# Lecture 35: Streams and Concurrency

# Aside: Threads and Processes

- Multiprocessing in general was out of the mainstream of programming practice for decades.
- As a result, it is packaged in a distressing variety of ways with terminology to match.
- The terms *thread*, *process*, and *light-weight process* all refer to kinds of logically or physically concurrent computations.
  - *Processes* are generally "heavy weight" objects that tend to be isolated from each other, communicating best through specialized data structures or I/O.
  - *Threads* are "lighter-weight" objects that generally share memory and processing resources with each other. They tend to have various arcane restrictions on what they can do in parallel with each other, and are often more about logical separation of function than parallelism.
  - The less said about light-weight processes, the better.
- Python provides both, with "processes" generally intended for actual parallel computation, and thread often used for "parallel waiting."

# Futures

- We've seen *lazy values:* expressions that do not get evaluated until their value is actually needed.
- A simple extension to this idea is the *future:*

  An expression that gets evaluated while the requesting process proceeds concurrently, and only later comes back to pick up the value.
- Full Scheme provides futures among its builtin features. Python provides them in its library.

# Basics of Futures in Python: concurrent.futures

- An object of type `Future` (in `concurrent.futures`) is created by an object called an `Executors` from a function and some arguments.
- It executes the function (in parallel), and returns the result (or exception) when its `.result()` method is called.
- Its `.exception()` method returns the exception that ended computation (or `None`).
- Both methods take an optional "timeout" parameter that limits the time one spends waiting for a result.
- The builtin executors that create future are `ThreadPoolExecutor`, which creates a future out of a thread, and a `ProcessPoolExecutor`, which creates a future out of a process.
- Unfortunately, considerations of when to use which depend on the OS and on the application: threads if you really need to share data in memory, and processes if you need isolation and concurrency.

# Example of Futures (I)

From the Python 3 documentation (adapted):

```
NUMBERS = [ 112272535095293, 112582705942171, 112272535095293,
            115280095190773, 115797848077099, 109972689928541 9]

def is_prime(n):
    """True iff non-negative integer N is prime."""
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def find_primes():
    for number in NUMBERS:
        print(number, 'is prime:'  is_prime(number))
```

# Parallelizing Example (I)

- As written, we check the numbers in sequence. No need in principal.

```
def find_primes():
    executor = concurrent.futures.ProcessPoolExecutor()
    # Create a future for testing each number.
    tests = [ executor.submit(is_prime, n) for n in NUMBERS ]
    # Print results
    for number, prime in zip(NUMBERS, tests):
        print(number, 'is prime:', prime.result())
    # Clean up.
    executor.shutdown()
```

- The `ProcessPoolExecutor` maintains a pool of worker processes that it wakes up to execute functions given to `.submit`.
- (`zip(L1, L2)` returns the sequence `(L1[0], L2[0]), (L1[1], L2[1]), ...`.)

# Parallelizing Example (I): Map

- Creating an array of futures is common, so...

```
def find_primes():
    executor = concurrent.futures.ProcessPoolExecutor()
    for number, prime in zip(NUMBERS, executor.map(is_prime, NUMBERS)
        print(number, 'is prime:', prime)
    executor.shutdown()
```

- `executor.map` is like regular map, but gets its results by creating futures and then applying `.result` to them.

# Side Note: Insuring Clean Up

- Executors use up limited processor resources, so programmers usually want to make sure things are cleaned up reliably.
- Here's the standard way:

```
def find_primes():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(NUMBERS, executor.map(is_prime, NUM
                print(number, 'is prime:', prime)
```

# Parallelizing Example (II)

Another example from the Python documentation (sequentially):

```
URLS = ['http://www.foxnews.com/', 'http://www.cnn.com/',
        'http://europe.wsj.com/',  'http://www.bbc.co.uk/',
        'http://some-made-up-domain.com/']

def load_url(url, timeout):
    """Read the result of a GET request to URL, waiting up to
    TIMEOUT seconds."""
    return urllib.request.urlopen(url, timeout=timeout).read()

def check_urls():
    for url in URLS:
        try:
            print("{0} page is {1} bytes"
                  .format(url, len(load_url(url,60))))
        except:
            print("{0} page generated an exception".format(url))
```

# Parallelizing Example (II): Overlapping waiting

- Sometimes, we use overt parallelism to avoid serial waiting.
- In the URL example, nothing happens while we wait for a remote site to respond. We could send out all requests and *then* wait:

```
from concurrent.futures import ThreadPoolExecutor, as_completed
def check_urls():
    with ThreadPoolExecutor(max_workers=5) as executor:
        future_to_url = \
          dict((executor.submit(load_url, url, 60), url)
               for url in URLS)
        for future in as_completed(future_to_url):
            url = future_to_url[future]
            if future.exception() is not None:
                print('{0} generated an exception: {1}'
                      .format(url, future.exception()))
            else:
                print('{0} page is {1} bytes'
                      .future(url, len(future.result())))
```

# Comments on Example II

- Threads are appropriate here, since we don't really need parallel computation: all threads will be waiting on I/O.
- The `as_completed` method takes an iterable of some kind that yields futures, and returns an iterator that produces them as they complete.

# Queues: More Communication among Processes

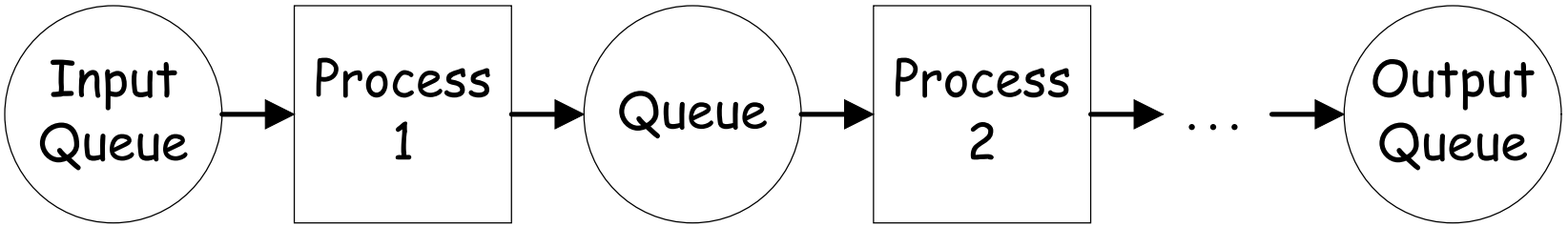
- A queue of values (like the Mailboxes in Lecture #33) is a convenient mechanism for communicating between processes.
- A process can wait for a value when it is ready and queue up a value for another process when it has one.
- Flexible as to whether one waits for a process to respond before continuing.
- In Python, the `multiprocessing` package provides `Processes` (which we've been using in disguised form right along) and `Queues`, which allow callers to wait for data.

```
def f(q):
    q.put([42, None, 'hello'])

def trivial_example():
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()          # Waits for p to finish
```

# Pipelines

- All kinds of arrangements of processes can be constructed with Process and Queue.
- One simple one: a *pipeline* is a sequence of processes, each of which sends data to the next in line and receives it from the preceding process.
- We can overlap the computations in each process, and get considerable speedup if they have comparable speed.



# Pipelines in Code

```
def accum(input, output, s):
    while True:
        p = input.get()
        if p is None:
            output.put(None)
        s += p
        output.put(s)
```

```
def mul(input, output, k):
    while True:
        p = input.get()
        if p is None:
            output.put(None)
        output.put(p*k)
```

```
Q1, Q2, Q3 = Queue(), Queue(), Queue()
stage1 = Process(target = accum, args=(Q1, Q2, 0))
stage2 = Process(target = mul, args=(Q2, Q3))
stage1.start(); stage2.start()
for i in range(10):
    Q1.put(i)
Q1.put(None)
while True:
    x = Q3.get()
    if x is None: break
    print(x)
stage1.join(); stage2.join()
```