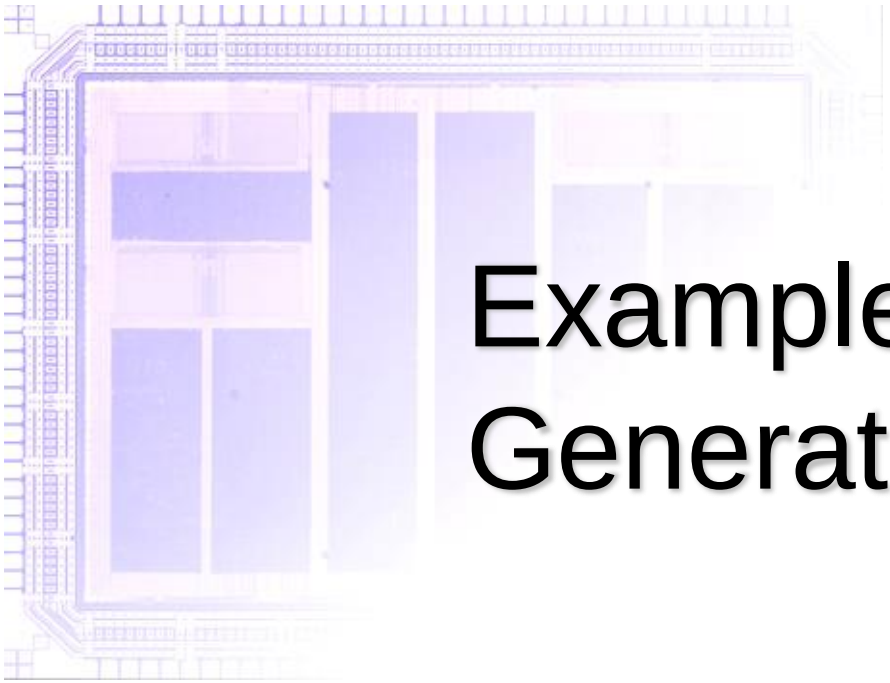
MSOC Lab #2

Yu-Sheng Lin

johnjohnlys@media.ee.ntu.edu.tw

In Lab #2, you will learn how SystemC works.

However, we will use Python because it has some good language features.



Example #0
Generator / Coroutine

Outline

- In SystemC, you use SC_THREAD and SC_METHOD as if they execute in parallel.
- However, they are indeed coroutines.
  - Coroutines look like parallel threads.
  - Coroutines actually execute sequentially.
  - Coroutines are cheap.
- Reference
  - https://johnjohnlin.github.io/nicotb/concurrent.html

In Short

- You will learn how to implement a simple SystemC-like library with Python.

- I think I don't have to illustrate how to install Python. Figure it out by yourself~

Coroutine in Languages

- Coroutines are very common.
  - Python, JS, and very likely in C++20
- So I use Python 3 as an example.
  - Python is easier/more popular than C++.

- Note: Python uses indentation for code structure, which is done by braces in C++.
  - The most common indentation is 4 spaces.

A Function Returning Multiple Values?

```
def A():
    a, b = 1, 2
    return a
    return b
```

```
> A()
1
> A()
1
> A()
1
```

The second return is useless

Generator: Function That Can Stop Temporarily

```
def A():
    a, b = 1, 2
    yield a
    yield b
```

```
> A()
<generator object B at ...>
```

??

```
> aa = A()
> next(aa)
1
> next(aa)
2
```

Generator is the basic form of coroutine.

http://blog.blackwhite.tw/2013/05/python-yield-generator.html

Generator v.s. wait in SystemC (1)

```
def f():
    for i in range(5):
        print("Function f, {}".format(i))
        yield
    print("Function f finished")

def g():
    for i in range(3):
        print("Function g, {}".format(i))
        yield
    print("Function g finished")
```

- SC_THREAD(f); SC_THREAD(g);
  - What do you expect?
- The yield is almost the same as wait? Why?
  - See the next page

Generator v.s. wait in SystemC (2)

```
def f():
    for i in range(5):
        print("...")
        yield
    print("...")


def g():
    for i in range(3):
        print("...")
        yield
    print("...")
```

Scheduler

```
def main_loop():
    from itertools import zip_longest
    for dummy in zip_longest(f(), g()):
        pass
```

Schedule the Generators (1)

```
def f():
    for i in range(5):
        print("...")
        yield
    print("...")

def g():
    for i in range(3):
        print("...")
        yield
    print("...")
```

```
def main_loop():
    from itertools import zip_longest
    for dummy in zip_longest(f(), g()):
        print("clk")
```

Program entry

```
johnjohnlin /tmp  % python b.py
Function f, 0
Function g, 0
===== clk =====
Function f, 1
Function g, 1
===== clk =====
Function f, 2
Function g, 2
===== clk =====
Function f, 3
Function g finished
===== clk =====
Function f, 4
===== clk =====
Function f finished
johnjohnlin /tmp  %
```

Schedule the Generators (2)

```
def f():
    for i in range(5):
        print("...")
        yield
    print("...")

def g():
    for i in range(3):
        print("...")
        yield
    print("...")
```

```
def main_loop():
    from itertools import zip_longest
    for dummy in zip_longest(f(), g()):
        print("clk")
```

zip_longest first goes inside f

```
johnjohnlin /tmp  % python b.py
Function f, 0
Function g, 0
===== clk =====
Function f, 1
Function g, 1
===== clk =====
Function f, 2
Function g, 2
===== clk =====
Function f, 3
Function g finished
===== clk =====
Function f, 4
===== clk =====
Function f finished
johnjohnlin /tmp  %
```

Schedule the Generators (3)

```
def f():
    for i in range(5):
        print("...")
        yield
    print("...")

def g():
    for i in range(3):
        print("...")
        yield
    print("...")
```

```
def main_loop():
    from itertools import zip_longest
    for dummy in zip_longest(f(), g()):
        print("clk")
```

zip_longest sees the yield and then goes into g

```
johnjohnlin /tmp  % python b.py
Function f, 0
Function g, 0
===== clk =====
Function f, 1
Function g, 1
===== clk =====
Function f, 2
Function g, 2
===== clk =====
Function f, 3
Function g finished
===== clk =====
Function f, 4
===== clk =====
Function f finished
johnjohnlin /tmp  %
```

Schedule the Generators (4)

```
def f():
    for i in range(5):
        print("...")
        yield
    print("...")
```

```
def g():
    for i in range(3):
        print("...")
        yield
    print("...")
```

```
def main_loop():
    from itertools import zip_longest
    for dummy in zip_longest(f(), g()):
        print("clk")
```

OK, nothing to do

```
johnjohnlin /tmp  % python b.py
Function f, 0
Function g, 0
===== clk =====
Function f, 1
Function g, 1
===== clk =====
Function f, 2
Function g, 2
===== clk =====
Function f, 3
Function g finished
===== clk =====
Function f, 4
===== clk =====
Function f finished
johnjohnlin /tmp  %
```

Schedule the Generators (5)

```
def f():
    for i in range(5):
        print("...")
        yield
    print("...")

def g():
    for i in range(3):
        print("...")
        yield
    print("...")
```

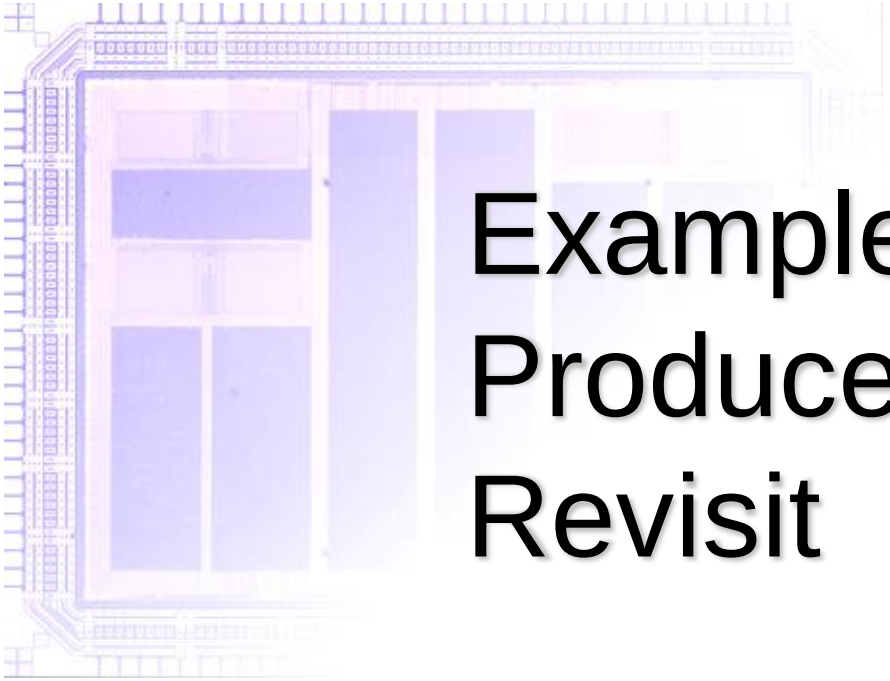
```
def main_loop():
    from itertools import zip_longest
    for dummy in zip_longest(f(), g()):
        print("clk")
```

zip_longest doesn't schedule anymore since it finishes

```
johnjohnlin /tmp  % python b.py
Function f, 0
Function g, 0
===== clk =====
Function f, 1
Function g, 1
===== clk =====
Function f, 2
Function g, 2
===== clk =====
Function f, 3
Function g finished
===== clk =====
Function f, 4
===== clk =====
Function f finished
johnjohnlin /tmp  %
```

In example #0, just find a machine with Python 3 (I recommend 3.6) and run the code.

Note: you should call main_loop() somewhere in your code



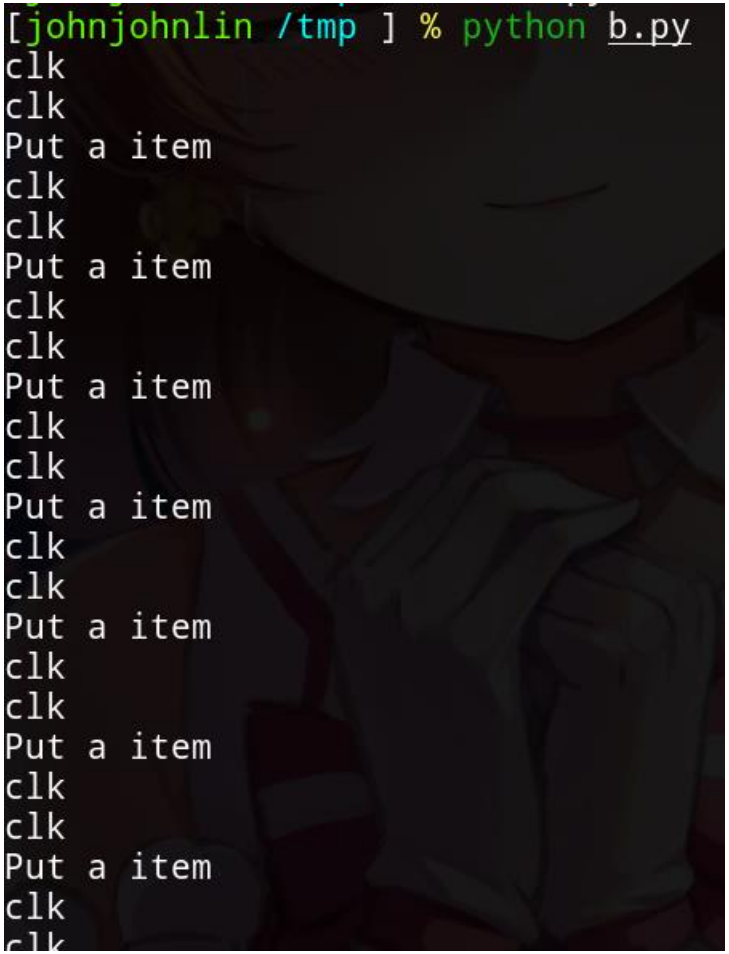
Example #1 Producer/Consumer Revisit

Producer Revisit

```
n_item = 0

def Producer(n):
    global n_item
    for i in range(n):
        yield
        yield
        print("Put an item")
        n_item +=1
```

Wait 2 cycles



Run the Producer

- Use the same main function in Example #0
- We do a little modification to make it more flexible
- ```
  def main_loop(threads):
      from itertools import zip_longest
      for dummy in zip_longest(*threads):
          print("clk")


  main_loop([Producer(10)])
  ```
  ```

Consumer

```
def Consumer(n):
    global n_item
    n_get = 0
    while n_get < n:
        if ???:
            ???
```

TODO

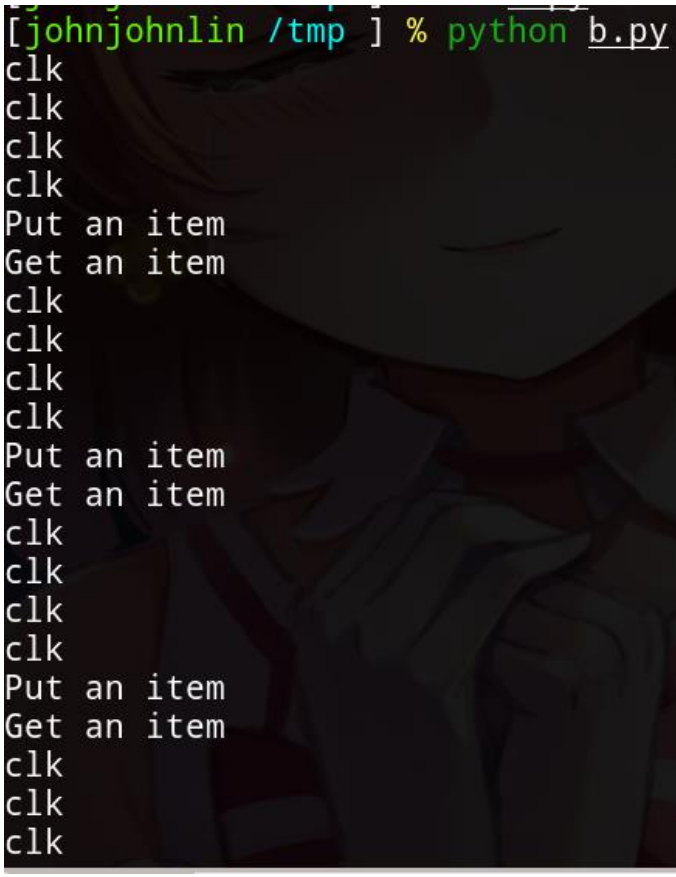
```
main_loop([Producer(10), Consumer(10)])
```

Add Consumer to scheduled threads

- Now use 3 yields for Producer and 2 yields for Comsumer, such that Producer is slower than Comsumer.
- Do not let the Comsumer consumes data when there is no item!!

Run the Producer

- What if we use Comsumer(11)?
  - Why?
- What if we swap the order in main_loop()?
  - `main_loop([Consumer(10), Producer(10)])`
  - Is there any difference?
  - Do you think this is reasonable?



Example #2
Event

Improve Consumer by Events

- You learnt Event in SystemC of Lab#1.
- Now Consumer checks whether something are produced every cycle.
- Can Consumer wait until something are produced?
  - We need event!

A Basic Event

```
def f():
    yield 1
    print("done 1")
    yield 0
    print("done 0")
    yield 2
    print("done 2")
```

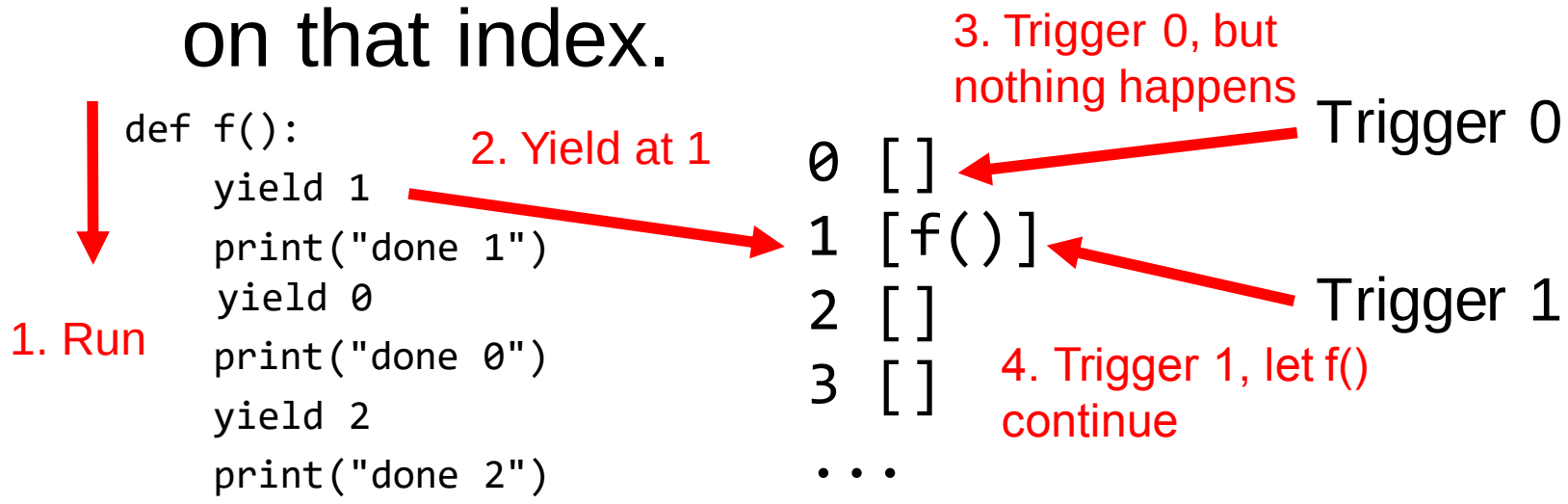
```
johnjohnlin /drive/DATA/msoccc/MSOC_v2/lab2  % python 2
handling 0
handling 1
handling 2
Done 2!
handling 3
handling 0
Done 0!
handling 1
Done 1!
handling 2
johnjohnlin /drive/DATA/msoccc/MSOC_v2/lab2  %
```

We trigger 0, 1, 2, 3, 0, 1, 2 in order

Give an index for each event

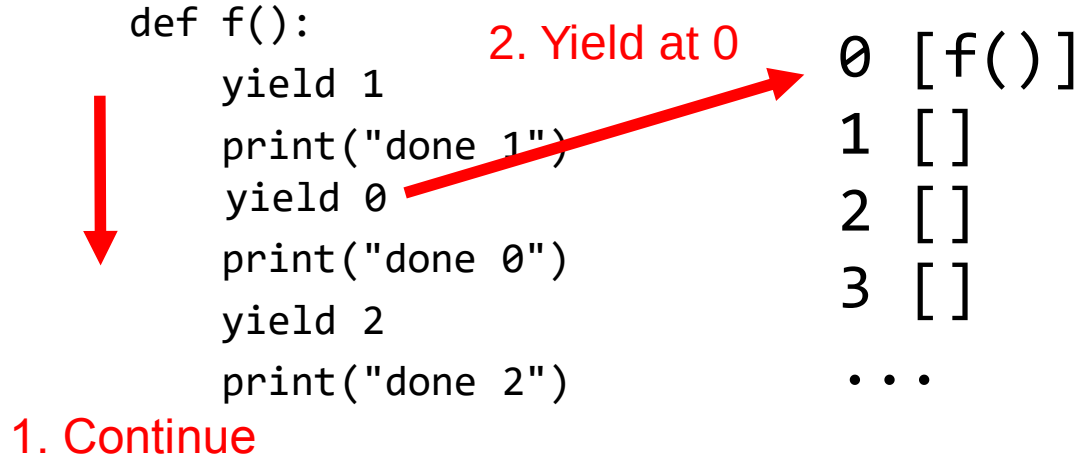
Implement Events (1)

- We have to store which thread is waiting on that index.



Implement Events (2)

- We have to store which thread is waiting on that index.



A More Complex Scheduler (1)

The most important function

```
def WaitOnNextEvent(t):
    try:
        waiting_on = next(t)
        event_pending_on_list[waiting_on].append(t)
    except StopIteration:
        pass
```

Continue/Run the threads

Make it wait on the yielded index

The thread might return

```
def main_loop(threads):
    TRIGGER = [0, 1, 2, 3, 0, 1, 2]
    for t in threads:
        WaitOnNextEvent(t)
    for trigger in TRIGGER:
        print("handling {}".format(trigger))
        handling, pending[trigger] = pending[trigger], list()
        for t in handling:
            WaitOnNextEvent(t)
```

A More Complex Scheduler (2)

```
def WaitOnNextEvent(t):
    try:
        waiting_on = next(t)
        event_pending_on_list[waiting_on].append(t)
    except StopIteration:
        pass

def main_loop(threads):
    TRIGGER = [0, 1, 2, 3, 0, 1, 2]
    for t in threads:
        WaitOnNextEvent(t)
    for trigger in TRIGGER:
        print("handling {}".format(trigger))
        handling, pending[trigger] = pending[trigger], list()
        for t in handling:
            WaitOnNextEvent(t)
```

Initially, all threads is pending, so we run all of them

Implementation details: we should use a swap since a thread can wait on the same event twice

Run all pending threads

No Hard-Coded Events (1)

```
def main_loop(threads):
    TRIGGER = [0, 1, 2, 3, 0, 1, 2]
    for t in threads:
        WaitOnNextEvent(t)
    for trigger in TRIGGER:
        ...

def MyThread(t):
      TriggerEvent(22)
```

We hard-codes the events here

But we should be able to trigger events in our own threads.

No Hard-Coded Events (2)

```
from collections import queue
TRIGGER = queue()

def main_loop(threads):
      for t in threads:
       WaitOnNextEvent(t)
   for trigger in TRIGGER:
       ...

def TriggerEvent(idx):
    ???

def MyThread(t):
    TriggerEvent(22)
```

Need something like that

Example: Consumer Revisits

```
def Consumer(n):
    global n_item
    n_get = 0
    while n_get < n:
        if XXX:
            yield 33
        # consume an item
        yield 22

def Producer(n):
   ...
   TriggerEvent(33)
```

Empty?

Give the event an ID you like
Let 33 = wait FIFO write

Wait for a cycle
Let 22 = clock

You should be familiar with that.

Implement Events (3)

- If we trigger an event, we still must process all threads pending on this event before we wake it up.

```
def Producer():
    ...
    Trigger(33)
```

1-2. Here

1-1. We are here

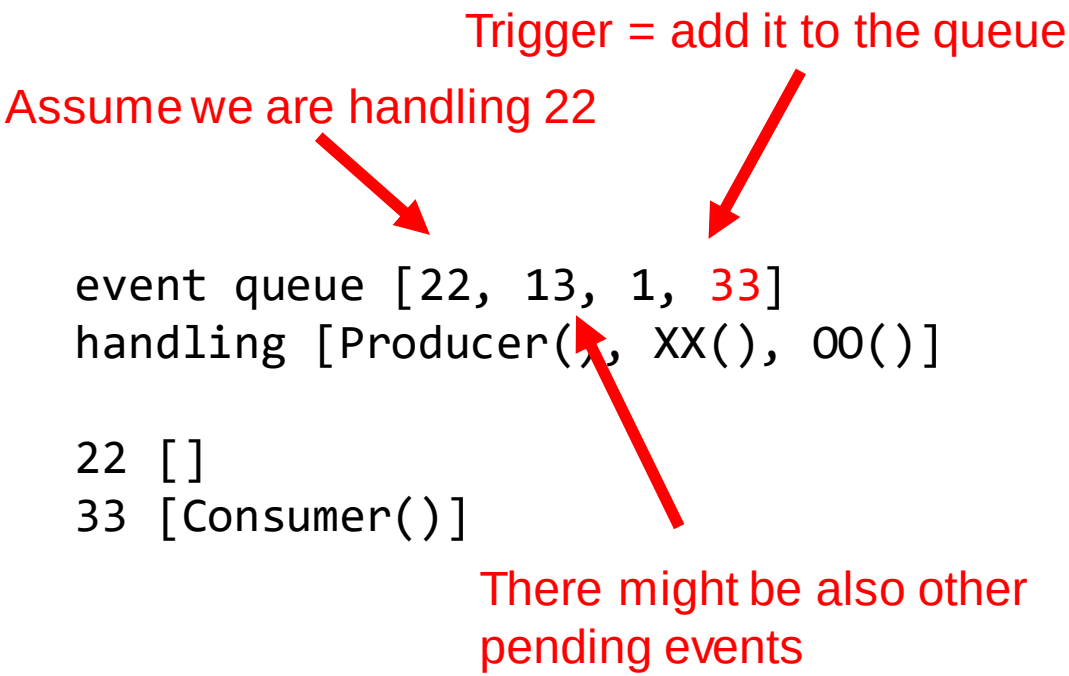
```
handling [Producer(), XX(), OO()]

22 []
33 [Consumer()]
```

2. Who should go first?

Implement Events (4)

- Usually, events are implemented as a queue.



Scheduler with Event (1)

```
def WaitOnNextEvent(t):
    try:
        waiting_on = next(t)
        event_pending_on_list[waiting_on].append(t)
    except StopIteration:
        pass


def main_loop(threads):
    for t in threads:
        WaitOnNextEvent(t)
    while ???:
        trigger = ???
        print("handling {}".format(trigger))
        handling, pending[trigger] = pending[trigger], list()
        for t in handling:
            WaitOnNextEvent(t)
```

No modification!!

No more hard-coded events

TODO

Scheduler with Event (2)

```
INIT_EVENT, WRITE_EVENT, CLOCK = 10, 20, 30

from collections import deque
TRIGGER = deque()
TRIGGER.append(INIT_EVENT)

def main_loop(threads):
    for t in threads:
        WaitOnNextEvent(t)
    while ???:
        trigger = ???
        print("handling {}".format(trigger))
        handling, pending[trigger] = pending[trigger], list()
        for t in handling:
            WaitOnNextEvent(t)
```

We name the events.

Queue type in Python.
We add an auto initialization event before the whole simulation.

TODO:
Now we have to obtain an event from the event queue TRIGGER.

Scheduler with Event (3)

```
INIT_EVENT, WRITE_EVENT, CLOCK = 10, 20, 30

def Clock(n):
    yield INIT_EVENT
    ???


def Consumer(n):
    ...
    if ???:
        yield WRITE_EVENT
    ...

main_loop([Clock(100), Producer(10), Consumer(10)])
```

This one generates 100 cycles after the initialization phase.

Wait if FIFO is empty.

This read is always valid.

We have 3 threads now.

Simplify the Interface

It can be such simple!
Note: an yield from is required for calling a generator.

```
def Consumer(n):
    ...
    if ???:
        yield WRITE_EVENT
    ...
```

This is too complex.
Remember that we don't want to care about the low-level things like what happens in every cycle?

```
def Consumer(n):
    n_get = 0
    while n_get < n:
        yield from GetAnItem()
        n_get += 1


def GetAnItem():
    ...
    if ???:
        yield WRITE_EVENT
    ...
```

Cut from Consumer.
Congrats, you just implement sc_fifo::read() from zero!

Example #3
dont_initialize

Example #3 Bonus Part

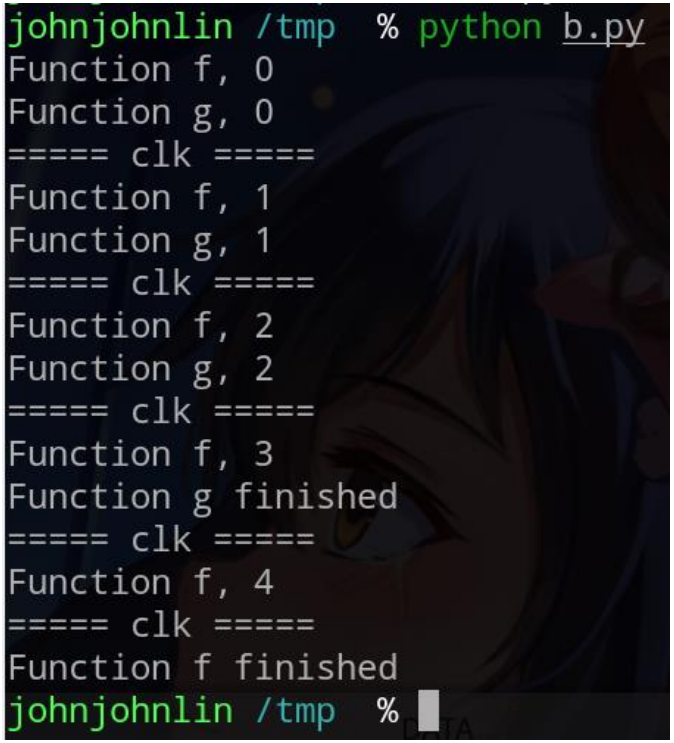
- (Bonus part!!!)
- Think about this: What is dont_initialize in SystemC?
- Actually you can implement based on Example #2 easily.
  - Hint: what is the effect of yield INIT_EVENT in Example #2?

Assignment #2

Requirements

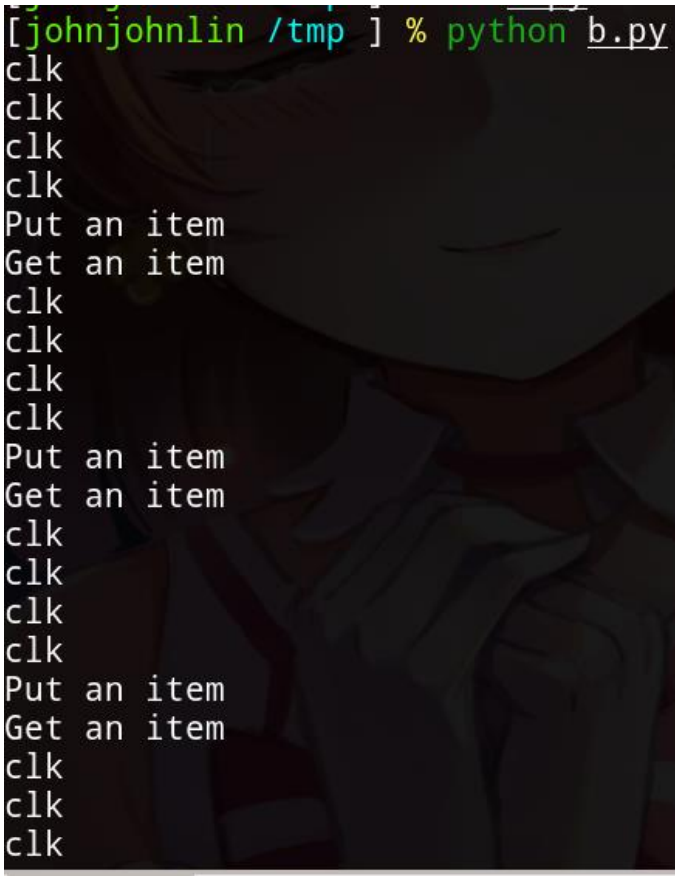
Lab Requirement (Example #0)

- Build and run it, an example is shown here.
- Discuss in report.pdf
  - Please draw a complete program execution flow.
  - Also, we use the zip_iterator to schedule generators, please explain how it works.
    - Do not just copy the document!
    - (in Python2, its name is izip_iterator)



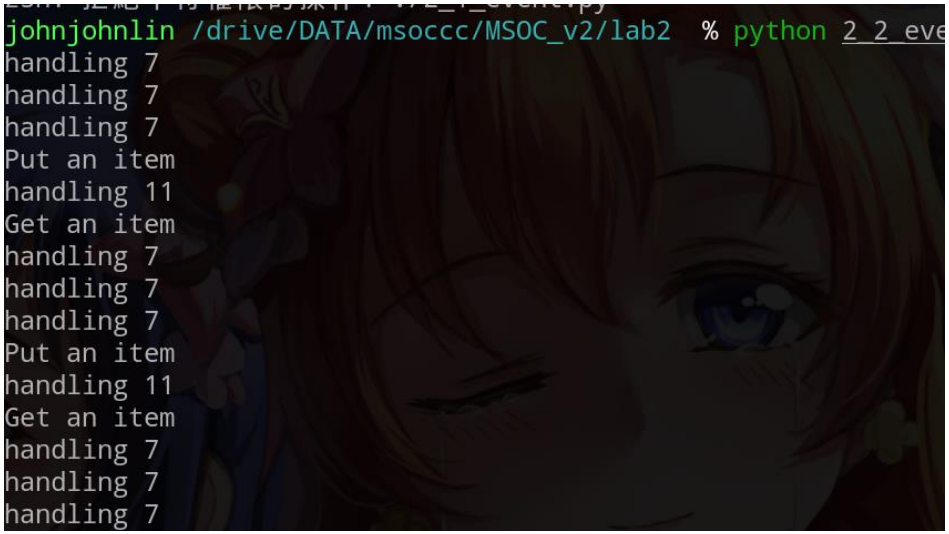
Lab Requirement (Example #1)

- Submission:
  - 1_prod_con.py
- Discuss in report.pdf
  - What if we use Comsumer(11)?
  - What if we swap the order in main_loop()?
  - (See also the last page of Example #1)



Lab Requirement (Example #2)

- Implement all TODOs in 2_2_event_fifo.py
  - Python 3 deque document
- Submission
  - 2_2_event_fifo.py



Take-Home HW (Example #3)

- (Bonus part!!!)
- Think about this: what is dont_initialize in SystemC?
- Submission
  - 3_dont.py (based your example #2)
- Discuss in report.pdf:
  - What does dont_initizlize mean in SystemC?
  - How do you implement it?
  - Give an example about the difference with/without dont_initialize.
  - You won't get any bonus if you do not submit a report.

Grading Rule

- Example #0 (0/15%)
- Example #1 (20/20%)
- Example #2 (45/0%)
- Example #3 (20/20%)

- (Program Score/Report Score)