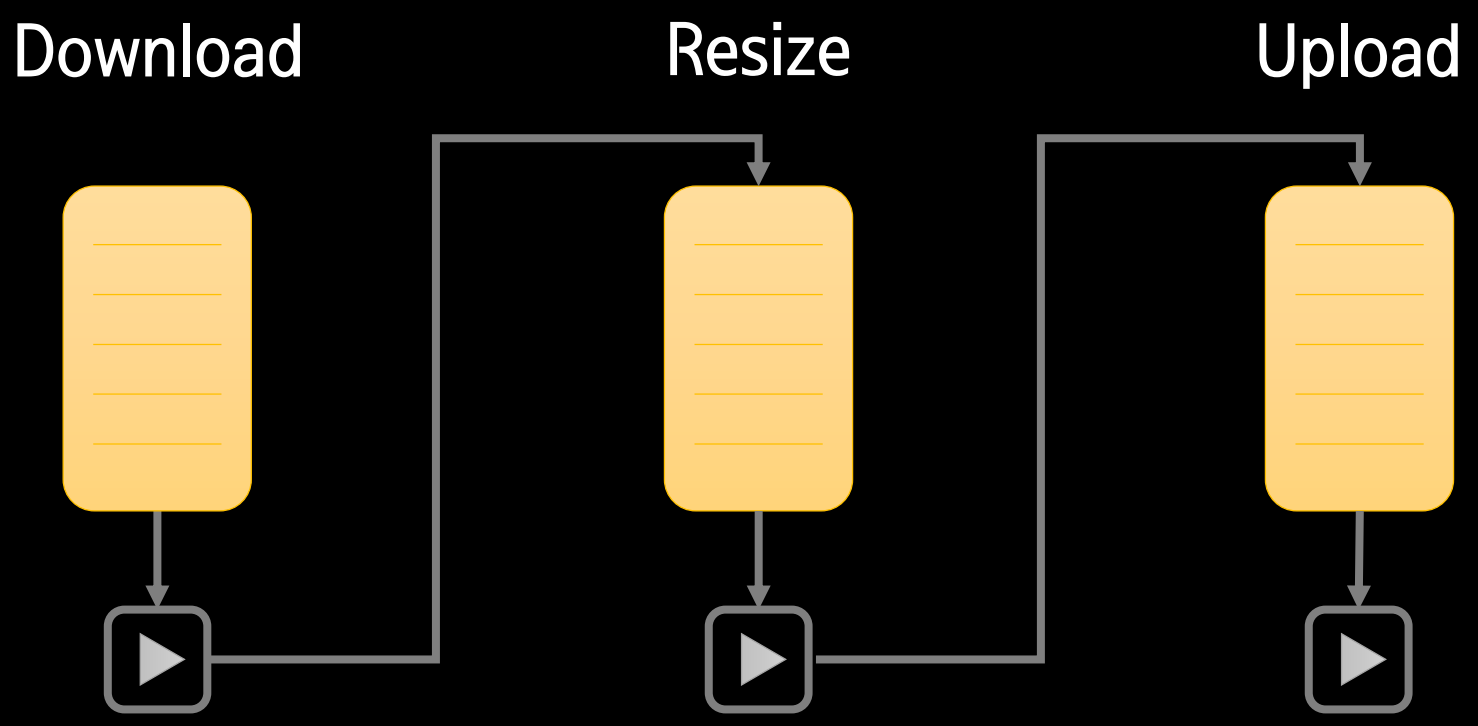
CHAPTER 39

스레드간의 작업을 조율하려면 Queue를 사용하자

동시성 작업에서는 함수의 파이프라인화가 유용하다

이미지 처리를 파이프라인화 한다고 하면

Download Resize Upload


Download
Resize
Upload

WorkerClass

```
class Worker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super.__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue
        self.polled_count = 0
        self.work_done = 0
```

WorkerClass

```python
class Worker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super.__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue
        self.polled_count = 0
        self.work_done = 0
```

```python
def put(self, item):
    with self.lock:
        self.items.append(item)

def get(self):
    with self.lock:
        return self.items.popleft()
```

queue는 collection.dequeue에
Lock을 걸어 사용

WorkerClass

```
class Worker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super.__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue
        self.polled_count = 0
        self.work_done = 0
```

WorkerClass

```
def run(self):
    while True:
        self.polled_count += 1
        try:
            item = self.in_queue.get()
        except IndexError:
            sleep(0.01)  # No work to do
        except AttributeError:
            # The magic exit signal
            return
        else:
            result = self.func(item)
            self.out_queue.put(result)
            self.work_done += 1
```

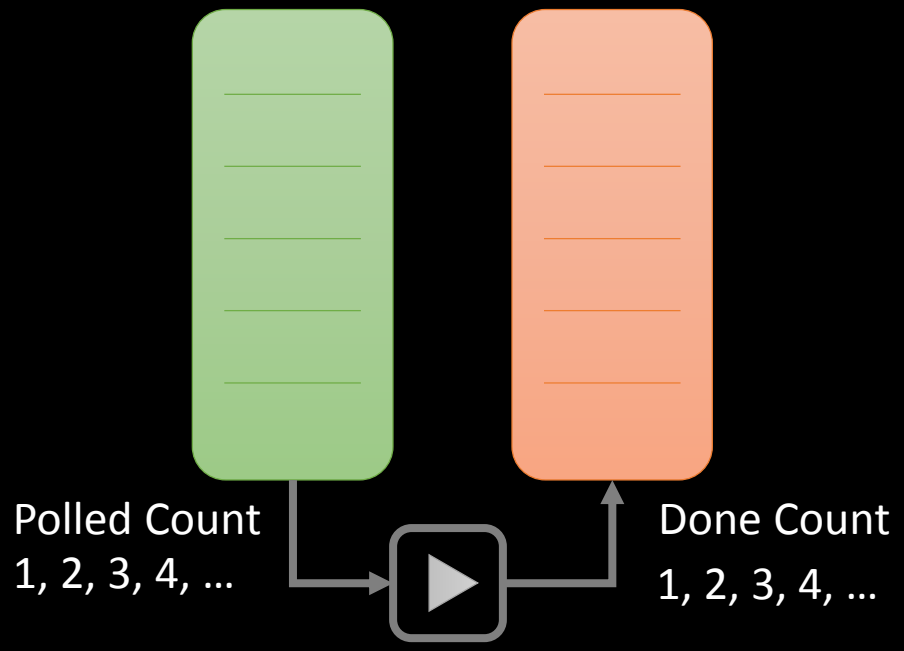
WorkerClass

```
def run(self):
    while True:
        self.polled_count += 1
        try:
            item = self.in_queue.get()
        except IndexError:
            sleep(0.01)  # No work to do
        except AttributeError:
            # The magic exit signal
            return
        else:
            result = self.func(item)
            self.out_queue.put(result)
            self.work_done += 1
```

WorkerClass

```
def run(self):
    while True:
        self.polled_count += 1
        try:
            item = self.in_queue.get()
        except IndexError:
            sleep(0.01)  # No work to do
        except AttributeError:
            # The magic exit signal
            return
        else:
            result = self.func(item)
            self.out_queue.put(result)
            self.work_done += 1
```

Class Worker

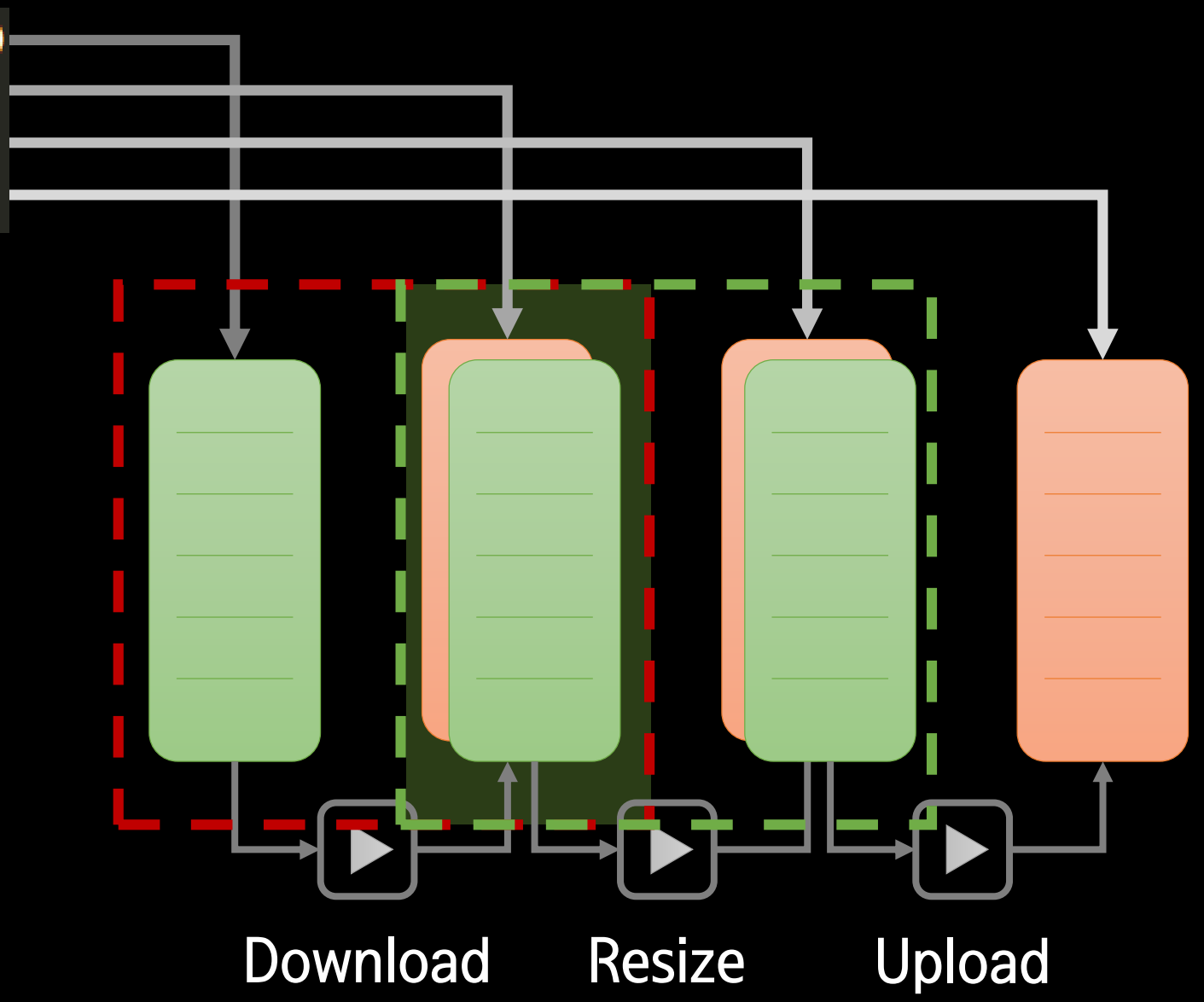


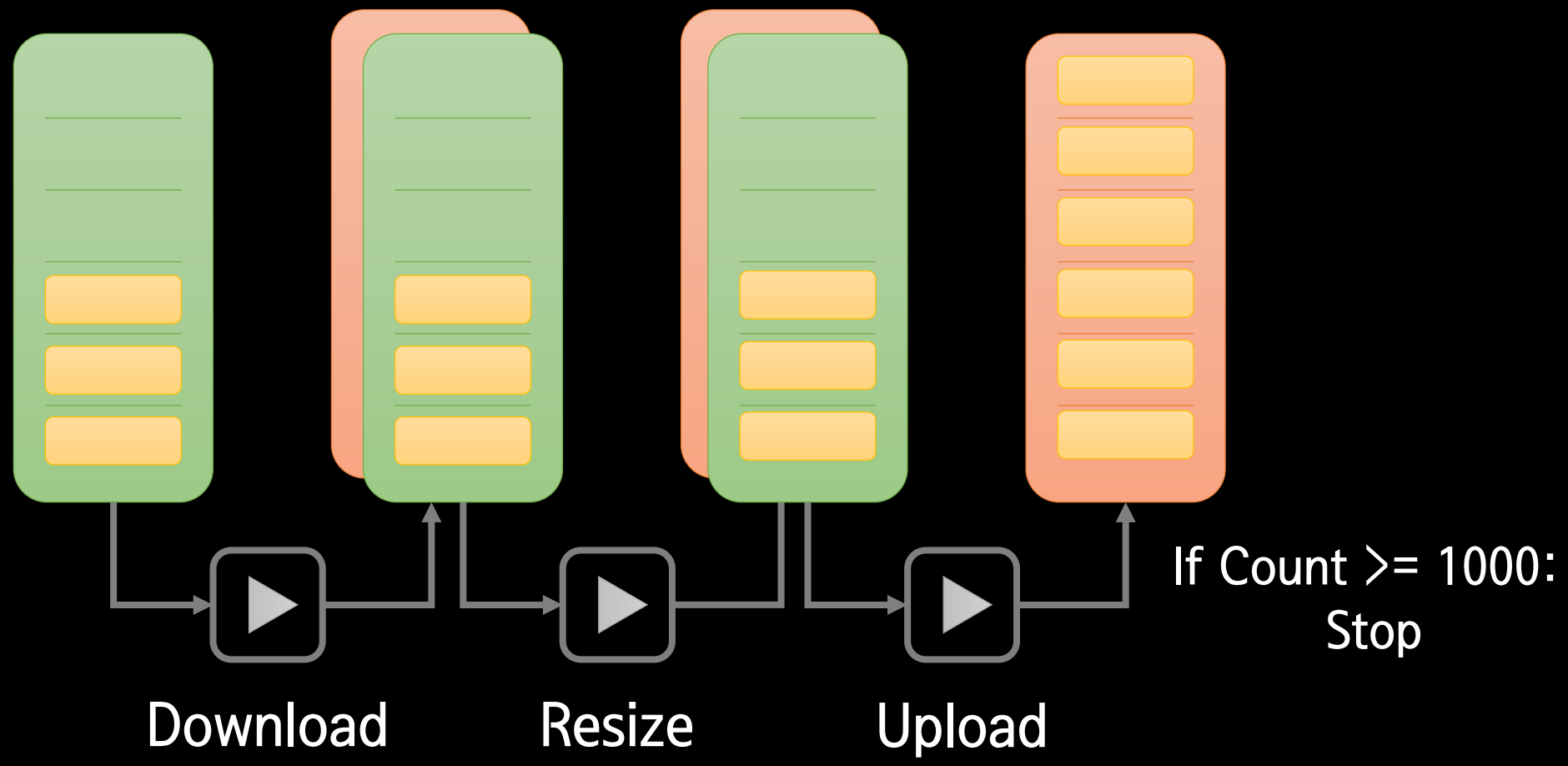
Class Worker

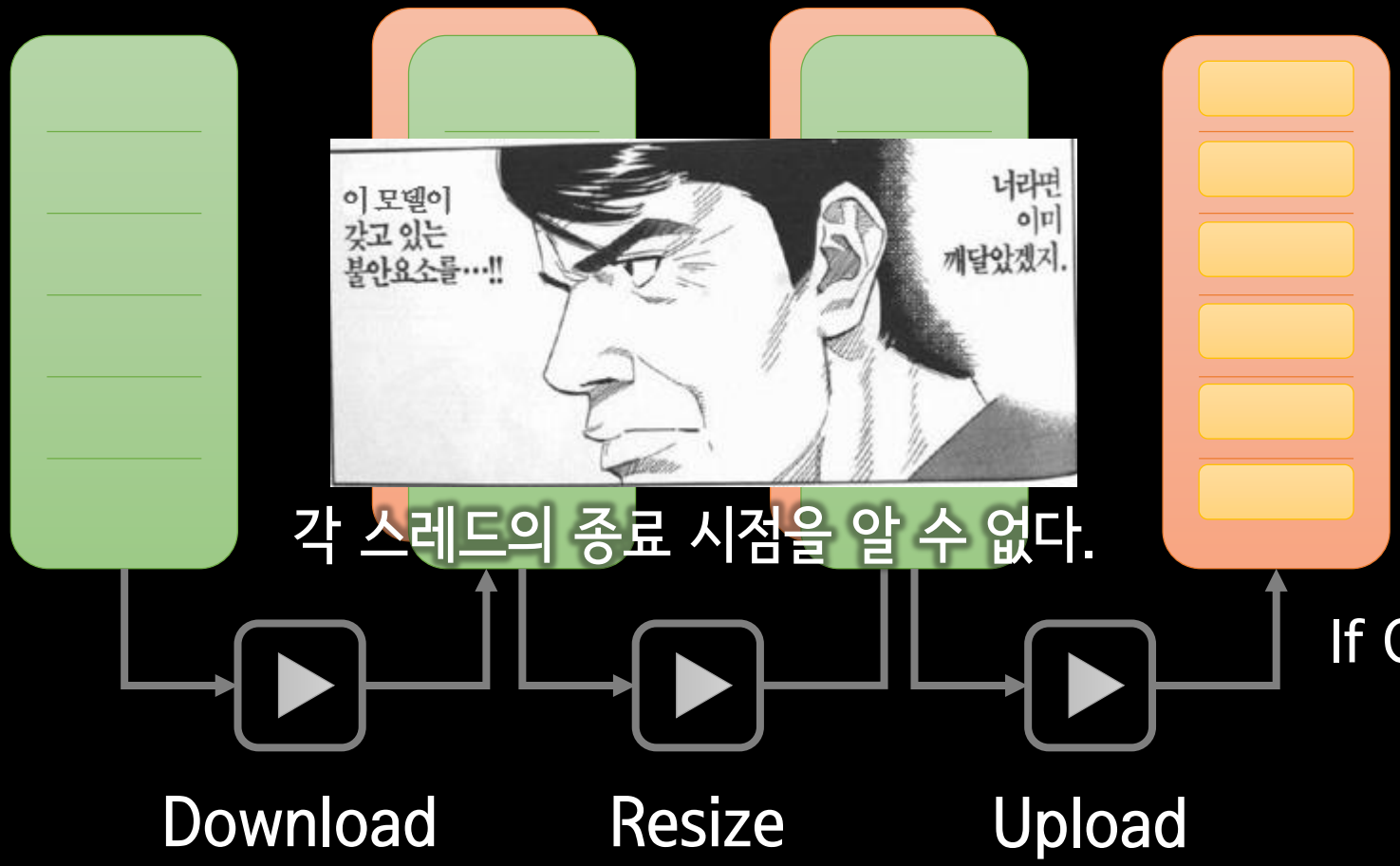
```
download_queue = MyQueue()
resize_queue = MyQueue()
upload_queue = MyQueue()
done_queue = MyQueue()
threads = [
    Worker(download, download_queue, resize_queue),
    Worker(resize, resize_queue, upload_queue),
    Worker(upload, upload_queue, done_queue),
]

for thread in threads:
    thread.start()
for _ in range(1000):
    download_queue.put(object())
```

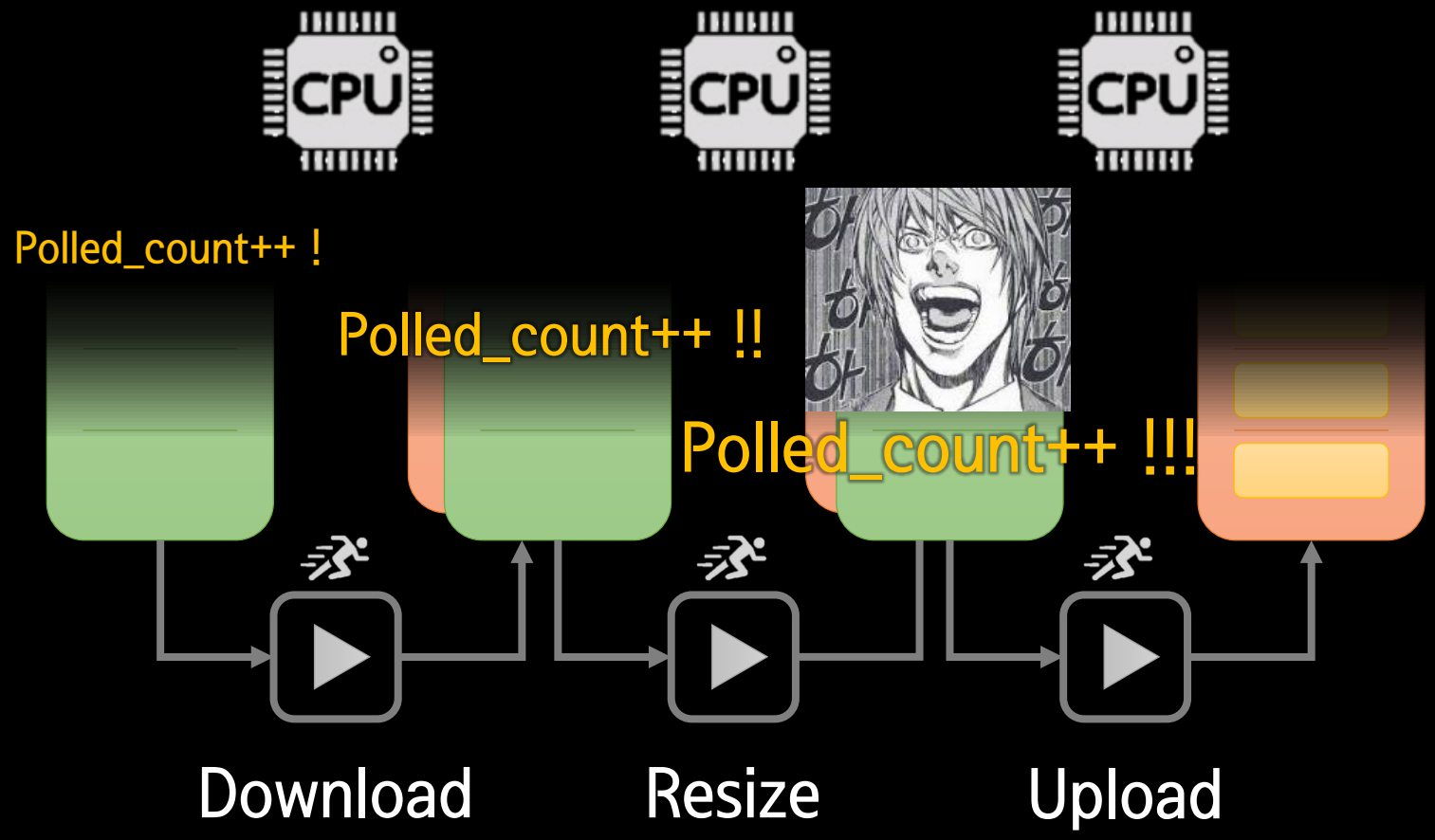
```
download_queue = MyQueue()
resize_queue = MyQueue()
upload_queue = MyQueue()
done_queue = MyQueue()
```



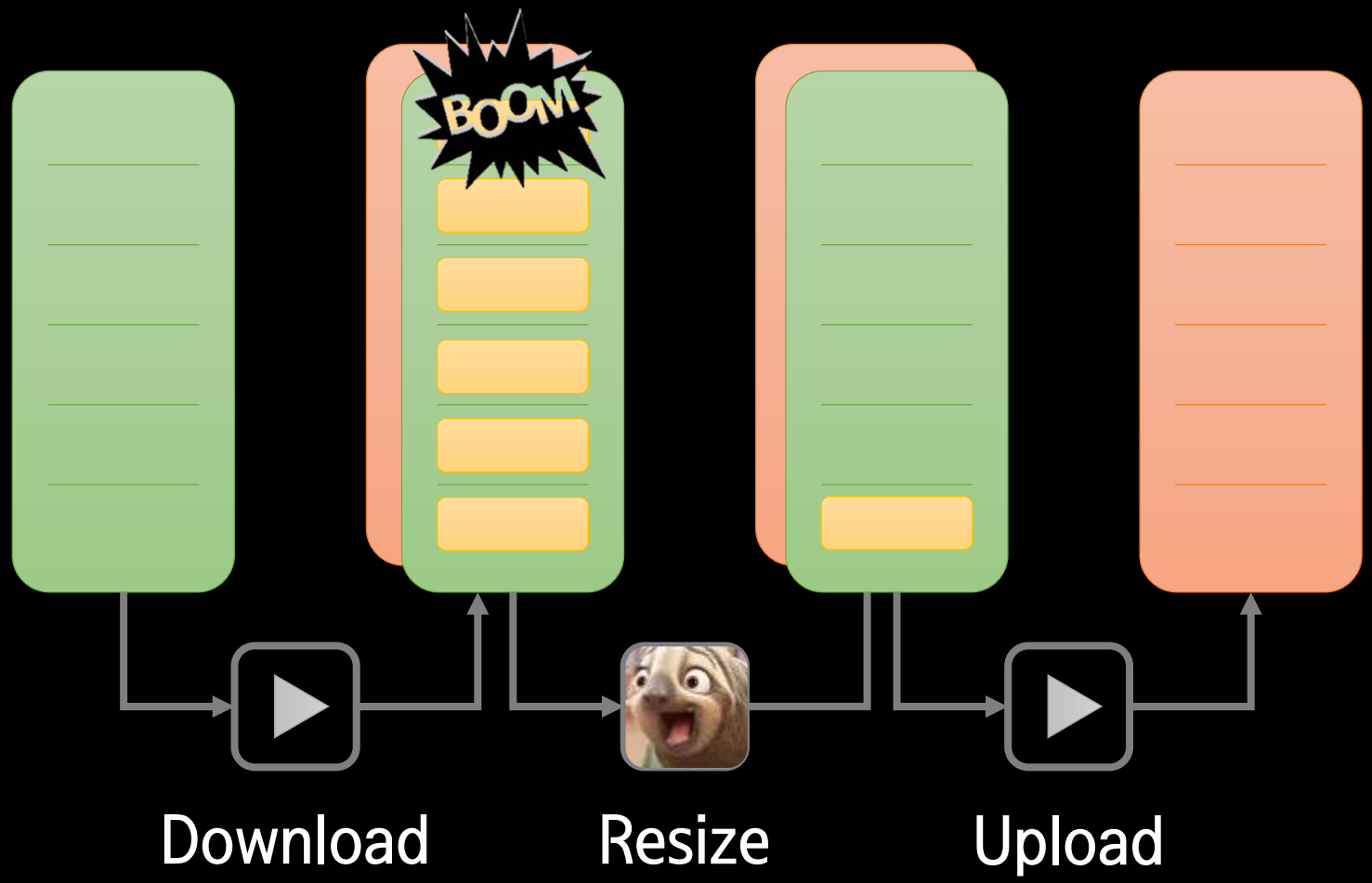

Download
Resize
Upload
If Count >= 1000:
Stop



If Count >= 1000:
Stop



불필요한 CPU 사용이 발생한다



메모리가 터질 수 있다

Queue를 사용하자!

1. get이 호출될 때 Queue가 비어있으면 자동으로 블록됨

```
from Queue import Queue

testQueue = Queue()
temp = testQueue.get()
testQueue.put(object())
```

*get이 먼저 호출되면 DEADLOCK

Queue를 사용하자!

1. get이 호출될 때 Queue가 비어있으면 자동으로 블록됨
2. Queue에 삽입된 아이템 수 만큼 task_done()이 호출되어야 join()이 종료됨

```
from queue import Queue

download_queue = Queue()
for _ in range(10):
    download_queue.put(object())

for _ in range(10):
    download_queue.task_done()

download_queue.join()
```

9, 8, 7 ...

* get 호출수는 상관없음

```python
class ClosableQueue(Queue):
    def __init__(self, name):
        super().__init__()
        self.SENTINEL = object()

    def close(self):
        self.put(self.SENTINEL)

    def __iter__(self):
        while True:
            item = self.get()
            try:
                if item is self.SENTINEL:
                    return
                yield item
            finally:
                self.task_done()
```

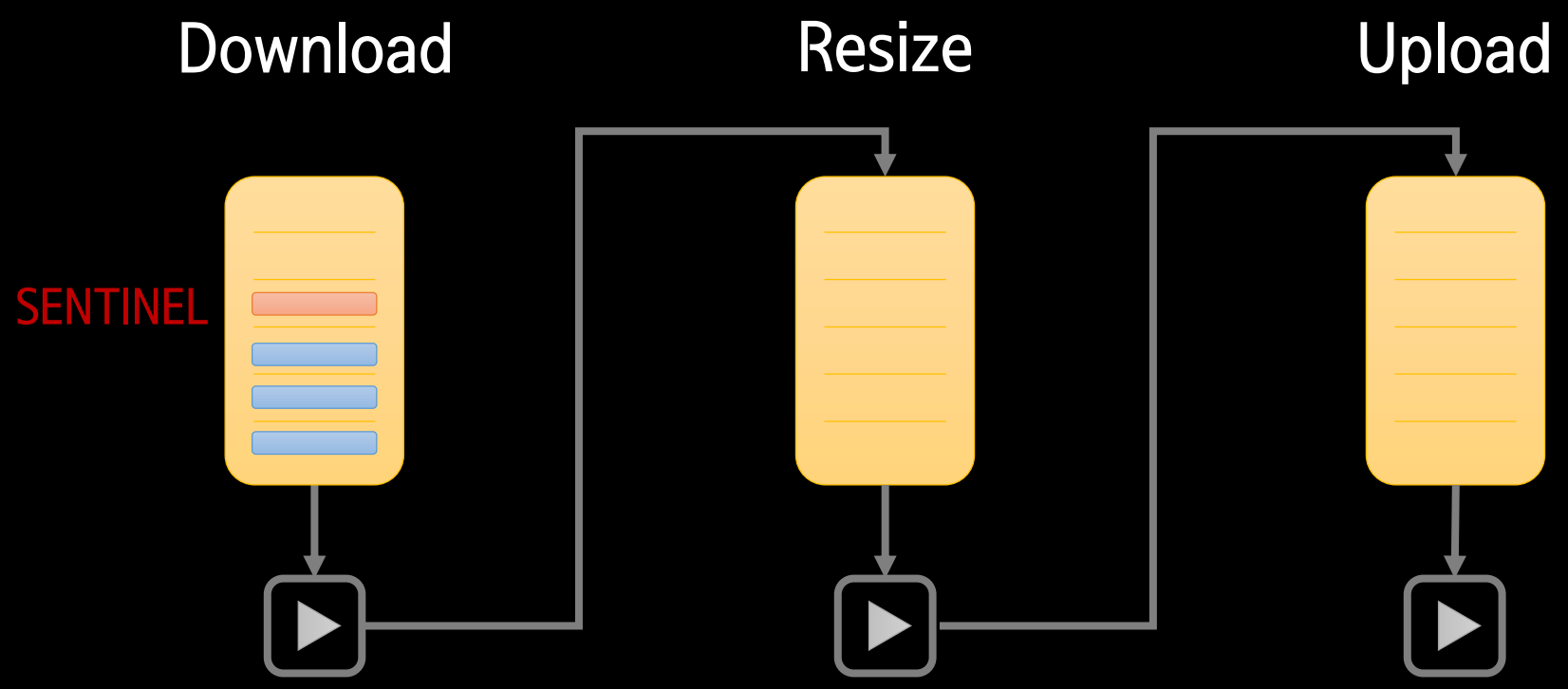
```python
class StoppableWorker(Thread):
    def run(self):
        for item in self.in_queue:
            result = self.func(item)
            self.out_queue.put(result)
```

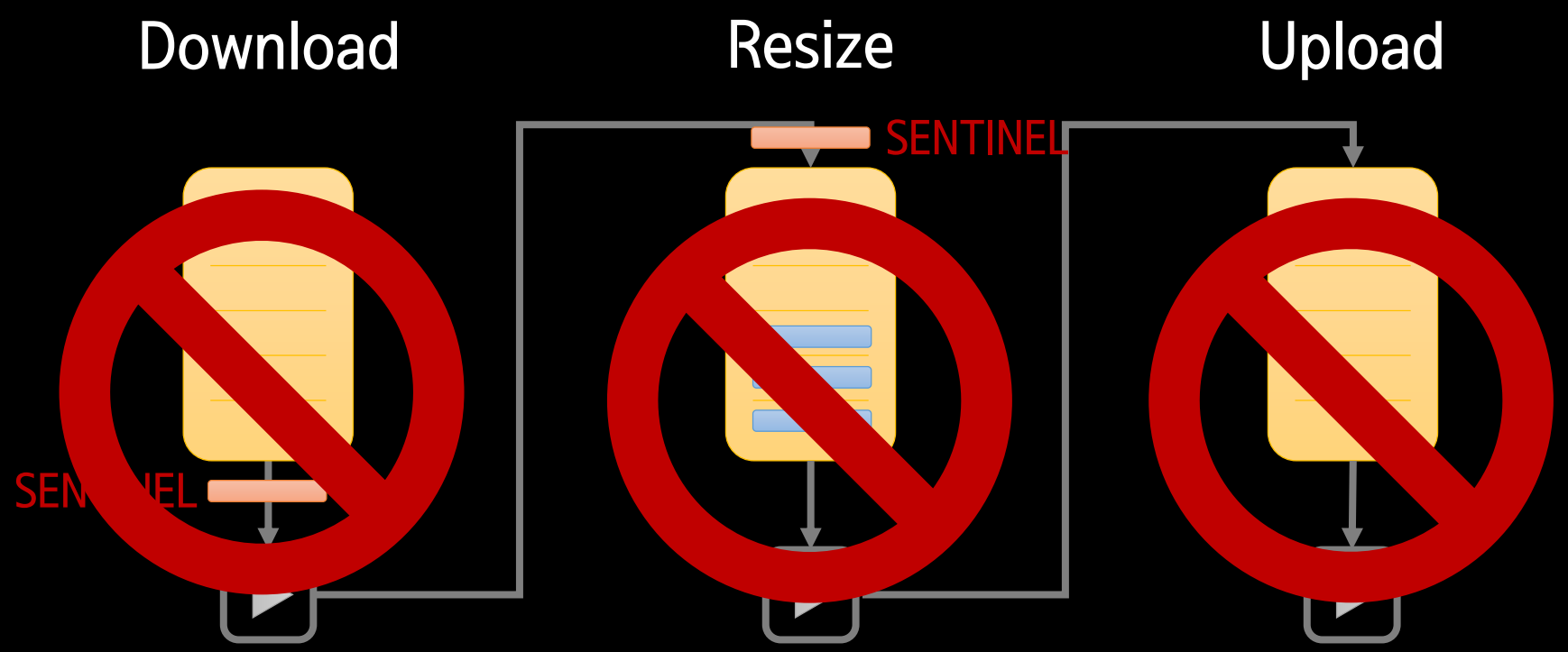
```python
for thread in threads:
    thread.start()

for _ in range(10):
    download_queue.put(object())
download_queue.close()

download_queue.join()
resize_queue.close()
resize_queue.join()
upload_queue.close()
upload_queue.join()
print(done_queue.qsize(), 'items finished')
```

큐를 모두 비웠다는 것은
다음 큐에 대한 삽입도 끝났다는 것


Download
Resize
Upload
SENTINEL



각 스레드의 종료 시점을 알 수 없어
불필요한 CPU 사용이 발생한다

메모리가 터질 수 있다

이건 여전히 해결되지 않는 것 같다

작업이 오래걸리면 더 작게 분할해주는게 필요할듯…

CHAPTER 42

functools.wraps로 함수 데코레이터를 정의하자

데코레이터?

```
@mydecorator
def funcA():
```

```
funcA = mydecorator(funcA)
```

* funcA가 mydecorator로 변경되기 때문에 디버거를 붙이거나
직렬화를 할때 문제가 발생할 수 있다

```python
def deco(func):
    def wrapper(*args, **kwargs):
        print('call wrapper')
        result = func(*args, **kwargs)
        print('result is ', result)
        return result
    return wrapper

@deco
def funcA():
    print('function name is ', funcA.__name__)
    return 10

funcA()
```

```
call wrapper
function name is wrapper
result is  10
```

```python
funcA = mydecorator(funcA)
```

functools.wraps로 함수 데코레이터를 정의하자

```
from functools import wraps
def trace(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print('%s(%r, %r) -> %r' %
              (func.__name__, args, kwargs, result))
        return result
    return wrapper
```

```
call wrapper
function name is funcA
result is 10
```

CHAPTER 43

재사용 가능한 try/finally 동작을 만들려면 contextlib와 with를 고려하자

@contextmanager

```
with funcA():
    #do something
```

@contextmanager

```python
from contextlib import contextmanager
@contextmanager
def funcA():
    #pre logic
    try:
        yield
    finally:
        #post logic
```

```python
with funcA():
    #do something
```

@contextmanager

```python
@contextmanager
def MyPrinter():
    print('before')
    try:
        yield
    finally:
        print('after')

with MyPrinter():
    print('do something')
```

```
before
do something
after
```

객체 넘기기

```python
from contextlib import contextmanager

@contextmanager
def MyPrinter():
    print('before')
    print_handler = print
    try:
        yield print_handler
    finally:
        print('after')

with MyPrinter() as p:
    p('do something')
```

```
before
do something
after
```

끝