Concurrent Programming (Part 3)

Robert M. Dondero, Ph.D.

Princeton University

Objectives

- We will cover:
  - Thread conditions
  - Environment variables

Agenda

- **Thread conditions**
- Environment variables

Thread Conditions

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

Recall locking.py

Not realistic!

Thread Conditions

- **Observation** (concerning locking.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

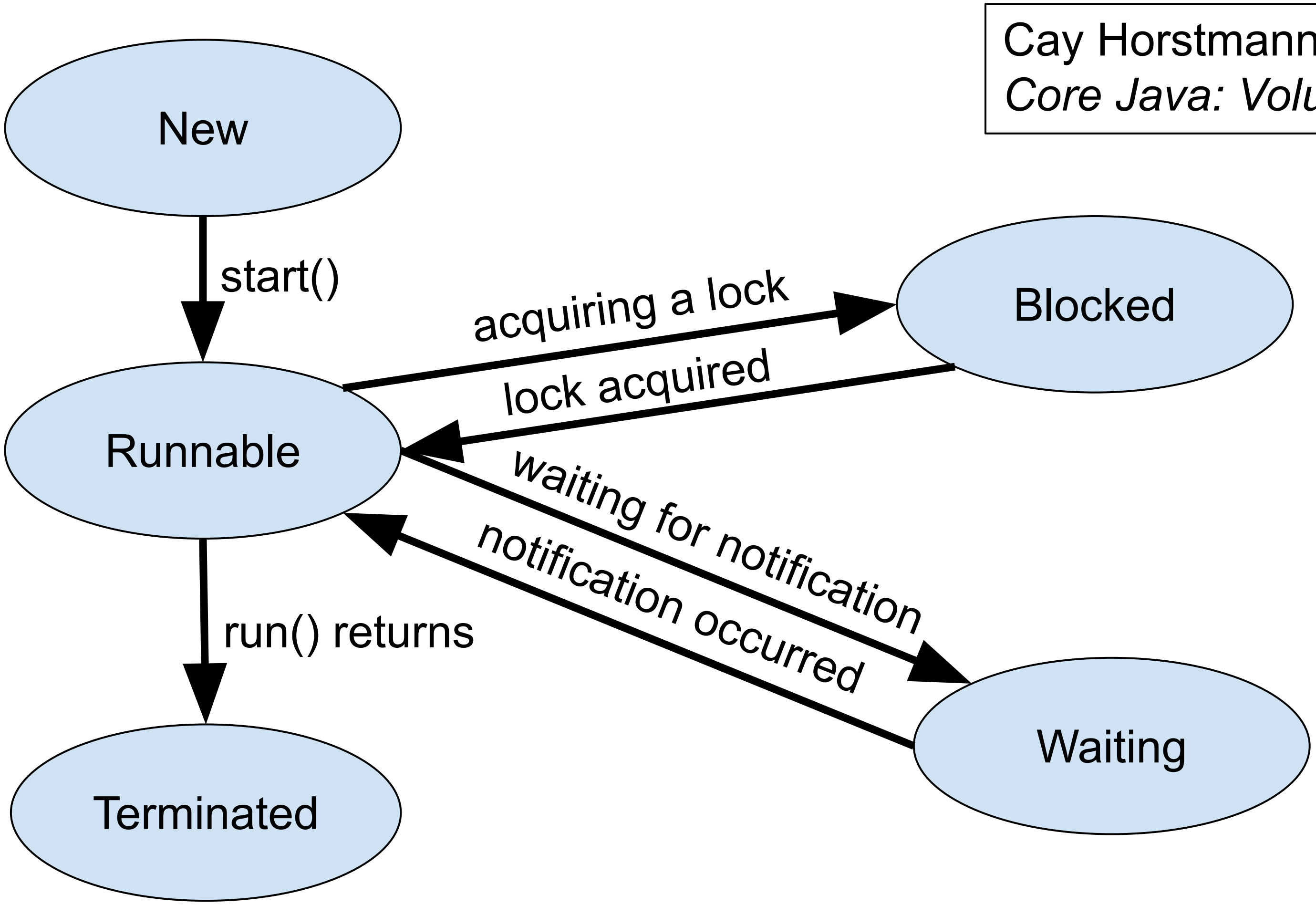
Thread conditions pattern:

```
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 conditions
- **Environment variables**

Environment Vars

- ***Environment variables***
  - Each process has a set of environment variables
    - `PATH=...`
    - `SHELL=...`
    - `QUERY_STRING=`...
    - ...
  - Each child process inherits the environment variables of its parent process

Environment Vars

In the Bash shell (on Linux or Mac):

```
$ export XXX=yyy
$ printenv
…
XXX=yyy
…
$ printenv XXX
yyy
$ echo $XXX
yyy
$ unset XXX
$ printenv XXX
$
```

Environment Vars

In a Command Prompt window (on MS Windows):

```
C:\>set XXX=yyy
C:\>echo %XXX%
yyy
C:\>set XXX=
C:\>echo %XXX%
C:\>
```

Environment Vars

In Python:

```
$ python
>>> import os
>>> os.environ['XXX'] = 'yyy'
>>> os.environ['XXX']
'yyy'
>>> os.environ['ZZZ']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<frozen os>", line 714, in
__getitem__ KeyError: 'ZZZ'
>>> os.environ.get('XXX')
'yyy'
>>> os.environ.get('ZZZ')
>>> os.environ.get('ZZZ', 'somedefault')
'somedefault'
>>> quit()
$
```

Environment Vars

- **Question**:
  - How can a Python process accept data from its user?
- **Answers**:
  - By reading it (from stdin, a file, a socket, or a pipe)
  - Through a command-line argument
  - Through an **environment variable**

Environment Vars

- See **envvar1.py**

```
$ export GREETING=hello
$ python envvar1.py
hello
$ unset GREETING
$ python envvar1.py
hi
$
```

Environment Vars

- The Python ***dotenv*** module
  - Python-specific mechanism for setting/getting env vars
  - To install:

```
$ python -m pip install python-dotenv
```

Environment Vars

- The Python ***dotenv*** module (cont.)
  - To use in Python code (step 1)

.env file:

```
SOMEVAR=somevalue
…
```

Environment Vars

- The Python ***dotenv*** module (cont.)
  - To use in Python code (step 2)

    .py file:

```
import dotenv
…
dotenv.load_dotenv()
SOME_VAR = os.environ.get('SOMEVAR', default)
…
```

(1) Looks for *SOMEVAR* as env var; if not found…

(2) Looks for *SOMEVAR* in .env file, if not found…

(3) Uses *default*

Environment Vars

- See **envvar2.py**

Created first:

```
$ cat .env
GREETING=hello
$
```

Then:

```
$ export GREETING=bonjour
$ python envvar2.py
bonjour
$ unset GREETING
$ python envvar2.py
hello
$ rm .env
$ python envvar2.py
hi
$
```

Lecture Summary

- In this lecture we covered:
  - Thread conditions
  - Environment variables
- See also:
  - **Appendix 1:** Inter-Process Communication
  - **Appendix 2:** Inter-Thread Communication
  - **Appendix 3**: Thread Conditions in Java

Appendix 1:
Inter-Process Communication

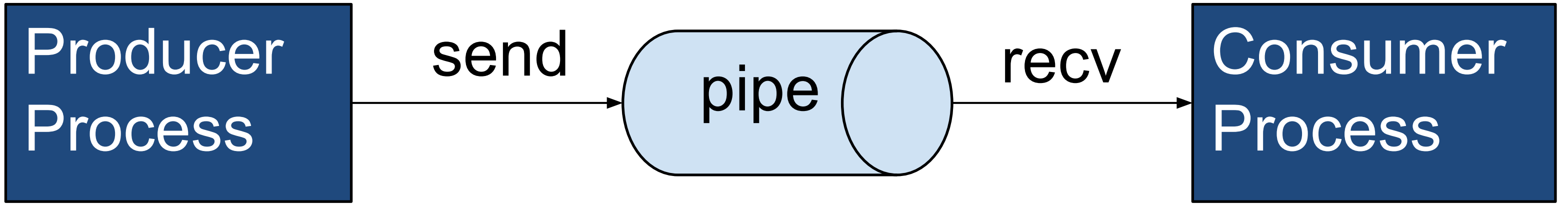
Inter-Process Communication

- 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 calls `send()`

    Blocks when pipe is full

Consumer process calls `recv()`

    Blocks when 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
$
```

Appendix 2:
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

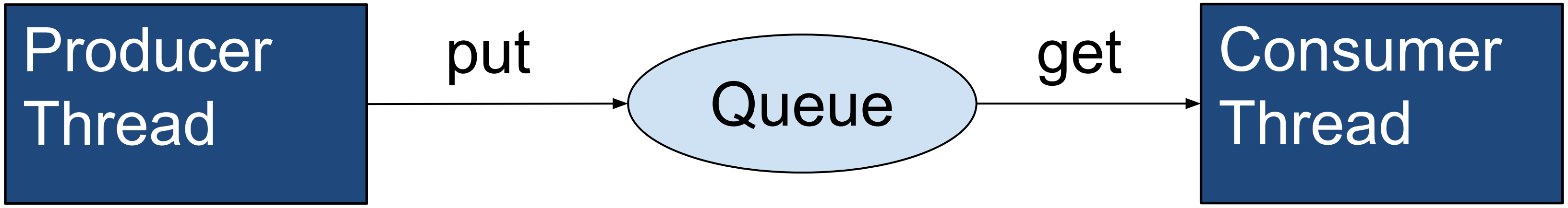
Inter-Thread Communication

- 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 calls `put()`

Consumer thread calls `get()`

- `get()` throws exception if `Queue` object is empty

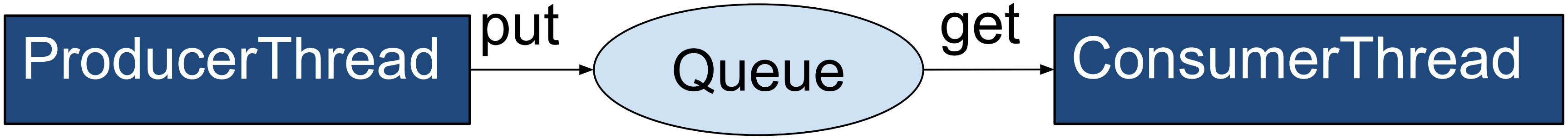
Inter-Thread Communication

- Use case 2:

```
…
q = queue.Queue(n)
…
q.put(item)
…
item = q.get()
…
```

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

Inter-Thread Communication



`Queue` object has a finite size

- Specified by Python pgm

Producer thread calls `put()`

- Waits when `Queue` object is full
- Notifies when finished

Consumer thread calls `get()`

- Waits when `Queue` object is empty
- 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

Appendix 3:
Thread Conditions in Java

Thread Conditions 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
$
```