

## 61A Lecture 33

18th November, 2011

## Last time

Why is parallel computation important?

What is parallel computation?

Some examples in Python

Some problems with parallel computation

## Parallel computation terminology

### Processor

- One of (possibly) many pieces of hardware responsible for executing instructions

### Thread

- One of (possibly) many simultaneous sequences of instructions, being executed in a shared memory environment

### Shared memory

- The environment in which threads are executed, containing variables that are accessible to all the threads.

## Today: dealing with shared memory

### "Vulnerable sections" of a program

- Critical Sections
- Atomicity

### Correctness

- What does "correctness" mean for parallel computation?

### Protecting vulnerable sections

- Locks
- Semaphores
- Conditions

### Deadlock

## Parallel computing example: bank balance

```
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            print('Insufficient funds')
        else:
            balance = balance - amount
            print(balance)
    return withdraw
```

```
w = make_withdraw(10)
balance = 10
```

w(8)

w(7)

```
print('Insufficient funds')
```

## Parallel computing example: bank balance

```
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            print('Insufficient funds')
        else:
            balance = balance - amount
            print(balance)
    return withdraw
```

```
w = make_withdraw(10)
balance = 10
```

w(8)

w(7)

```
read balance: 10
read amount: 8
8 > 10: False
if False
10 - 8: 2
write balance -> 2
print 2
```

```
read balance: 10
read amount: 7
7 > 10: False
if False
10 - 7: 3
write balance -> 3
print 3
```

\$15 withdrawn from a \$10 account?  
With \$3 left? Inconceivable!

### Parallel computing example: bank balance

```
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            print('Insufficient funds')
        else:
            balance = balance - amount
            print(balance)
        return withdraw
    return withdraw
```

```
w = make_withdraw(10)
balance = 10 2 or 3
```

w(8)

w(7)

```
print('Insufficient funds')
```

### Another problem: vector mathematics

$$A = B+C$$

$$V = M \times A$$

### Vector mathematics

$$A = B+C$$

$$V = M \times A$$

$$A = \begin{pmatrix} 2 \\ 5 \end{pmatrix} \quad V = \begin{pmatrix} 12 \\ 12 \end{pmatrix} \quad B = \begin{pmatrix} 2 \\ 0 \end{pmatrix} \quad C = \begin{pmatrix} 0 \\ 5 \end{pmatrix} \quad M = \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix} \quad A = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$$

$$A_1 = B_1 + C_1$$

$$V_1 = M_{1.} \cdot A$$

$$A_2 = B_2 + C_2$$

$$V_2 = M_{2.} \cdot A$$

P1

```
read B1: 2
read C1: 0
calculate 2+0: 2
write 2 -> A1
read M1: (1 2)
read A: (2 0)
calculate (1 2).(2 0): 2
write 2 -> V1
```

P2

```
read B2: 0
read C2: 5
calculate 5+0: 5
write 5 -> A2
read M2: (1 2)
read A: (2 5)
calculate (1 2).(2 5):12
write 12 -> V2
```

$$V = \begin{pmatrix} 2 \\ 12 \end{pmatrix}$$

### Vector mathematics

Step 1

$$A = B+C$$

Step 2

$$V = M \times A$$

Threads must wait for each other.  
Only move on when all have finished previous step.

### Correctness

The outcome should always be equivalent to some serial ordering of individual steps.

**serial ordering:** if the threads were executed individually, from start to finish, one after the other instead of in parallel.

### Problem 1: inconsistent values

Inconsistent values

- A thread reads a value and starts processing
- Another thread changes the value
- The first thread's value is inconsistent and out of date

### Problem 2: unsynchronized threads

Unsynchronized threads

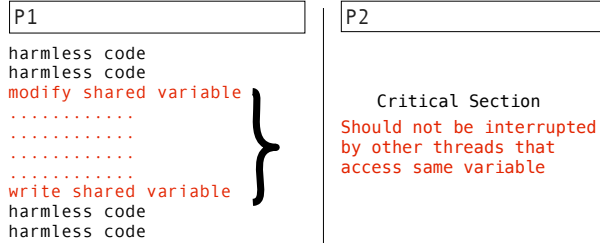
- Operations is a series of steps
- Threads must wait until all have finished previous step

Need ways to make threads wait.

## Problem 1: inconsistent values

Inconsistent values

- A thread reads a value and starts processing
- Another thread changes the value
- The first thread's value is inconsistent and out of date



## Terminology

“Critical section”

- A section of code that should not be interrupted
- Should be executed as if it is a single statement

“Atomic” and “Atomicity”

- Atomic: cannot be broken down into further pieces
- Atomic (when applied to code): cannot be interrupted, like a single hardware instruction.
- Atomicity: a guarantee that the code will not be interrupted.

Critical sections need to have atomicity.

## Protecting shared state with shared state

Use shared state to store signals

Signals can indicate:

- A variable is in use
- A step is complete (or not)
- How many threads are using a resource
- Whether or not a condition is true

Signals:

- Locks or mutexes (mutual exclusions)
- Semaphores
- Conditions

Don't physically protect shared state

Convention and shared rules for signals protect shared state.

- Like traffic signals “protect” an intersection

## Locks

Implemented using real atomic hardware instructions.

Used to signal that a shared resource is in use.

acquire()

- “set” the signal.
- No other threads will be able to acquire()
- They will automatically wait until ...

release()

- “unset” a signal.
- Any one thread that was waiting for acquire() will now succeed

## Using locks: bank balance example

```
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            print('Insufficient funds')
        else:
            balance = balance - amount
            print(balance)
    return withdraw
```

critical section

```
w = make_withdraw(10)
balance = 10
```

w(8)

```
read balance: 10
read amount: 8
8 > 10: False
if False
10 - 8: 2
write balance -> 2
print 2
```

w(7)

```
read balance: 10
read amount: 7
7 > 10: False
if False
10 - 7: 3
write balance -> 3
print 3
```

## Using locks: bank balance example

```
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            print('Insufficient funds')
        else:
            balance = balance - amount
            print(balance)
    return withdraw
```

critical section

New code

```
def make_withdraw(balance)
    balance_lock = Lock()
    def withdraw(amount):
        nonlocal balance
        # try to acquire the lock
        balance_lock.acquire()
        # once successful, enter the critical section
        if amount > balance:
            print("Insufficient funds")
        else:
            balance = balance - amount
            print(balance)
        # upon exiting the critical section, release the lock
        balance_lock.release()
```

## Using locks: bank balance example

```
w = make_withdraw(10)
balance = 10
balance_lock = Lock() acquired by p2
```

w(8)

**P1**  
acquire balance\_lock: ok  
read balance: 10  
read amount: 8  
8 > 10: False  
if False  
10 - 8: 2  
write balance -> 2  
print 2  
release balance\_lock

w(7)

**P2**  
acquire balance\_lock: wait  
wait  
wait  
wait  
wait  
wait  
wait  
wait  
acquire balance\_lock:ok  
read balance: 2  
read amount: 7  
7 > 2: True  
if True  
print 'Insufficient funds'  
release balance\_lock

## Quiz: does this solution enforce correctness?

```
def make_withdraw(balance):
    balance_lock = Lock()
    def withdraw(amount):
        nonlocal balance
        # try to acquire the lock
        balance_lock.acquire()
        # once successful, enter the critical section
        if amount > balance:
            print("Insufficient funds")
        else:
            balance = balance - amount
            print(balance)
        # upon exiting the critical section, release the lock
        balance_lock.release()
```

## Answer: yes

remember: always  
release your locks.

```
def make_withdraw(balance):
    balance_lock = Lock()
    def withdraw(amount):
        nonlocal balance
        # try to acquire the lock
        balance_lock.acquire()
        # once successful, enter the critical section
        if amount > balance:
            print("Insufficient funds")
        else:
            balance = balance - amount
            print(balance)
        # upon exiting the critical section, release the lock
        balance_lock.release() important, allows others
        to proceed
```

No two processes can be in the critical section at the same time.

Whichever gets to balance\_lock.acquire() first gets to finish.

All others have to wait until it's finished.

## Semaphores

Used to protect access to limited resources

Each has a limit, N

Can be acquire()'d N times

After that, processes trying to acquire() automatically wait

Until another process release()'s

## Semaphores example: database

A database that can only support 2 connections at a time.

```
# set up the semaphore
db_semaphore = Semaphore(2)
```

```
def insert(data):
    # try to acquire the semaphore
    db_semaphore.acquire()
    # if successful, proceed
    database.insert(data)
    #release the semaphore
    db_semaphore.release()
```

## Example: database

```
db_semaphore = Semaphore(2)
```

```
def insert(data):
    db_semaphore.acquire()
    database.insert(data)
    db_semaphore.release()
```

insert(7)

**P1**

acquire db\_semaphore: ok  
read data: 7  
read global database  
insert 7 into database  
release db\_semaphore: ok

insert(8)

**P2**

acquire db\_semaphore: wait  
wait  
wait  
wait  
acquire db\_semaphore: ok  
read data: 8  
read global database  
insert 8 into database  
release db\_semaphore: ok

insert(9)

**P3**

acquire db\_semaphore: ok  
read data: 9  
read global database  
insert 9 into database  
release db\_semaphore: ok

## Conditions

Conditions are signals used to coordinate multiple processes

Processes can wait() on a condition

Other processes can notify() processes waiting for a condition.

25

## Conditions example: vector mathematics

step1\_finished = 0  
start\_step2 = Condition()  $A = B+C$   
 $V = MxA$

```
def do_step_1(index):  
    A[index] = B[index] + C[index]  
    start_step2.acquire()  
    step1_finished += 1  
    if(step1_finished == 2):  
        start_step2.notifyAll()  
    start_step2.release()
```

```
def do_step_2(index):  
    start_step2.wait()  
    V[index] = M[index] . A
```

25

## Conditions example: vector mathematics

step1\_finished=2     $B = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$      $C = \begin{pmatrix} 0 \\ 5 \end{pmatrix}$      $M = \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$      $A = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$   
start\_step2 = Condition()

$A_1 = B_1 + C_1$   
 $V_1 = M_1 . A$

P1

```
read B1: 2  
read C1: 0  
calculate 2+0: 2  
write 2 -> A1  
acquire start_step2: ok  
write 1 -> step1_finished  
step1_finished == 2: false  
release start_step2: ok  
start_step2: wait  
start_step2: wait  
start_step2: wait  
read M1: (1 2)  
read A: (2 5)  
calculate (1 2) . (2 5): 12
```

$A_2 = B_2 + C_2$   
 $V_2 = M_2 . A$

P2

```
read B2: 0  
read C2: 0  
calculate 5+0: 5  
write 5 -> A2  
acquire start_step2: ok  
write 2 -> step1_finished  
step1_finished == 2: true  
notifyAll start_step2: ok  
read M2(1 2)  
read A: (2 5)
```

25

## Deadlock

A condition in which threads are stuck waiting for each other forever

25

## Deadlock example

```
>>> x_lock = Lock()  
>>> y_lock = Lock()  
>>> x = 1  
>>> y = 0  
>>> def compute():  
    x_lock.acquire()  
    y_lock.acquire()  
    y = x + y  
    x = x * x  
    y_lock.release()  
    x_lock.release()  
>>> def anti_compute():  
    y_lock.acquire()  
    x_lock.acquire()  
    y = y - x  
    x = sqrt(x)  
    x_lock.release()  
    y_lock.release()
```

25

## Deadlock: example

```
def compute():  
    x_lock.acquire()  
    y_lock.acquire()  
    y = x + y  
    x = x * x  
    y_lock.release()  
    x_lock.release()  
def anti_compute():  
    y_lock.acquire()  
    x_lock.acquire()  
    y = y - x  
    x = sqrt(x)  
    x_lock.release()  
    y_lock.release()
```

compute()

```
P1  
acquire x_lock: ok  
acquire y_lock: wait  
wait  
wait  
wait  
wait  
...
```

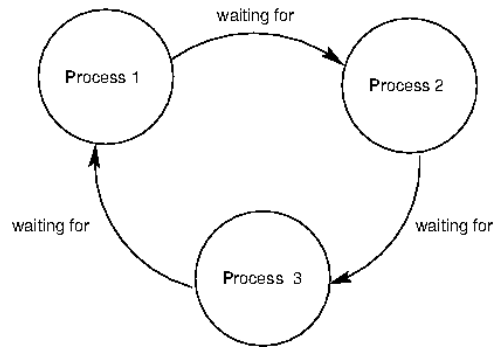
anti\_compute()

```
P2  
acquire y_lock: ok  
acquire x_lock:  
wait  
wait  
wait  
wait  
...
```

25

## Deadlock

---



## Next time

---

Sequences and Streams