[544] Locks

Tyler Caraza-Harter

Learning Objectives

- identify critical sections in code
- protect critical sections with locks
- write code that avoids concurrency bugs, such as race conditions and deadlocks

Outline

Critical Sections and Locks

Worksheet and Demos

Advanced Topics

- Global Interpreter Lock
- Instruction Reordering and Caching

Critical Sections

```
1     # in dollars
2     bank_accounts = {"x": 25, "y": 100, "z": 200}
3
4     def transfer_euros(src, dst, euros):
5         dollars = euros_to_dollars(euros)
6         success = False
7
8         if bank_accounts[src] >= dollars:
9             bank_accounts[src] -= dollars
10            bank_accounts[dst] += dollars
11            success = True
12
13        print("transferred" if success else "denied")
```

If two threads are calling transfer_euros concurrently, *during which lines would a context switch between those two be problematic?*

A section of code we don't want interrupted by certain other code is a "critical section"

Critical Sections

```
1     # in dollars
2     bank_accounts = {"x": 25, "y": 100, "z": 200}
3
4     def transfer_euros(src, dst, euros):
5         dollars = euros_to_dollars(euros)
6         success = False
7
8         if bank_accounts[src] >= dollars:
9             bank_accounts[src] -= dollars
10            bank_accounts[dst] += dollars
11            success = True
12
13        print("transferred" if success else "denied")
```

critical section

Goals:

Atomicity: want withdrawal+deposit seen together (never seen half done).

Consistency: rules (called "invariants") like "no account goes negative" must be enforced

Locks

```
1     # in dollars
2     bank_accounts = {"x": 25, "y": 100, "z": 200}
3     lock = threading.Lock() # protects bank_accounts
4
5     def transfer_euros(src, dst, euros):
6         lock.acquire()
7         dollars = euros_to_dollars(euros)
8         success = False
9         if bank_accounts[src] >= dollars:
10            bank_accounts[src] -= dollars
11            bank_accounts[dst] += dollars
12            success = True
13        print("transferred" if success else "denied")
14        lock.release()
```

Lock Rules

- between acquire and release, a lock is held by the thread that acquired it
- **a lock may only be held by one thread at a time**
- if T2 wants to acquire a lock held by T1, T2 blocks until T1 releases it

Locks

```
1     # in dollars
2     bank_accounts = {"x": 25, "y": 100, "z": 200}
3     lock = threading.Lock() # protects bank_accounts
4
5     def transfer_euros(src, dst, euros):
6         dollars = euros_to_dollars(euros)
7         success = False
8         lock.acquire()
9         if bank_accounts[src] >= dollars:
10            bank_accounts[src] -= dollars
11            bank_accounts[dst] += dollars
12            success = True
13        lock.release()
14        print("transferred" if success else "denied")
```

Tradeoffs

- different patterns may accomplish the same goal
- some are more efficient; some are simpler

Locks

```
1     # in dollars
2     bank_accounts = {"x": 25, "y": 100, "z": 200}
3     lock = threading.Lock() # protects bank_accounts
4
5     def transfer_euros(src, dst, euros):
6         dollars = euros_to_dollars(euros)
7         success = False
8         if bank_accounts[src] >= dollars:
9             lock.acquire()
10            bank_accounts[src] -= dollars
11            bank_accounts[dst] += dollars
12            lock.release()
13            success = True
14        print("transferred" if success else "denied")
```

Tradeoffs

- different patterns may accomplish the same goal
- some are more efficient; some are simpler
- be careful! (this incorrect version provides atomicity but not consistency)

Worksheet and Demos...