Setting Python Threading Free

David Hoese

Who am I?

- Software Developer at UW Space Science and Engineering Center (SSEC)
- Open Source Maintainer and Contributor
- Processing tools and weather instrument data ingest

My experience with parallel processing

- Processing of large image-like arrays
  Chunk data to process in parallel (dask)
- Low-level Cython/C extensions
- Lots of data files - embarrassingly parallel
- Asynchronous event processing (asyncio)
- Communicate with multiple instruments from one command

"Python is slow"

"Python is slow"

- "Glue language"

"Python is slow"

- "Glue language"
- Use C extensions where possible (past talk on Cython)

"Python is slow"

- "Glue language"
- Use C extensions where possible (past talk on Cython)
- Parallel processing → Threads

"Python is slow"

- "Glue language"
- Use C extensions where possible (past talk on Cython)
- Parallel processing → Threads
  - "It's not real threading"

"Python is slow"

- "Glue language"
- Use C extensions where possible (past talk on Cython)
- Parallel processing → Threads
  - "It's not real threading"
  - "Global Interpreter Lock…blah blah blah"

"Python is slow"

- "Glue language"
- Use C extensions where possible (past talk on Cython)
- Parallel processing → Threads
  - "It's not real threading"
  - "Global Interpreter Lock…blah blah blah"
- Use multiprocessing instead of threading - **no longer necessary**

this talk

Python's simple beginnings

- Scripting language - easier than C
- Performance not the priority
- Over 35 years old
- Performance "speed bumps"
  - Everything is an object
    - Large objects, no primitives
    - Slow to move, hard to cache
  - Reference Counting/Garbage Collection
  - Interpreted
    - Convert Python to bytecode

```c
struct _longobject {
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
    _PyLongValue long_value;
};
```

Interpreted

```python
def some_func(input):
    result = []
    for row in input:
        weight = get_weight(row)
        avg = mean(row)
        res = avg * weight
        result.append(res)
    return max(result)
```

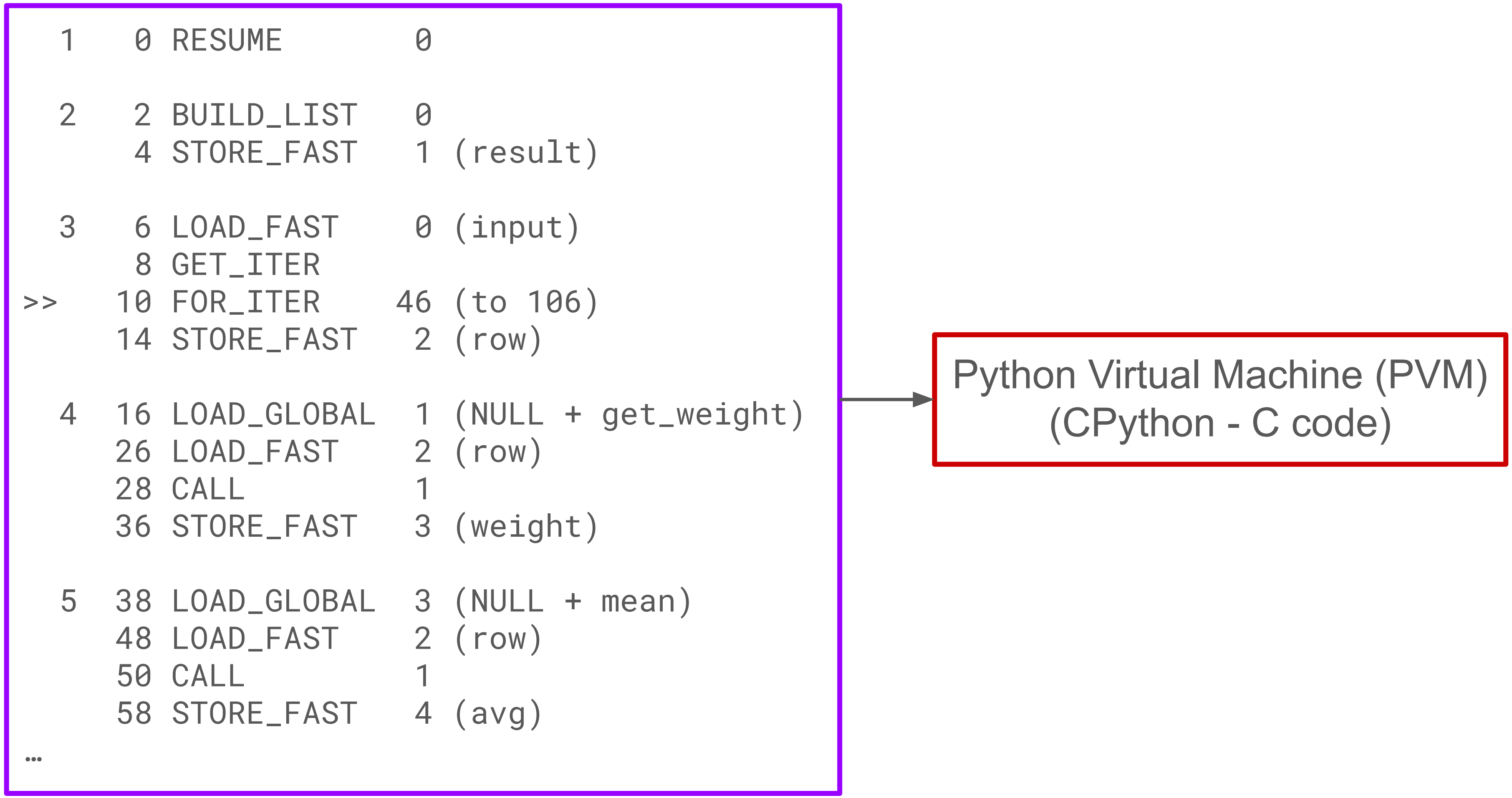
```
  1     0 RESUME          0

  2     2 BUILD_LIST      0
        4 STORE_FAST      1 (result)

  3     6 LOAD_FAST       0 (input)
        8 GET_ITER
>>     10 FOR_ITER       46 (to 106)
       14 STORE_FAST      2 (row)

  4    16 LOAD_GLOBAL     1 (NULL + get_weight)
       26 LOAD_FAST       2 (row)
       28 CALL            1
       36 STORE_FAST      3 (weight)

  5    38 LOAD_GLOBAL     3 (NULL + mean)
       48 LOAD_FAST       2 (row)
       50 CALL            1
       58 STORE_FAST      4 (avg)
…
```



```
  1     0 RESUME          0

  2     2 BUILD_LIST      0
        4 STORE_FAST      1 (result)

  3     6 LOAD_FAST       0 (input)
        8 GET_ITER
 >>    10 FOR_ITER       46 (to 106)
       14 STORE_FAST      2 (row)

  4    16 LOAD_GLOBAL     1 (NULL + get_weight)
       26 LOAD_FAST       2 (row)
       28 CALL            1
       36 STORE_FAST      3 (weight)

  5    38 LOAD_GLOBAL     3 (NULL + mean)
       48 LOAD_FAST       2 (row)
       50 CALL            1
       58 STORE_FAST      4 (avg)
…
```

Python's simple beginnings

- Scripting language - easier than C
- Performance not the priority
- Over 35 years old
- Performance "speed bumps"
  - Everything is an object
    - Large objects, no primitives
    - Slow to move, hard/impossible to cache
  - Reference Counting/Garbage Collection
  - Interpreted
    - Convert Python to bytecode
    - Hard to specialize instructions

Python's simple beginnings

- Scripting language - easier than C
- Performance not the priority
- Over 35 years old
- Performance "speed bumps"
    - Everything is an object
        - Large objects, no primitives
        - Slow to move, hard/impossible to cache
    - Reference Counting/Garbage Collection
    - Interpreted
        - Convert Python to bytecode
        - Hard to specialize instructions
        - Global Interpreter Lock (GIL)

Free-threading to the rescue!

Understanding free-threading

- What is free-threading?
    - Allow Python threads to execute in parallel by removing the GIL
- Concurrency versus Parallelism - "at the same time"
- Threads versus Processes - enable "parallel" work
- How it works now
- Removing the GIL
- What workflows should improve
- Examples

Parallelism versus Concurrency

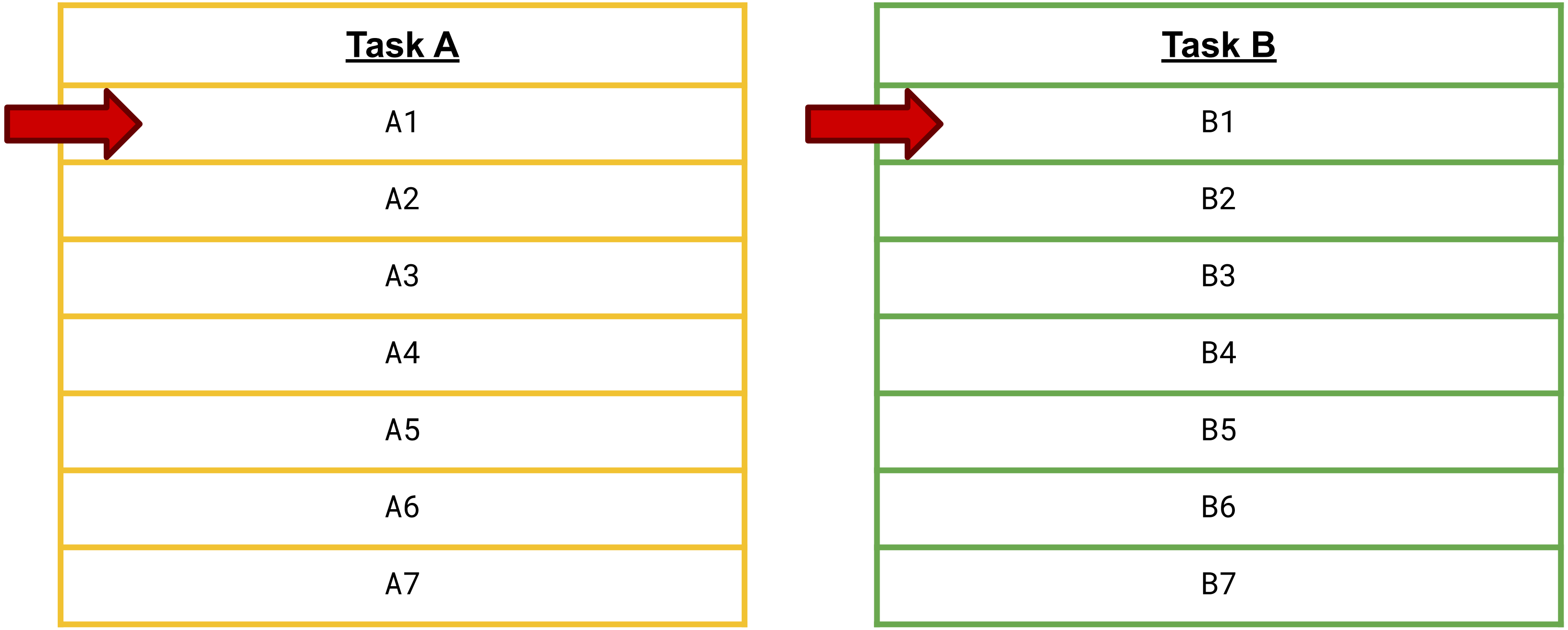
- Multiple sets of tasks/commands
- Single application (process) versus multiple executions
- Single core versus multiple cores
- Running at the "same time"

Parallelism versus Concurrency

Task A
A1
A2
A3
A4
A5
A6
A7

Task B
B1
B2
B3
B4
B5
B6
B7

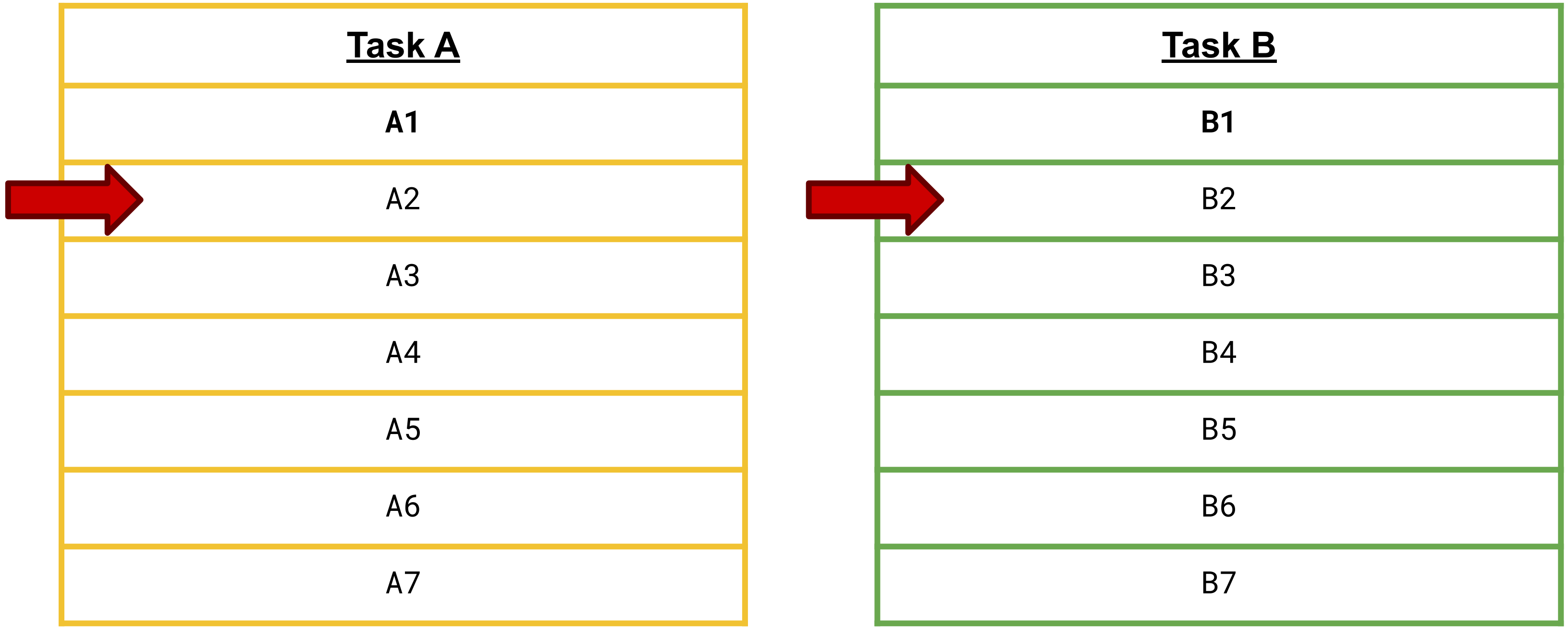
Parallelism - Simultaneous



Task A
A1
A2
A3
A4
A5
A6
A7

Task B
B1
B2
B3
B4
B5
B6
B7

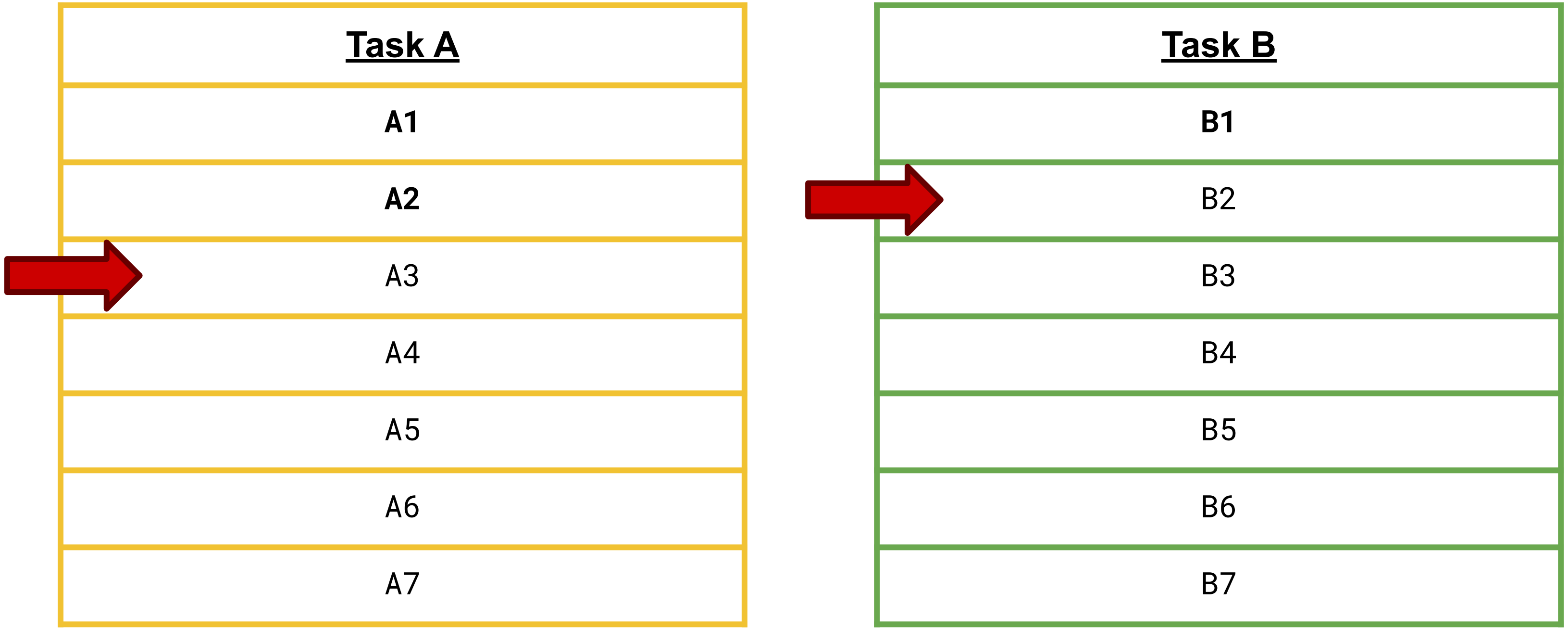
Parallelism - Simultaneous



Task A
A1
A2
A3
A4
A5
A6
A7

Task B
B1
B2
B3
B4
B5
B6
B7

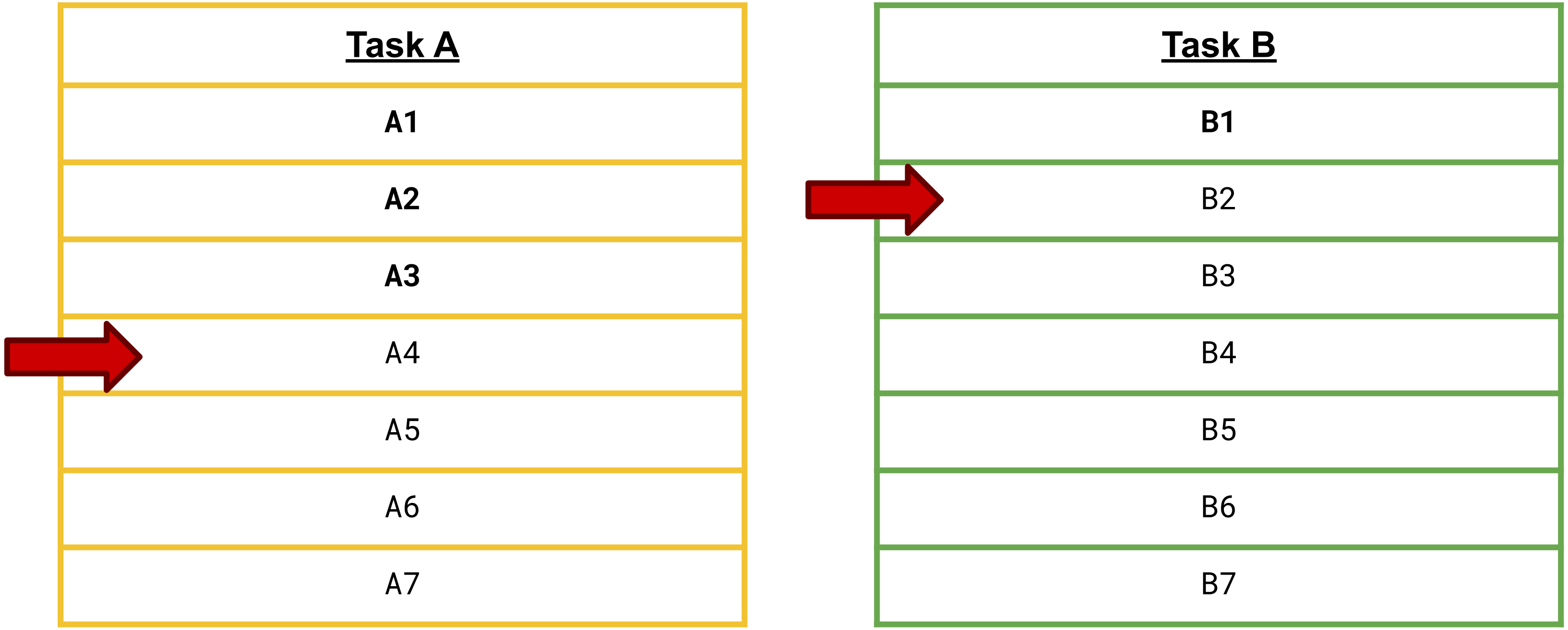
Parallelism - Simultaneous



Task A
A1
A2
A3
A4
A5
A6
A7

Task B
B1
B2
B3
B4
B5
B6
B7

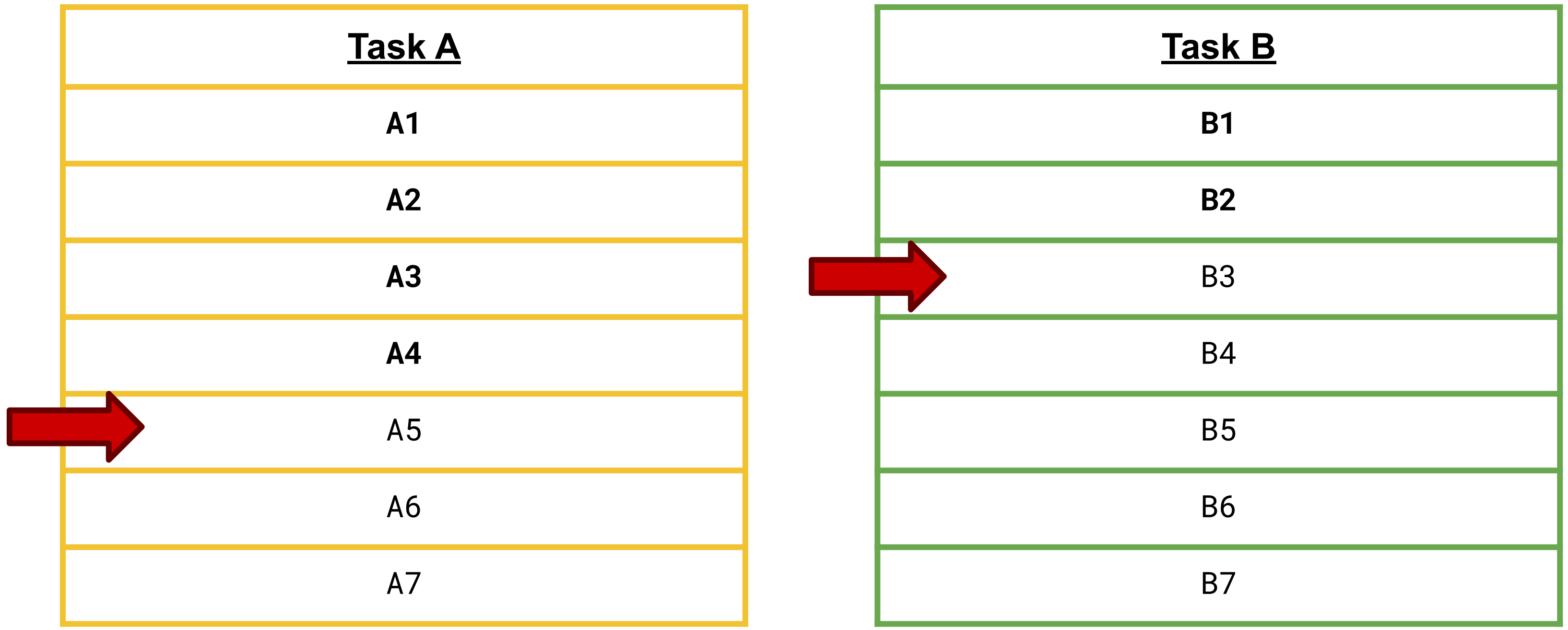
Parallelism - Simultaneous



Task A
A1
A2
A3
A4
A5
A6
A7

Task B
B1
B2
B3
B4
B5
B6
B7

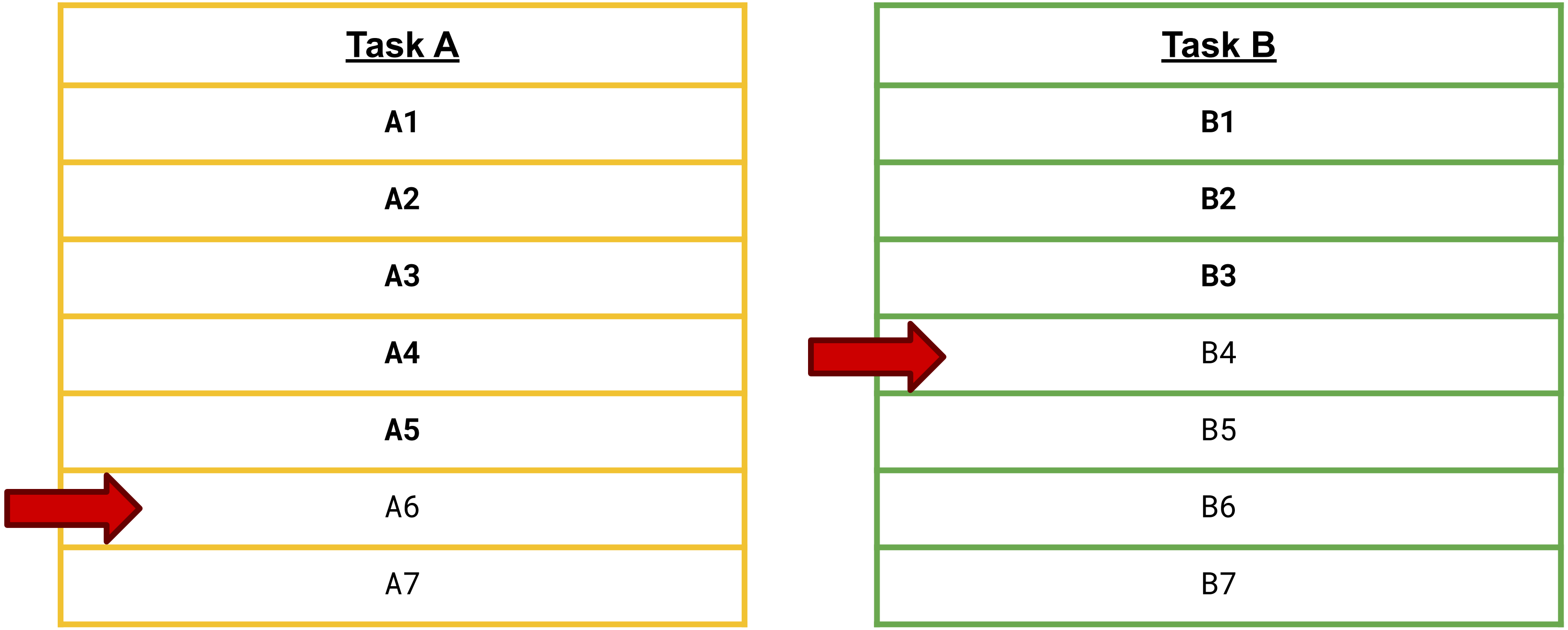
Parallelism - Simultaneous



Task A
A1
A2
A3
A4
A5
A6
A7

Task B
B1
B2
B3
B4
B5
B6
B7

Parallelism - Simultaneous



Task A
A1
A2
A3
A4
A5
A6
A7

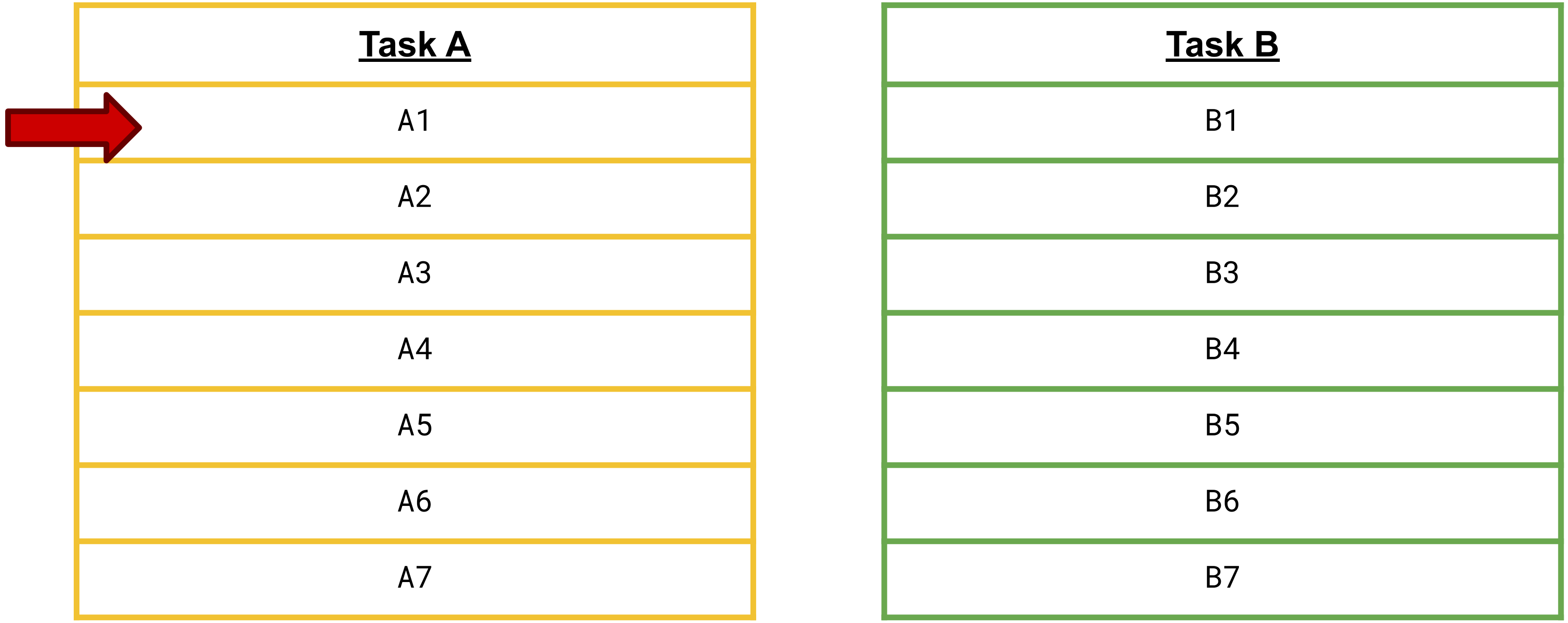
Task B
B1
B2
B3
B4
B5
B6
B7

Concurrency - Interleave

Task A
A1
A2
A3
A4
A5
A6
A7

Task B
B1
B2
B3
B4
B5
B6
B7

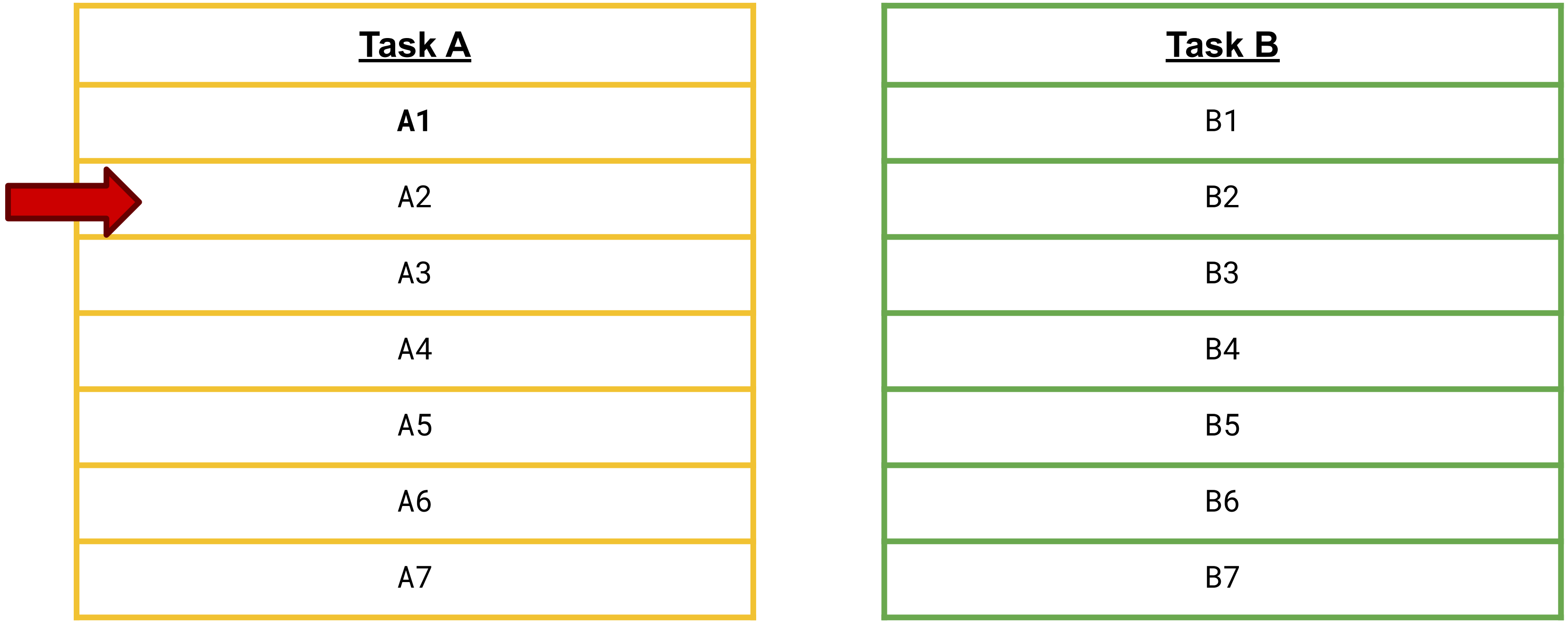
Concurrency - Interleave



Task A
A1
A2
A3
A4
A5
A6
A7

Task B
B1
B2
B3
B4
B5
B6
B7

Concurrency - Interleave



Task A
A1
A2
A3
A4
A5
A6
A7

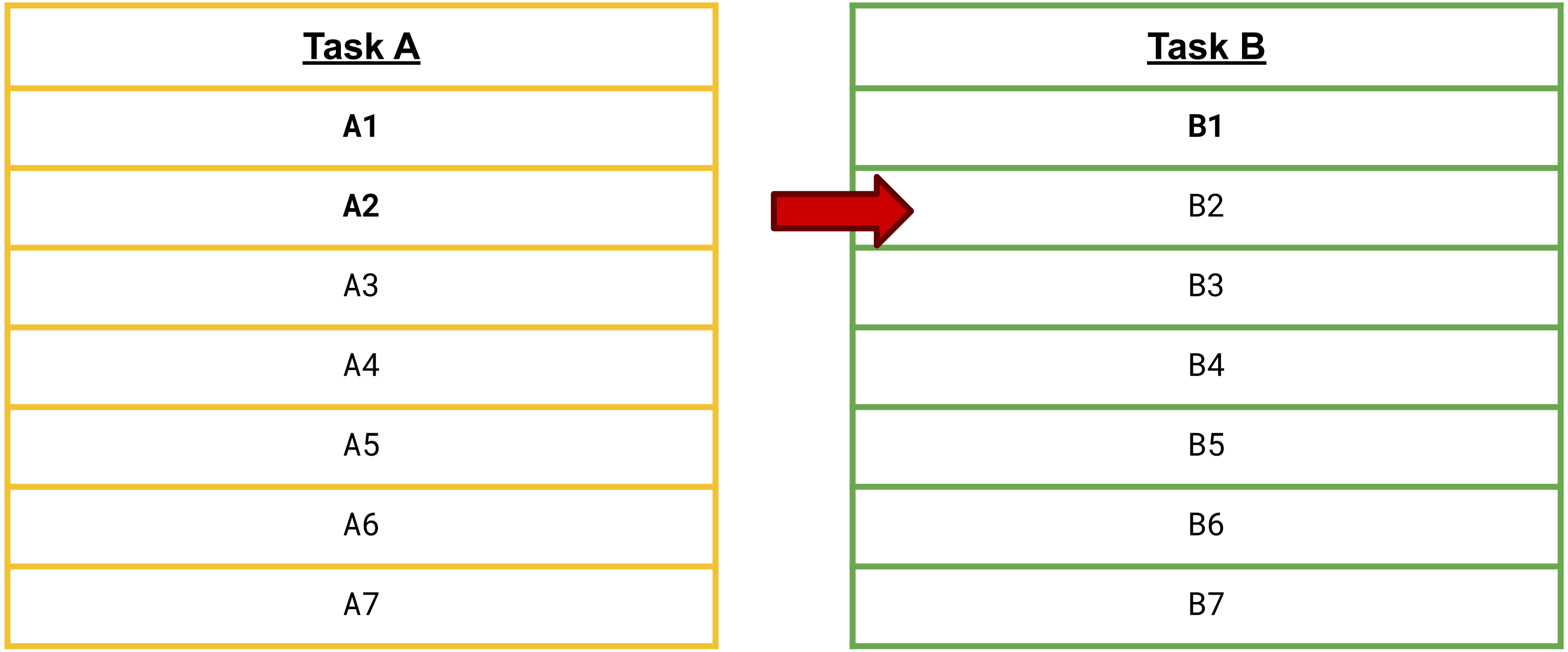
Task B
B1
B2
B3
B4
B5
B6
B7

Concurrency - Interleave

Task A
A1
A2
A3
A4
A5
A6
A7

Task B
B1
B2
B3
B4
B5
B6
B7

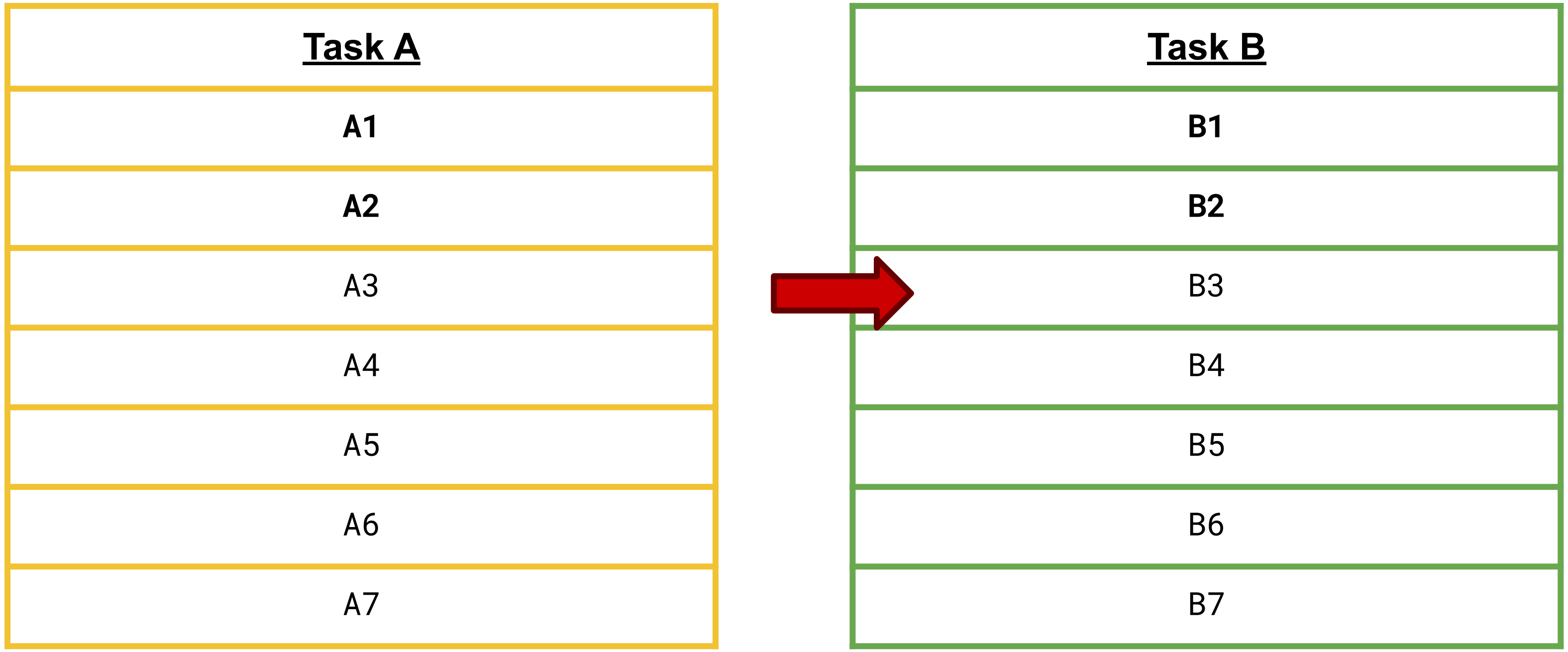
Concurrency - Interleave



Task A
A1
A2
A3
A4
A5
A6
A7

Task B
B1
B2
B3
B4
B5
B6
B7

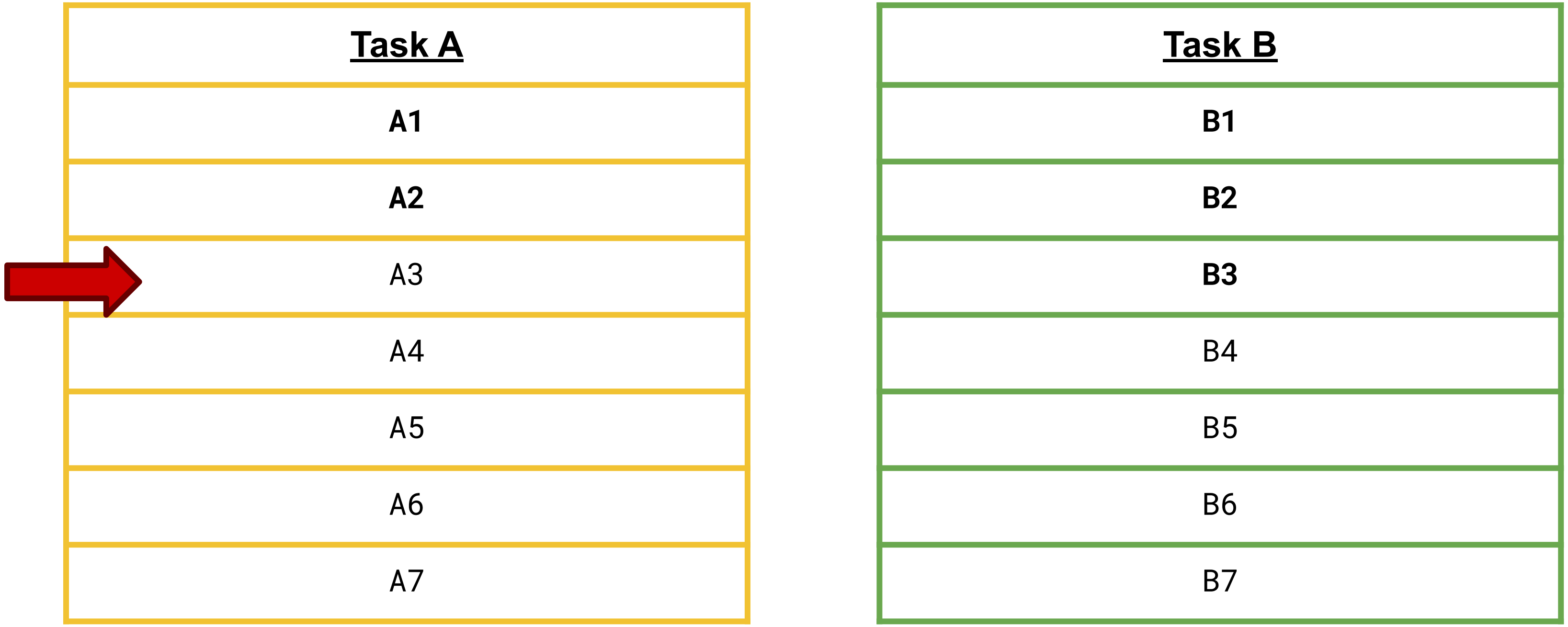
Concurrency - Interleave



Task A
A1
A2
A3
A4
A5
A6
A7

Task B
B1
B2
B3
B4
B5
B6
B7

Concurrency - Interleave



Task A
A1
A2
A3
A4
A5
A6
A7

Task B
B1
B2
B3
B4
B5
B6
B7

Concurrency - Interleave

	Task A
	A1
	A2
	A3
➡	A4
	A5
	A6
	A7

Task B
B1
B2
B3
B4
B5
B6
B7

Parallelism

- Accomplish multiple things at once
- Achieve more with more cores

Examples

- Python multiprocessing
- Linux multiple processes
- Python threading with no GIL (free-threading)

Concurrency

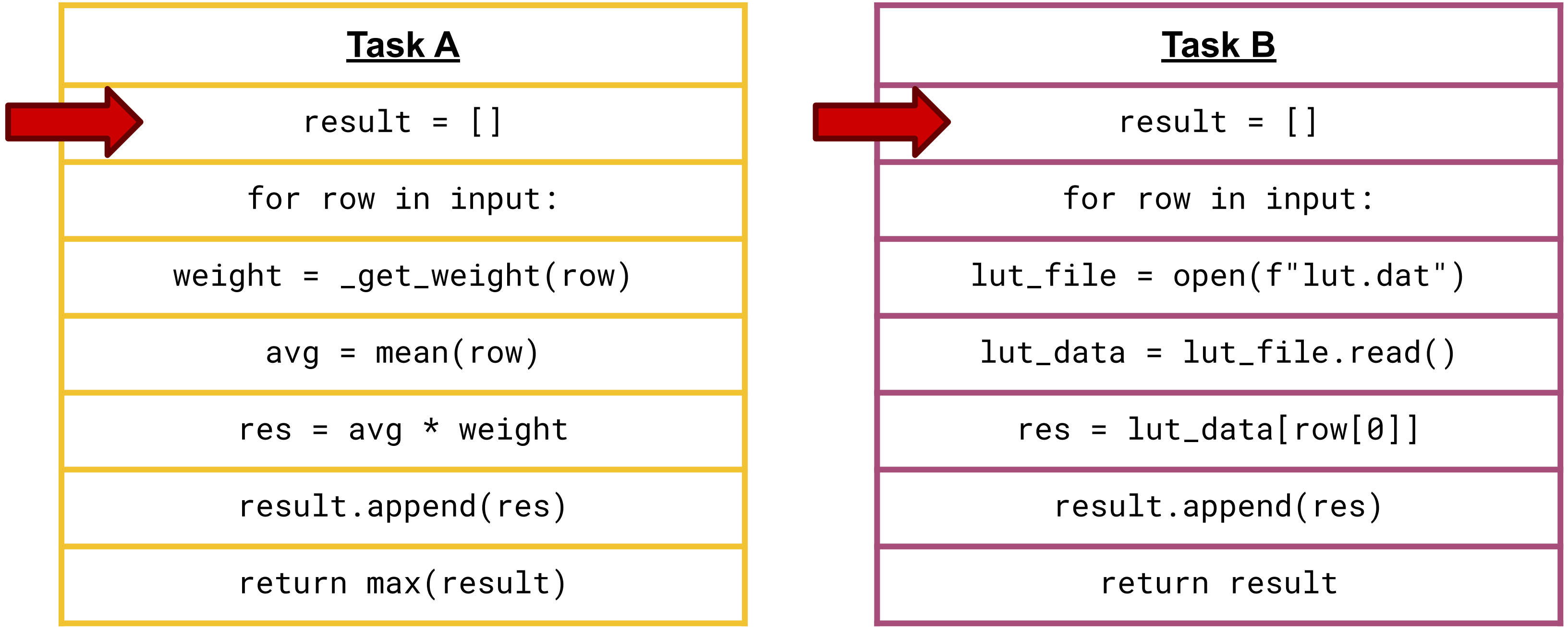
- Accomplish multiple things at once
- Achieve more on limited cores

Examples

- Python threading with GIL - acts as a single executor
- Python's (basic) asyncio
- Linux context switching

Possible to have concurrent and parallel tasks

Global Interpreter Lock (GIL)



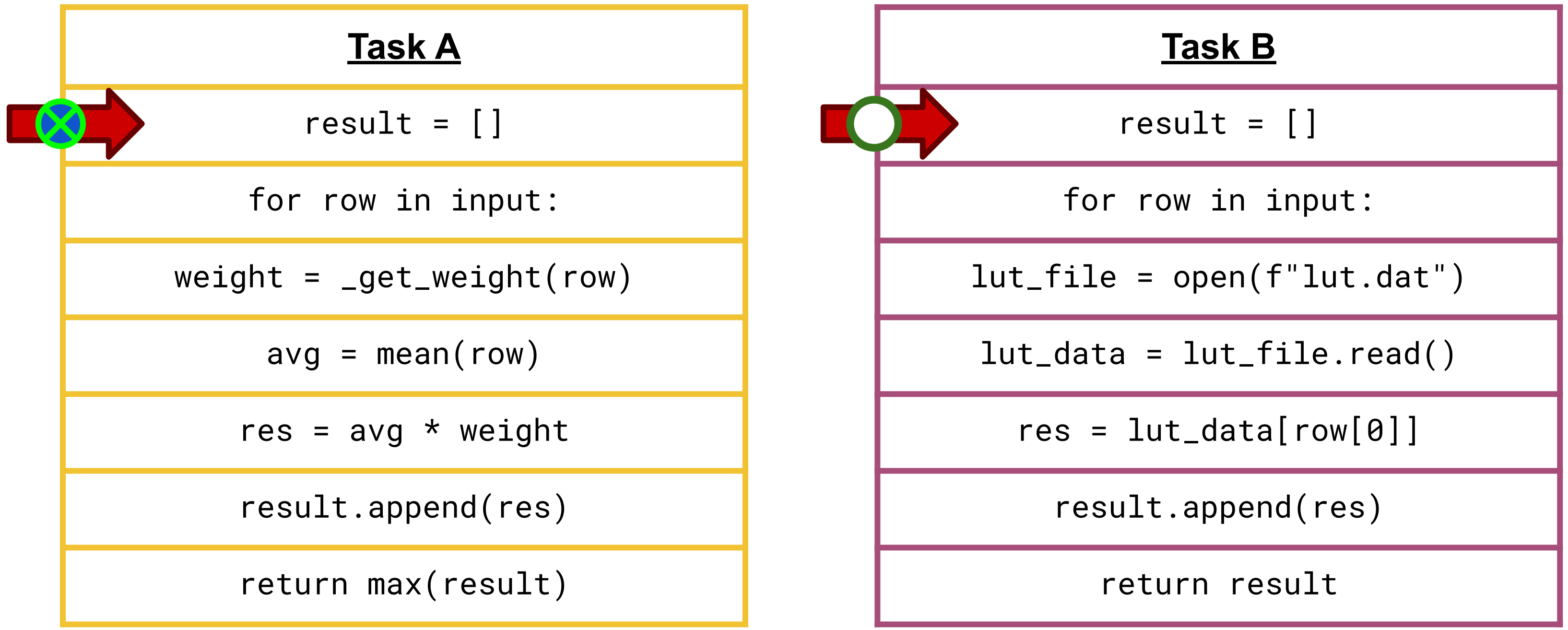
Task A

```
result = []
for row in input:
weight = _get_weight(row)
avg = mean(row)
res = avg * weight
result.append(res)
return max(result)
```

Task B

```
result = []
for row in input:
lut_file = open(f"lut.dat")
lut_data = lut_file.read()
res = lut_data[row[0]]
result.append(res)
return result
```

Global Interpreter Lock (GIL)



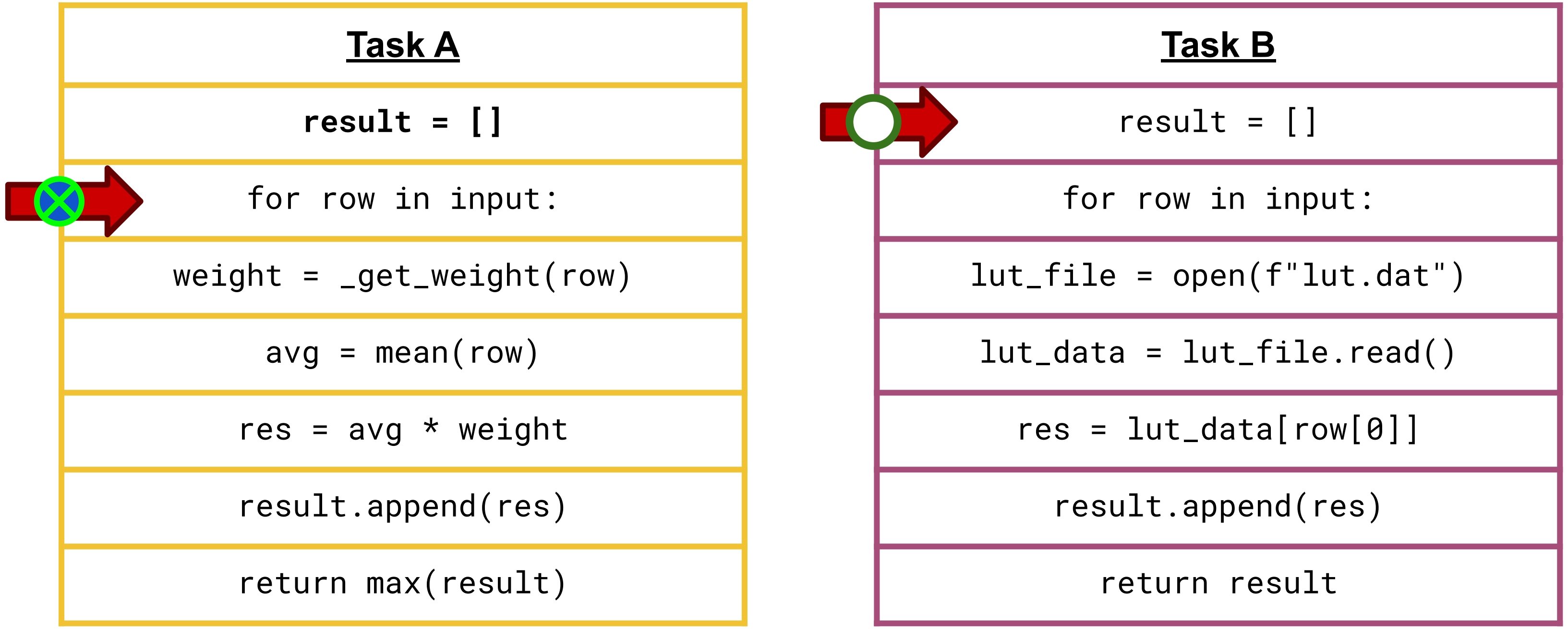
Task A

```
result = []
for row in input:
weight = _get_weight(row)
avg = mean(row)
res = avg * weight
result.append(res)
return max(result)
```

Task B

```
result = []
for row in input:
lut_file = open(f"lut.dat")
lut_data = lut_file.read()
res = lut_data[row[0]]
result.append(res)
return result
```

Global Interpreter Lock (GIL)



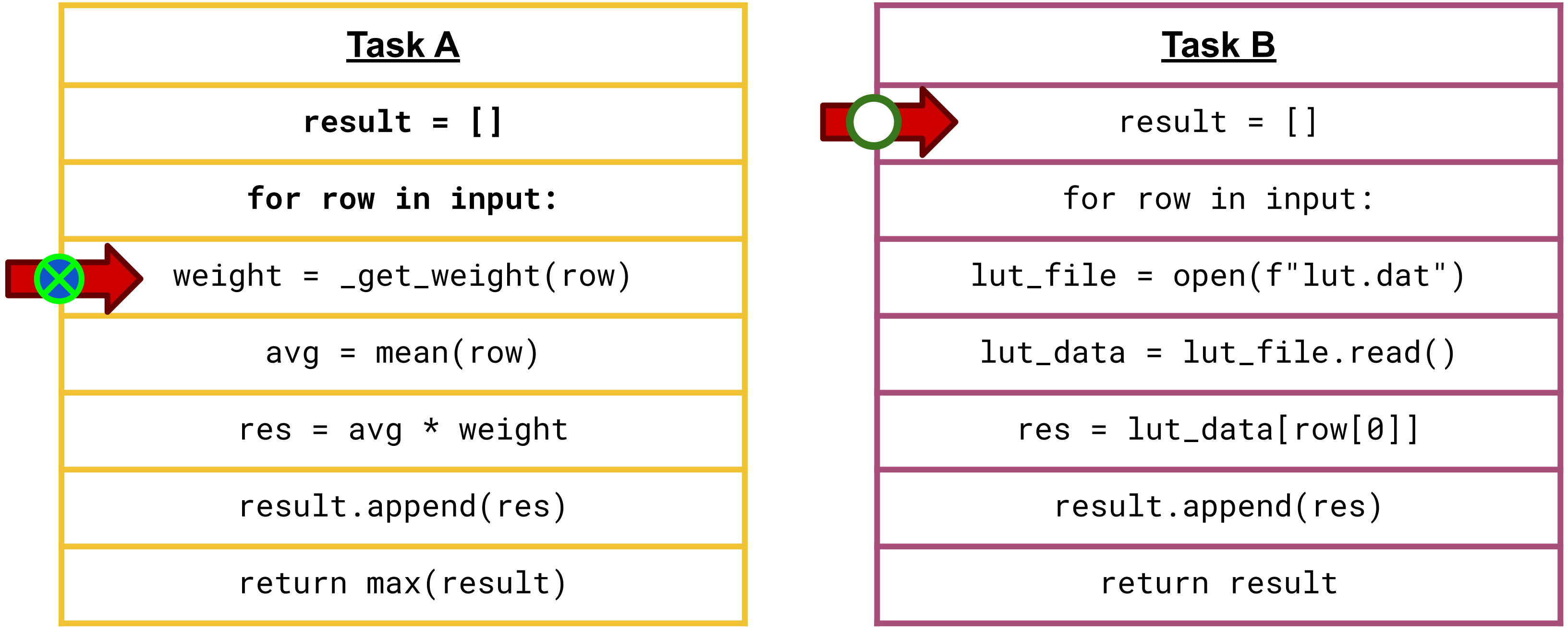
Task A

```
result = []
for row in input:
weight = _get_weight(row)
avg = mean(row)
res = avg * weight
result.append(res)
return max(result)
```

Task B

```
result = []
for row in input:
lut_file = open(f"lut.dat")
lut_data = lut_file.read()
res = lut_data[row[0]]
result.append(res)
return result
```

Global Interpreter Lock (GIL)



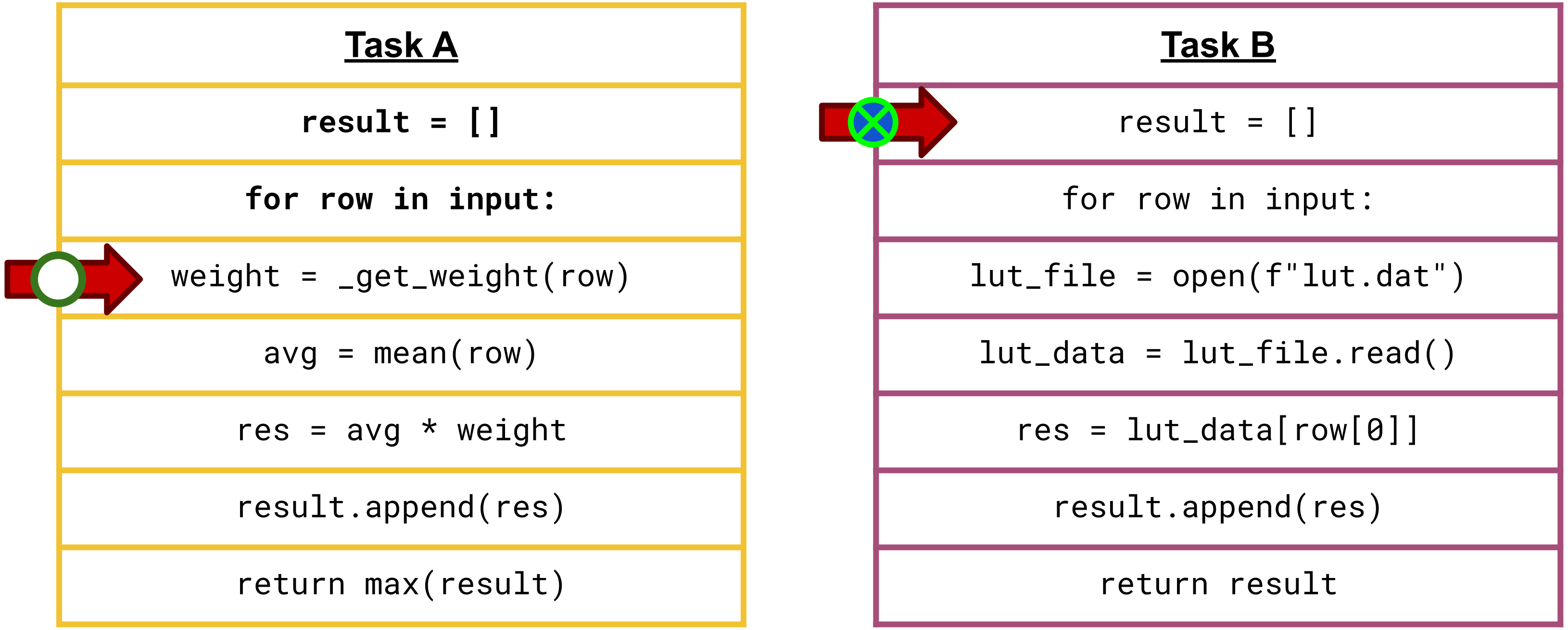
Task A

```
result = []
for row in input:
weight = _get_weight(row)
avg = mean(row)
res = avg * weight
result.append(res)
return max(result)
```

Task B

```
result = []
for row in input:
lut_file = open(f"lut.dat")
lut_data = lut_file.read()
res = lut_data[row[0]]
result.append(res)
return result
```

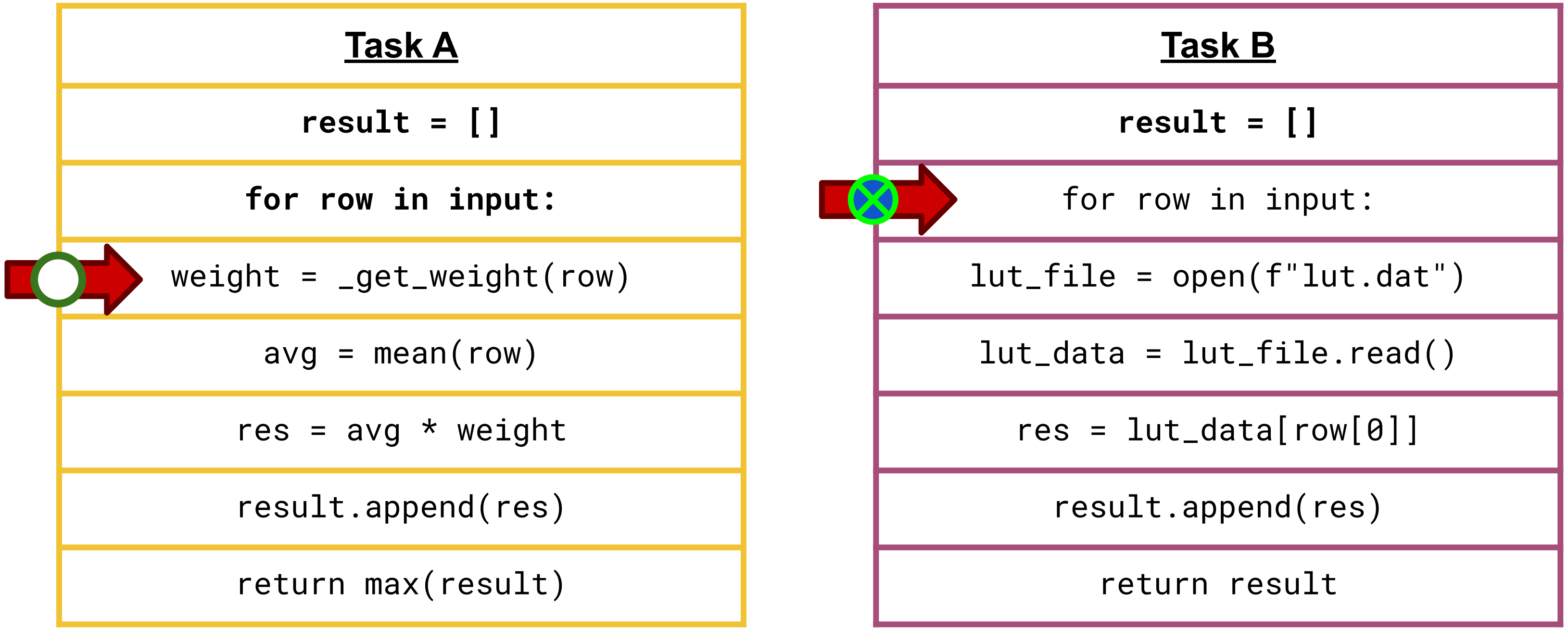
Global Interpreter Lock (GIL)



Task A
result = []
for row in input:
weight = _get_weight(row)
avg = mean(row)
res = avg * weight
result.append(res)
return max(result)

Task B
result = []
for row in input:
lut_file = open(f"lut.dat")
lut_data = lut_file.read()
res = lut_data[row[0]]
result.append(res)
return result

Global Interpreter Lock (GIL)



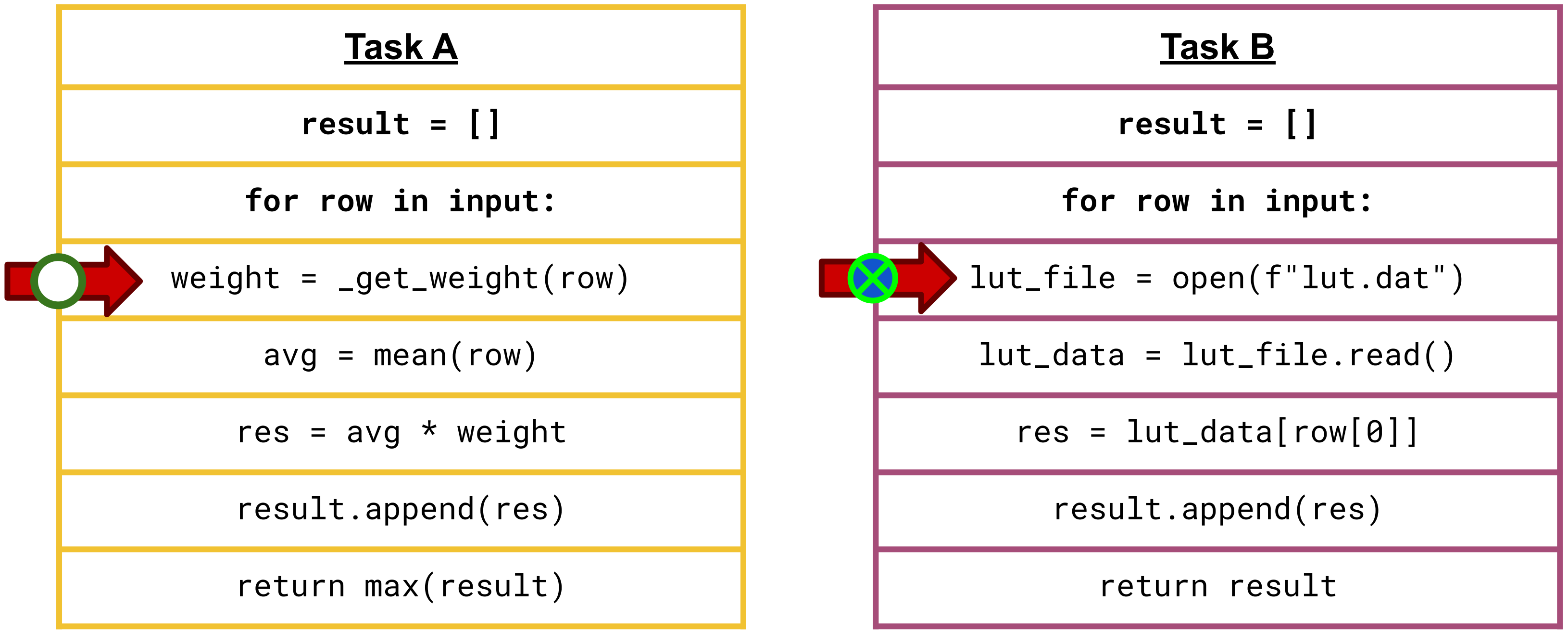
Task A

```
result = []
for row in input:
weight = _get_weight(row)
avg = mean(row)
res = avg * weight
result.append(res)
return max(result)
```

Task B

```
result = []
for row in input:
lut_file = open(f"lut.dat")
lut_data = lut_file.read()
res = lut_data[row[0]]
result.append(res)
return result
```

Global Interpreter Lock (GIL)



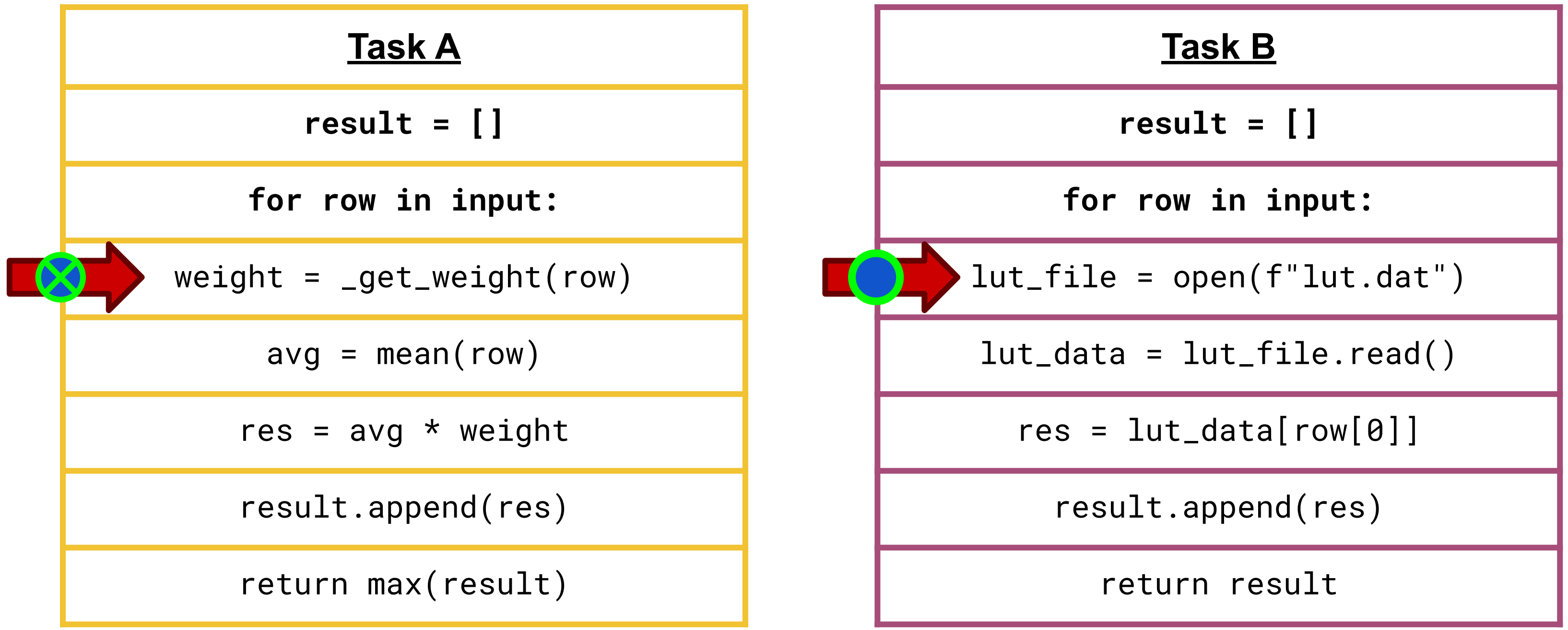
Task A

```
result = []
for row in input:
weight = _get_weight(row)
avg = mean(row)
res = avg * weight
result.append(res)
return max(result)
```

Task B

```
result = []
for row in input:
lut_file = open(f"lut.dat")
lut_data = lut_file.read()
res = lut_data[row[0]]
result.append(res)
return result
```

Global Interpreter Lock (GIL)



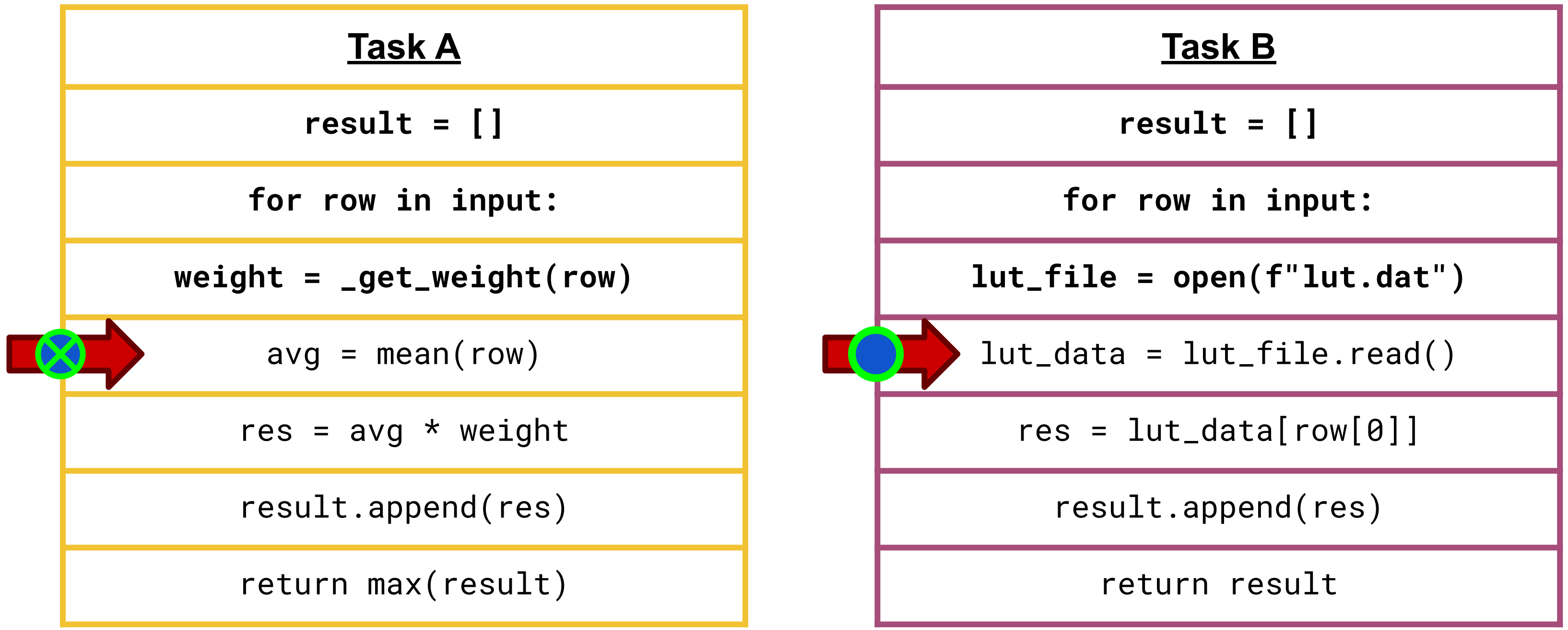
Task A

```
result = []
for row in input:
weight = _get_weight(row)
avg = mean(row)
res = avg * weight
result.append(res)
return max(result)
```

Task B

```
result = []
for row in input:
lut_file = open(f"lut.dat")
lut_data = lut_file.read()
res = lut_data[row[0]]
result.append(res)
return result
```

Global Interpreter Lock (GIL)



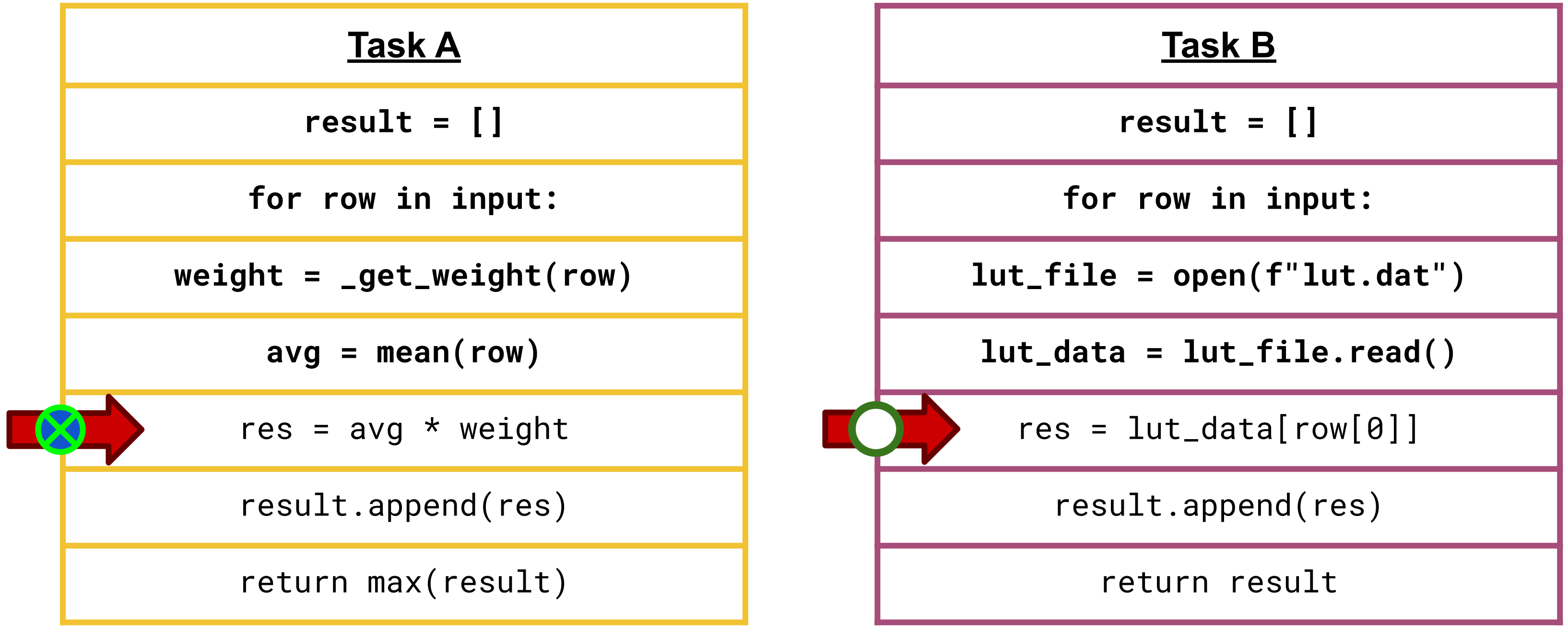
Task A

```
result = []
for row in input:
weight = _get_weight(row)
avg = mean(row)
res = avg * weight
result.append(res)
return max(result)
```

Task B

```
result = []
for row in input:
lut_file = open(f"lut.dat")
lut_data = lut_file.read()
res = lut_data[row[0]]
result.append(res)
return result
```

Global Interpreter Lock (GIL)



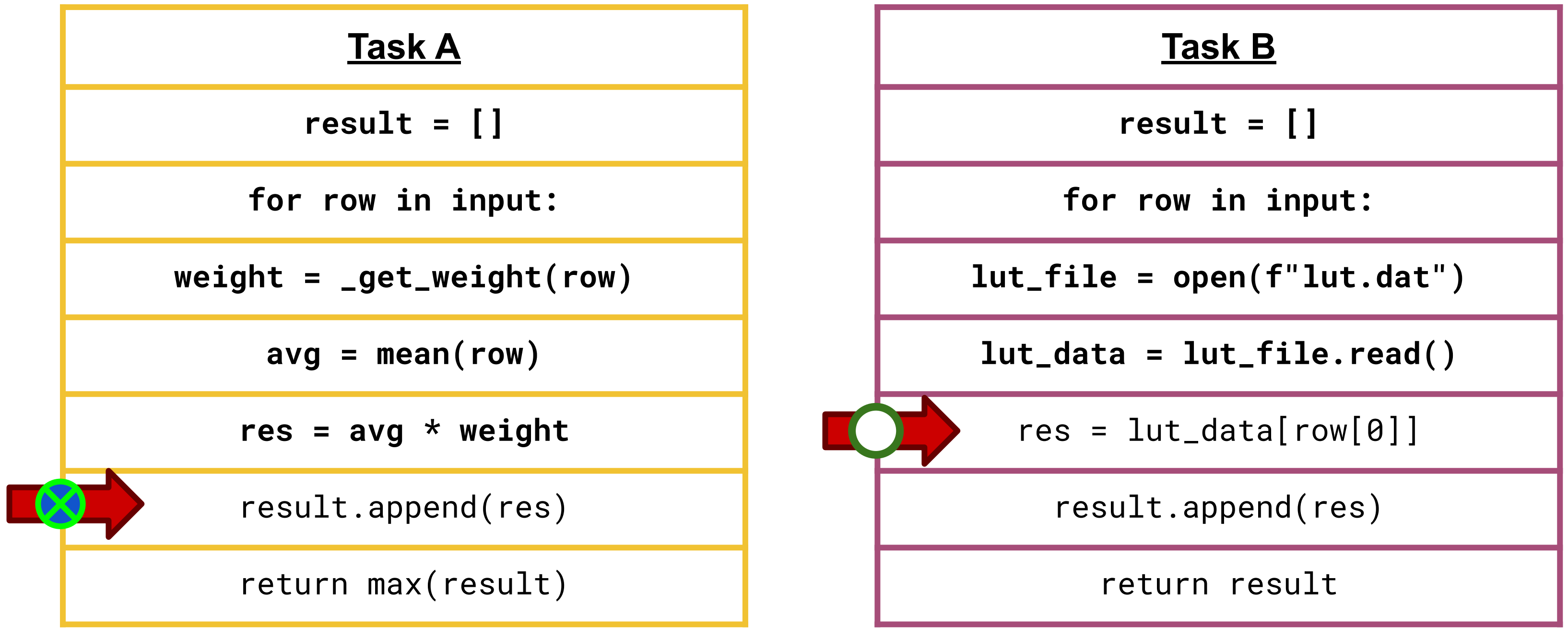
Task A

```
result = []
for row in input:
weight = _get_weight(row)
avg = mean(row)
res = avg * weight
result.append(res)
return max(result)
```

Task B

```
result = []
for row in input:
lut_file = open(f"lut.dat")
lut_data = lut_file.read()
res = lut_data[row[0]]
result.append(res)
return result
```

Global Interpreter Lock (GIL)

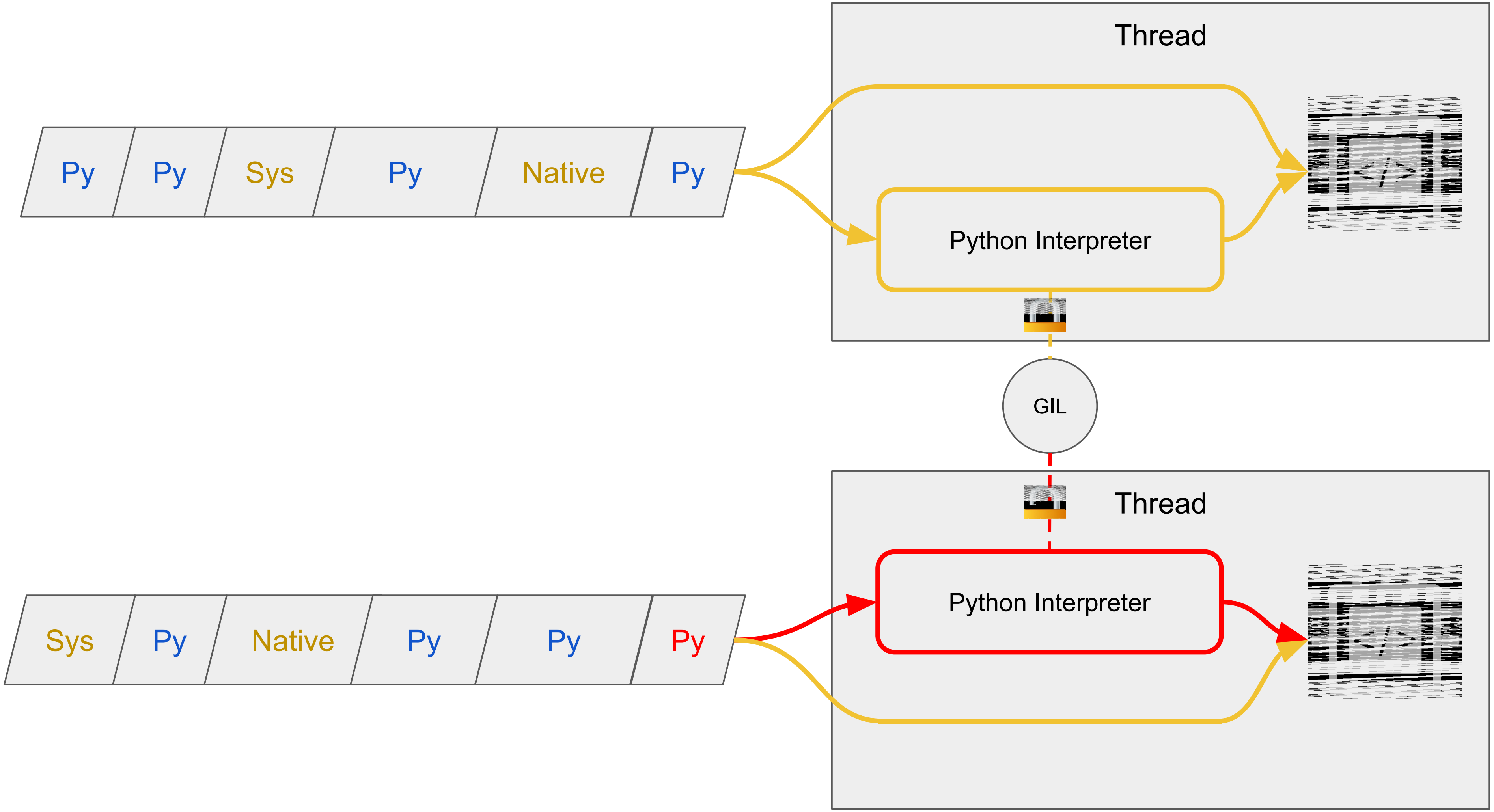


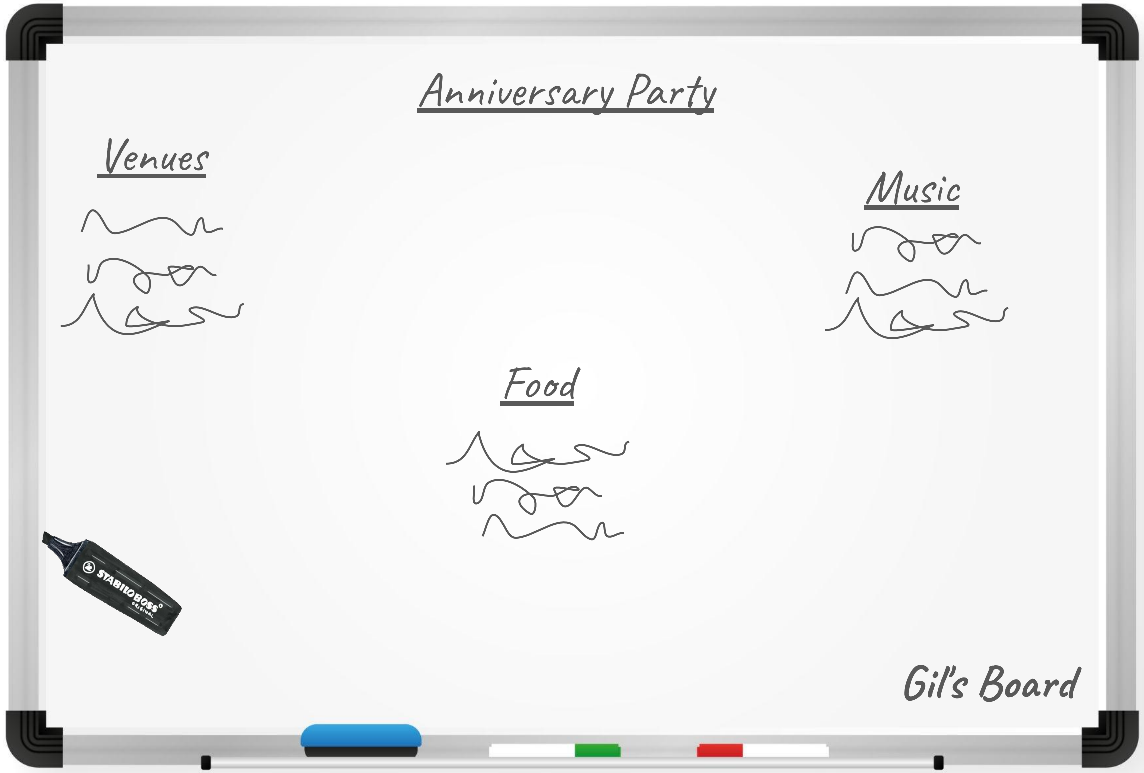
Task A

```
result = []
for row in input:
weight = _get_weight(row)
avg = mean(row)
res = avg * weight
result.append(res)
return max(result)
```

Task B

```
result = []
for row in input:
lut_file = open(f"lut.dat")
lut_data = lut_file.read()
res = lut_data[row[0]]
result.append(res)
return result
```


Py
Py
Sys
Py
Native
Py
Thread
Python Interpreter
GIL
Sys
Py
Native
Py
Py
Py
Thread
Python Interpreter


Anniversary Party
Venues
Music
Food
Gil's Board
STABILO BOSS ORIGINAL

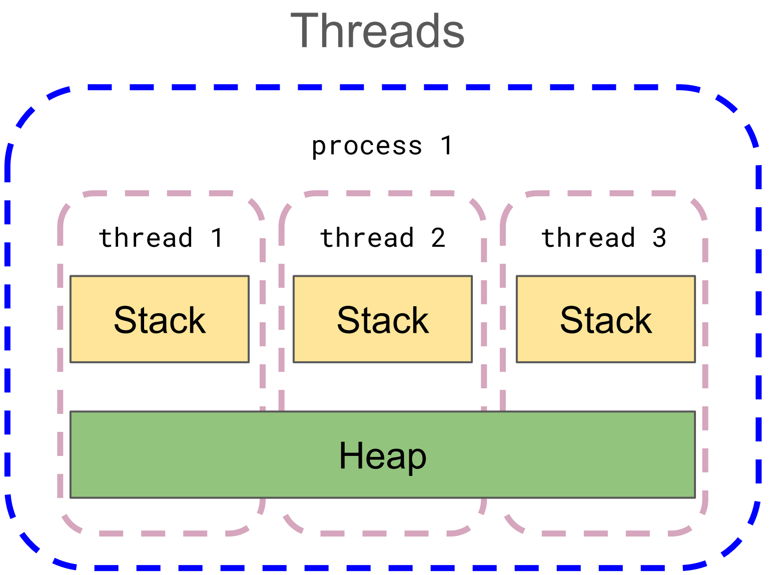
When the GIL isn't a problem

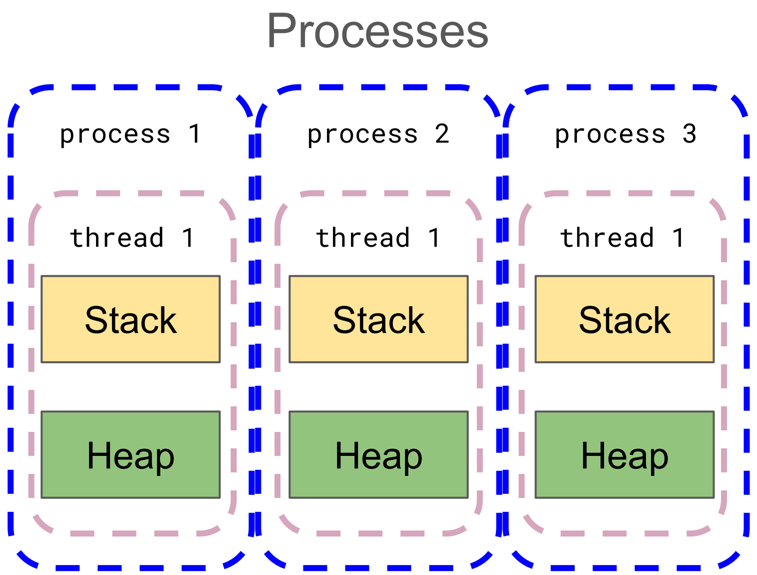
- Input/Output and other system calls
  - Disk
  - Network
  - `time.sleep`
- Explicit releasing of the GIL
  - Cython/C/other extensions
  - Third-party libraries
    - Ex. Numpy
    - "number crunching"
- Non-CPython interpreter
  - Jython
  - IronPython

Threads versus Processes

- Both allow you to accomplish things "at the same time"
- Threads are "lighter", processes are "heavier"
- In non-python languages, threads can be the best of both worlds
    - Easier to share data
    - Easier to context switch
- OS-dependent, version/kernel dependent, etc
- Use "locks" to control access to shared resources/values
    - acquire → release → acquire → release
- Use atomic operations to avoid locking

Threads versus Processes - Memory





Python's threading (traditionally)

- `threading.Thread(target=some_func)`
- Thread Pools
  - `concurrent.futures.ThreadPoolExecutor`
  - `multiprocessing.pool.ThreadPool`
  - third-party library
- "Everything is an object" means everything in the shared heap
- Locks still available for coordinating access to shared resources
- GIL:
  - Pro: Each bytecode operation is "atomic"
  - Con: "To avoid the GIL, use multiple processes"

Threading in Python

```
import threading

def some_func(some_input):
    …

if __name__ == "__main__":
    t = threading.Thread(
        target=some_func,
        args=("my_file.txt",),
    )
    t.start()  # ← start thread 2

    # other stuff - running in thread 1

    t.join()
```

Threading in Python

Questions before free-threading?

Removing the GIL

Changes in CPython 3.13 (PEP 703) - Experimental:

- Reference counting and Immortalization
- Memory management and Garbage Collection
- Container thread-safety (dict, list, etc) - per-object locks
- Locking and atomic APIs
- Turn off some interpreter optimizations

Changes in CPython 3.14:

- Convert temporary workarounds into long-term solutions
- Specializing adaptive interpreter enabled

Python's threading (free-threading)

- Experimental in Python 3.13
- Fully-supported in Python 3.14
- Special ***separate*** build and binary of CPython
    - Optional: Need to explicitly install it
    - Single threaded applications are generally slower
        - 5-10% performance penalty in Python 3.14
        - Even worse in Python 3.13
- No GIL!

What cases does free-threading improve?

- Short answer: Every multi-threaded workflow
- Long answer:
  - I/O bound tasks should see minimal improvement
  - CPU bound pure-python (minimal extensions) will see the greatest improvement

How do I use it?

- `conda install -c conda-forge python-freethreading`
- `python3.13t`
- https://docs.python.org/3/howto/free-threading-python.html
- https://py-free-threading.github.io/ (by Quansight Labs and Meta)
- Re-enable GIL:

```
python3.13t -X gil=1 my_script.py
PYTHONGIL=1 python3.13t
```

- Check if build supports no-GIL:

```
print(sys.version)
'3.13.5 experimental free-threading build | packaged by conda-forge…'
```

- At runtime:

```
sys._is_gil_enabled()
```

Examples

https://github.com/djhoese/free-threading-examples

Free-threading

- Try it out!