Concurrent Programming (Part 3)

Robert M. Dondero, Ph.D.
Princeton University

Objectives

- We will cover:
  - Thread safety
  - Thread conditions
  - Inter-process communication
  - Inter-thread communication

Agenda

- **Thread safety**
- Thread conditions
- Inter-process communication
- Inter-thread communication

Thread Safety

- Recall **lockinresource.py**
  - A context switch can occur between any 2 **machine lang** instructions
  - Implications:
    - `get_balance()` should be protected by locking
    - _balance should be private
      - But cannot be

Thread Safety

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

Thread Safety

- **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

Agenda

- Thread safety
- **Thread conditions**
- Inter-process communication
- Inter-thread communication

Thread Conditions

- **Observation** (concerning lockinresource.py):
  - 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

- Observation (in general):
  - Sometimes a **consumer** thread must **wait** for a condition on a shared object to become true
  - Sometimes a **producer** thread must change the condition, and **notify** waiting threads that they can try again

- Implementation: ***Thread conditions***

Thread Conditions

- See **conditions.py**

```
$ 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
$
```

Thread Conditions

- See **conditions.py** (cont.)
  - **`condition.notify_all()`**
    - Moves all threads waiting on this object from waiting state to runnable state
  - **`condition.wait()`**
    - Releases the lock
    - Moves current thread from runnable state to waiting state
    - Upon return, reacquires lock

Thread Conditions

- See **conditionsw.py**
  - Uses `with` statement

Thread Conditions

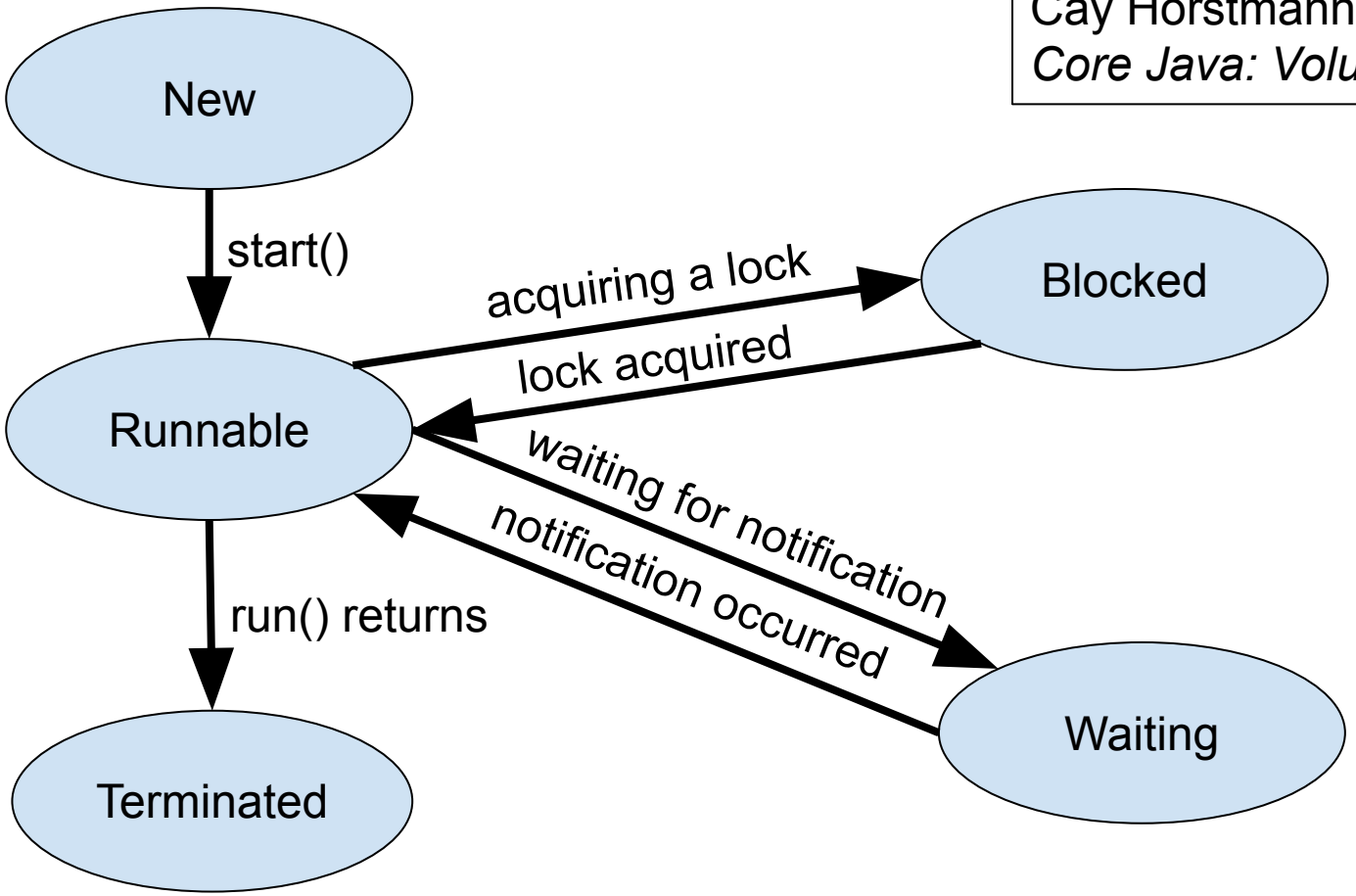
Thread conditions pattern:

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

producer thread
   // Change objectstate.
   condition.notify_all();
```

Aside: Thread States

Cay Horstmann.
Core Java: Volume 1



At any time OS gives processor(s) to Runnable thread(s)

Agenda

- Thread safety
- Thread conditions
- **Inter-process communication**
- Inter-thread communication

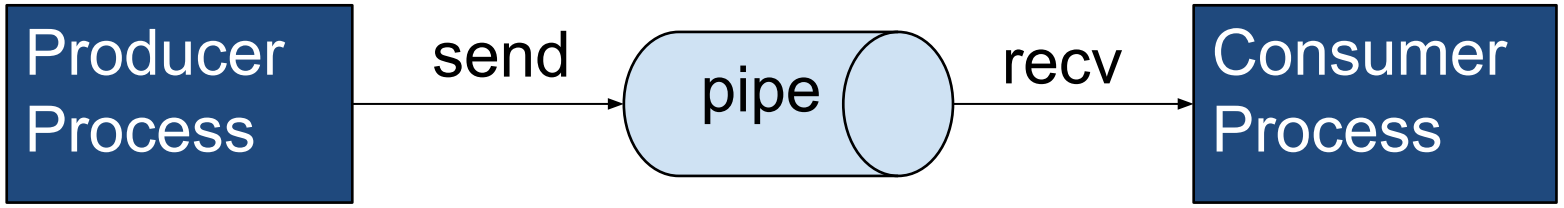
Inter-Process Communication

- Processes **do not** share objects, so…
- Inter-process comm **cannot** be accomplished via a shared object…

Inter-Process Communication

- ***Pipe***
  - An operating system (not a Python) feature

Inter-Process Communication



Pipe has a finite size (determined by OS)
Producer process "sends" to pipe
    `send()` blocks while pipe is **full**
Consumer process "receives" from pipe
    `recv()` blocks while `pipe` is **empty**

Inter-Process Communication

- See **prodconprocesses.py**

```
$ python prodconprocesses.py
...
Produced: 95
Consumed: 95
Produced: 96
Consumed: 96
Produced: 97
Consumed: 97
Produced: 98
Consumed: 98
Produced: 99
Consumed: 99
Finished
$
```

Agenda

- Thread safety
- Thread conditions
- Inter-process communication
- **Inter-thread communication**

Inter-Thread Communication

- Threads share objects, so…
- Inter-thread comm can be accomplished via a shared object…

Inter-Thread Communication

- Python `Queue` class
  - Semi-thread-safe
  - Designed for inter-thread comm

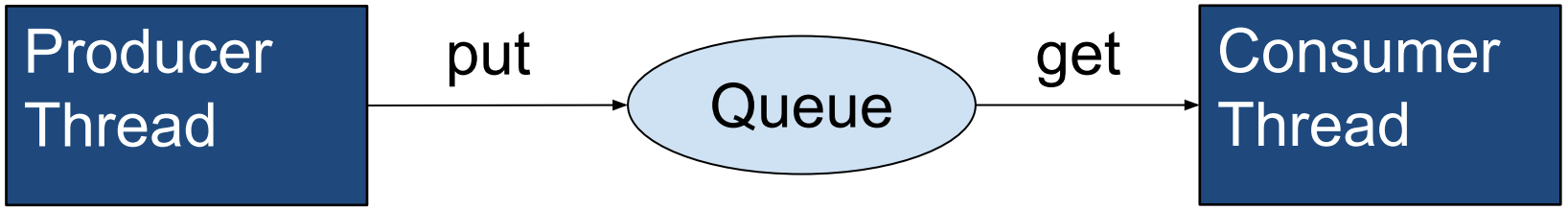
Inter-Thread Communication

- Use case 1:

```
…
q = queue.Queue()
…
q.put(item)
…
try:
    item = q.get(block=False)
except queue.Empty:
    # The queue is empty.
```

`Queue` object can contain an unlimited number of items

Inter-Thread Communication



Producer thread “puts” data to `Queue` object
Consumer thread “gets” data from `Queue` object
`get()` throws exception if `Queue` object is empty

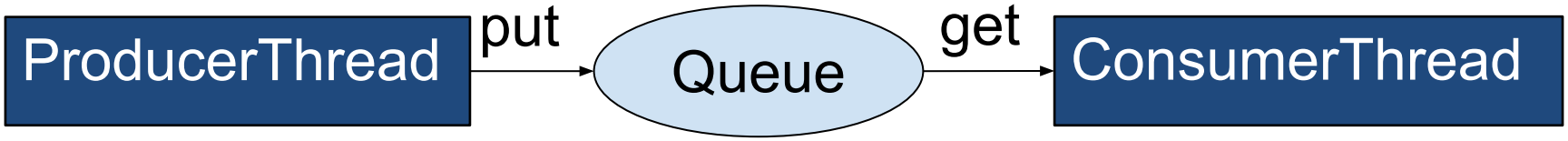
Inter-Thread Communication

- Use case 2:

```
…
q = queue.Queue(n)
…
q.put(item)
# Waits if q is full.
# Notifies when finished.
#    Some other thread might be
#    waiting for q to have some items.
…
item = q.get()
# Waits if q is empty.
# Notifies when finished.
#    Some other thread might be
#    waiting for q to have some room.
…
```

`Queue` object can contain up to `n` items

Inter-Thread Communication



`Queue` object has a finite size (determined by Python pgm)

Producer thread "puts" to `Queue` object

- `put()` **waits** while `Queue` object is **full**
- `put()` **notifies** when finished

Consumer thread "gets" from `Queue` object

- `get()` **waits** while `Queue` object is **empty**
- `get()` method **notifies** when finished

Inter-Thread Communication

- See **prodconthreads.py**

```
$ python prodconthreads.py
...
Produced: 97
Consumed: 93
Produced: 98
Consumed: 94
Produced: 99
Consumed: 95
Consumed: 96
Consumed: 97
Consumed: 98
Consumed: 99
Finished
$
```

Inter-Thread Communication

- See **prodconthreads.py** (cont.)
  - Observation: It's a good thing that `Queue` objects are semi-thread-safe

Summary

- We have covered:
  - Thread safety
  - Thread conditions
  - Inter-process communication
  - Inter-thread communication
- See also:
  - **Appendix 1**: Threads in Java
  - **Appendix 2**: Threads in C

Appendix 1:
Threads in Java

Threads in Java

- See **Conditions.java**

```
$ javac Conditions.java
$ java Conditions
1
2
3
4
5
6
7
8
9
10
8
6
4
2
0
Final balance: 0
$
```

Appendix 2: Threads in C

Threads in C

- See **conditions.c**

```
$ gcc -pthread conditions.c -o conditions
$ ./conditions
1
2
3
4
5
6
7
8
9
10
8
6
4
2
0
Final balance: 0
$
```