

Effective Python

- Ch36 자식 프로세스를 관리하려면 `subprocess` 를 사용하자
- Ch37 스레드를 블로킹 I/O 용으로 사용하고, 병렬화용으로는 사용하지 말자
- Ch38 스레드에서 데이터 경쟁을 막으려면 `Lock` 을 사용하자

Chapter 36

자식 프로세스를 관리하려면 subprocess 를 사용하자

Subprocess ?

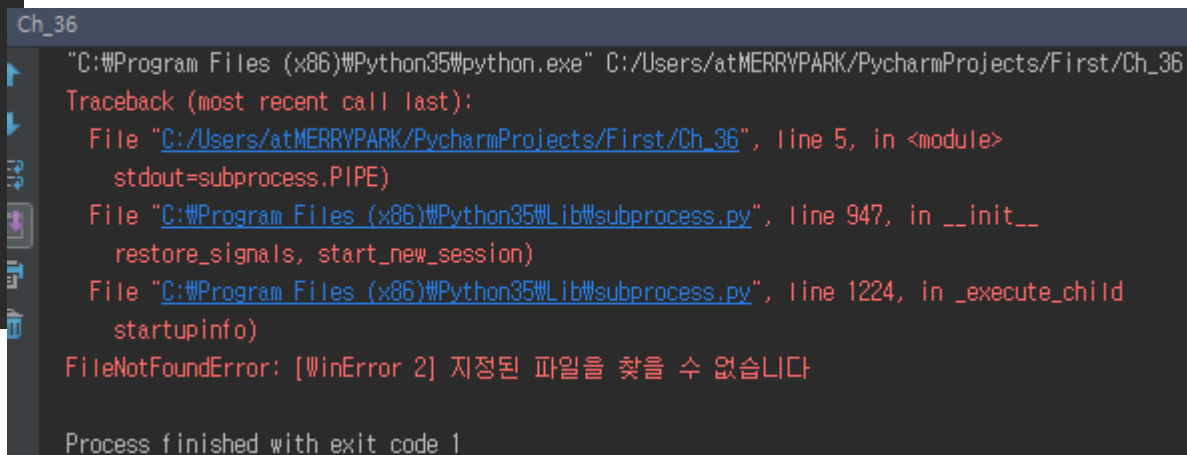
Python 의 내장모듈 중 하나

현재 Python 에서 자식 프로세스를 관리하는 방법 중에
최선이자 간단한 방법

Example - Subprocess

```
import subprocess

proc = subprocess.Popen(
    ['echo', 'Hello from the child!'],
    stdout=subprocess.PIPE)
out, err = proc.communicate()
print( out.decode('utf-8') )
```



```
Ch_36
"C:\Program Files (x86)\Python35\python.exe" C:/Users/atMERRY PARK/PycharmProjects/First/Ch_36
Traceback (most recent call last):
  File "C:/Users/atMERRY PARK/PycharmProjects/First/Ch_36", line 5, in <module>
    stdout=subprocess.PIPE)
  File "C:\Program Files (x86)\Python35\Lib\subprocess.py", line 947, in __init__
    restore_signals, start_new_session)
  File "C:\Program Files (x86)\Python35\Lib\subprocess.py", line 1224, in _execute_child
    startupinfo)
FileNotFoundError: [WinError 2] 지정된 파일을 찾을 수 없습니다

Process finished with exit code 1
```

Effective Python 의 SampleSource 는 윈도우 플랫폼 기반이 아니었다 ^-^

Example - Subprocess

```
proc = subprocess.Popen(  
    ['echo', 'Hello from the child!'],  
    stdout=subprocess.PIPE)  
  
out, err = proc.communicate()  
print( out.decode('utf-8') )
```

Linux/Posix

```
proc = subprocess.Popen(  
    'echo hello from the child!',  
    stdout=subprocess.PIPE, shell=True)  
  
out, err = proc.communicate()  
print( out.decode('utf-8') )
```

Windows

책의 소스들을 모두 윈도우 버전으로 돌아가게끔 바꾸기 위해 인터넷을 헤매고 ^-^
(그대로 돌아가는게. 한 손으로 셀 수. 있었어요.)

그래서 이 PPT 에서는 윈도우 플랫폼에서 돌아가는 샘플소스 입니다

Example - Subprocess

```
# 1
proc = subprocess.Popen(
    'echo hello from the child!',    # args    - 실행시킬 프로그램과 넘겨줄 인자
    stdout=subprocess.PIPE,         # stdout  - 자식프로세스에서 사용할 standard output file handle
    shell=True)                     # shell   - shell 을 이용하여 프로세스 실행

out, err = proc.communicate()       # 자식 프로세스의 output 을 read
print( out.decode('utf-8') )        # 받아온 output 을 출력

>> hello from the child!
```

```
#2
proc2 = subprocess.Popen('sleep 0.1', shell=True)
while proc2.poll() is None:        # 자식 프로세스가 종료된 상태가 아니면, return code 가 None
    print('Working...')
    # 시간이 걸리는 작업 몇 개를 수행함
    # ...
print( 'Exit status', proc2.poll())

>> Working...
    Working...
    Exit status 1
```

Example - Subprocess

```
#3
import time

def run_sleep(period):
    cmd = 'sleep %d' % period
    proc3 = subprocess.Popen(cmd, shell=True)
    return proc3

# 병렬 실행 할 경우.
start = time.time()

procs = []
for _ in range(10):
    proc = run_sleep(1)
    procs.append(proc)

for proc in procs:
    proc.communicate() # 자식 프로세스가 terminate 될 때 까지 기다린다

end = time.time()
print('Finished in %.3f seconds' % ( end - start ))

# 순차 실행 할 경우.
start2 = time.time()

for _ in range(10):
    proc = run_sleep(1)
    proc.communicate()

end2 = time.time()
print('Finished in %.3f seconds' % ( end2 - start2 ))
```

```
>> Finished in 0.049 seconds
    Finished in 0.141 seconds
```

병렬 실행이 순차 실행보다 빠르다

Example - Subprocess

#4

```
def findstr():  
    proc = subprocess.Popen('findstr "aaa"',  
                             shell=True,  
                             stdin=subprocess.PIPE,      # standard input file handle 을 pipe 로 생성  
                             stdout=subprocess.PIPE,     # standard output file handle 을 pipe 로 생성  
                             universal_newlines=True)     # stdin, stdout, stderr 데이터를 string 으로 받는다  
  
    return proc
```

```
out, err = findstr().communicate('aaa bbb ccc') # 자식프로세스로 'aaa bbb ccc' 데이터 write  
                                                # communicate 로 read. 자식프로세스가 종료될 때까지 wait
```

```
if len( out.strip() ) == 0:  
    print('cant find')
```

```
else:  
    print( out.strip() )
```

```
out2, err2 = findstr().communicate('abcedf')
```

```
if len( out2.strip() ) == 0:  
    print('canW't find')
```

```
else:  
    print( out2.strip() )
```

```
>> aaa bbb ccc  
can't find
```

Example - Subprocess

#5 Subprocess Chaining

```
p1 = subprocess.Popen('netstat -ano',  
                        stdin=subprocess.PIPE,  
                        stdout=subprocess.PIPE,  
                        universal_newlines=True,  
                        )  
  
p2 = subprocess.Popen('findstr "LISTENING"',  
                        stdin=p1.stdout, # p1의 output 과 p2의 input 을 연결  
                        # stdin=subprocess.PIPE,  
                        stdout=subprocess.PIPE,  
                        universal_newlines=True)  
  
print( p1.communicate()[0].strip())  
p1.stdout.close()  
print('=====')  
print( p2.communicate()[0].strip() )
```

Example - Subprocess

#5 Subprocess Chaining

```
p1 = subprocess.Popen('netstat -ano',  
                      stdin=subprocess.PIPE,  
                      stdout=subprocess.PIPE,  
                      universal_newlines=True,  
                      )  
  
p2 = subprocess.Popen('findstr "LISTENING"',  
                      stdin=p1.stdout,  
                      # stdin=subprocess.PIPE,  
                      stdout=subprocess.PIPE,  
                      universal_newlines=True)  
  
print( p1.communicate()[0].strip()  
p1.stdout.close()  
print( '=====  
print( p2.communicate()[0].strip() )
```



활성 연결

프로토콜	로컬 주소	외부 주소	상태	PID
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING	896
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING	4
TCP	0.0.0.0:945	0.0.0.0:0	LISTENING	17792
TCP	0.0.0.0:1536	0.0.0.0:0	LISTENING	620
TCP	0.0.0.0:1537	0.0.0.0:0	LISTENING	980
TCP	0.0.0.0:1538	0.0.0.0:0	LISTENING	808
TCP	0.0.0.0:1539	0.0.0.0:0	LISTENING	2080
TCP	0.0.0.0:1552	0.0.0.0:0	LISTENING	740
TCP	0.0.0.0:1569	0.0.0.0:0	LISTENING	756
TCP	0.0.0.0:3973	0.0.0.0:0	LISTENING	7544
TCP	0.0.0.0:65004	0.0.0.0:0	LISTENING	2632
TCP	121.157.218.183:139	0.0.0.0:0	LISTENING	4
TCP	121.157.218.183:3489	111.221.29.160:443	ESTABLISHED	15124
TCP	121.157.218.183:3833	27.0.238.248:5223	ESTABLISHED	14368
TCP	121.157.218.183:3942	192.99.63.220:80	CLOSE_WAIT	18032
TCP	121.157.218.183:3959	72.246.184.11:443	ESTABLISHED	7544
=====				
TCP	0.0.0.0:4433	0.0.0.0:0	LISTENING	2868
TCP	0.0.0.0:4433	0.0.0.0:0	LISTENING	2868
TCP	0.0.0.0:5357	0.0.0.0:0	LISTENING	4
TCP	0.0.0.0:5644	0.0.0.0:0	LISTENING	1420
TCP	0.0.0.0:5645	0.0.0.0:0	LISTENING	1420
TCP	0.0.0.0:14430	0.0.0.0:0	LISTENING	332
TCP	0.0.0.0:14440	0.0.0.0:0	LISTENING	332
TCP	0.0.0.0:21201	0.0.0.0:0	LISTENING	17792
TCP	0.0.0.0:28263	0.0.0.0:0	LISTENING	2632

About - Subprocess

Subprocess 가 자식 프로세스를 관리하는 방법

1. run() function

Python3.5 이후에 생김

모든 경우에서 자식프로세스 제어가 가능하므로 추천한다

2. Popen Class

더 디테일하게 제어해야하는 경우, Popen interface 가 run() 보다 상대적으로 좋다

Popen ?

class 이다

Popen Constructor

```
class Popen(object):  
  
    _child_created = False # Set here since __del__ checks it  
  
    def __init__(self, args, bufsize=-1, executable=None,  
                  stdin=None, stdout=None, stderr=None,  
                  preexec_fn=None, close_fds=_PLATFORM_DEFAULT_CLOSE_FDS,  
                  shell=False, cwd=None, env=None, universal_newlines=False,  
                  startupinfo=None, creationflags=0,  
                  restore_signals=True, start_new_session=False,  
                  pass_fds=()):
```

Popen Constructor

```
proc = subprocess.Popen('findstr "LISTENING"',  
                        stdin=subprocess.PIPE,  
                        stdout=subprocess.PIPE,  
                        universal_newlines=True,  
                        shell=True  
                        )
```

args :

프로세스에 넘겨주는 파라미터 타입은, sequence type or single string
sequence 일 경우, first item 이 실행시키고자 하는 프로그램
single string 경우, 플랫폼에 따라 인자의 순서가 의미하는 바가 다를 수 있다

sequence type ? str, bytes, bytearray, list, tuple, range

(<https://docs.python.org/3.1/library/stdtypes.html#sequence-types-str-bytes-bytearray-list-tuple-range>)

! 윈도우에서는 args (sequence data) 를 string 으로 Converting 해주어야한다

(<https://docs.python.org/3/library/subprocess.html#converting-argument-sequence>)

Popen Constructor

```
proc = subprocess.Popen('findstr "LISTENING"',  
                        stdin=subprocess.PIPE,  
                        stdout=subprocess.PIPE,  
                        universal_newlines=True,  
                        shell=True  
                        )
```

shell :

True - shell 을 이용하여 프로그램을 실행. 이 경우, args type 을 sequence 보다 string 추천
False - shell 을 사용하지 않음

stdin, stdout, stderr :

윈도우에서는 Windows file handles 이다
PIPE - pipe
DEVNULL - os.devnull

universal_newlines :

True - FileObject stdin, stdout, stderr 가 text stream 을 취급한다
False - FileObject stdin, stdout, stderr 가 binary stream 을 취급한다

Popen.wait() vs Popen.poll()

자식 프로세스의 종료 상태를 알 수 있는 function

Popen.poll()

함수 호출 시점에, 자식 프로세스의 종료 여부를 리턴 한다
return code 가 None 이면 종료된 것을 의미

```
#2
proc2 = subprocess.Popen('sleep 0.1', shell=True)
while proc2.poll() is None:    # 자식 프로세스가 종료된 상태가 아니면, return code 가 None
    print('Working...')
    # 시간이 걸리는 작업 몇 개를 수행할
    # ...
print('Exit status', proc2.poll())
```

Popen.wait()

프로세스가 종료 될 때까지 blocking (hang)

Popen.wait() 은 deadlock 위험이 있다

대표 사례.

stdout = PIPE 일 경우.

pipe 의 buffer 가 꽉 차면 (전형적으로 4kb)

Writing process 는 Reading Process 가 일정량의 data 를 read 할 때까지 기다린다

Reading process 는 wait() ... Writing Process 의 종료를 기다린다

그래도 stdout=PIPE 를 쓰고 싶고, deadlock 을 피하고 싶다면?

자식 프로세스가 tempfile 에 output 하게 하여, 작업이 완료되면
부모 프로세스가 tempfile 을 read 한다

Popen.communicate() 를 쓰면 deadlock 을 피할 수 있다?

Note: This will deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use `Popen.communicate()` when using pipes to avoid that.

근데..

`Popen.communicate(input=None, timeout=None)`

Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate.

게다가..

책의 핵심정리에서도 communicate deadlock 조심하라고 한다

Popen.communicate() 와 Popen.wait() 은 timeout 값을 받을 수 있다

```
Popen . wait (timeout=None)
```

```
Popen . communicate (input=None, timeout=None)
```

지정된 time 이 지나도 자식프로세스가 종료되지 않으면,
TimeoutExpired Exception 을 발생 시킨다

Popen.communicate()

- 자식 프로세스와의 IO Interface
- Input ? communicate('input data here')
 Outout ? out, err = communicate()
- communicate() return type 은 tuple (stdout_data, stderr_data)
 -> universal_newlines bool 값에 따라 string or bytes
- 자식 프로세스가 종료 될 때 까지 대기
- eof 가 나올 때 까지 read

Chapter 36 _ 핵심

- 자식 프로세스를 실행하고 자식 프로세스의 입출력 스트림을 관리하려면 subprocess 모듈을 사용하자
- 자식 프로세스는 파이썬 인터프리터에서 병렬로 실행되어 CPU 사용을 극대화 하게 해준다
- communicate 에 timeout 파라미터를 사용하여, 자식 프로세스들이 deadlock 되지 않게 하자

Chapter 37

스레드를 블로킹 I/O 용으로 사용하고, 병렬화용으로 사용하지 말자

```
import time

# multi-thread vs single-thread

def factorize(number):
    for i in range(1, number + 1):
        if number % i == 0:
            yield i

numbers = [2139079, 1214759, 1516637, 1852285]
```

```
#1 single-thread
start = time.time()

for number in numbers:
    list(factorize(number))

end = time.time()
print('Took %.3f seconds' % (end - start))
```

```
>>Took 1.374 seconds
```

```
# 2 multi-thread
from threading import Thread

class FactorizeThread(Thread):
    def __init__(self, number):
        super().__init__()
        self.number = number

    def run(self):
        self.factors = list(factorize(self.number))

start = time.time()
threads = []
for number in numbers:
    thread = FactorizeThread(number)
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join()

end = time.time()
print('Took %.3f seconds' % (end - start))
```

```
>>Took 1.400 seconds
```

Multi-thread 처리시간이 더 느리다

Single-thread vs multi-thread

Multi-thread 가 더 느린 이유?

GIL (Global Interpreter Lock)

Lock 은
Detail 에 따라
두 종류로 나눌 수 있다

Coarse-grained-lock	:	거칠게 그냥 싹 다
Fine-grained-lock	:	세세하게

Ex.

디스크에 파일을 쓰고 싶은데,
모든 디스크에 락을 거는게 coarse-grained-lock
특정 디스크에 락을 거는게 fine-grained-lock

Coarse-grained-lock

Fine-grained-lock

GIL

GIL

어떤 시점에서든
하나의 `bytecode` 만
실행하도록
강제

note. Interpreter 를 실행하면
1. Compile – `bytecode` 변환
2. `bytecode` 한 줄씩 실행

multi-thread 일 지라도
실제로는 한 시점에 1개의 `thread` 만 수행되고 있다

GIL

Mutex (와 비슷하다)

(<http://www.dabeaz.com/python/UnderstandingGIL.pdf>)

Python Locks

- The Python interpreter only provides a single lock type (in C) that is used to build all other thread synchronization primitives
- It's not a simple mutex lock
- It's a binary semaphore constructed from a pthreads mutex and a condition variable
- The GIL is an instance of this lock

아니 그럼
Multi-thread 는 왜 지원하는거야?

아니 그럼
GIL 이 왜 있는거야?

는
뒤에서

책에서는,
System Call 의 경우, GIL 의 영향을 받지 않아서 multi-thread 효과를 볼 수 있다
라고 설명합니다

```
import select

def slow_systemcall():
    select.select([], [], [], 0.1)

start = time.time()

for _ in range(5):
    slow_systemcall()

end = time.time()
print('Took %.3f seconds' % (end - start))

>> Took 0.503 seconds
```

```
from threading import Thread

start = time.time()
threads = []

for _ in range(5):
    thread = Thread(target=slow_systemcall)
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join()

end = time.time()
print('Took %.3f seconds' % (end - start))

>> Took 0.102 seconds
```

하지만 예제가 윈도우에서 돌아가지 않습니다

예제에서 언급한 SystemCall 의 한 종류, Blocking I/O 의 종류를 찾고,
예제를 바꿔서 테스트 해보면..

Blocking I/O ?

- File read/write
- Network Interaction
- Display Device Interaction

Blocking I/O – File read/write_single-thread vs multi-thread

```
def factorize(number):
    for i in range(1, number + 1):
        if number % i == 0:
            yield i
numbers = [2139079, 1214759, 1516637, 1852285]

def slowsystem_call(number):
    txtName = 'blockingio%d.txt' % number
    f = open(txtName, 'w')
    for i in range(1000000):
        data = '%d line.Wn' % i
        f.write(data)
    f.close()
    print(number)
    print(txtName)

start = time.time()
for i in range(5):
    slowsystem_call(i)
for number in numbers:
    list(factorize(number))
end = time.time()
print('Took %.3f seconds' % (end - start))
```

0
blockingio0.txt
1
blockingio1.txt
2
blockingio2.txt
3
blockingio3.txt
4
blockingio4.txt
Took 6.592 seconds

```
from threading import Thread

start_2 = time.time()
threads = []
for i in range(5):
    thread = Thread(target=slowsystem_call, args=(i,))
    thread.start()
    threads.append(thread)

for number in numbers:
    list(factorize(number))
for thread in threads:
    thread.join()
end_2 = time.time()
print('Took %.3f seconds' % (end_2 - start_2))
```

0
blockingio0.txt
4
blockingio4.txt
1
blockingio1.txt
3
blockingio3.txt
2
blockingio2.txt
Took 6.505 seconds

Blocking I/O – File read/write_single-thread vs multi-thread

왜 책처럼 차이가 안 나는 거지...
(Multi-thread 가 왜 더 안 빠르지)

책 설명을 다시 보면.

GIL 은 파이썬 코드가 병렬로 실행하지 못하게 한다.
하지만 SystemCall 에서는 이런 부정적인 영향이 없다.
이는 파이썬 스레드가 시스템 호출을 만들기 전에 GIL 을 풀고
시스템 호출의 작업이 끝나는 대로 GIL 을 다시 얻기 때문이다

Blocking I/O – File read/write_single-thread vs multi-thread

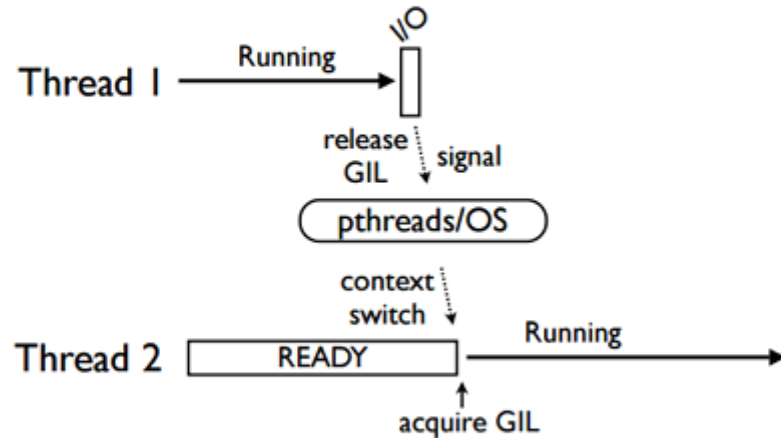
왜 책처럼 차이가 안 나는 거지...
(Multi-thread 가 왜 더 안 빠르지)

책 설명을 다시 보면.

GIL 은 파이썬 코드가 병렬로 실행하지 못하게 한다.
하지만 SystemCall 에서는 이런 부정적인 영향이 없다.
이는 파이썬 스레드가 시스템 호출을 만들기 전에 GIL 을 풀고
시스템 호출의 작업이 끝나는 대로 GIL 을 다시 얻기 때문이다

Thread Switching

- Easy case : Thread 1 performs I/O (read/write)



GIL 을 얻은 Thread 는 Run

GIL 을 못 얻은 Thread 는 Ready

- Release of GIL results in a signaling operation
- Handled by thread library and operating system

(<http://www.dabeaz.com/python/UnderstandingGIL.pdf>)

System Call 이 GIL 의 영향을 안 받는 이유?

1. A Thread 에서,
SystemCall 한 operation 는 OS 에서 할 테니
A Thread 는 GIL 을 놓고
쉬고 있어
2. B Thread 에게 GIL 을 주자
B Thread 일 해라
3. System Operation 작업이 끝나면
A Thread 에게 다시 GIL 을 주자

파이썬의 바이트 코드와 별개로 SystemOperation 이 수행된다

라고 이해하면 되지 않을까.

Windows 에서 돌아가게끔 바꾼 샘플소스는 왜 차이가 안 났을까?

```
def factorize(number):
    for i in range(1, number + 1):
        if number % i == 0:
            yield i
numbers = [2139079, 1214759, 1516637, 1852285]

def slowsystem_call(number):
    txtName = 'blockingio%d.txt' % number
    f = open(txtName, 'w')
    for i in range(1000000):
        data = '%d line.Wn' % i
        f.write(data)
        f.close()
    print(number)
    print(txtName)

start = time.time()
for i in range(5):
    slowsystem_call(i)
for number in numbers:
    list(factorize(number))
end = time.time()
print('Took %.3f seconds' % (end - start))
```

이 for 문 때문 일 것 같다 (결국 이게 파이썬 바이트코드)
f.write() 로 넘기는 데이터양이 많아야
샘플의 올바른 결과가 나올 듯 (multi-thread가 더 빠른 결과)

Python 에서 Multi-thread 를 지원하는 이유?

1. 프로그램이 동시에 여러 작업을 하는 것처럼 보이게 할 수 있다
GIL 때문에 한 번에 한 스레드만 진행하여,
CPython 은 균형감 있게 스레드들을 실행시킨다
2. 특정 SystemCall 시, 발생하는 Blocking I/O 를 다루기 위해서 이다

Python 에서 GIL 을 만든 이유?

1. Cpython 의 메모리 매니징은 thread-safe 하지 않다

2. fine-grained-lock 에 비해서 ~

- 1. single thread 일 때 훨씬 빠르다

- 2. i/o bound program 에 한해 multi-thread 인 경우 빠를 수 있다

 - > GIL 은 blocking i/o call 의 경우 해제되므로 성능에 영향이 없다

- 3. C library 를 사용하는 연산이 많은 cpu bound program 을 multi-thread 로 돌린 경우 더 빠를 수 있다

3. C extension 을 쓰기에 용이하다

 - > C extension 에 의해서 GIL 은 해제된다

4. C library 의 wrapping 에 용이하다

 - > C library 의 thread-safety 를 고려하지 않아도 된다

 - GIL 을 이용해서 lock 을 잡으면, 정상적으로 다 돌기 때문이다

Chapter 37 _ 핵심

- 파이썬 스레드는 전역 인터프리터 잠금(GIL) 때문에 여러 CPU 코어에서 병렬로 바이트코드를 실행 할 수 없다
- GIL 에도 불구하고 파이썬 스레드는 동시에 여러 작업을 하는 것처럼 보여주기 쉽게 해주므로 여전히 유용하다
- 여러 시스템 호출을 병렬로 수행하려면 파이썬 스레드를 사용하자
이렇게 하면 계산을 하면서도 블로킹 I/O 를 수행 할 수 있다

Chapter 38

스레드에서 데이터 경쟁을 막으려면 Lock 을 사용하자

GIL 은 프로그램의 데이터들을 thread-safe 하게 보호하지 않는다

```

from threading import Thread

class Counter(object):
    def __init__(self):
        self.count = 0

    def increment(self, offset):
        self.count += offset

def worker(sensor_index, how_many, counter):
    for _ in range(how_many):
        # 센서에서 읽어옴
        # ..
        counter.increment(1)

def run_threads(func, how_many, counter):
    threads = []
    for i in range(5):
        args = (i, how_many, counter)
        thread = Thread(target=func, args=args)
        threads.append(thread)
        thread.start()
    for thread in threads:
        thread.join()

```

```

how_many = 10**5
counter = Counter()
run_threads(worker, how_many, counter )
print('Counter should be %d, found %d' %
      (5 * how_many, counter.count) )

>> Counter should be 500000, found 357331

```

thread-safe 하지 않다!

```
# Counter 의 increment()  
counter.count += offset
```

```
# >> 파이썬이 보이지 않게 연산 3 개를 수행한다  
value = getattr(counter, 'count')  
result = value + offset  
setattr(counter, 'count', result)
```

```
# 스레드 A 에서 실행  
value_a = getattr(counter, 'count')  
# 스레드 B 로 컨텍스트 스위칭  
value_b = getattr(counter, 'count')  
result_b = value_b + 1  
setattr(counter, 'count', result_b)  
# 스레드 A 로 컨텍스트 스위칭  
result_a = value_a + 1  
setattr(counter, 'count', result_a)
```

Python 은, 모든 스레드가 균등하게 실행시키기 위하여 스레드를 잠시 중지하고 다른 스레드를 재개한다

데이터를 thread-safe 하게 만들기 위해, Lock 을 이용하자

```
from threading import Lock

class LockingCounter(object):
    def __init__(self):
        self.lock = Lock()
        self.count = 0

    def increment(self, offset):
        with self.lock:
            self.count += offset
```

with 문으로 Mutex 를 얻고 해제

Chapter 38 _ 핵심

- 파이썬에 전역 인터프리터 잠금이 있다고 해도 프로그램 안에서 실행되는 스레드 간의 데이터 경쟁으로부터 보호할 책임은 프로그래머에게 있다
- 여러 스레드가 잠금 없이 같은 객체를 수정하면 프로그램의 자료 구조가 오염된다
- 내장 모듈 threading 의 Lock 클래스는 파이썬의 표준 상호 배제 잠금 구현이다