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

```python
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)
w(8)
w(7)
```

$15 withdrawn from a $10 account?  
With $3 left? Inconceivable!
Parallel computing example: bank balance

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

w = make_withdraw(10)
balance = 10
w(8)
w(7)
```

print('Insufficient funds')

Another problem: vector mathematics

\[ A = B + C \]
\[ V = MA \]

Vector mathematics

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

\[ A_1 = B_1 + C_1 \]
\[ V_1 = M_1 A \]
\[ A_2 = B_2 + C_2 \]
\[ V_2 = M_2 A \]

P1
read \( B \): 2
read \( C \): 0
calculate \( 2 + 0 \): 2
write 2 \rightarrow A1
read \( M_1 \): \( (1 \ 2) \)
read \( A \): \( (2 \ 0) \)
calculate \( (1 \ 2) \cdot (2 \ 0) \): 2
write 2 \rightarrow V1
write 2 \rightarrow V1

P2
read \( B \): 0
read \( C \): 5
calculate \( 5 + 0 \): 5
write 5 \rightarrow A2
read \( M_2 \): \( (1 \ 2) \)
read \( A \): \( (2 \ 5) \)
calculate \( (1 \ 2) \cdot (2 \ 5) \): 12
write 12 \rightarrow V2
write 12 \rightarrow V2

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

```
P1
harmless code
harmless code
modify shared variable
............
............
............
............
write shared variable
harmless code
harmless code

P2
Critical Section
Should not be interrupted
by other threads that
access same variable
```

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
```

Locks

```
def make_withdraw(balance):
    balance = 10
    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()
    return withdraw
```

Using locks: bank balance example

```
def make_withdraw(balance):
    balance = 10
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            print('Insufficient funds')
        else:
            balance = balance - amount
            print(balance)
    return withdraw
```

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()
    return withdraw
```

Using locks: bank balance example

```
def make_withdraw(balance):
    balance = 10
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            print('Insufficient funds')
        else:
            balance = balance - amount
            print(balance)
    return withdraw
```

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()
    return withdraw
```

Using locks: bank balance example

```
def make_withdraw(balance):
    balance = 10
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            print('Insufficient funds')
        else:
            balance = balance - amount
            print(balance)
    return withdraw
```

New code
Using locks: bank balance example

\[
\begin{align*}
  w &= \text{make\_withdraw}(10) \\
  \text{balance} &= 10 \\
  \text{balance\_lock} &= \text{Lock()} \text{ acquired by p2}
\end{align*}
\]

\[
\begin{align*}
  w(8) & & \text{P1} & \text{acquire balance\_lock: ok} \\
  \text{read balance: } 10 & \text{ read amount: } 8 & \text{if False} & \text{10 - 8 = 2} \\
  \text{write balance} & \rightarrow 2 & \text{print } 2 & \text{wait} \\
  \text{release balance\_lock} & & & \text{released by p2}
\end{align*}
\]

Quiz: does this solution enforce correctness?

\[
\begin{align*}
\text{def make\_withdraw(balance)} \\
\text{balance\_lock} &= \text{Lock()} \\
\text{def withdraw(amount):} \\
\text{nonlocal} \text{ balance} \\
\text{# try to acquire the lock} \\
\text{balance\_lock.acquire()} \\
\text{# once successful, enter the critical section} \\
\text{if amount > balance:} \\
\text{print('Insufficient funds')} \\
\text{else:} \\
\text{balance} &= \text{balance - amount} \\
\text{print(balance)} \\
\text{# upon exiting the critical section, release the lock} \\
\text{balance\_lock.release()}
\end{align*}
\]

Answer: yes

\[
\begin{align*}
\text{def make\_withdraw(balance)} \\
\text{balance\_lock} &= \text{Lock()} \\
\text{def withdraw(amount):} \\
\text{nonlocal} \text{ balance} \\
\text{# try to acquire the lock} \\
\text{balance\_lock.acquire()} \\
\text{# once successful, enter the critical section} \\
\text{if amount > balance:} \\
\text{print('Insufficient funds')} \\
\text{else:} \\
\text{balance} &= \text{balance - amount} \\
\text{print(balance)} \\
\text{# upon exiting the critical section, release the lock} \\
\text{balance\_lock.release()}
\end{align*}
\]

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

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

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

All others have to wait until it’s finished.

Semaphores example: database

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

\[
\begin{align*}
\text{# set up the semaphore} \\
\text{db\_semaphore} &= \text{Semaphore(2)} \\
\text{def insert(data):} \\
\text{# try to acquire the semaphore} \\
\text{db\_semaphore.acquire()} \\
\text{# if successful, proceed} \\
\text{database.insert(data)} \\
\text{db\_semaphore.release()}
\end{align*}
\]

Example: database

\[
\begin{align*}
\text{db\_semaphore} &= \text{Semaphore(2)} \\
\text{def insert(data):} \\
\text{db\_semaphore.acquire()} \\
\text{database.insert(data)} \\
\text{db\_semaphore.release()}
\end{align*}
\]

\[
\begin{align*}
\text{insert(7)} & & \text{P1} & \text{acquire db\_semaphore: ok} \\
\text{read data: 7} & \text{read global database} & \text{insert 7 into database} & \text{release db\_semaphore: ok} \\
\text{insert(8)} & & \text{P2} & \text{acquire db\_semaphore: wait} \\
\text{read data: 8} & \text{read global database} & \text{insert 8 into database} & \text{release db\_semaphore: ok} \\
\text{insert(9)} & & \text{P3} & \text{acquire db\_semaphore: ok} \\
\text{read data: 9} & \text{read global database} & \text{insert 9 into database} & \text{release db\_semaphore: ok}
\end{align*}
\]
Conditions

Conditions are signals used to coordinate multiple processes. Processes can wait() on a condition. Other processes can notify() processes waiting for a condition.

Conditions example: vector mathematics

\[
\begin{align*}
A &= B + C \\
V &= MA \\
A_1 &= B_1 + C_1 \\
V_1 &= M_1 A \\
A_2 &= B_2 + C_2 \\
V_2 &= M_2 A \\
\end{align*}
\]

Deadlock

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

Deadlock: example

```python
>>> 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)
...     y_lock.release()
...     x_lock.release()
```

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

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

Next time

Sequences and Streams