.wait_for_event();
P4      count = e1.count;
P5      // do work
P6      e2.signal_event();
  }}}

class Environment extends
Thread{
     public void run(){
E1     while(true){
E2       e1.signal_event();
E3       e2.wait_for_event
  }}}
```

Very large state! How do we solve that?

JPF: Example Abstracted

```
class Event{
  Boolean count = 0;

  public synchronized
  void wait_for_event(){
V1  wait();
  }

  public synchronized
  void signal_event(){
V2  count = !count;
V3  notifyAll();
  }
}
```

Hunt the Bug!

```
    class Planner extends Thread{
      Boolean count = 0;

      public void run(){
P1      while(true){
P2        if(count == e1.count)
P3          e1.wait_for_event();
P4        count = e1.count;
P5        // do work
P6        e2.signal_event();
    }}}

class Environment extends
Thread{
      public void run(){
E1      while(true){
E2        e1.signal_event();
E3        e2.wait_for_event
    }}}
```

JPF: Example Abstracted

```
class Event{
  Boolean count = 0;

  public synchronized
  void wait_for_event(){
V1  wait();
  }

  public synchronized
  void signal_event(){
V2  count = !count;
V3  notifyAll();
  }
}
```

```
   class Planner extends Thread{
     Boolean count = 0;

     public void run(){
P1     while(true){
P2       if(count == e1.count)
P3         e1.wait_for_event();
P4       count = e1.count;
P5       // do work
P6       e2.signal_event();
     }}}

class Environment extends
Thread{
     public void run(){
E1     while(true){
E2       e1.signal_event();
E3       e2.wait_for_event
     }}}
```

Hunt the Bug!

State Graph

Legend
Planner line, count, e1.count |
Env line e2.count

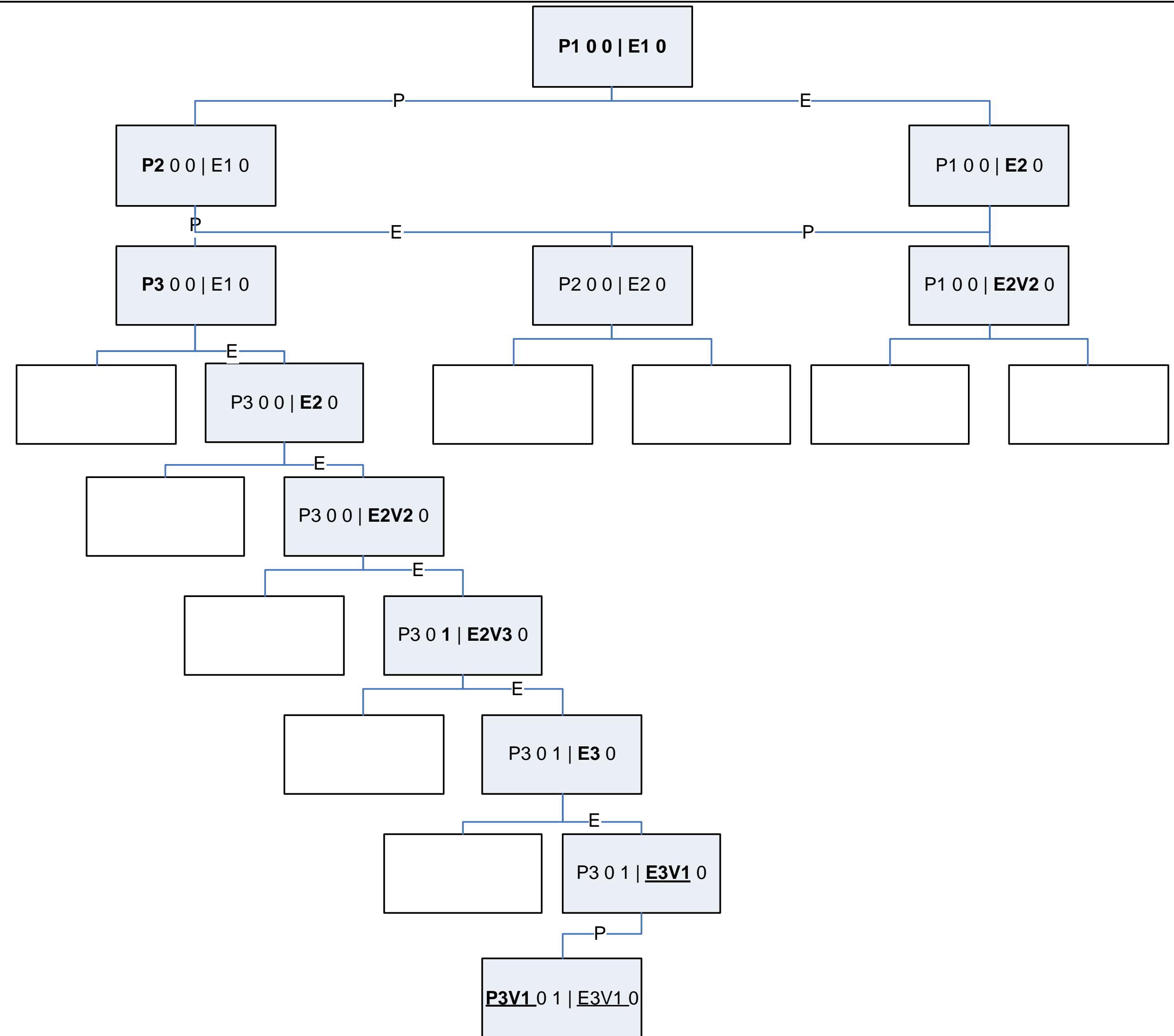
underlined means waiting

```
class Event{
  Boolean count = 0;

  public synchronized
  void wait_for_event(){
V1  wait();
  }

  public synchronized
  void signal_event(){
V2  count = !count;
V3  notifyAll();
  }
}
```

```
class Planner extends Thread{
   Boolean count = 0;

   public void run(){
P1    while(true){
P2      if(count == e1.count)
P3        e1.wait_for_event();
P4      count = e1.count;
P5      // do work
P6      e2.signal_event();
   }}}
```

```
class Environment extends
Thread{
   public void run(){
E1    while(true){
E2      e1.signal_event();
E3      e2.wait_for_event
   }}}
```

State Graph



Legend
Planner line, count, e1.count |
Env line e2.count

underlined means waiting

JPF Backtracking JVM

Keep track of states we have visited.

What is state?

- Call stack for every java thread
- Static info (in classes)
- Dynamic info (objects on the heap)
- Java operand stack

JPF Backtracking JVM

```
// depth first search
visit(s){
  if s in hashtable, return;
  enter current state in hash table
  for each possible next state s
    visit(s);
}
```

Performance:

20 states/second, 50k states in 512MB

Note: this is Depth-First Search. We can also do Breadth-First Search. What are benefits & disadvantages?

BFS & DFS

BFS

DFS

BFS & DFS

BFS

- Finds shortest path to error
- Does not get stuck in infinite loop
- Typically uses more memory

DFS

- May not find error at all
- Often uses less memory

Example of DFS Infinite Loop

- assert false is a “bug” that should be found
- Always schedule thread 1:
  - never returns to same state
  - Search does not end
- BFS finds bug after one or two schedulings

thread 1:

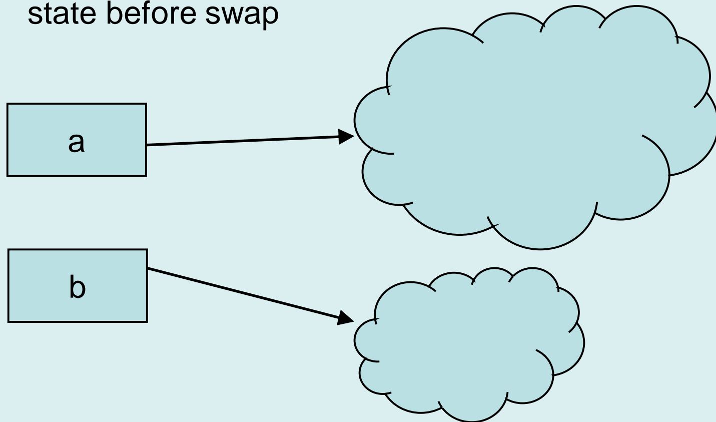
```
Vector a = new Vector();
while(true){
  a.add(new Object());
}
```

thread 2:

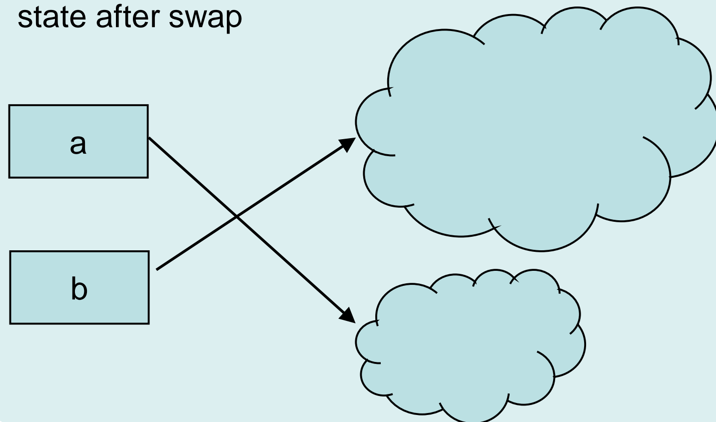
```
assert(false);
```

Collapsing example

Without Collapsing

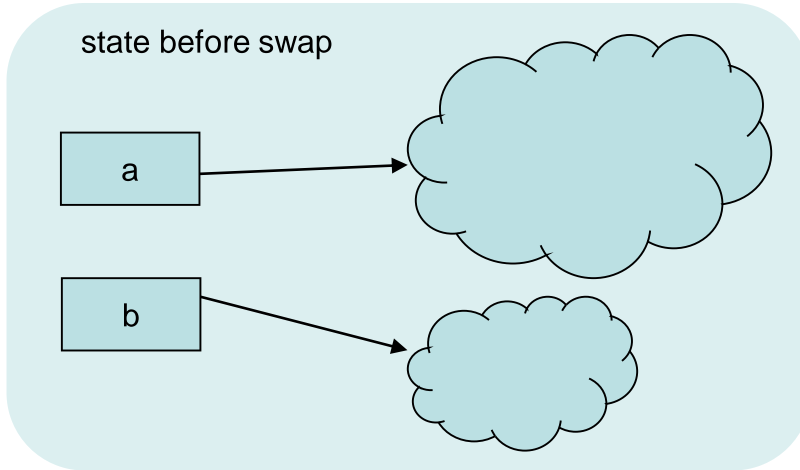


swap(a,b)

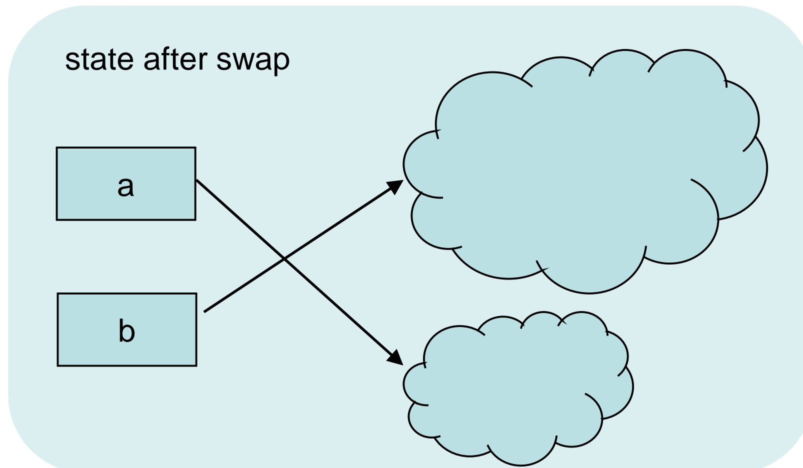


Collapsing example

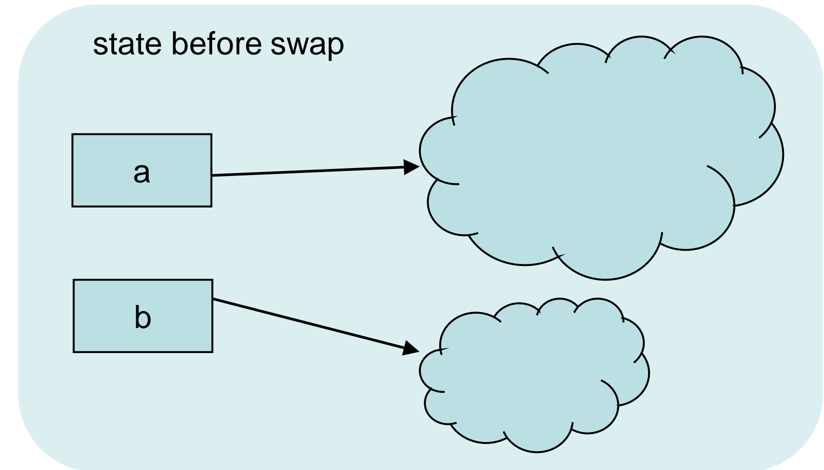
Without Collapsing



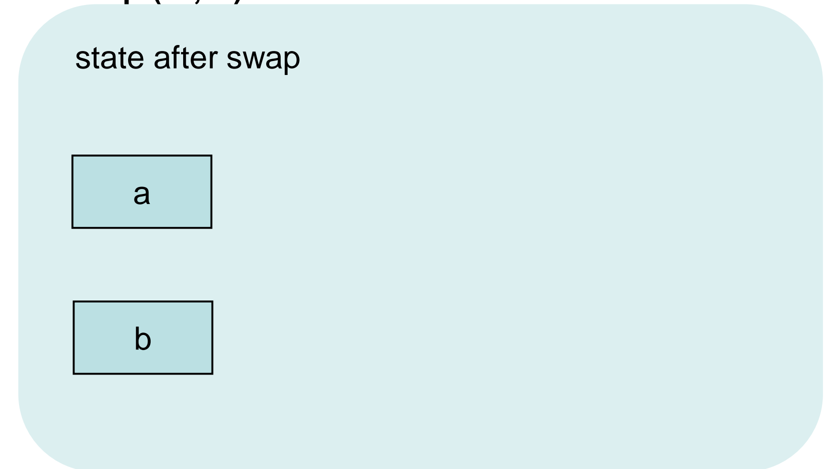
swap(a,b)



With Collapsing

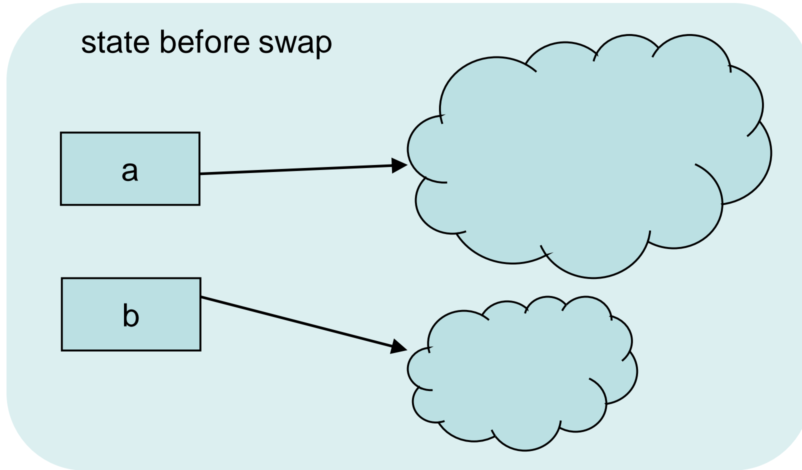


swap(a,b)

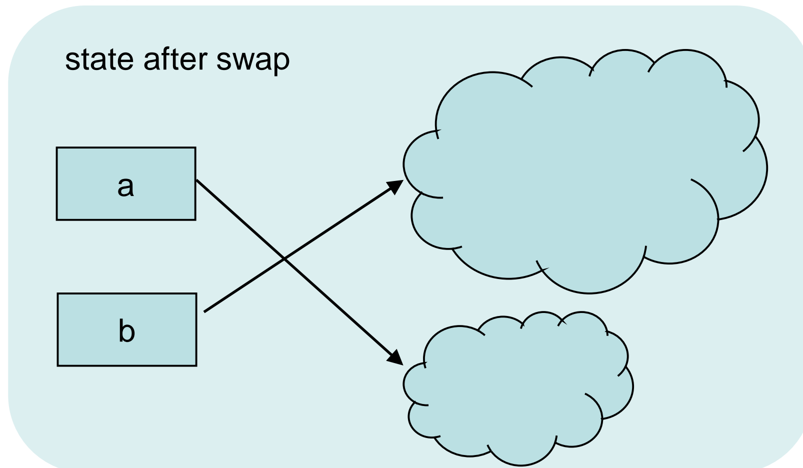


Collapsing example

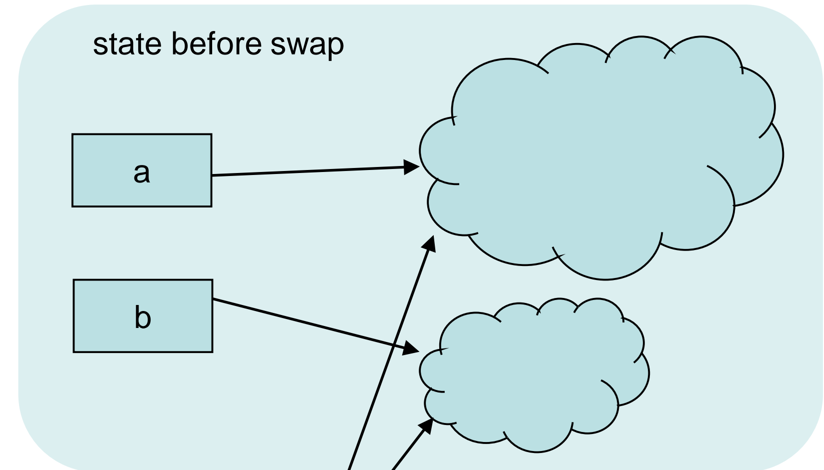
Without Collapsing



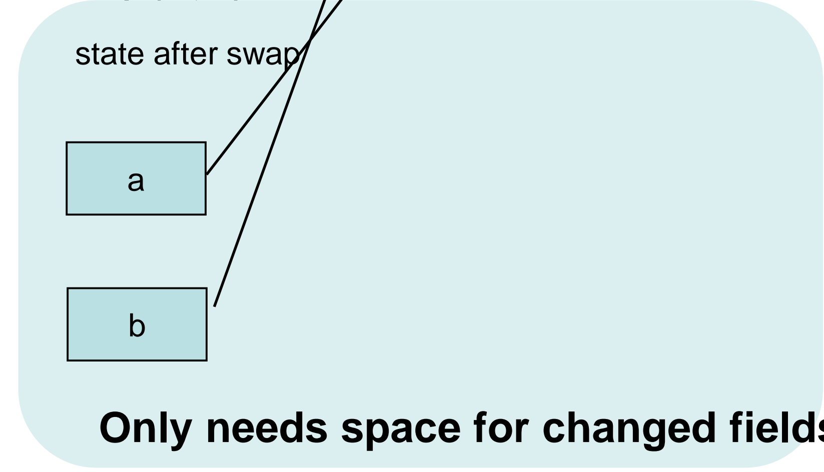
swap(a,b)



With Collapsing



swap(a,b)



Only needs space for changed fields!

Collapsing States

- States are structured and changes are local
- Objects are hashed on pointer value **and** content
- Store every object only once, use hash indices instead of memory addresses for fields

- Performance: 500-1500 states/second, millions of states in 512MB

- If we use uncollapsing to reconstruct states from the hash tables, we can use collapsed states on the DFS stack as well.

- Performance: 6k-10k states/second, 4X memory improvement

- Collapsing works well for DFS, but not necessarily for BFS.

Symmetry

- Do not consider a state if you have seen an *equivalent* state before
- Example: loading classes
- Suppose thread T1 loads class A, T2 loads class B.
- Depending on the order of execution, A and B end up in different locations. We have equivalent but different states:
  - (A,B) v. (B,A)

Symmetry: Class Loading

When loading a class, check if a slot number is assigned to it, and if the slot is free

- If yes, load class into that slot
- If no, load class into first free slot, assign a slot number

With n classes you have n instead of n! possible states.

Example: first you try an interleaving that loads A then B. You remember: A goes into slot 0, B into slot 1.

Then you backtrack and try an interleaving that loads B first. It goes to slot 1 even though slot 0 is available

Symmetry

Same problem for object creation.

Solution: Try to always put objects in the same place

An objects is identified by

1. The new() statement that creates it and
2. The invocation of that statement (numbered consecutively)

Symmetry

```
class S1 { int x; }

class T1 extends Thread {
  public void run(){
    S1 s1; int x = 1;
    s1 = new S1();
    x = 3;
  }
}

class Main{
  public static void main(…){
  T1 t1 = new T1();
  T2 t2 = new T2();
  t1.start(); t2.start();
}
```

```
class S2 { int y; }

class T2 extends Thread {
  public void run(){
    S2 s2; int x = 1;
    s2 = new S2();
    x = 3;
  }
}
```

Symmetry reduction reduces state space from 258 to 105 states.

More Tricks

- Multithreaded Slicing
- Abstraction
- Integration with Eraser and Locktree

Experience

- Found bug in Remote Agent Space Craft Controller
- Found bug in Real Time OS for aircraft