Programming with Concurrent Threads (Part 2)

Robert M. Dondero, Ph.D.
Princeton University

Objectives

- We will cover:
  - Preventing race conditions
  - Thread conditions

Agenda

- **Preventing race conditions**
- Thread conditions

Preventing Race Conditions

- **Recall from last lecture**
  - race.py
- **Problem**
  - ***Race condition***
  - Use of shared resource (`BankAcct` object) by multiple threads causes unpredictable behavior

Preventing Race Conditions

- **Observation**:
  - While a thread is executing `deposit()` or `withdraw()` on a particular `bankAcct` object…
  - No other thread should be able to execute `deposit()` or `withdraw()` on that `bankAcct` object

Preventing Race Conditions

- **Solution**: ***Locking***
  - Each object has a lock
  - All threads that will use object X agree to a "pact": must acquire lock on X before using X
  - Current thread acquires lock on X
  - Other threads cannot acquire lock on X until current thread releases lock on X
  - (Adds lots of overhead)

Preventing Race Conditions

- **Approach 1**: Locking in **user** of shared object

Preventing Race Conditions: Example

- See **lockinuser.py**
  - The job:
    - Same as race.py

8

Locking: User-Level

Code notes:

Study code statically

`self._lock = RLock()`
In `BankAcct` constructor
Creates a lock for the newly instantiated object

`self._lock.acquire()`
In each thread
"Give me the lock on the `bankAcct` object"
Request granted => current thread proceeds
Request denied => current thread blocks

`self._lock.release()`
In each thread
"Release the lock on the `bankAcct` object"

Study code dynamically

Preventing Race Conditions: Example

- See **lockinuser.py** (cont.)

```
$ python lockinuser.py
1
2
3
4
5
6
7
8
9
10
8
6
4
2
0
Final balance: 0
$
```

```
$ python lockinuser.py
1
2
3
4
2
0
-2
-4
-6
-5
-4
-3
-2
-1
0
Final balance: 0
$
```

It works!

Preventing Race Conditions: Example

- See **lockinuserw.py**
  - The job:
    - Same as lockinuser.py
  - The difference:
    - Uses `with` statement

Preventing Race Conditions

- **Approach 2**: Locking in shared resource/object itself

Preventing Race Conditions: Example

- See **lockinresource.py**
  - The job:
    - Same as race.py

12

Locking: Resource-Level

Code notes

Study code statically
Same, except...
Locking is performed by `BankAcct` object rather than by thread objects
Study code dynamically

Note:
`get_balance()` should be protected too
At the bytecode level, it may not be atomic

Preventing Race Conditions: Example

- See **lockinresource.py** (cont.)

```
$ python lockinresource.py
1
2
3
4
5
6
7
8
9
10
8
6
4
2
0
Final balance: 0
$
```

```
$ python lockinresource.py
1
2
3
1
-1
-3
-5
-7
-6
-5
-4
-3
-2
-1
0
Final balance: 0
$
```

It works!

Preventing Race Conditions: Example

- See **lockinresource.py** (cont.)
  - Note:
    - At the machine language level, `get_balance()` method is not necessarily atomic
    - `get_balance()` method should be protected by "locking"

Preventing Race Conditions: Example

- See **lockinresource.py** (cont.)
  - Note:
    - At the machine language level, direct access of `_balance` field is not necessarily atomic
    - `_balance` field should be protected by locking
      - But cannot be in Python
    - `_balance` field should be private
      - But cannot be in Python

Preventing Race Conditions: Example

- See **lockinresourcew.py**
  - The job:
    - Same as lockinresource.py
  - The difference:
    - Uses `with` statement

Preventing Race Conditions

- Which locking approach is better?
  - **User**-level locking: sometimes **faster**
  - **Resource**-level locking: **safer**

17

Locking: Strategies

Which locking approach is better?

User-level locking: **faster**

Avoids locking when unnecessary

Only choice if you don't control the shared resource code

Resource-level locking: **safer**

Forces all threads to agree to the "pact"

Preventing Race Conditions

- ***Thread-safety***
  - Oversimplification…
  - An object is **thread-safe** if all of its methods are “locked” & all of its fields are private

Preventing Race Conditions

- **Java**
  - Methods can be locked (`synchronized`)
  - Fields **can** be private
  - Objects can be thread-safe
- **Python**
  - Methods can be locked
  - Fields **cannot** be private
  - Any object that has fields cannot be thread-safe
- **Commentary**: Python for large apps???!!!

19

Locking: Thread Safety

[see slide]

I worry about using Python for large apps:

Weak type checking, especially for parameters
Lack of private fields and methods
Impossibility of thread safety

Agenda

- Preventing race conditions
- **Thread conditions**

Thread Conditions

- Problem in bank example:

```
$ python lockinresource.py
1
2
3
4
5
3
1
-1
-3
-5
-4
-3
-2
-1
0
Final balance: 0
$
```

Unrealistic for bank accounts to have negative balances

Thread Conditions

- Solution:
  - Before withdrawing, withdraw thread should **wait** for the bank account balance to be sufficiently large
  - After depositing, deposit thread should **notify** waiting threads that they can try again

Thread Conditions

- Solution in general:
  - **Consumer** thread **waits** for some condition on shared object to become true
  - **Producer** thread changes condition, and **notifies** waiting threads that they can try again
- Implementation: ***Thread conditions***

Thread Conditions: Example

- See **conditions.py**
  - The job:
    - Same as lockinresource.py, except...
    - Disallows negative bank balances
      - `withdraw()` calls `condition.wait()`
        - » `withdraw()` must wait until balance is sufficiently large
      - `deposit()` calls `condition.notifyAll()`
        - » `deposit()` informs waiting threads to try again

24

Conditions: Example

[see slide]

Code notes:

Study code statically

Condition wraps around lock

`condition.wait()`

Releases the lock

Moves current thread from **runnable** state to **waiting** state

Upon return, reacquires lock

`condition.notifyAll()`

Moves all threads waiting for lock on this object from **waiting** state to **runnable** state

Study code dynamically

`withdraw()` calls `condition.wait()`

`withdraw()` must wait until balance is sufficiently large

`deposit()` calls `condition.notifyAll()`

`deposit()` informs waiting threads to try again

Thread Conditions: Example

- See **conditions.py** (cont.)

```
$ python conditions.py
1
2
3
4
5
6
7
8
9
10
8
6
4
2
0
Final balance: 0
$
```

```
$ python conditions.py
1
2
3
4
5
3
1
2
3
4
5
6
4
2
0
Final balance: 0
$
```

25

Conditions: Example

[see slide]

Note that balances are never negative

Thread Conditions: Example

- See **conditionsw.py**
  - The job:
    - Same as conditions.py
  - The difference:
    - Uses `with` statement

Thread Conditions: Pattern

Thread conditions pattern:

```
consumer thread
   while (! condition)
      wait();
   // Do what should be done when
   // condition is true.

producer thread
   // Change condition.
   notifyAll();
```

27

Conditions: Pattern

`condition.wait()`

Releases the lock

Moves current thread from **runnable** state to **waiting** state

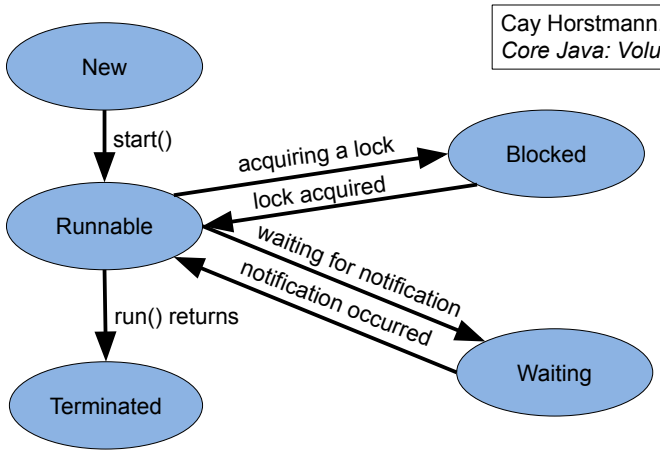
Upon return, reacquires lock

`condition.notifyAll()`

Moves all threads waiting for lock on this object from **waiting** state to **runnable** state

Aside: Thread States

Cay Horstmann.
Core Java: Volume 1



At any time thread scheduler gives CPU to one Runnable thread

Summary

- We have covered:
  - Preventing race conditions
  - Thread conditions