Concurrency

EECS 4315

www.eecs.yorku.ca/course/4315/

The readers-writers problem

The readers and writers problem, due to Courtois, Heymans and Parnas, is a classical concurrency problem. It models access to a database. There are many competing threads wishing to read from and write to the database. It is acceptable to have multiple threads reading at the same time, but if one thread is writing then no other thread may either read or write. A thread can only write if no thread is reading.

Reader class

```
public class Reader extends Thread {
  private Database database;

  public Reader(Database database) {
    this.database = database;
  }

  public void run() {
    this.database.read();
  }
}
```

Writer class

```java
public class Writer extends Thread {
  private Database database;

  public Writer(Database database) {
    this.database = database;
  }

  public void run() {
    this.database.write();
  }
}
```

Database class

```
public class Database {
  ...
  public Database() { ... }
  public void read() { ... }
  public void write() { ... }
}
```

Main class

```
Database database = new Database();

final int READERS = Integer.parseInt(args[0]);
for (int r = 0; r < READERS; r++) {
  (new Reader(database)).start();
}

final int WRITERS = Integer.parseInt(args[1]);
for (int w = 0; w < WRITERS; w++) {
  (new Writer(database)).start();
}
```

Recap

Quick overview of what we discussed in the last lecture.

Attributes of Database class

```
private boolean writing; // is any Writer writing?
private int readers; // number of Readers that are reading

public Database() {
  this.writing = false;
  this.readers = 0;
}
```

Waiting when a writer is writing

```
private synchronized void beginRead() {
  if (this.writing) {
    try {
      this.wait();
    } catch (InterruptedException e) {}
  }
}

public void read() {
  this.beginRead();
  // read
  ...
}
```

Modifying the writing attribute

```
public void write() {
  ...
  this.writing = true;
  // write
  this.writing = false;
  ...
}
```

Waiting when . . .

. . . another writer is writing or a reader is reading

```
private synchronized void beginWrite() {
  if (this.writing || this.readers > 0) {
    try {
      this.wait();
    } catch (InterruptedException e) {}
  }
}
```

Modifying the readers attribute

```
private synchronized void beginRead() {
  ...
  this.readers++;
}

private synchronized void endRead() {
  this.readers--;
}
```

End of recap

What remains to be done?

Wait and notify

General questions to ask:

- When does a thread have to wait?
- When can a waiting thread potentially continue?

Waking up waiting readers

Question

Readers may be waiting because a writer is writing. Where and how do we wake up these waiting readers?

Waking up waiting readers

Question

Readers may be waiting because a writer is writing. Where and how do we wake up these waiting readers?

Answer

Use the `notifyAll` once the writer is done with writing.

Waking up waiting readers

```
private synchronized void endWrite() {
  this.writing = false;
  this.notifyAll();
}
```

Waking up waiting readers

```
private synchronized void endWrite() {
  this.writing = false;
  this.notifyAll();
}
```

Question

Besides waiting readers, does the above `notifyAll` also wake up waiting writers?

Waking up waiting readers

```
private synchronized void endWrite() {
  this.writing = false;
  this.notifyAll();
}
```

Question

Besides waiting readers, does the above `notifyAll` also wake up waiting writers?

Answer

Yes.

Waking up waiting writers

Question

Writers may be waiting because (1) a writer is writing or (2) readers are reading. We have already seen that waiting writers are woken up once a writer is done with writing – capturing (1). Where and how do we wake up a waiting writer – capturing (2)?

Waking up waiting writers

Question

Writers may be waiting because (1) a writer is writing or (2) readers are reading. We have already seen that waiting writers are woken up once a writer is done with writing – capturing (1). Where and how do we wake up a waiting writer – capturing (2)?

Answer

Use the `notify` once a reader is done with reading.

Waking up waiting writers

Question

Writers may be waiting because (1) a writer is writing or (2) readers are reading. We have already seen that waiting writers are woken up once a writer is done with writing – capturing (1). Where and how do we wake up a waiting writer – capturing (2)?

Answer

Use the `notify` once a reader is done with reading.

Question

Any reader?

Waking up waiting writers

Question

Writers may be waiting because (1) a writer is writing or (2) readers are reading. We have already seen that waiting writers are woken up once a writer is done with writing – capturing (1). Where and how do we wake up a waiting writer – capturing (2)?

Answer

Use the `notify` once a reader is done with reading.

Question

Any reader?

Answer

Only the last reader.

Waking up waiting writers

```
private synchronized void endRead() {
  this.readers--:
  if (this.readers == 0) {
    this.notify();
  }
}
```

Correct?

Let us use JPF to try to find bugs in the `Database` class.

Configuration file

```
target=ReadersAndWriters
target.args=5,2
classpath=<path to ReadersAndWriters.class>
```

JPF report

```
JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 U
====================================================== syst
ReadersAndWriters.main("5","2")
====================================================== sear
====================================================== erro
gov.nasa.jpf.vm.NotDeadlockedProperty
deadlock encountered:
  thread Writer:{id:6,name:Thread-6,status:WAITING,priority
  thread Writer:{id:7,name:Thread-7,status:WAITING,priority
...
====================================================== stat
elapsed time:     00:00:01
states:           new=264,visited=209,backtracked=438,end=2
search:           maxDepth=50,constraints=0
choice generators: thread=263 (signal=18,lock=77,sharedRef=
heap:             new=417,released=952,maxLive=394,gcCycles
instructions:     7066
```

Deadlock

Question

When does a deadlock occur?

Deadlock

Question

When does a deadlock occur?

Answer

A deadlock occurs if the complete system cannot make any progress, although at least one thread has not terminated yet.

Deadlock

Question

When does a deadlock occur?

Answer

A deadlock occurs if the complete system cannot make any progress, although at least one thread has not terminated yet.

A typical deadlock scenario occurs when threads mutually wait for each other to progress.

Smallest instance

Rather than analyzing a model of 264 states, let us try to find a smallest instance (in terms of number of readers and writers) for which a deadlock occurs.

Smallest instance

Rather than analyzing a model of 264 states, let us try to find a smallest instance (in terms of number of readers and writers) for which a deadlock occurs.

READERS $=$ 1 and WRITERS $=$ 1: no deadlock

Smallest instance

Rather than analyzing a model of 264 states, let us try to find a smallest instance (in terms of number of readers and writers) for which a deadlock occurs.

READERS $=$ 1 and WRITERS $=$ 1: no deadlock
READERS $=$ 2 and WRITERS $=$ 1: no deadlock

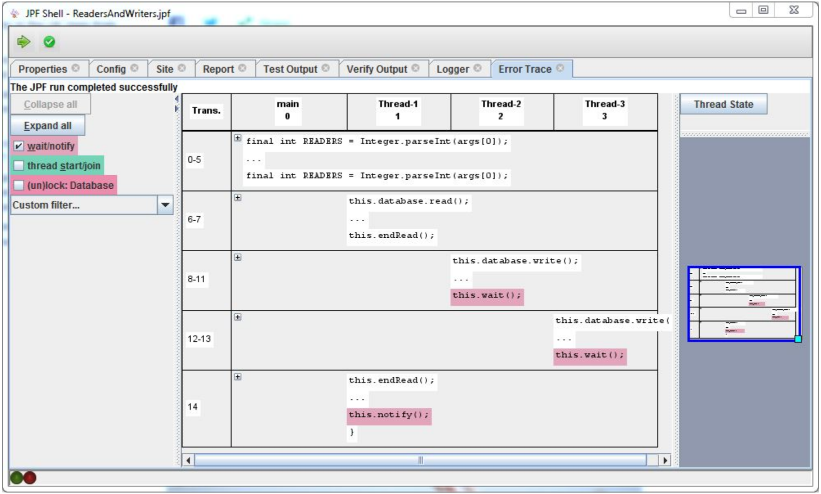
Smallest instance

Rather than analyzing a model of 264 states, let us try to find a smallest instance (in terms of number of readers and writers) for which a deadlock occurs.

READERS $=$ 1 and WRITERS $=$ 1: no deadlock
READERS $=$ 2 and WRITERS $=$ 1: no deadlock
READERS $=$ 1 and WRITERS $=$ 2: deadlock

The model has 244 states.

jpf-visual



The JPF run completed successfully

☑ wait/notify

☐ thread start/join

☐ (un)lock: Database

Trans.	main 0	Thread-1 1	Thread-2 2	Thread-3 3
0-5	final int READERS = Integer.parseInt(args[0]); ... final int READERS = Integer.parseInt(args[0]);			

6-7		this.database.read();
		...
		this.endRead();		

8-11			this.database.write();
			...
			this.wait();	

12-13				this.database.write(... this.wait();

14		this.endRead();
		...
		this.notify();
		}		

Notify versus notifyall

Replace

```
private synchronized void endRead() {
  this.readers--:
  if (this.readers == 0) {
    this.notify();
  }
}
```

with

```
private synchronized void endRead() {
  this.readers--:
  if (this.readers == 0) {
    this.notifyAll();
  }
}
```

JPF report

```
JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 U
====================================================== syst
ReadersAndWriters.main("1","2")
====================================================== sear
====================================================== resu
no errors detected
====================================================== stat
elapsed time:      00:00:02
states:            new=2842,visited=4523,backtracked=7365,en
search:            maxDepth=37,constraints=0
choice generators: thread=2842 (signal=243,lock=732,sharedR
heap:              new=1408,released=17116,maxLive=378,gcCyc
instructions:      63725
max memory:        113MB
loaded code:       classes=65,methods=1482
```

No writer

Question

How can we use JPF to check that there is no writer writing when a reader is reading?

No writer

Question

How can we use JPF to check that there is no writer writing when a reader is reading?

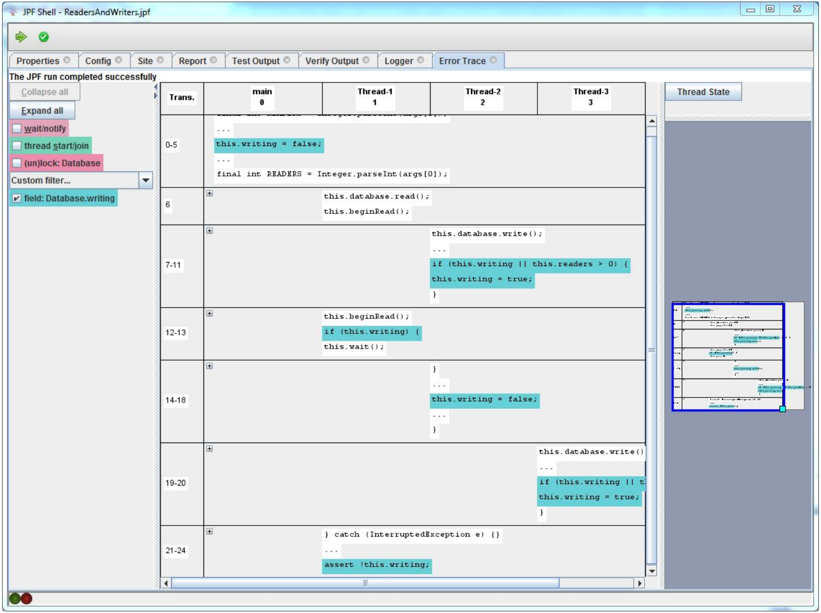
Answer

Add `assert !this.writing` in the `read` method where the database is read. If the assertion fails, an exception is thrown. JPF detects exceptions that are thrown and not caught.

JPF report

```
JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 U
====================================================== syst
ReadersAndWriters.main("1","2")
====================================================== sear
====================================================== erro
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.AssertionError
  at Database.read(Database.java:25)
  at Reader.run(Reader.java:22)
====================================================== snap
thread Reader:{id:1,name:Thread-1,status:RUNNING,priority:5
  call stack:
    at Database.read(Database.java:25)
    at Reader.run(Reader.java:22)
```

jpf-visual



The JPF run completed successfully

- ☐ wait/notify
- ☐ thread start/join
- ☐ (un)lock: Database
- Custom filter...
- ☑ field: Database.writing

Trans.	main 0	Thread-1 1	Thread-2 2	Thread-3 3

0-5	...
	this.writing = false;
	...
	final int READERS = Integer.parseInt(args[0]);			

6	⊞	this.database.read();
		this.beginRead();		

7-11	⊞		this.database.write();
			...
			if (this.writing || this.readers > 0) {
			this.writing = true;
			}	

12-13	⊞	this.beginRead();
		if (this.writing) {
		this.wait();		

14-18	⊞		}
			...
			this.writing = false;
			...
			}	

19-20	⊞			this.database.write()
				...
				if (this.writing || t
				this.writing = true;
				}

21-24	⊞	} catch (InterruptedException e) {}
		...
		assert !this.writing;		

Trace

main: running

```
Database database = new Database();

final int READERS = 1;
for (int r = 0; r < READERS; r++) {
  (new Reader(database)).start();
}
```

main: running, Reader: runnable

Trace

main: running, Reader: runnable

```
final int WRITERS = 2;
for (int w = 0; w < WRITERS; w++) {
  (new Writer(database)).start();
}
```

main: running, Reader: runnable, Writer: runnable, Writer: runnable

Trace

main: runnable, Reader: running, Writer: runnable, Writer: runnable

```
// this refers to the Reader object
this.database.read();
```

main: runnable, Reader: running, Writer: runnable, Writer: runnable

Trace

main: runnable, Reader: running, Writer: runnable,
Writer: runnable

```
// this refer to the Database object
this.beginRead();
```

main: runnable, Reader: running, Writer: runnable,
Writer: runnable

Trace

main: runnable, Reader: runnable, Writer: running, Writer: runnable

```
// this refer to the Writer object
this.database.write();
```

main: runnable, Reader: runnable, Writer: running, Writer: runnable

Trace

main: runnable, Reader: runnable, Writer: running,
Writer: runnable

```
// this refers to the Database object
this.beginWrite();
```

main: runnable, Reader: runnable, Writer: running,
Writer: runnable

Trace

main: runnable, Reader: runnable, Writer: running,
Writer: runnable

```
// this refers to the Database object
if (this.writing || this.readers > 0) {
  try {
    this.wait();
  } catch (InterruptedException e) {}
}
this.writing = true;
```

main: runnable, Reader: runnable, Writer: running,
Writer: runnable

Trace

main: runnable, Reader: running, Writer: runnable, Writer: runnable

```
// this refers to the Database object
if (this.writing) {
  try {
    this.wait();
  } catch (InterruptedException e) {}
}
```

main: runnable, Reader: blocked, Writer: runnable, Writer: runnable

Trace

main: runnable, Reader: blocked, Writer: running, Writer: runnable

```
// this refers to the Database object
this.endWrite();
```

main: runnable, Reader: blocked, Writer: running, Writer: runnable

Trace

main: runnable, Reader: blocked, Writer: running,
Writer: runnable

```
// this refers to the Database object
this.writing = false;
this.notifyAll();
```

main: runnable, Reader: runnable, Writer: running,
Writer: runnable

Trace

`main`: runnable, `Reader`: runnable, `Writer`: runnable,
`Writer`: running

```
// this refers to the Writer object
this.database.write();
```

`main`: runnable, `Reader`: runnable, `Writer`: runnable,
`Writer`: running

Trace

main: runnable, Reader: runnable, Writer: runnable,
Writer: running

```
// this refers to the Writer object
this.beginWrite();
```

main: runnable, Reader: runnable, Writer: runnable,
Writer: running

Trace

main: runnable, Reader: runnable, Writer: runnable,
Writer: running

```
// this refers to the Database object
if (this.writing || this.readers > 0) {
  try {
    this.wait();
  } catch (InterruptedException e) {}
}
this.writing = true;
```

main: runnable, Reader: runnable, Writer: runnable,
Writer: running

Trace

main: runnable, Reader: running, Writer: runnable, Writer: runnable

```
// this refers to the Database object
this.readers++;
assert !this.writing;
```

main: runnable, Reader: running, Writer: runnable, Writer: runnable

Bug

```
if (this.writing) {
  try {
    this.wait();
  } catch (InterruptedException e) {}
}
this.readers++;
```

Although the attribute `waiting` was `false` when the state of the `Reader` thread changed from blocked to runnable, it was not any more when the state of the `Reader` thread changed from runnable to running.

Bug

```
if (this.writing) {
  try {
    this.wait();
  } catch (InterruptedException e) {}
}
this.readers++;
```

Question

How do we modify the above code so that we check that `waiting` is `false` when the state of the `Reader` thread changed from runnable to running?

Bug

```
if (this.writing) {
  try {
    this.wait();
  } catch (InterruptedException e) {}
}
this.readers++;
```

Bug

```
if (this.writing) {
  try {
    this.wait();
  } catch (InterruptedException e) {}
}
this.readers++;
```

Answer

Replace `if` with `while`.

JPF report

```
JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 U
====================================================== syst
ReadersAndWriters.main("1","2")
====================================================== sear
====================================================== resu
no errors detected
====================================================== stat
elapsed time:      00:00:02
states:            new=2842,visited=4523,backtracked=7365,e
search:            maxDepth=37,constraints=0
choice generators: thread=2842 (signal=243,lock=732,shared
heap:              new=1408,released=17116,maxLive=378,gcCy
instructions:      63725
max memory:        113MB
loaded code:       classes=65,methods=1482
```

No reader

Question

How can we use JPF to check that there is no reader reading when a writer is writing?

No reader

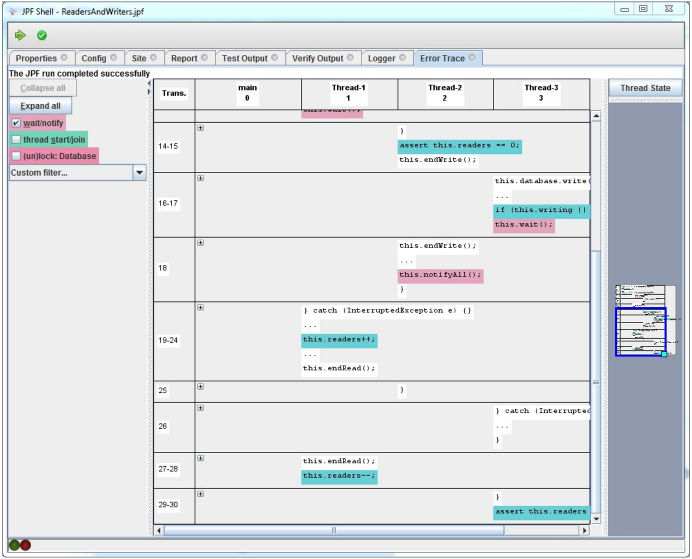
Question

How can we use JPF to check that there is no reader reading when a writer is writing?

Answer

Add `assert this.readers == 0` in the `write` method where the database is written.

Another bug



The JPF run completed successfully

☑ wait/notify
□ thread start/join
□ (un)lock: Database

Trans.	main 0	Thread-1 1	Thread-2 2	Thread-3 3
		this.wait();		

14-15	⊞		}
			assert this.readers == 0;
			this.endWrite();	

16-17	⊞			this.database.write(
				...
				if (this.writing ||
				this.wait();

18	⊞		this.endWrite();
			...
			this.notifyAll();
			}	

19-24	⊞	} catch (InterruptedException e) {}
		...
		this.readers++;
		...
		this.endRead();		

25	⊞		}	
26	⊞			} catch (Interrupted ... }
27-28	⊞	this.endRead(); this.readers--;		
29-30	⊞			} assert this.readers

Another bug

```
if (this.writing || this.readers > 0) {
  try {
    this.wait();
  } catch (InterruptedException e) {}
}
this.writing = true;
```

Another bug

```
if (this.writing || this.readers > 0) {
  try {
    this.wait();
  } catch (InterruptedException e) {}
}
this.writing = true;
```

Fix

Replace `if` with `while`.

JPF report

```
JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 U
====================================================== syst
ReadersAndWriters.main("1","2")
====================================================== sear
====================================================== resu
no errors detected
====================================================== stat
elapsed time:      00:00:02
states:            new=2842,visited=4523,backtracked=7365,en
search:            maxDepth=37,constraints=0
choice generators: thread=2842 (signal=243,lock=732,shared
heap:              new=1408,released=17116,maxLive=378,gcCy
instructions:      63725
max memory:        113MB
loaded code:       classes=65,methods=1482
```

No other writer

Question

How can we use JPF to check that there is no other writer writing when a writer is writing?

No other writer

Question

How can we use JPF to check that there is no other writer writing when a writer is writing?

Answer

- Add attribute `writers` ("ghost variable").
- Initialize `writers` to zero.
- Increment and decrement `writers` in the `write` method.
- Add `assert this.writers == 1` in the `write` method where the database is written.

Wait

In most cases, wrap a `wait` in a `while` loop.

Notify versus notifyAll

In most cases, use `notifyAll` instead of `notify`.

Synchronized blocks

Instead of synchronized methods, one can also use synchronized blocks.

```
synchronized (someObjectReference) {
  ... // executed once the lock of someObjectReference
      // has been acquired
}
```

Synchronized blocks

```
public void read() {
  synchronized(this) {
    while (this.writing) {
      this.wait();
    }
    this.readers++;
  }
  // read
  assert !this.writing;
  synchronized (this) {
    this.readers--;
    if (this.readers == 0) {
      this.notifyAll();
    }
  }
}
```

State space

Number of states

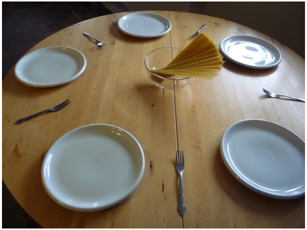
	0	1	2	3	4	5
0	1	30	374	4,046	41,115	400,033
1	24	338	3,833	39,791	391,614	3,711,014
2	356	3,894	40,009	394,027	3,745,232	
3	4,352	42,856	413,962	3,913,381		
4	47,786	452,488				
5	493,298					

Columns: number of readers

Rows: number of writers

The dining philosophers problem

In the dining philosophers problem, due to Dijkstra, five philosophers are seated around a round table. Each philosopher has a plate of spaghetti. A philosopher needs two forks to eat it. The layout of the table is as follows.



The life of a philosopher consists of alternative periods of eating and thinking. When philosophers get hungry, they try to pick up their left and right fork, one at a time, in either order. If successful in picking up both forks, the philosopher eats for a while, then puts down the forks and continues to think.

The dining philosophers problem

```
public class Philosopher extends Thread {
  private int id;
  private Table table;

  public Philosopher(int id, Table table) {
    this.id = id;
    this.table = table;
  }

  public void run() {
    while (true) {
      this.table.pickUp(id);
      this.table.pickUp(id + 1);
      // eat
      this.table.putDown(id);
      this.table.putDown(id + 1);
    }
```

The dining philosophers problem

```
public class Table {
  private int size;

  public Table(int size) { ... }
  public void pickUp(int id) { ... }
  public void putDown(int id) { ... }
}
```

The dining philosophers problem

```java
public class Philosophers {
  public static void main(String[] args) {
    final int PHILOSOPHERS = 5;
    Table table = new Table(PHILOSOPHERS);
    for (int id = 0; id < PHILOSOPHERS; id++) {
      (new Philosopher(id, table)).start();
    }
  }
}
```

Wait and notify

General questions to ask:

- When does a thread have to wait?
- When can a waiting thread potentially continue?

Wait

Question

When does a philosopher have to wait?

Wait

Question

When does a philosopher have to wait?

Answer

When either fork is not available.

Table

Question

Of what information about the forks should we keep track?

Table

Question

Of what information about the forks should we keep track?

Answer

Whether it has been picked up.

Table

Question

Of what information about the forks should we keep track?

Answer

Whether it has been picked up.

Question

How do we represent this information?

Table

Question

Of what information about the forks should we keep track?

Answer

Whether it has been picked up.

Question

How do we represent this information?

Answer

As an attribute of type `boolean[]`.

Table

Question

Where and how do we initialize the attribute?

Table

Question

Where and how do we initialize the attribute?

Answer

```
private boolean[] pickedUp;

public Table(int size) {
  this.size = size;
  this.pickedUp = new boolean[size]; // all false
}
```

Table

Question

Implement the method `pickUp(int id)`.

- When does a `Philosopher` have to wait?
- How does the array `pickedUp` need to be updated?

Table

Question

Implement the method pickUp(int id).

- When does a Philosopher have to wait?
- How does the array pickedUp need to be updated?

Answer

```
while (this.pickedUp[id % table.size]) {
  try {
    this.wait();
  } catch (InterruptedException e) {}
}
this.pickedUp[id % table.size] = true;
```

Notify

Question

When is a philosopher woken up?

Notify

Question

When is a philosopher woken up?

Answer

When a fork is put down.

Table

Question

Implement the method `putDown(int id)`.

- How does the array `pickedUp` need to be updated?
- Do `Philosopher`s need to be notified?

Table

Question

Implement the method `putDown(int id)`.

- How does the array `pickedUp` need to be updated?
- Do `Philosopher`s need to be notified?

Answer

```
this.pickedUp[id % table.size] = false;
this.notifyAll();
```

The dining philosophers problem

Question

Does this solve the problem?

The dining philosophers problem

Question

Does this solve the problem?

Answer

No.

The dining philosophers problem

Question

Does this solve the problem?

Answer

No.

Question

Why not?

The dining philosophers problem

Question

Does this solve the problem?

Answer

No.

Question

Why not?

Answer

Deadlock.

JPF

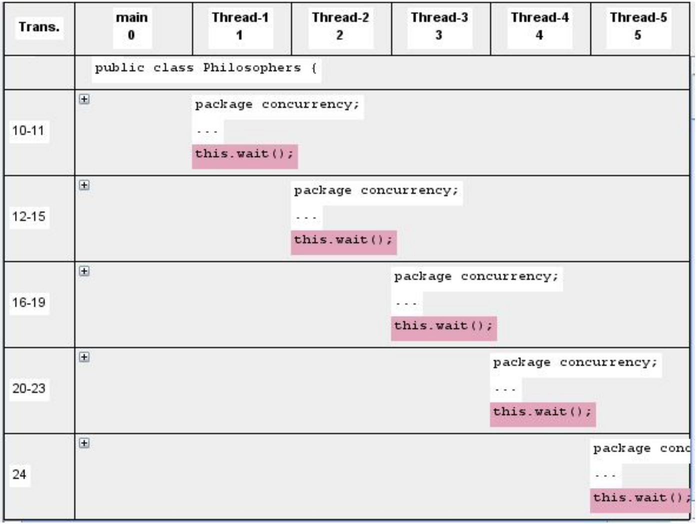
```
JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 U
====================================================== syst
concurrency.Philosophers.main()
====================================================== sear
====================================================== erro
gov.nasa.jpf.vm.NotDeadlockedProperty
deadlock encountered:
thread concurrency.Philosopher:{id:1,name:Thread-1,status:W
thread concurrency.Philosopher:{id:2,name:Thread-2,status:W
thread concurrency.Philosopher:{id:3,name:Thread-3,status:W
thread concurrency.Philosopher:{id:4,name:Thread-4,status:W
thread concurrency.Philosopher:{id:5,name:Thread-5,status:W
...
```

jpf-visual

```
target=Philosophers
classpath=<path to Philosophers.class>
sourcepath=<path to Philosophers.java>

@using jpf-visual
report.errorTracePrinter.property_violation=trace
report.publisher+=,errorTracePrinter
report.errorTracePrinter.class=ErrorTracePrinter
shell=gov.nasa.jpf.shell.basicshell.BasicShell
shell.panels+=,errorTrace
shell.panels.errorTrace=ErrorTracePanel
```

jpf-visual



Trans.	main 0	Thread-1 1	Thread-2 2	Thread-3 3	Thread-4 4	Thread-5 5
	public class Philosophers {					

10-11	⊞	package concurrency;
		...
		this.wait();				

12-15	⊞		package concurrency;
			...
			this.wait();			

16-19	⊞			package concurrency;
				...
				this.wait();		

20-23	⊞				package concurrency;
					...
					this.wait();	

24	⊞					package conc ... this.wait();

Bug

All five philosophers pick up their left fork first and then all wait for their right fork.