Writing Faster Python

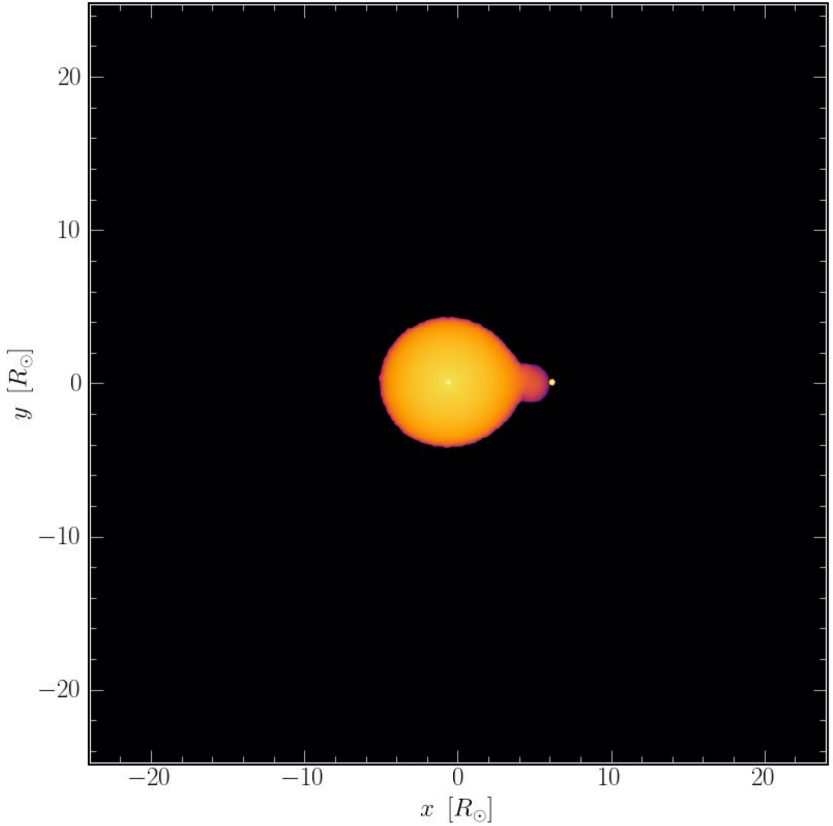
Roger Hatfull

15th Annual Symposium for Graduate Physics Research

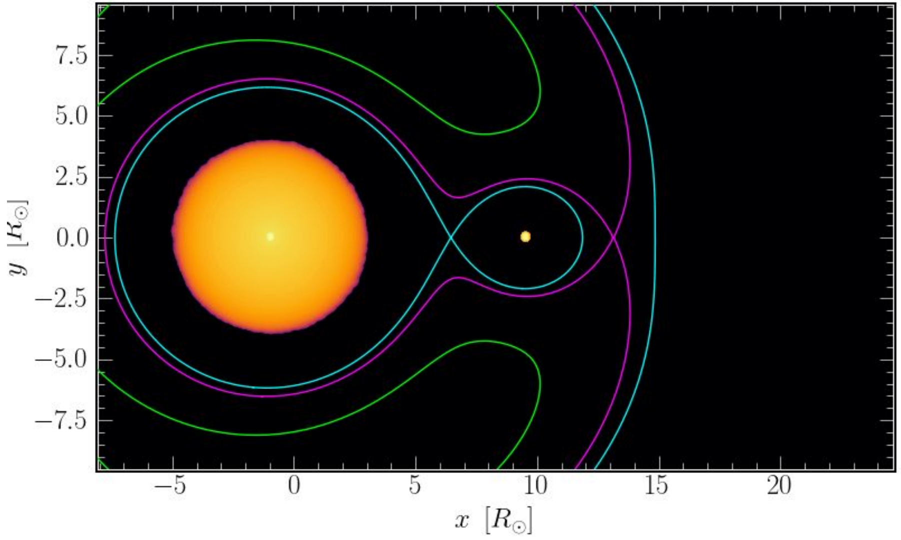
GPSA, University of Alberta

https://tinyurl.com/wfpgpsa2024

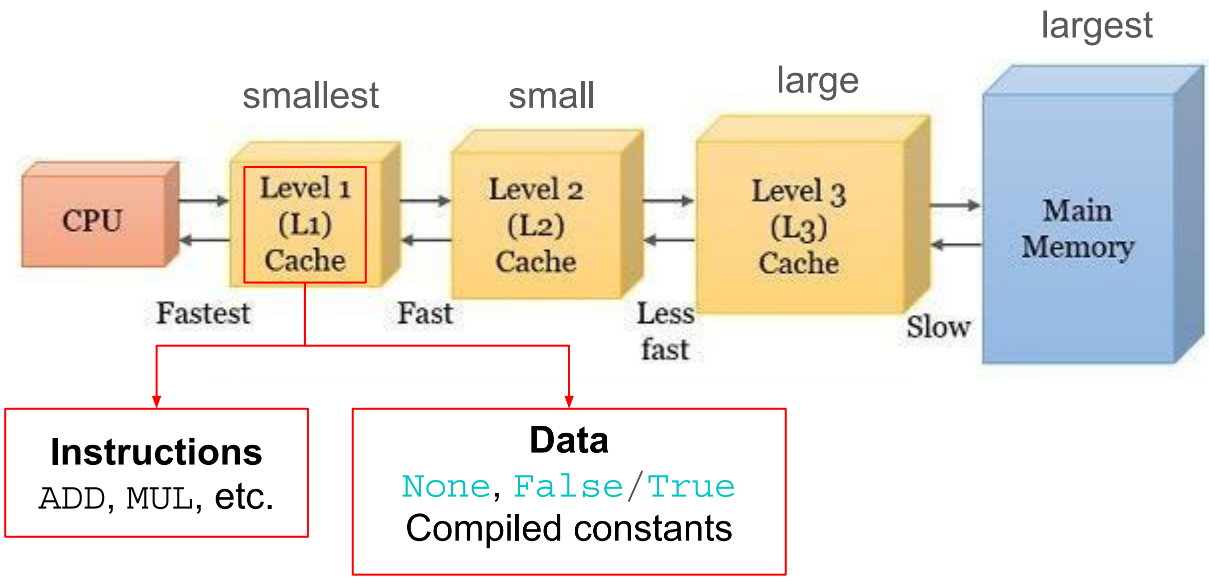
https://github.com/hatfullr/writing-faster-python



~20 TB data
~300 CPU years
~50 GPU years



CPUs



smallest

small

large

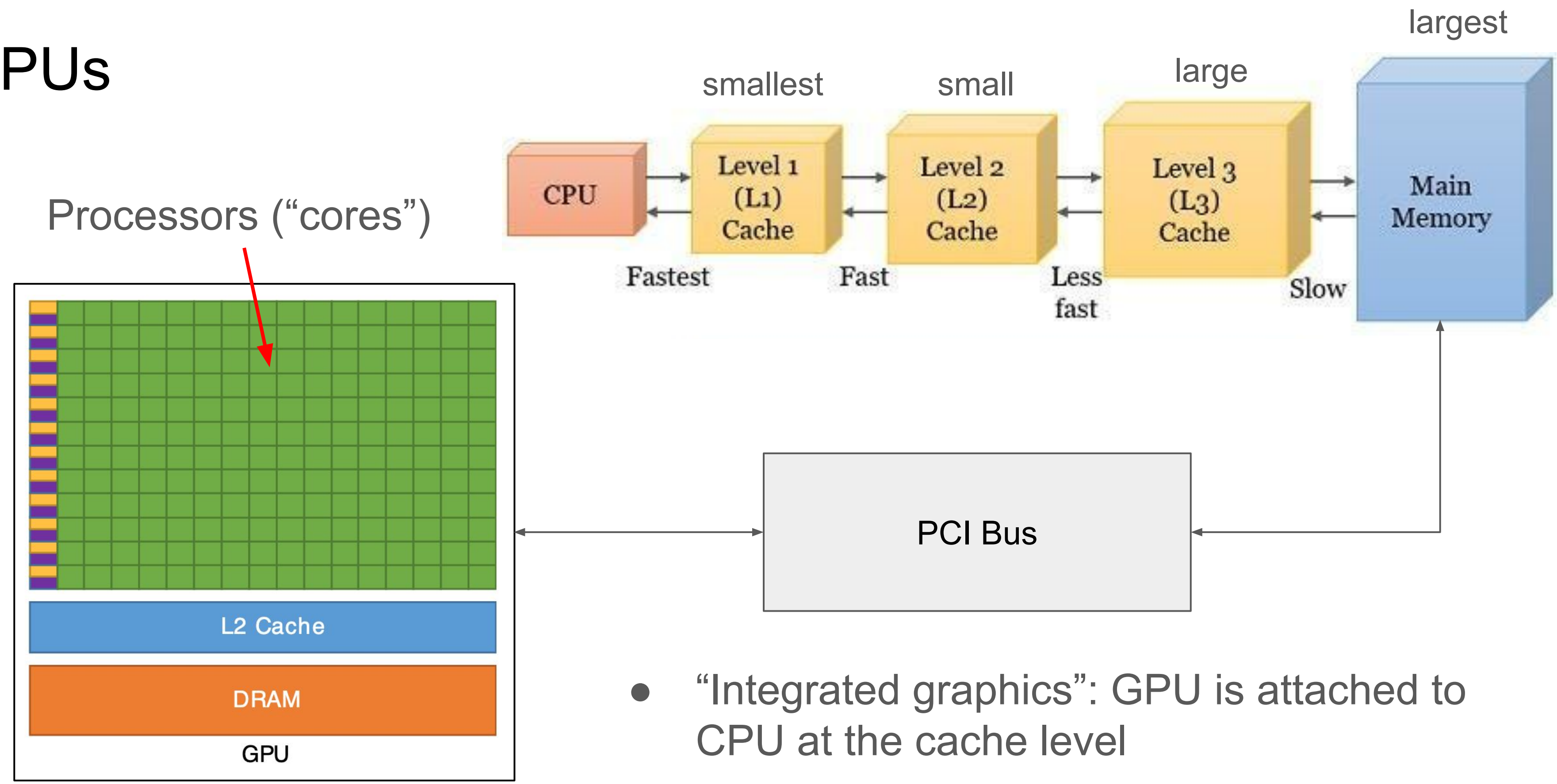
largest

Instructions
`ADD`, `MUL`, etc.

Data
`None`, `False`/`True`
Compiled constants

- (Some) manufacturer tricks:
  - “Cache lines”: data from caches are loaded 64 bytes at a time
  - “Pre-fetching”: Predict and load to low-level caches before needed
    - <u>Scripts run faster on consecutive executions</u>: data is being cached

GPUs



- "Integrated graphics": GPU is attached to CPU at the cache level

Lowest-hanging fruit

- **Programmer time > execution time**
- Make shortest syntax a habit
- Modern computations are typically memory-bound
  - Performing calculations can be much faster than storing calculations

scripts/syntax/list.py

```
0)    0.528331 seconds   "for i in range(len(a)): b += [a[i]]"
1)    0.485454 seconds   "for i, ai in enumerate(a): b += [ai]"
2)    0.479977 seconds   "for i in range(len(a)): b[i] = a[i]"
3)    0.464579 seconds   "for i, ai in enumerate(a): b[i] = ai"
4)    0.424565 seconds   "for i in range(len(a)): b.append(a[i])"
5)    0.398427 seconds   "for i, ai in enumerate(a): b.append(ai)"
6)    0.362043 seconds   "for ai in a: b += [ai]"
7)    0.290544 seconds   "for ai in a: b.append(ai)"
8)    0.279091 seconds   "b = [ai for ai in a]"
9)    0.105188 seconds   "b = a.copy()"
```

(Some) coding practices

- Avoid indentations
  - Unless it makes the code easier to read
- Loops with conditionals:
  - Use `continue` and `break`

scripts/examples/practices.py

```
for i in range(3):
    if i == 0:
        print("Hello")
    else:
        if i == 1:
            print("What's up?")
        else:
            print("Not much")
```

scripts/examples/practices.py

```
for i in range(3):
    if i == 0:
        print("Hello")
        continue
    if i == 1:
        print("What's up?")
        continue
    print("Not much")
```

```
$ python3 practices.py
Hello
What’s up?
Not much
```

```
if condition: value = True
else: value = False
```

✗

```
value = condition
```

✓

Being careful

- What are pointers?
  - When the computer accesses a pointer in memory, it receives an instruction to access a different place in memory
- Python creates pointers *sometimes*
- If you aren't sure: `copy.deepcopy()`

scripts/examples/pointers.py

```
a = [False, True, False]
b = a
print(a)
b[1] = False
print(a)
```

```
$ python3 pointers.py
[False, True, False]
[False, False, False]
```

scripts/examples/pointers.py

```
import copy
a = [False, True, False]
b = copy.deepcopy(a)
print(a)
b[1] = False
print(a)
```

```
[False, True, False]
[False, True, False]
```

Memory efficiency

- Generators (“enumerators”)
  - yield
  - Evaluates expressions on-the-fly
  - Saves memory space
  - Allows to easily write complex algorithms

```
def find_files(directory):
    results = []
    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if os.path.isdir(path):
            results += find_files_regular(path)
            continue
        else: results += [path]
    return results
```

(snippet)scripts/examples/generators.py

```
def find_files(directory):
    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if os.path.isdir(path): yield from find_files(path)
        else: yield path
```

Faster `NumPy`

- Avoid “`np.`______” at all costs

`scripts/syntax/np.py`

```
0)    4.605282 seconds   "for i, ai in enumerate(a): b = np.append(b, ai)"
1)    0.374851 seconds   "for i, ai in enumerate(a): b[i] = ai"
2)    0.103949 seconds   "np.copy(a)"
3)    0.058146 seconds   "a.copy()"

0)    2.729569 seconds   "np.sum(a)"
1)    2.181445 seconds   "for i, ai in enumerate(a): b += ai"
2)    1.691387 seconds   "sum(a)"
3)    1.342459 seconds   "a.sum()"

0)    1.332396 seconds   "np.power(a, 2)"
1)    0.541881 seconds   "pow(a, 2)"
2)    0.525106 seconds   "a**2"
3)    0.509239 seconds   "a*a"
```

line_profiler

https://github.com/pyutils/line_profiler

scripts/examples/line_profiler_example.py

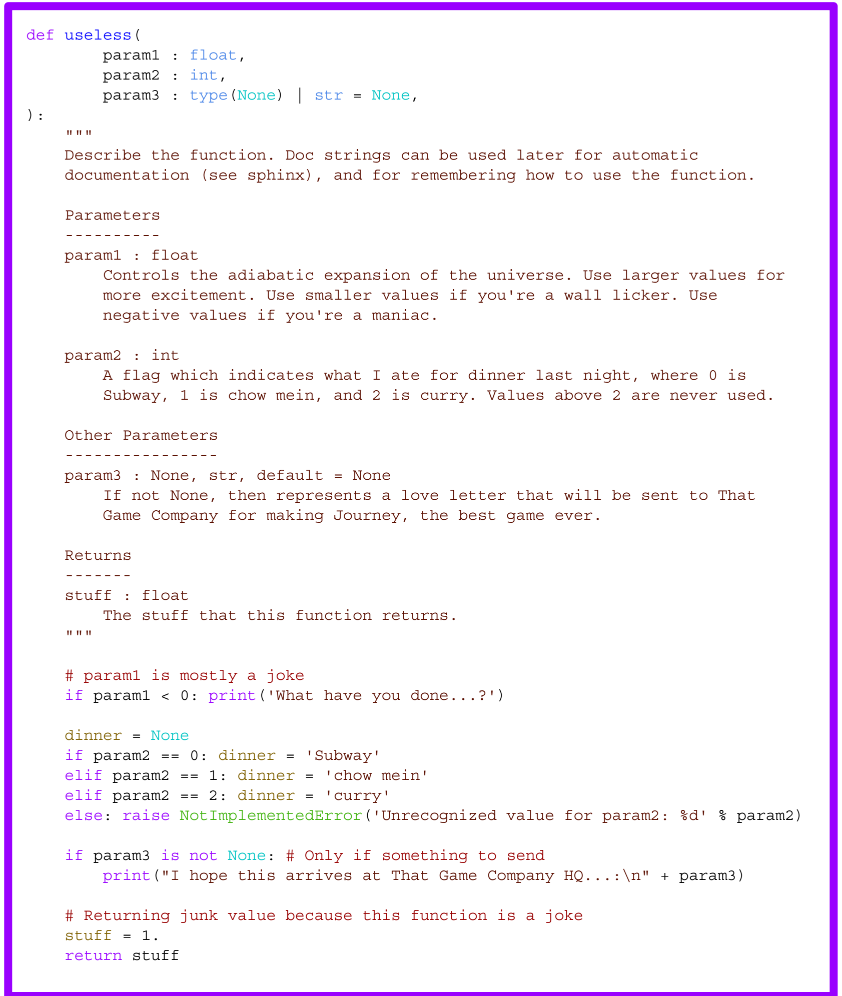
```python
@profile
def func(): # Some long operation
    for i in range(int(1e6)):
        i**2

func()
```

```
$ kernprof -l line_profiler_example.py
Wrote profile results to line_profiler_example.py.lprof
$ python3 -m line_profiler line_profiler_example.py.lprof
Timer unit: 1e-06 s

Total time: 0.912989 s
File: line_profiler_example.py
Function: func at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           @profile
     2                                           def func(): # Some long
operation
     3   1000001     330973.0      0.3     36.3      for i in range(int(1e6)):
     4   1000000     582016.0      0.6     63.7          i**2
```

Writing for the “user”

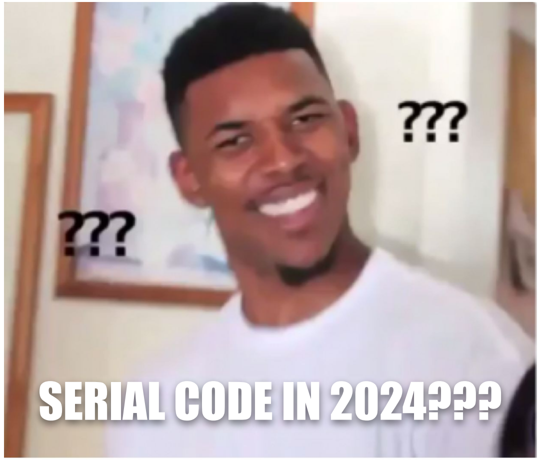
- **<u>YOU ARE THE USER</u>**
- Be kind to yourself!!
  - Function annotations
  - Doc strings
  - Code comments: “why” not “what”
- If not, you <u>WILL LOSE TIME LATER</u>
- Before writing, ask “is this something I might use a lot?”
- Most difficult task in programming is understanding someone else’s code
- You will be someone else in ~6 months
  - Your old code == someone else’s code

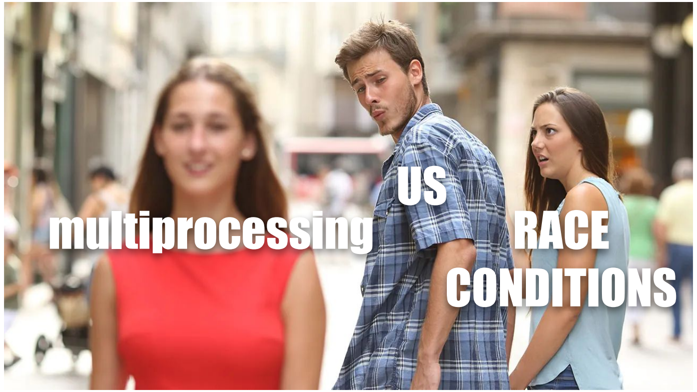
scripts/examples/user.py



```python
def useless(
        param1 : float,
        param2 : int,
        param3 : type(None) | str = None,
):
    """
    Describe the function. Doc strings can be used later for automatic
    documentation (see sphinx), and for remembering how to use the function.

    Parameters
    ----------
    param1 : float
        Controls the adiabatic expansion of the universe. Use larger values for
        more excitement. Use smaller values if you're a wall licker. Use
        negative values if you're a maniac.

    param2 : int
        A flag which indicates what I ate for dinner last night, where 0 is
        Subway, 1 is chow mein, and 2 is curry. Values above 2 are never used.

    Other Parameters
    ----------------
    param3 : None, str, default = None
        If not None, then represents a love letter that will be sent to That
        Game Company for making Journey, the best game ever.

    Returns
    -------
    stuff : float
        The stuff that this function returns.
    """

    # param1 is mostly a joke
    if param1 < 0: print('What have you done...?')

    dinner = None
    if param2 == 0: dinner = 'Subway'
    elif param2 == 1: dinner = 'chow mein'
    elif param2 == 2: dinner = 'curry'
    else: raise NotImplementedError('Unrecognized value for param2: %d' % param2)

    if param3 is not None: # Only if something to send
        print("I hope this arrives at That Game Company HQ...:\n" + param3)

    # Returning junk value because this function is a joke
    stuff = 1.
    return stuff
```

GET IN, LOSER
WE'RE CODING IN PARALLEL
Paramount

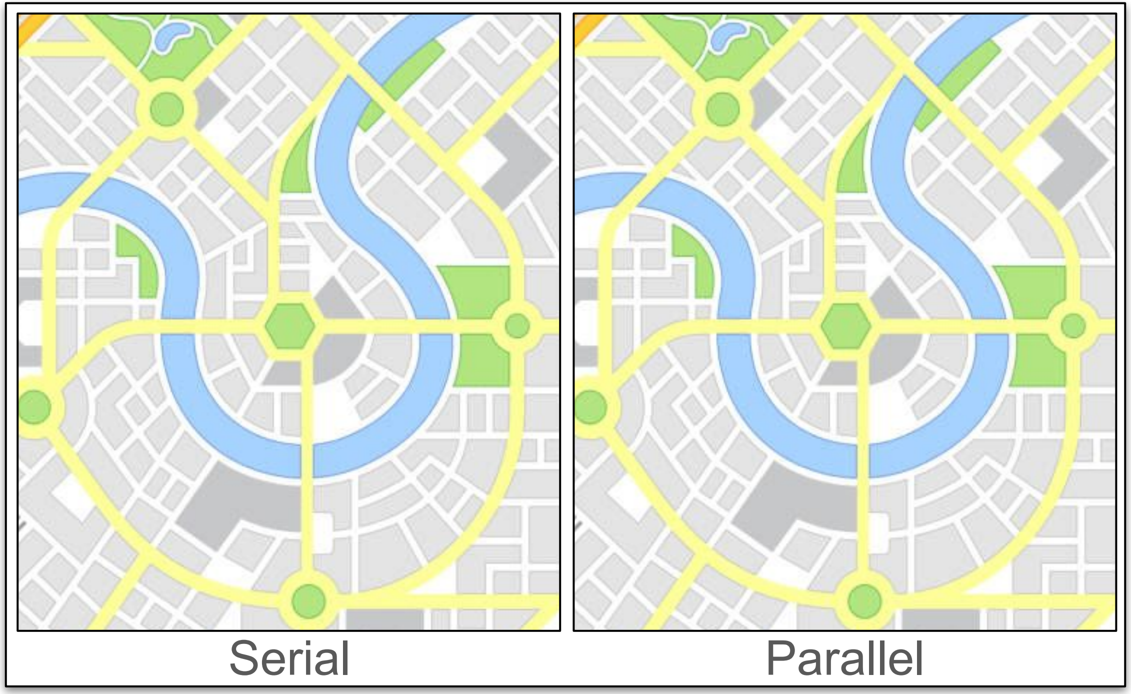

???
???
SERIAL CODE IN 2024???

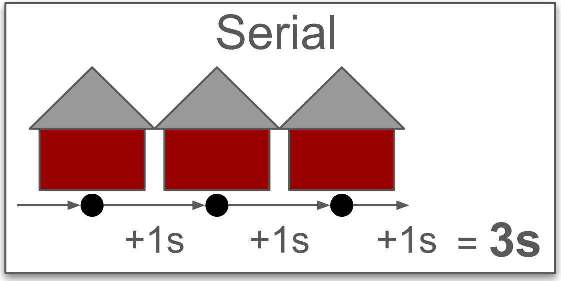

US
multiprocessing
RACE CONDITIONS

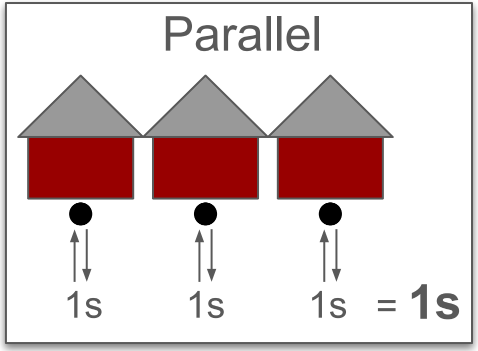
I CAN HAS FULL CPU UTILIZATION?

Thinking in parallel: mail delivery

- Deliver & pickup info
- You are the director @ HQ
- Serial (1 truck)
  - Simple
  - Takes too long
- Parallel (many trucks)
  - More complex
  - Much faster







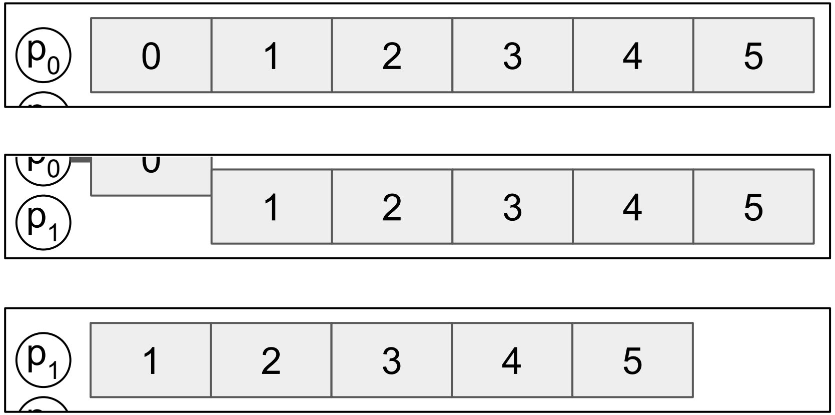
- Example with single “child” process
- When we run the script, we create the main process
- Main creates a child process
- If main dies, the child should die too
- “daemon = True”
- **This is still equivalent to serial**

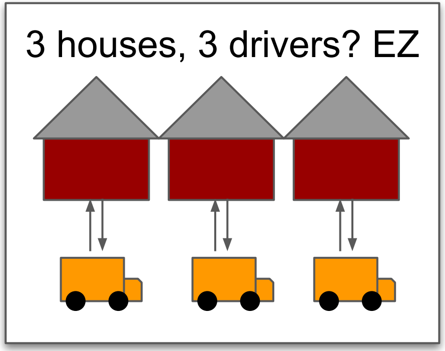
scripts/examples/single_process.py

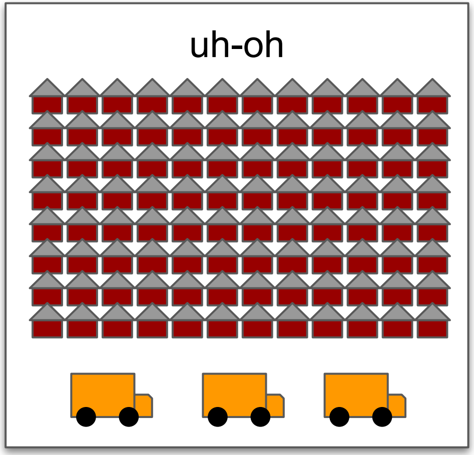
```
import multiprocessing, time

def func(): time.sleep(1)

if __name__ == '__main__': # Only if we are the main process
     # Create a child process
     process = multiprocessing.Process(
         target = func,
         daemon = True, # Terminate child when main exits
     )
     process.start()
     start_time = time.time() # Starting timestamp
     process.join() # Wait until child finishes
     end_time = time.time() # Stopping timestamp
     print(end_time - start_time) # Expect: 1 (s)
```

How to direct to the mail drivers?

- Create a “queue” at HQ
  - Driver calls for new address: take from the queue
- `multiprocessing.Queue`
  - **“First-in, first-out” (FIFO)**
    - “Last-in, first-out” (LIFO)







GOAL:

```
$ python3 queues.py
0
1
2
3
4
5
6
7
```

1) Make Queue

scripts/examples/queues.py

```python
import multiprocessing

if __name__ == '__main__':
    nprocs = 4
    queue = multiprocessing.Queue()

    # Put values in the queue
    for i in range(nprocs*2): queue.put(i)
    # Append end-of-queue signals
    for _ in range(nprocs): queue.put(None)

    # Create nprocs processes
    processes = [multiprocessing.Process(
        target = func,
        args = [queue],
        daemon = True,
    ) for _ in range(nprocs)]

    # Start processes
    for process in processes: process.start()
    # Wait for processes to finish
    for process in processes: process.join()
```

2) Fill Queue

When a process reads None from the queue, it stops running

scripts/examples/queues.py

```python
import multiprocessing

if __name__ == '__main__':
    nprocs = 4
    queue = multiprocessing.Queue()

    # Put values in the queue
    for i in range(nprocs*2): queue.put(i)
    # Append end-of-queue signals
    for _ in range(nprocs): queue.put(None)

    # Create nprocs processes
    processes = [multiprocessing.Process(
        target = func,
        args = [queue],
        daemon = True,
    ) for _ in range(nprocs)]

    # Start processes
    for process in processes: process.start()
    # Wait for processes to finish
    for process in processes: process.join()
```

3) Create processes

scripts/examples/queues.py

```
import multiprocessing

if __name__ == '__main__':
    nprocs = 4
    queue = multiprocessing.Queue()

    # Put values in the queue
    for i in range(nprocs*2): queue.put(i)
    # Append end-of-queue signals
    for _ in range(nprocs): queue.put(None)

    # Create nprocs processes
    processes = [multiprocessing.Process(
        target = func,
        args = [queue],
        daemon = True,
    ) for _ in range(nprocs)]

    # Start processes
    for process in processes: process.start()
    # Wait for processes to finish
    for process in processes: process.join()
```

3) Create processes

```
def func(queue):
    while True: # until no more tasks
        i = queue.get()
        if i is None: # end-of-queue
            return
        print(i)
```

scripts/examples/queues.py

```
import multiprocessing

if __name__ == '__main__':
    nprocs = 4
    queue = multiprocessing.Queue()

    # Put values in the queue
    for i in range(nprocs*2): queue.put(i)
    # Append end-of-queue signals
    for _ in range(nprocs): queue.put(None)

    # Create nprocs processes
    processes = [multiprocessing.Process(
        target = func,
        args = [queue],
        daemon = True,
    ) for _ in range(nprocs)]

    # Start processes
    for process in processes: process.start()
    # Wait for processes to finish
    for process in processes: process.join()
```

scripts/examples/queues.py

4) Start processes
5) Wait

```python
def func(queue):
    while True: # until no more tasks
        i = queue.get()
        if i is None: # end-of-queue
            return
        print(i)
```

```python
import multiprocessing

if __name__ == '__main__':
    nprocs = 4
    queue = multiprocessing.Queue()

    # Put values in the queue
    for i in range(nprocs*2): queue.put(i)
    # Append end-of-queue signals
    for _ in range(nprocs): queue.put(None)

    # Create nprocs processes
    processes = [multiprocessing.Process(
        target = func,
        args = [queue],
        daemon = True,
    ) for _ in range(nprocs)]

    # Start processes
    for process in processes: process.start()
    # Wait for processes to finish
    for process in processes: process.join()
```

6) Output

```python
def func(queue):
    while True: # until no more tasks
        i = queue.get()
        if i is None: # end-of-queue
            return
        print(i)
```

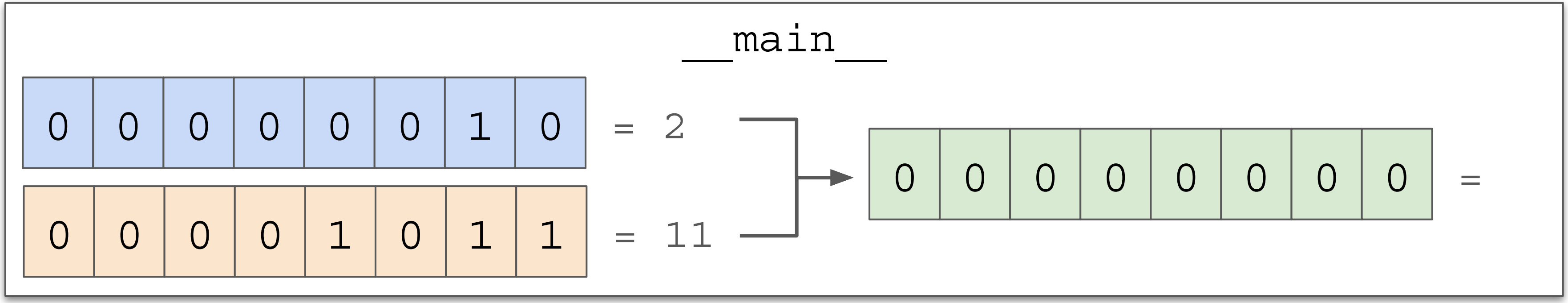
```
$ python3 queues.py
0
1
2
3
4
5
6
7
```

scripts/examples/queues.py

```python
import multiprocessing

if __name__ == '__main__':
    nprocs = 4
    queue = multiprocessing.Queue()

    # Put values in the queue
    for i in range(nprocs*2): queue.put(i)
    # Append end-of-queue signals
    for _ in range(nprocs): queue.put(None)

    # Create nprocs processes
    processes = [multiprocessing.Process(
        target = func,
        args = [queue],
        daemon = True,
    ) for _ in range(nprocs)]

    # Start processes
    for process in processes: process.start()
    # Wait for processes to finish
    for process in processes: process.join()
```

Complications

- Consider summation in **series**



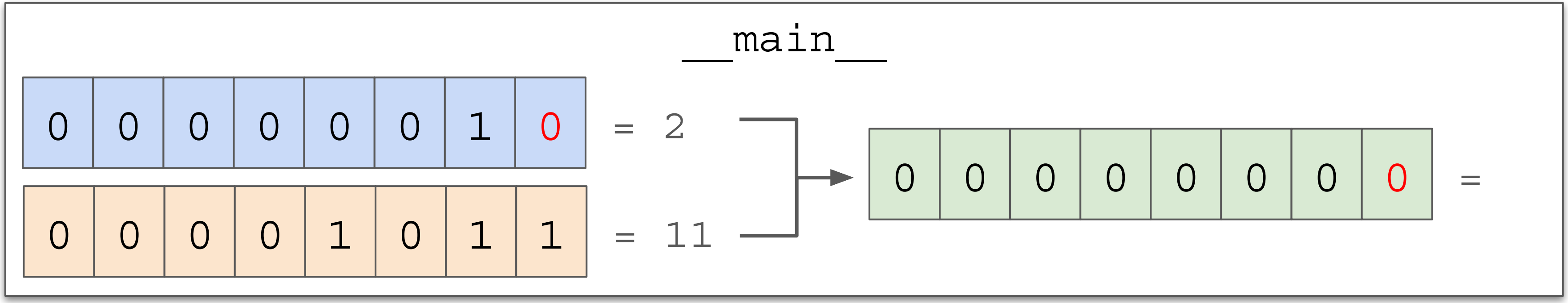
```
__main__
```

0	0	0	0	0	0	1	0	= 2
0	0	0	0	1	0	1	1	= 11

0	0	0	0	0	0	0	0	=

Complications

- Consider summation in **series**



__main__

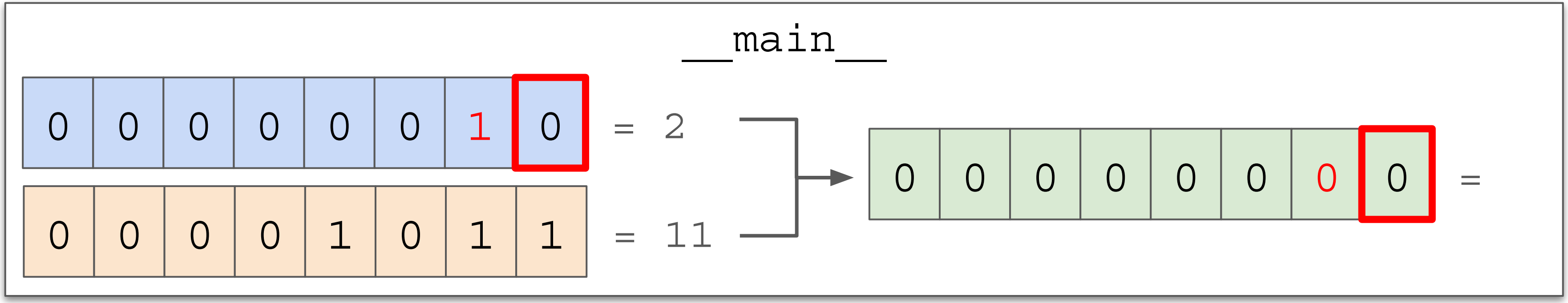
0 0 0 0 0 0 1 0 = 2

0 0 0 0 1 0 1 1 = 11

0 0 0 0 0 0 0 0 =

Complications

- Consider summation in **series**



__main__

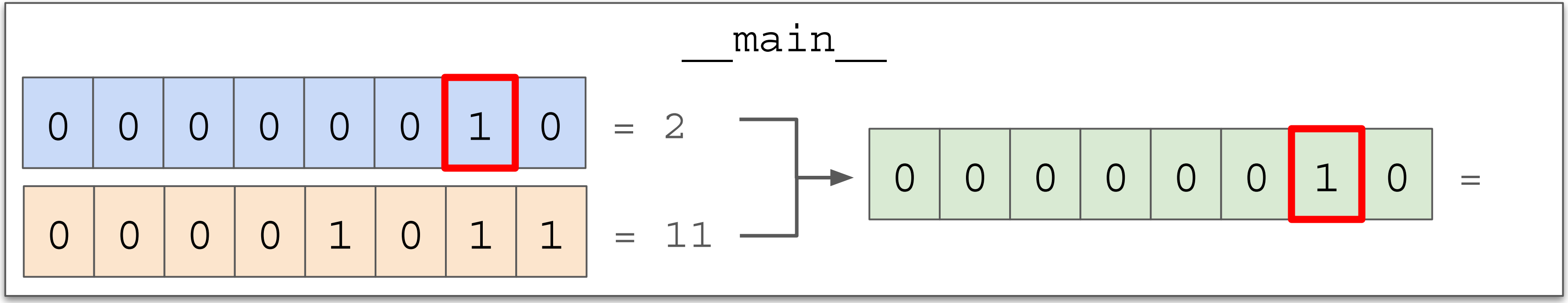
0 0 0 0 0 0 1 0 = 2

0 0 0 0 1 0 1 1 = 11

0 0 0 0 0 0 0 0 =

Complications

- Consider summation in **series**

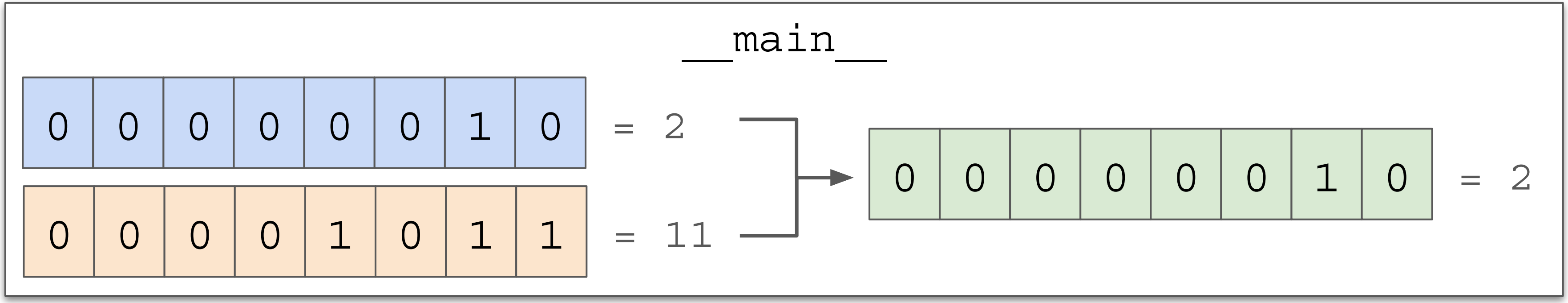


0	0	0	0	0	0	1	0	= 2
0	0	0	0	1	0	1	1	= 11

0	0	0	0	0	0	1	0	=

Complications

- Consider summation in **series**



__main__

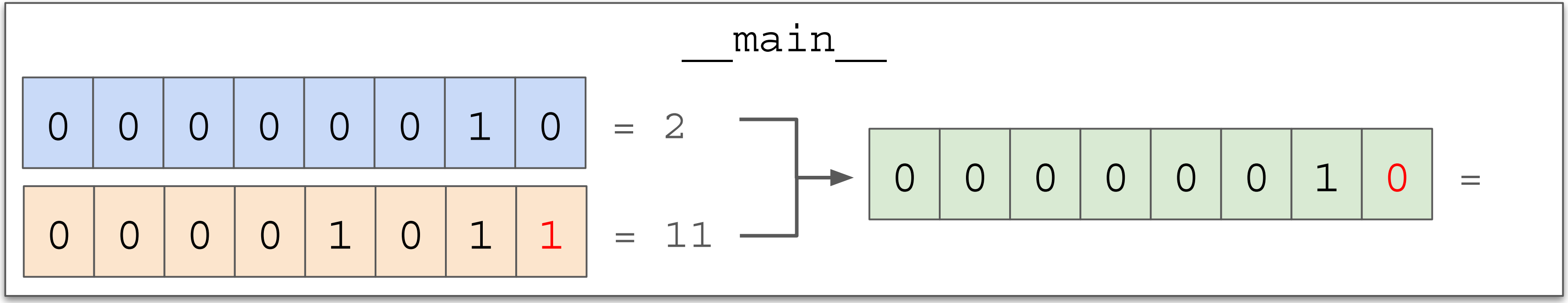
0 0 0 0 0 0 1 0 = 2

0 0 0 0 1 0 1 1 = 11

0 0 0 0 0 0 1 0 = 2

Complications

- Consider summation in **series**



__main__

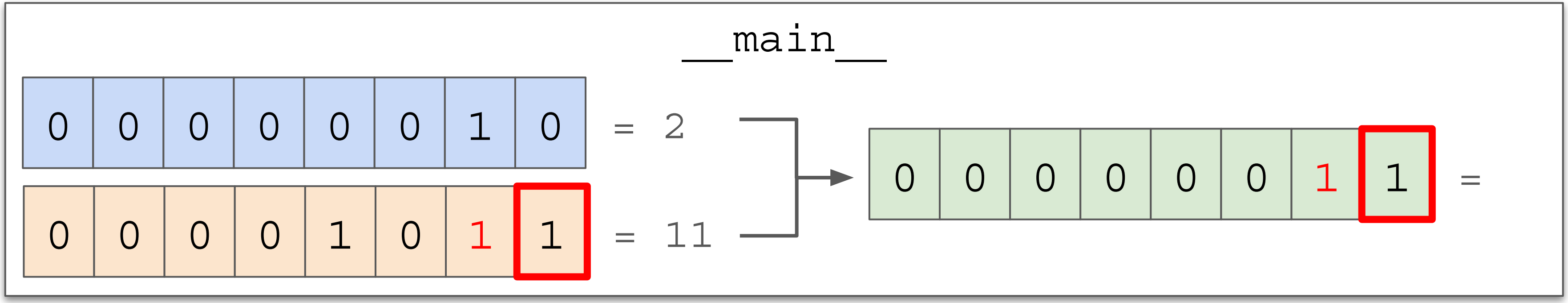
0 0 0 0 0 0 1 0 = 2

0 0 0 0 1 0 1 1 = 11

0 0 0 0 0 0 1 0 =

Complications

- Consider summation in **series**



__main__

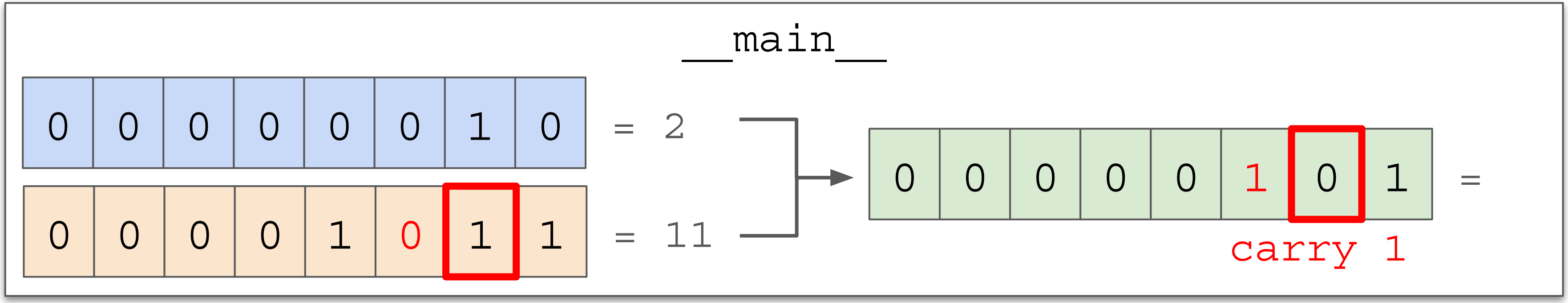
0 0 0 0 0 0 1 0 = 2

0 0 0 0 1 0 1 1 = 11

0 0 0 0 0 0 1 1 =

Complications

- Consider summation in **series**



__main__

0 0 0 0 0 0 1 0 = 2

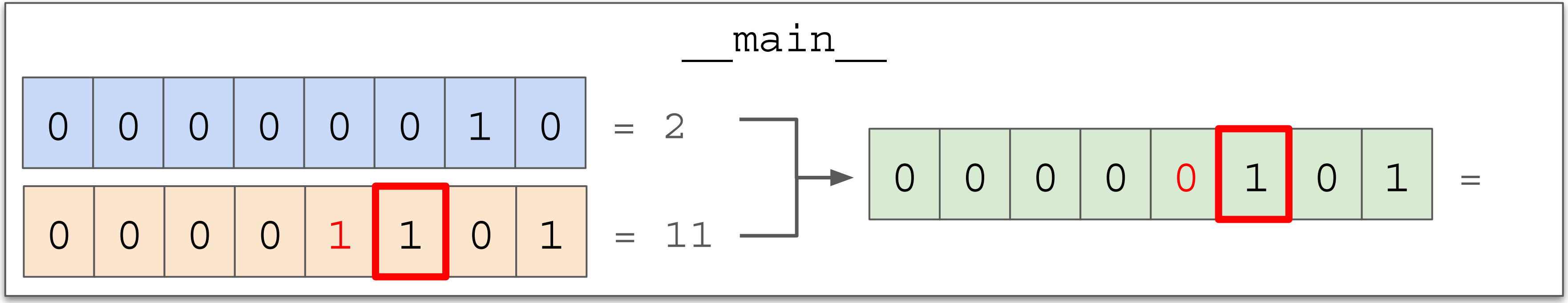
0 0 0 0 1 0 1 1 = 11

0 0 0 0 0 1 0 1 =

carry 1

Complications

- Consider summation in **series**



__main__

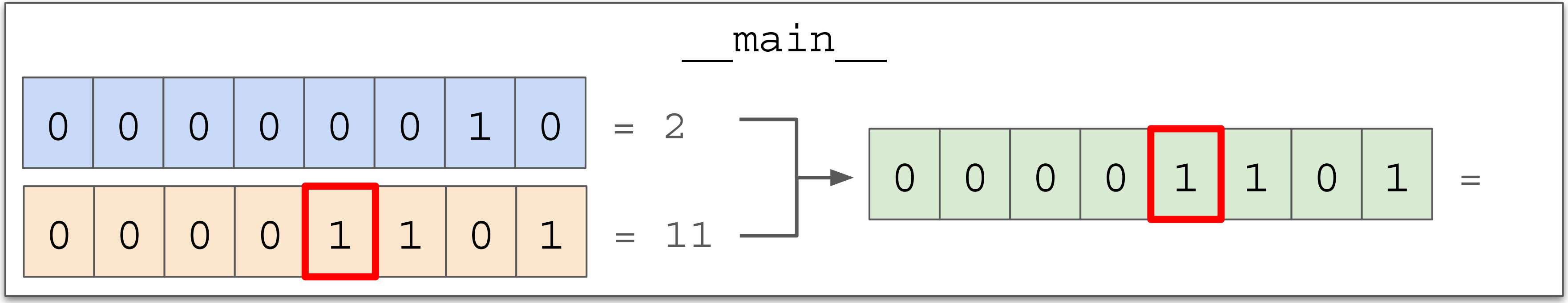
0 0 0 0 0 0 1 0 = 2

0 0 0 0 1 1 0 1 = 11

0 0 0 0 0 1 0 1 =

Complications

- Consider summation in **series**



__main__

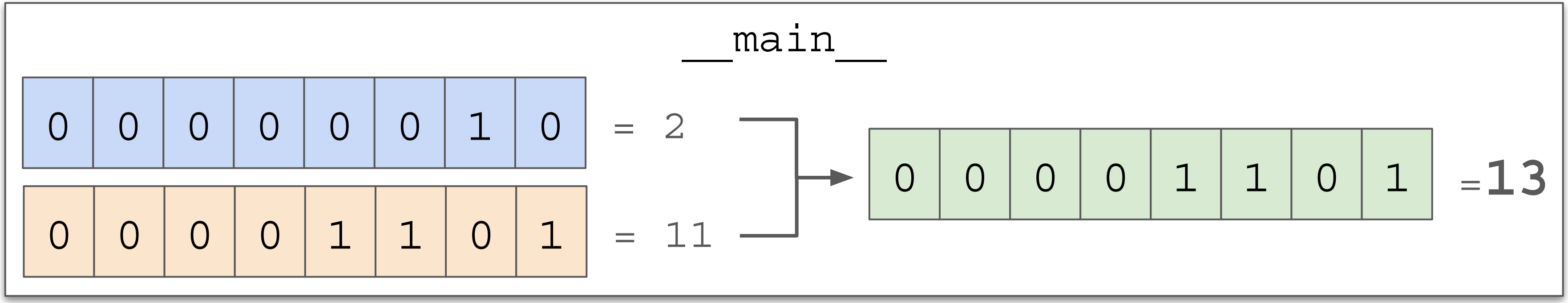
0 0 0 0 0 0 1 0 = 2

0 0 0 0 1 1 0 1 = 11

0 0 0 0 1 1 0 1 =

Complications

- Consider summation in **series**



__main__

0 0 0 0 0 0 1 0 = 2

0 0 0 0 1 1 0 1 = 11

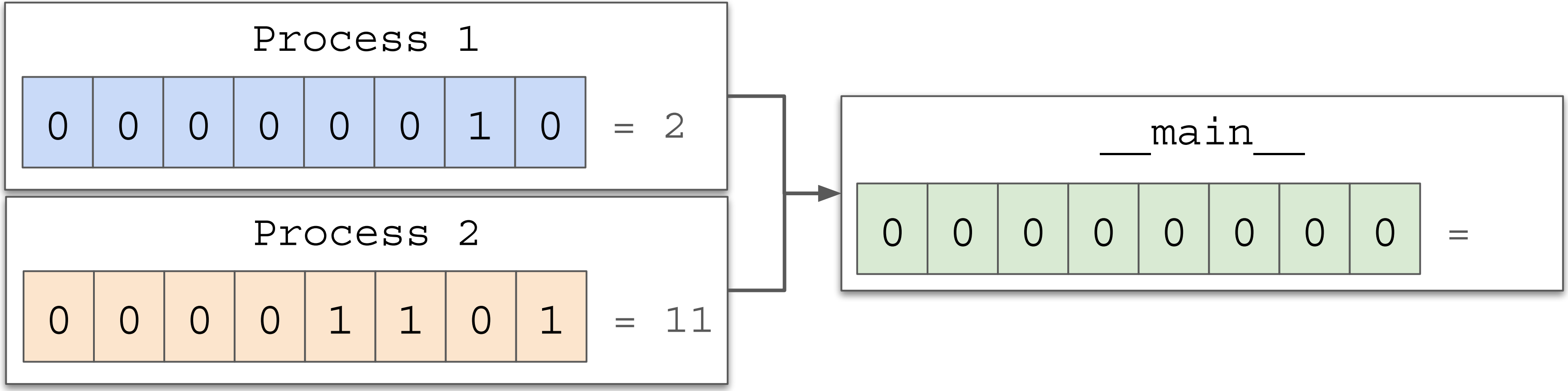
0 0 0 0 1 1 0 1 =13

Complications

- Consider summation in **parallel**
  - But processes each write to same place in memory

expected

0	0	0	0	1	1	0	1	=13



Process 1

0	0	0	0	0	0	1	0	= 2

Process 2

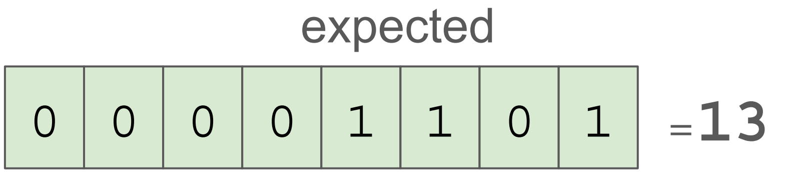
0	0	0	0	1	1	0	1	= 11

__main__

0	0	0	0	0	0	0	0	=

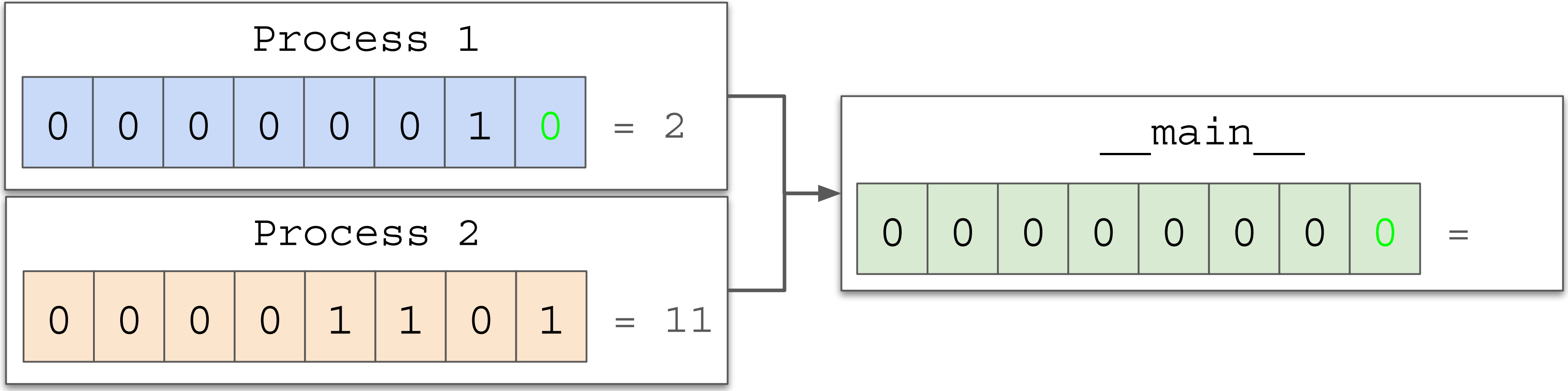
Complications

- Consider summation in **parallel**
  - But processes each write to same place in memory



expected

0 0 0 0 1 1 0 1 =13



Process 1

0 0 0 0 0 0 1 0 = 2

Process 2

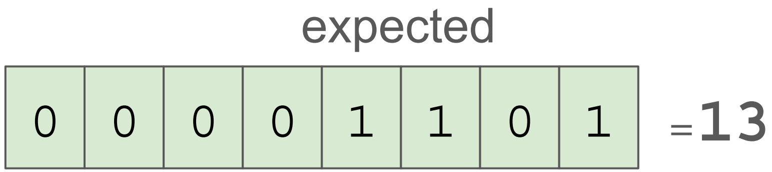
0 0 0 0 1 1 0 1 = 11

__main__

0 0 0 0 0 0 0 0 =

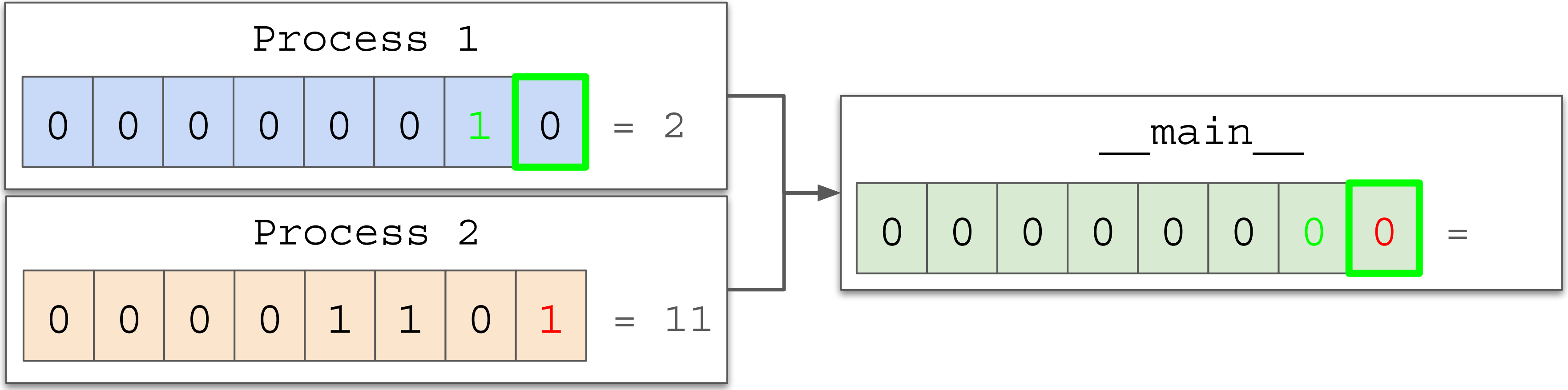
Complications

- Consider summation in **parallel**
  - But processes each write to same place in memory



expected

0	0	0	0	1	1	0	1	=13



Process 1

0	0	0	0	0	0	1	0	= 2

Process 2

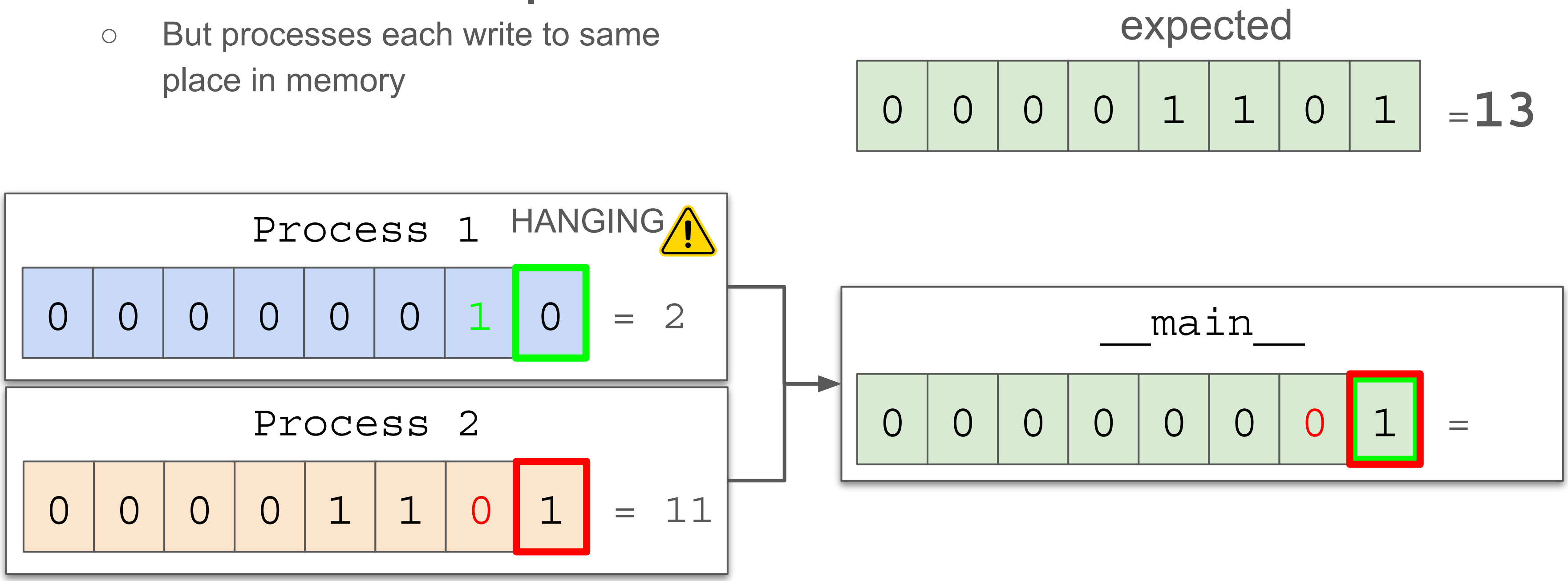
0	0	0	0	1	1	0	1	= 11

__main__

0	0	0	0	0	0	0	0	=

Complications

- Consider summation in **parallel**
    - But processes each write to same place in memory



expected

0 0 0 0 1 1 0 1 =13

Process 1 HANGING ⚠

0 0 0 0 0 0 1 0 = 2

Process 2

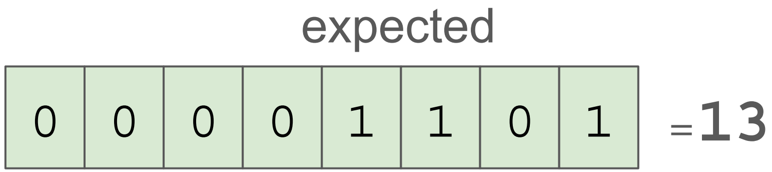
0 0 0 0 1 1 0 1 = 11

__main__

0 0 0 0 0 0 0 1 =

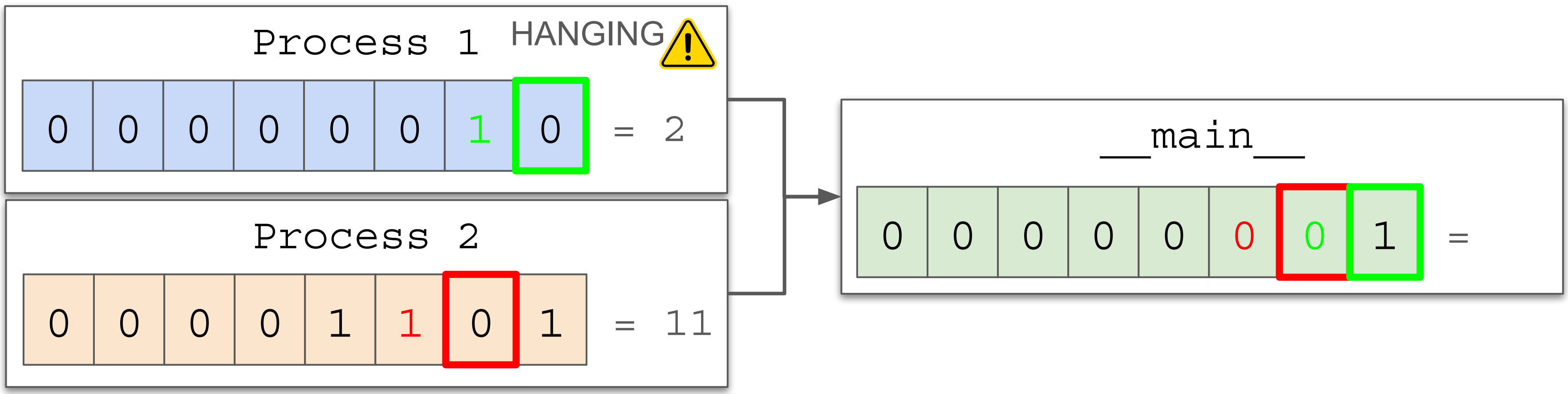
Complications

- Consider summation in **parallel**
    - But processes each write to same place in memory



expected

0 0 0 0 1 1 0 1 =13



Process 1 HANGING ⚠

0 0 0 0 0 0 1 0 = 2

Process 2

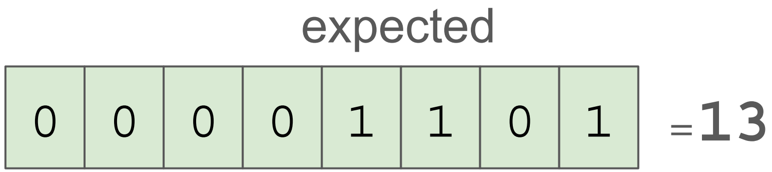
0 0 0 0 1 1 0 1 = 11

__main__

0 0 0 0 0 0 0 1 =

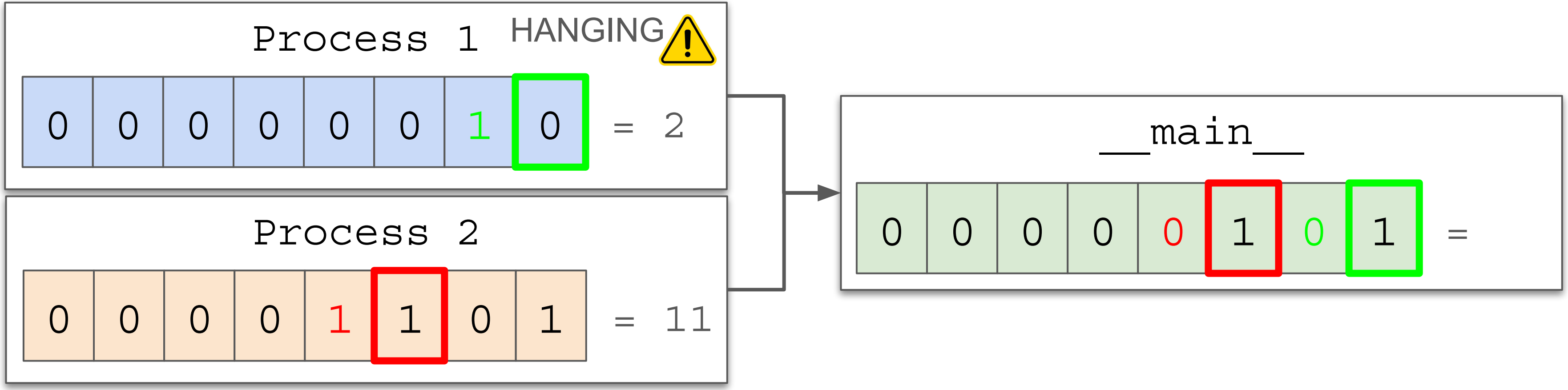
Complications

- Consider summation in **parallel**
  - But processes each write to same place in memory



expected

0 0 0 0 1 1 0 1 =13



Process 1 HANGING ⚠

0 0 0 0 0 0 1 0 = 2

Process 2

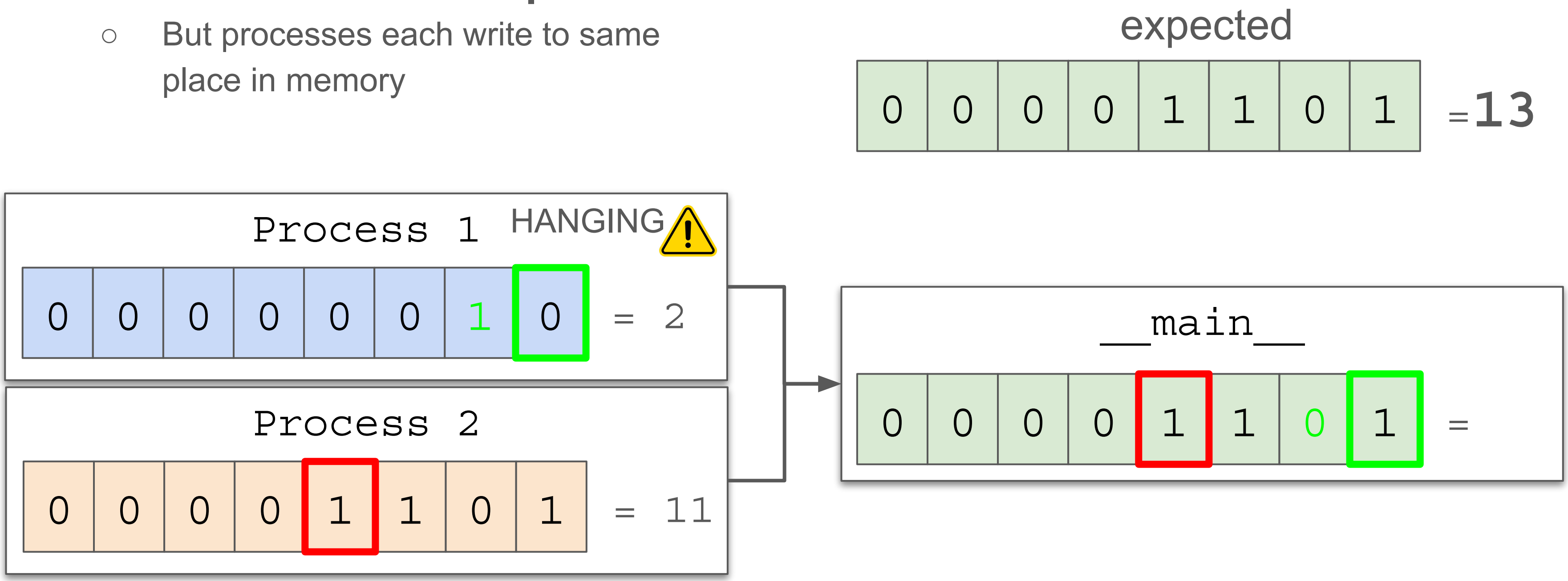
0 0 0 0 1 1 0 1 = 11

__main__

0 0 0 0 0 1 0 1 =

Complications

- Consider summation in **parallel**
  - But processes each write to same place in memory



expected

0 0 0 0 1 1 0 1 =13

Process 1 HANGING ⚠

0 0 0 0 0 0 1 0 = 2

Process 2

0 0 0 0 1 1 0 1 = 11

__main__

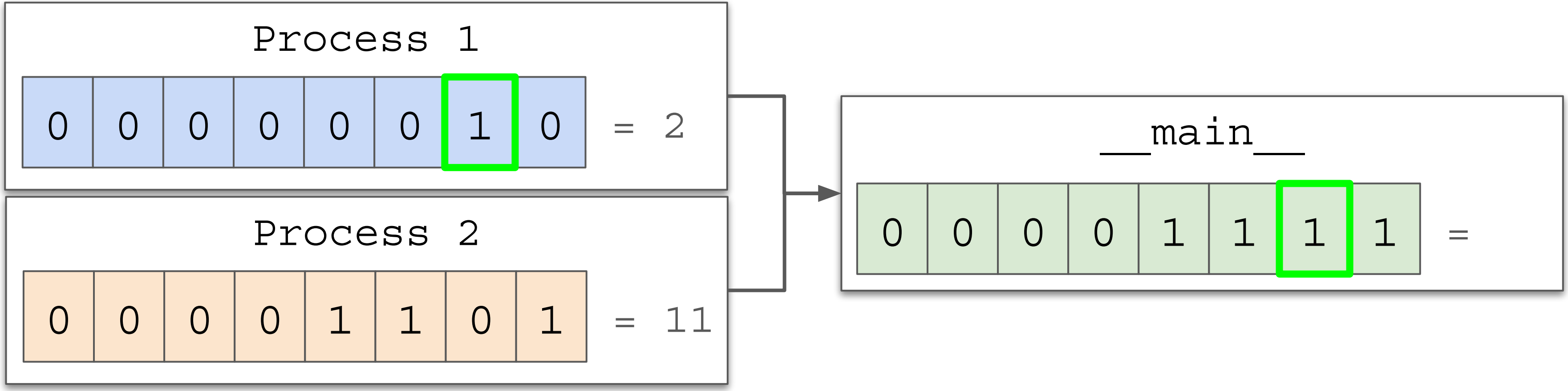
0 0 0 0 1 1 0 1 =

Complications

- Consider summation in **parallel**
  - But processes each write to same place in memory

expected

0	0	0	0	1	1	0	1	=13



Process 1

0	0	0	0	0	0	1	0	= 2

Process 2

0	0	0	0	1	1	0	1	= 11

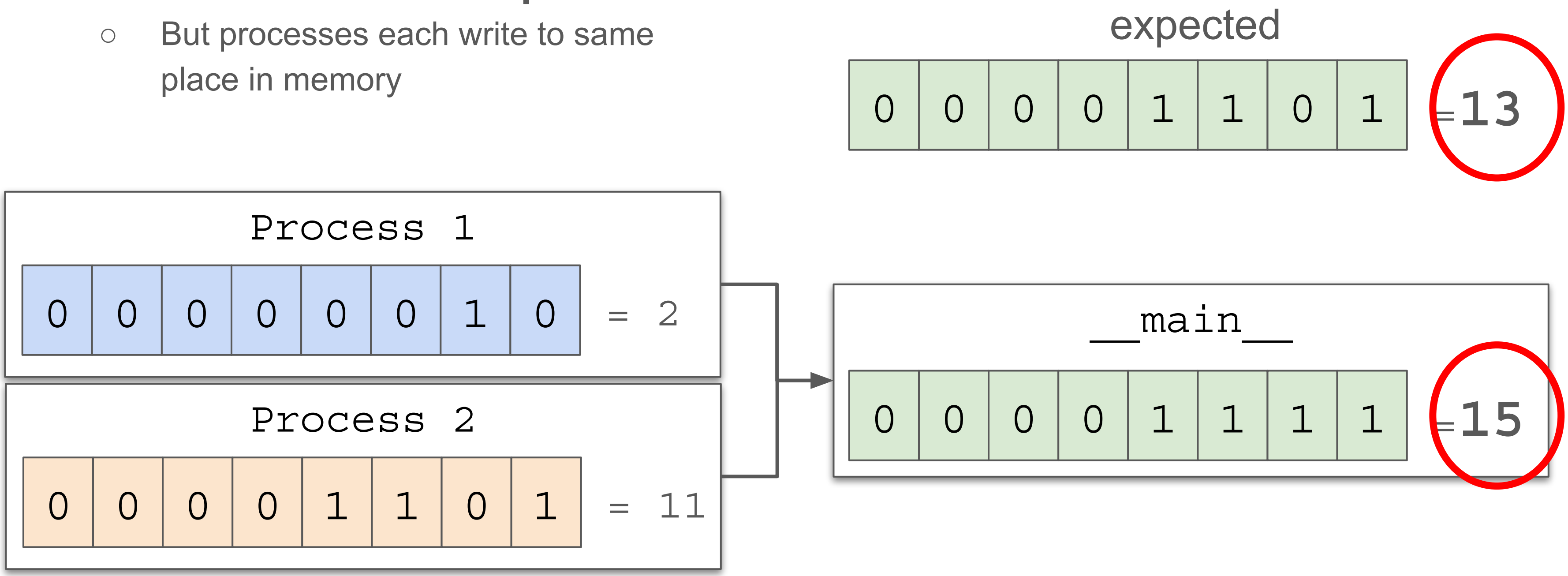
__main__

0	0	0	0	1	1	1	1	=

Complications

“Race condition”

- Consider summation in **parallel**
    - But processes each write to same place in memory



expected

0	0	0	0	1	1	0	1	=13

Process 1

0	0	0	0	0	0	1	0	= 2

Process 2

0	0	0	0	1	1	0	1	= 11

__main__

0	0	0	0	1	1	1	1	=15

File Management

- Race conditions could fatally corrupt files
- BUT: Drives are serial
  - Except for niche cases like RAID
- 1 “head”: reads and writes

- How parallel can help:
  - Main process reads data and sends to processes
  - Processes put results in a queue
  - Main process retrieves results from queue
- Only helpful for long-running calculations

(snippet) scripts/examples/fileio.py

```
for process in processes: process.start()

# Fill the input queue
for line in read(filename): input_queue.put(line)
for _ in processes:
    input_queue.put(None) # end-of-queue signal

# Get the results from the output queue
ndone = 0
while ndone < len(processes):
    process_errors() # Check for errors
    if output_queue.empty(): continue
    if output_queue.get() is None: ndone += 1
```

With great power comes great code

- Parallel sum (“reduce”)
- Suppose we want: `result = 0 + 1 + 2 + ... + n`
  - Say, `n` = 2x10^9 😈
  - In serial: `sum(range(int(2e9)) = 1999999999000000000`
- Try it yourself! `workspace/reduce.py`
  - Must use exactly 4 processes, and be able to prove it
  - Allowed modules: `multiprocessing`, `NumPy`, `time`, and `queue`
  - Cannot edit code marked with “`DO NOT EDIT`”
  - Local machines only
  - Everything else is allowed (go crazy)
  - **First code to execute in <1.55 seconds wins $10**
  - Hints:
    - Remember: pre-fetching!
    - Allocating too much memory on Ubuntu makes your kernel crash

(github page)

My solution

- “Chunking”: using 10 chunks

```
def func(input_queue, output_queue):
    while True:
        try: output_queue.put(
                np.einsum(
                    'i->',
                    np.arange(*input_queue.get()),
                 )
        )
        except (queue.Empty, TypeError):
            output_queue.put(None)
            return
```

```
$ python3 reduce.py
That took 1.5385827461723238 seconds
Correct result!
```

```
n = int(2e9)
nprocs = 4

input_queue = multiprocessing.Queue()
output_queue = multiprocessing.Queue()

chunklen = n // nchunks

for i in range(nchunks):
    input_queue.put([i*chunklen, (i+1)*chunklen])
for _ in range(nprocs): input_queue.put(None)

processes = [multiprocessing.Process(
    target = func,
    args = (input_queue, output_queue),
    daemon = True,
) for _ in range(nprocs)]

for process in processes: process.start()
result = 0
ndone = 0
while ndone < len(processes):
    if output_queue.empty(): continue
    r = output_queue.get()
    if r is None:
        ndone += 1
        continue
result += r
```