Threads

- Cute and furry beasts
Threads

- Must be restrained
Restrained concurrency models

- **Actors**
  - Good for independent tasks
  - Good for discriminate producers/consumers of data

- **Tuple spaces**
  - Good for indiscriminate producers/consumers of data

- **Map-reduce**
  - Good for data-intensive, parallelizable situations
Actors
Actor model

- Letterbox style (Ch 11) + Threads
- Actor = Object with its own thread
  - Aka “active object”
- Actors send messages to each other
  - Avoid shared memory
- Messages are placed in actors’ queues
  - Queues must be “thread-safe”
  - Sender places message and moves on
    - Asynchronous request
Active Object (Python)

```python
class ActiveWFOObject (Thread):
    def __init__(self):
        Thread.__init__(self)
        self.name = str(type(self))
        self.queue = Queue()
        self._stop = False
        self.start()

    def run(self):
        while not self._stop:
            message = self.queue.get()
            self._dispatch(message)
            if message[0] == 'die':
                self._stop = True

    def send(self, receiver, message):
        receiver.queue.put(message)
```

- **Thread-safe queue**
- **Block until there is a message**
- **Message loop**
- **Utility (could be a method)**
Active Object Queue
Queues

- Put / Enqueue / Send
- Get / Dequeue / Receive

- Operations must be thread safe
  - No items can be lost
Thread-safe queues

- Java: ArrayBlockingQueue
- C#: ConcurrentQueue
- C++ / Boost: message_queue
- Other langs: search for it or do it yourself
Actor example

```python
class WordFrequencyManager(ActiveWFOBJECT):
    """ Keeps the word frequency data """
    _word_freqs = {}  

def _dispatch(self, message):
    if message[0] == 'word':
        self._increment_count(message[1:])
    elif message[0] == 'top25':
        self._top25(message[1:])

def _increment_count(self, message):
    word = message[0]
    if word in self._word_freqs:
        self._word_freqs[word] += 1
    else:
        self._word_freqs[word] = 1

def _top25(self, message):
    recipient = message[0]
    freqs_sorted = sorted(self._word_freqs.items(), key=operator.itemgetter(1), reverse=True)
    send(recipient, ['top25', freqs_sorted])

Send messages to other actors
```
Actor model

- Concurrency constrained by
  - Associating [certain] objects with threads
  - Using message queues in each actor
  - Having threads on a loop

- Programmer needs to refrain from passing shared mutable objects around or else...
Tuple space model

- Concurrent threads
  - Consumers and producers of data items
- Shared data structures (queues, lists, trees, etc.)
  - Must be “thread-safe”
- Producers add items and move on
  - Asynchronous deposit
- Consumers take items and process them

- Similar to previous model, but where the queues are outside the objects/functions, and may not be queues
Tuple space model
TF Tuple spaces

4 # Two data spaces
5 word_space = Queue.Queue()
6 freq_space = Queue.Queue()
# Let's have this thread populate the word space

```python
for word in re.findall('[a-z]{2,}', open(sys.argv[1]).read().lower()):
    word_space.put(word)
```
# Worker function that consumes words from the word space and sends partial results to the frequency space

def process_words():
    word_freqs = {}

    while True:
        try:
            word = word_space.get(timeout=1)
        except Queue.Empty:
            break

        if not word in stopwords:
            if word in word_freqs:
                word_freqs[word] += 1
            else:
                word_freqs[word] = 1

        freq_space.put(word_freqs)
Let’s create the workers and launch them at their jobs

```python
workers = []
for i in range(5):
    workers.append(threading.Thread(target = process_words))
[t.start() for t in workers]
```

(functional style of creating threads in Python)
Tuple space model

- Can be functional or OOP style
- OOP style: worker functions are threaded objects
- Best fit: data processing parallelization

Akers example decomposition:
- DataStorageManager
- StopWordManager
- WordFrequencyManager
- WordFrequencyController

Tuple space example decomposition:
- Producers of words
- Consumers of words / producers of word frequencies
- Consumers of word frequencies
Tuple space model

- Concurrency constrained by
  - Having shared, thread-safe collections of items
  - Having producers/consumers of items in those collections
  - No further communication between threaded code
- Programmer needs to refrain from passing shared mutable objects around or else...
Map-Reduce
Map-Reduce model

- Big data situations
  - Problem at hand must be data-parallelizable
- Data is split into chunks
- Chunks are processed independently, produce partial results
  - A function is “mapped” to the chunks of data, potentially in parallel
- Partial results are then “reduced” to final result
  - This step is sequential
splits = \texttt{map}(\texttt{split\_words}, \texttt{partition}(\texttt{read\_file(sys.argv[1]), 200}))

\texttt{splits.insert(0, [])} \quad \# \texttt{normalize input to reduce}

\texttt{word\_freqs = sort(\texttt{reduce}(\texttt{count\_words, splits}))}
Data partitioning

```python
def partition(data_str, nlines):
    """
    Partitions the input data_str (a big string) into chunks of nlines.
    """
    lines = data_str.split('
')
    for i in xrange(0, len(lines), nlines):
        yield '
'.join(lines[i:i+nlines])
```
def split_words(data_str):
    """
    Takes a string, returns a list of pairs (word, 1), one for each word in the input, so
    [(w1, 1), (w2, 1), ..., (wn, 1)]
    """
    def _scan(str_data):
        pattern = re.compile('[\W_]+')
        return pattern.sub(' ', str_data).lower().split()
    def _remove_stop_words(word_list):
        with open('..\stop_words.txt') as f:
            stop_words = f.read().split(',,')
        stop_words.extend(list(string.ascii_lowercase))
        return [w for w in word_list if not w in stop_words]

    # The actual work of splitting the input into words
    result = []
    words = _remove_stop_words(_scan(data_str))
    for w in words:
        result.append((w, 1))
    return result
def count_words(pairs_list_1, pairs_list_2):
    """
    Takes two lists of pairs of the form
    [(w1, 1), ...]
    and returns a list of pairs [(w1, frequency), ...],
    where frequency is the sum of all the reported occurrences
    """
    mapping = dict((k, v) for k, v in pairs_list_1)
    for p in pairs_list_2:
        if p[0] in mapping:
            mapping[p[0]] += p[1]
        else:
            mapping[p[0]] = 1
    return mapping.items()
Map-Reduce, Hadoop

- The previous style allows for parallelization of the map step, but requires serialization of the reduce step. Google map-reduce and Hadoop use a slight variation that makes the reduce step also potentially parallelizable. The main idea is to regroup, or reshuffle, the list of results from the map step so that the regroupings are amenable to further mapping of a reducible function.
splits = map(split_words, partition(read_file(sys.argv[1]), 200))
splits_per_word = regroup(splits)
word_freqs = sort(map(count_words, splits_per_word.items()))
def regroup(pairs_list):
    """
    Takes a list of lists of pairs of the form
    
    [[(w1, 1), (w2, 1), ..., (wn, 1)],
     [(w1, 1), (w2, 1), ..., (wn, 1)],
     ...
    
    and returns a dictionary mapping each unique word to the corresponding list of pairs, so
    
    { w1 : [(w1, 1), (w1, 1)...],
      w2 : [(w2, 1), (w2, 1)...],
      ...
    }
    """

    mapping = {}
    for pairs in pairs_list:
        for p in pairs:
            if p[0] in mapping:
                mapping[p[0]].append(p)
            else:
                mapping[p[0]] = [p]

    return mapping
Map-reduce

- **Java**: try [Functional Java](https://www.functional-java.com) library
  - Or do “mapper” and “reducer” classes yourself

- **PHP**: try [this](https://example.com)
Map-Reduce model

- Concurrency constrained by
  - Having worker threads work on mutually exclusive chunks of data
  - No communication between threaded code