Locks and Synchronization

1. Threads for Simulation
   - simulating arrival of customers
   - the code for the simulation with threads

2. Two Players taking Turns
   - implemented as two concurrently running threads
   - code for the script `taketurns.py`

3. Restaurant with one Waiter
   - multiple threads require synchronization
   - code for restaurant simulation

MCS 275 Lecture 31
Programming Tools and File Management
Jan Verschelde, 29 March 2017
Locks and Synchronization

1. Threads for Simulation
   - simulating arrival of customers
   - the code for the simulation with threads

2. Two Players taking Turns
   - implemented as two concurrently running threads
   - code for the script `taketurns.py`

3. Restaurant with one Waiter
   - multiple threads require synchronization
   - code for restaurant simulation
Threads for Simulation

simulating arrival of customers

Customers arriving at a fastfood restaurant:
1. wait for their order to be ready;
2. spend some time eating and then leave.

We use *threads* to implement the customers. Inheriting from the class `Thread`, the data object attributes are:
1. the name of the customer;
2. waiting and eating time.

The names are entered when threads are born. Waiting and eating times are generated via `random.randint()` in the `__init__`. 
$ python fastfood.py
give #customers : 3
  name of customer 0 : A
  name of customer 1 : B
  name of customer 2 : C
starting the simulation
A is waiting for 1 time units
B is waiting for 1 time units
C is waiting for 4 time units
simulation has started
A waited 1 time units
B waited 1 time units
B ate for 2 time units
C waited 4 time units
A ate for 6 time units
C ate for 5 time units
$
from threading import Thread

class Customer(Thread):
    
    Customer orders food and eats.
    
    def __init__(self, t):
        
        Initializes customer with name t, generates waiting and eating time.
        
    def run(self):
        
        Customer waits for order and eats. writes three messages.
        
def main():
    
    Defines and starts threads.
1. Threads for Simulation
   - simulating arrival of customers
   - the code for the simulation with threads

2. Two Players taking Turns
   - implemented as two concurrently running threads
   - code for the script `taketurns.py`

3. Restaurant with one Waiter
   - multiple threads require synchronization
   - code for restaurant simulation
def main():
    """
    Defines and starts threads.
    """
    nbr = int(input('give #customers : '))
    threads = []
    for i in range(nbr):
        prompt = 'name of customer %d :' % i
        name = input(' ' + prompt)
        threads.append(Customer(name))
    print("starting the simulation")
    for customer in threads:
        customer.start()
    print("simulation has started")
from random import randint

class Customer(Thread):
    
    Customer orders food and eats.
    
    def __init__(self, t):
        
        Initializes customer with name t, generates waiting and eating time.
        
        Thread.__init__(self, name=t)
        self.wait = randint(1, 6)
        self.eat = randint(1, 6)
overriding the run in Customer (Thread)
The data attributes are wait and eat.

from time import sleep

def run(self):
    """
    Customer waits for order and eats, writes three messages.
    """

    name = self.getName()
    print(name + ' is waiting for %d time units' \
          % self.wait)
sleep(self.wait)
    print(name + ' waited %d time units' % self.wait)
sleep(self.eat)
    print(name + ' ate for %d time units' % self.eat)
Evaluation: Benefits and Limitations of the first script `fastfood.py`

Benefits of using threads:

1. **object oriented design separates stages**
   → three stages: born, started, active

2. **customers seen as independently acting**
   → behavior is defined locally

At least two limitations:

1. **role of server(s) not represented**
   → scheduling of order taking ignored

2. **hard to get statistics**
   → all data are gone with thread ends
Locks and Synchronization

1. Threads for Simulation
   - simulating arrival of customers
   - the code for the simulation with threads

2. Two Players taking Turns
   - implemented as two concurrently running threads
   - code for the script `taketurns.py`

3. Restaurant with one Waiter
   - multiple threads require synchronization
   - code for restaurant simulation
Two Players taking Turns
as concurrently running threads

Consider two players, taking turns:
  first player thinks, makes a move
  second player thinks, makes a move
  first player thinks, makes a move
  second player thinks, makes a move, etc...

Instead of one main program regulating turns, we want:

1. two threads independently running,
2. checking and changing a shared variable.

→ main program plays no role anymore, the two players are in control of the game.
running the script `taketurns.py`

```
$ python taketurns.py
player 0 born
player 1 born
starting the game
game has started
0 checks value 3
1 checks value 3
1 thinks 7 time units
0 checks value 3
1 changes value to 2
0 checks value 2
0 thinks 2 time units
0 changes value to 1
1 checks value 1
1 thinks 8 time units
0 checks value 1
1 changes value to 0
0 checks value 0
1 checks value 0
```
the algorithm for threads in `taketurns.py`

After making move, the players enter a *busy-waiting loop*:

```python
while True:
    time.sleep(5)
    if value % 2 == 'player name':
        break
```

The sleep time of 5 units is necessary to prevent one thread from absorbing all CPU cycles.

About the *shared* `value`:

- It determines the total number of moves the two players make.
- After each move, `value` is decreased by one.
Locks and Synchronization

1. Threads for Simulation
   - simulating arrival of customers
   - the code for the simulation with threads

2. Two Players taking Turns
   - implemented as two concurrently running threads
   - code for the script `taketurns.py`

3. Restaurant with one Waiter
   - multiple threads require synchronization
   - code for restaurant simulation
def main():
    ""
    Defines two players '0' and '1'
    and starts the game.
    ""

    shared = []
    one = Player('0', shared)
    two = Player('1', shared)
    print('starting the game')
    shared.append(3)
    one.start()
    two.start()
    print('game has started')
Shared Variable
how threads communicate with each other

Relations between references and objects:

<table>
<thead>
<tr>
<th>reference</th>
<th>object</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared</td>
<td>[ shared[0] ]</td>
</tr>
<tr>
<td>shared[0]</td>
<td>value</td>
</tr>
</tbody>
</table>

With `shared = []`, we pass only the reference when we create `p1 = Player('0', shared)`. The value in `shared` (i.e.: `shared[0]`) is set later. We pass the reference, not the values to the data object attributes of the threads.
from threading import Thread

class Player(Thread):
    """
    Player waits turn and makes move.
    """
    def __init__(self, t, v):
        """
        Initializes player with name t, and stores the shared value v.
        """
        Thread.__init__(self, name=t)
        print('player ' + t + ' born')
        self.sv = v
def run(self):
    ""
    Player checks value every 5 time units.
    If parity matches, value is decreased.
    The game is over if value == 0.
    ""

    p = self.getName()
    n = int(p)
    while True:
        while True:
            sleep(5)
            v = self.sv[0]
            print(p + ' checks value %d ' % v)
            if v <= 0 or v % 2 == n:
                break

→ inner loop is busy waiting
if v <= 0:
    break # game over
nbr = randint(1, 10)
s = p + ' thinks %d' % nbr
print(s + ' time units')
sleep(nbr)
v = self.sv.pop(0) - 1
print(p + ' changes value to %d' % v)
self.sv.append(v)

Changing the shared variable in

    v = self.sv.pop(0) - 1
    self.sv.append(v)

is a critical section of the program.
Thread Safety

Code is **thread safe** if its simultaneous execution by multiple threads is correct. 

**Only one thread can change shared data.**

Once only a concern for the operating system programmer ...

Some illustrations of thread safety concerns:

1. shared bank account
2. do not block intersection
3. access same mailbox from multiple computers
4. threads access same address space in memory
The Dining Philosophers Problem
an example problem to illustrate synchronization

The problem setup, rules of the game:

1. Five philosophers are seated at a round table.
2. Each philosopher sits in front of a plate of food.
3. Between each plate is exactly one chop stick.
4. A philosopher thinks, eats, thinks, eats, ...
5. To start eating, every philosopher
   a. first picks up the left chop stick, and
   b. then picks up the right chop stick.

Why is there a problem?

The problem of the starving philosophers:

- every philosopher picks up the left chop stick, at the same time,
- there is no right chop stick left, every philosopher waits, ...
using locks for a critical section

Recall *critical section* in `taketurns.py`:

```python
v = self.sv.pop(0) - 1
self.sv.append(v)
```

To ensure that *only one* thread executes code in critical section, we use a *lock*:

```python
import _thread
lock = _thread.allocate_lock()

lock.acquire()
# code in critical section
lock.release()
```
the Python interpreter lock

In CPython, the \textit{global interpreter lock}, or GIL, prevents multiple native threads from executing Python bytecodes in parallel.

Why is the GIL necessary? \\
→ The memory management in CPython is not thread safe.

Consequence: degrading performance on multicore processors.

For parallel code in Python, use \\
• the multiprocessing module (if sufficient memory available); or \\
• depend on multithreaded libraries that run outside the GIL.
Locks and Synchronization

1. Threads for Simulation
   - simulating arrival of customers
   - the code for the simulation with threads

2. Two Players taking Turns
   - implemented as two concurrently running threads
   - code for the script `taketurns.py`

3. Restaurant with one Waiter
   - multiple threads require synchronization
   - code for restaurant simulation
Restaurant with one Waiter

various stages and states

Customers pass through various stages:

→ 0 enters restaurant
  1 waits for table ←
  2 orders food
  3 waits for food ←
→ 4 eats the food
  5 waits for bill ←
→ 6 pays and leaves

Game of taking turns:

→ waiter waits for action of customer
← customer waits for action of waiter
Simulation Algorithm
object oriented multithreading

Object data attribute for customer is its state.
The waiter knows the state of every customer.

Moving from one state to the next:
1. even states: customer does it
2. odd states: action of waiter needed

One class Customer and one class Waiter.
→ all threads run autonomously
running the simulation restaurant.py

give number of customers : 3
simulation starts...
1 waits for table
2 waits for table
2 orders food
0 waits for table
1 orders food
2 waits for food
0 orders food
1 waits for food
0 waits for food
1 eats food
1 waits for bill
2 eats food
0 eats food
2 waits for bill
1 pays and leaves
0 waits for bill
2 pays and leaves
0 pays and leaves
Locks and Synchronization

1. Threads for Simulation
   - simulating arrival of customers
   - the code for the simulation with threads

2. Two Players taking Turns
   - implemented as two concurrently running threads
   - code for the script taketurns.py

3. Restaurant with one Waiter
   - multiple threads require synchronization
   - code for restaurant simulation
def main():
    """
    Simulation of restaurant with one waiter.
    """

    shared = []
server = Waiter('w', shared)
nbr = int(input('give number of customers : '))
eaters = []
for i in range(nbr):
    eaters.append(Customer(str(i), shared))
for eater in eaters:
    shared.append(0)
server.start()
for eater in eaters:
    eater.start()
print("simulation starts...")
class Customer(Thread):
    
    Customer enters, waits for table, orders food, waits for food, eats, waits for bill and then leaves.
    
def __init__(self, t, S):
    
    Initializes customer with name t, sets initial state to zero, and stores the shared state list S.
    
    Thread.__init__(self, name=t)
    self.state = 0
    self.shs = S
def __str__(self):
    """
    Returns the string representation of self.state.
    """
    result = self.getName()
    if self.state == 0:
        result += ' enters restaurant'
    elif self.state == 1:
        result += ' waits for table'
    elif self.state == 2:
        result += ' orders food'
    elif self.state == 3:
        result += ' waits for food'
    elif self.state == 4:
        result += ' eats food'
    elif self.state == 5:
        result += ' waits for bill'
    else:
        result += ' pays and leaves'
    return result
what a customer does
Depending on whether the state number is even or odd,
- the customer moves to the next state, or
- the customer waits for the waiter.

```python
def run(self):
    """
    Customer passes through stages.
    Moves at even numbered states
    and waits at odd numbered states.
    """
    while self.state < 6:
        if self.state % 2 == 0:
            self.move()
        else:
            self.wait()
    print(self)
```
def move(self):
    """
    In even numbered states, 
    the customer moves up after a delay.
    """
    idn = self.getName()
    nbr = randint(1, 10)
    sleep(nbr)
    self.state = self.state + 1
    self.shs[int(idn)] = self.state
a busy waiting loop

```python
def wait(self):
    """
    Needs waiter for odd stages.
    """
    idx = int(self.getName())
    while True:
        sleep(5)
        if self.state < self.shs[idx]:
            break
    self.state = self.state + 1
```

Programming Tools (MCS 275)
code for the class `Waiter`:

class Waiter(Thread):
    """
    Waiter checks the state list and advances odd states to next level.
    """
    def __init__(self, t, S):
        """
        Initializes waiter with name t, and stores the shared state list S.
        """
        Thread.__init__(self, name=t)
        self.shs = S
def run(self):
    
    Advances odd states of customers, while otherwise busy waiting.

    while True:
        sleep(5)
        done = True
        for i in range(len(self.shs)):
            if self.shs[i] % 2 == 1:
                sleep(randint(1, 3))
                self.shs[i] = self.shs[i] + 1
            if self.shs[i] < 6:
                done = False
        if done:
            break
Multithreaded programming models customers and server as autonomous agents. What happens is event driven, with the events triggered by individual actors. There is no controlling `main()`.

Many extensions needed:

1. customized distributions instead of `random.randint()`;
2. better data encapsulation of shared state list:
   1. separate class to control access,
   2. customer can only access its own state,
   3. for thread safety locks are needed;
3. taking turns is too rigid, allow for interrupts, e.g.: customer asking for refill or more drinks.
Assignments:

1. Extend `taketurns.py`: the shared value is the data object attribute of a class, imported by both players.

2. Use a class to represented the shared state list in `restaurant.py`.

3. Describe how `restaurant.py` should be modified to model a restaurant with multiple waiters.

4. Develop a multithreaded model to simulate how elevators run like in SEO: 12 floors, 4 elevators. Start thinking in the small: one elevator serving 3 floors and design for change.