PYTHON 병렬 프로그래밍

Moon Yong Joon

CONTENTS

1. PARALLEL COMPUTERS

Moon Yong Joon

PARALLEL COMPUTERS 용어 이해하기

Moon Yong Joon

Parallel computers

Parallel computers

Multiprocessor/multicore:

several processors work on data stored in shared memory

Cluster:

several processor/memory units work together by exchanging data over a network

Co-processor:

a general-purpose processor delegates specific tasks to a special-purpose processor (GPU, FPGA, ...)

Other:

- Grid computing
- Cluster of multicore nodes with GPUs
- NUMA (non-uniform memory access) architecture

grid computing

분산 병렬 컴퓨팅의 한 분야로서, 원거리 통신망으로 연결된 서로 다른 기종의 컴퓨터들을 묶어 가상의 대용량 고성능 컴퓨터를 구성하여 고도의 연산 작업(computation intensive jobs) 혹은 대용량 처리(data intensive jobs)를 수행하는 것을 일컫는다.

Parallelism vs. concurrency

Parallelism:

use multiple processors to make a computation faster.

Concurrency:

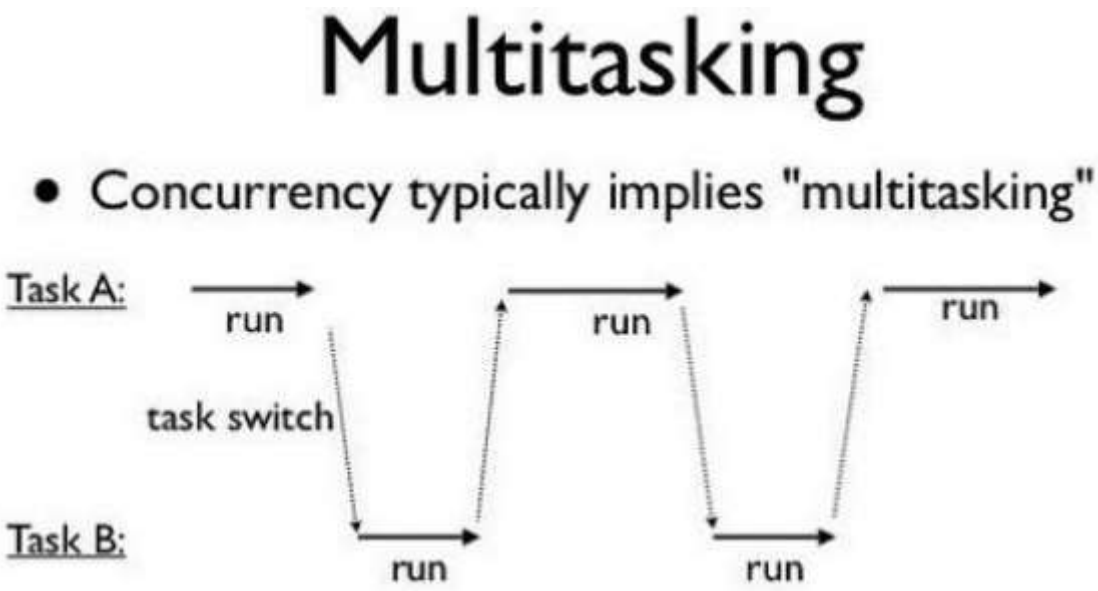
permit multiple tasks to proceed without waiting for each other.

Different goals that share implementation aspects. Scientific computing cares more about parallelism.

Concurrency is rarely needed.

concurrency

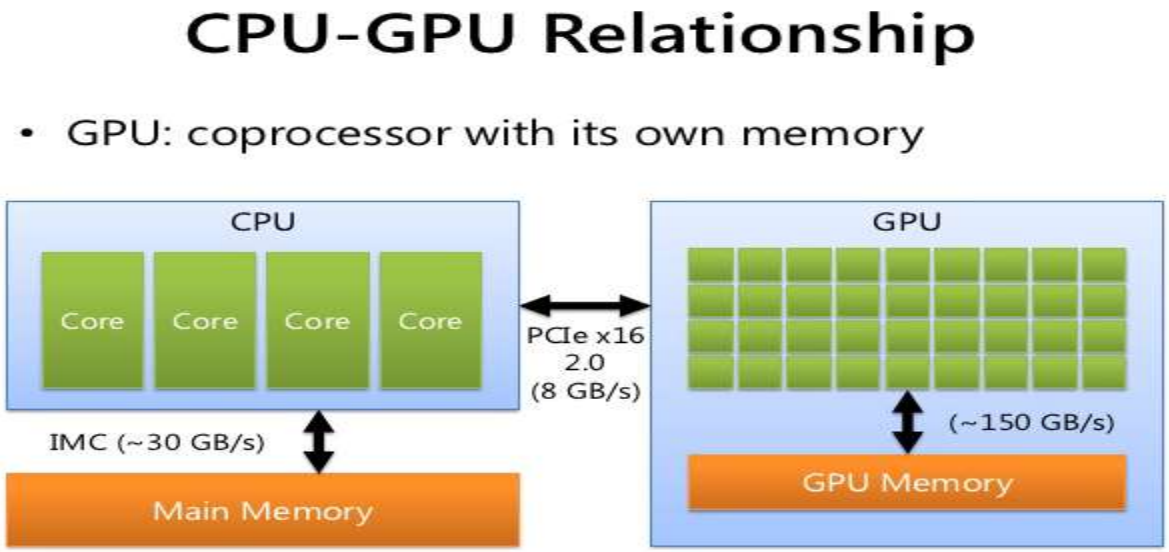
동시성 처리는 멀티태스킹 처리



HPC

GPU가속 컴퓨팅

GPU가속 컴퓨팅은 그래픽 처리 장치(GPU)와 CPU를 함께 이용하여 과학, 분석, 공학, 소비자 및 기업 애플리케이션의 처리속도를 높이는 것



grid computing

분산 병렬 컴퓨팅의 한 분야로서, 원거리 통신망으로 연결된 서로 다른 기종의 컴퓨터들을 묶어 가상의 대용량 고성능 컴퓨터를 구성하여 고도의 연산 작업(computation intensive jobs) 혹은 대용량 처리(data intensive jobs)를 수행하는 것을 일컫는다.

grid computing의 응용

- 컴퓨팅 그리드(Computational Grid): CPU나 GPU등의 기능을 이용해 복잡한 연산을 수행하는 것.
- 데이터 그리드(Data Grid): 대용량의 분산 데이터를 공유하고 관리하는 것.
- 액세스 그리드(Access Grid): 지리적으로 떨어진 곳에 있는 사용자들 간에 오디오와 비디오를 사용하여 업무 협력을 가능하게 하는 것.
- 장비 그리드(Equipment Grid) 망원경 등의 주요 장비를 원격 조정하며 장비로부터 얻은 데이터를 분석하는 것.

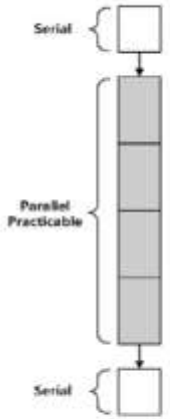
PARALLEL PROGRAMMING

Moon Yong Joon

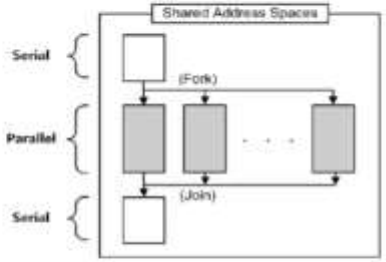
Parallel Programming

Parallel Programming

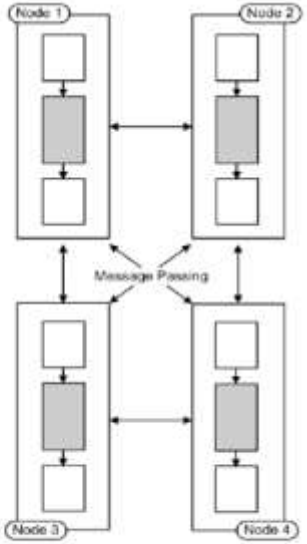
병렬 프로그래밍은 코드 명령어를 동시에 실행하기 위해 준비된 환경에서 프로그램을 만들고 실행하는 것.즉, 프로세스들이 동시에 프로그램을 실행



(a) Single Thread Process



(b) Shared-Memory Parallel Programmin Model



(c) Massage Passing Parallel Programming Model

Parallel Programming

- **Decomposition** of the complete task into independent subtasks and the data flow between them.
- **Distribution** of the subtasks over the processors minimizing the total execution time.
- For **clusters**: distribution of the data over the nodes minimizing the communication time.
- For **multiprocessors**: optimization of the memory access patterns minimizing waiting times.
- **Synchronization** of the individual processes

parallel programming problems

Deadlock

- **Deadlock:**
  – Two processes are waiting for each other to finish.
  - **Usually caused by locks or by blocking communication.**

Starvation

- **Starvation:**
  - **Now, imagine that a process A with high priority constantly consumes the CPU, while a lower priority process B never gets the chance.**

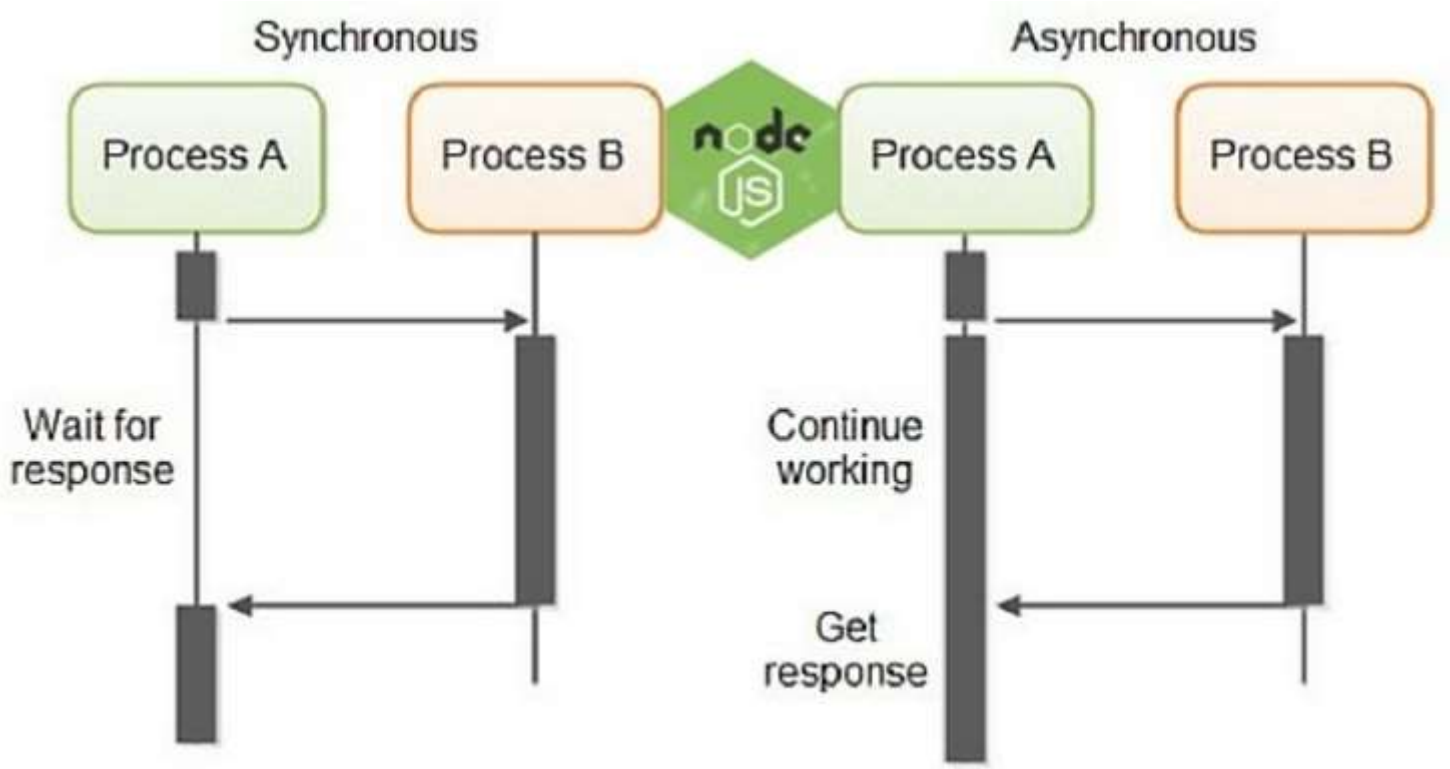
Race condition

- **Race condition:**
  - Two or more processes modify a shared resource (variable, file, ...)
  - Result depends on which process comes first.
  - Can be avoided using locks, but...
  - ... handling locks is very difficult and mistakes often cause deadlocks.

Sync/async

Sync/async

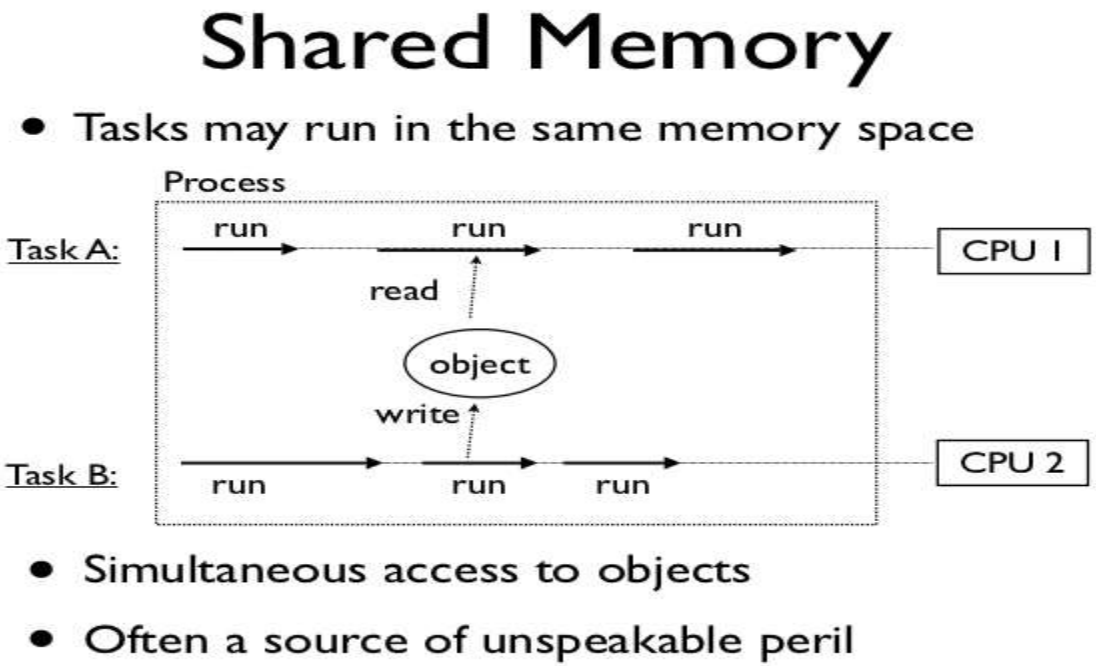
프로세스 등의 처리시 동기화 처리에 대한 기준



Shared memory communication

Shared memory

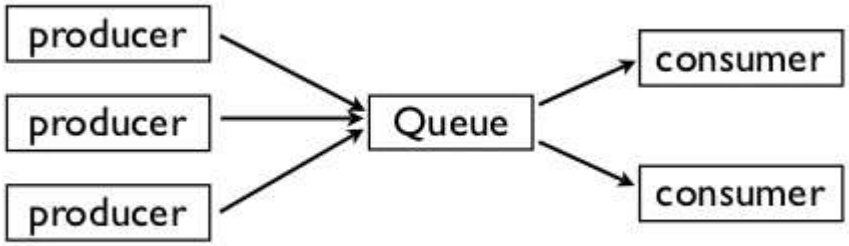
- **Shared memory (threading, multiprocessing)**
  Requires locks for safe modification

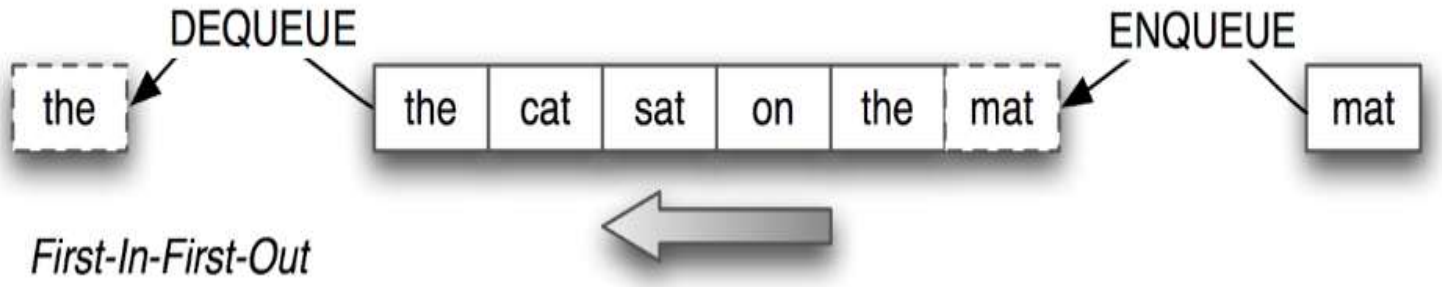


Message passing communication

Queue programming

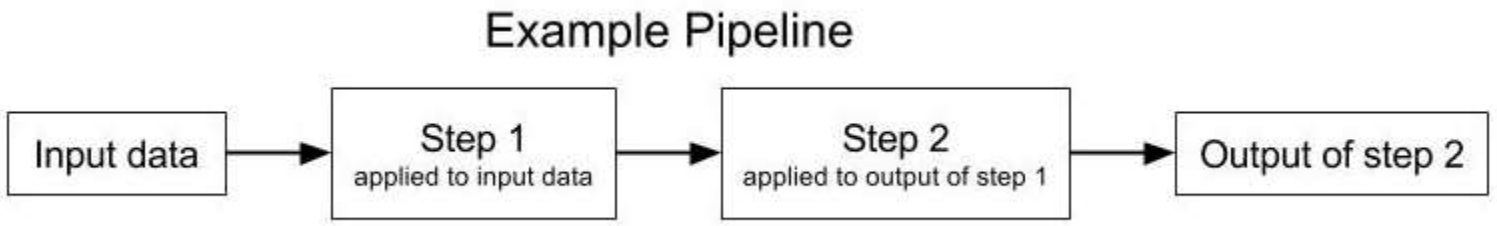
Queue를 중심으로 생산자와 소비자간의 통신을 통해 연결해 주는 프로그램





pipeline programming

Task를 pipeline으로 분리해서 각 단계별로 처리 후에 결과를 다음 단계의 input으로 처리하는 방법



프로세스와 쓰레딩 이해하기

Moon Yong Joon

process

프로세스 란

A process consists of

- a block of memory
- some executable code
- one or more threads that execute code independently but work on the same memory

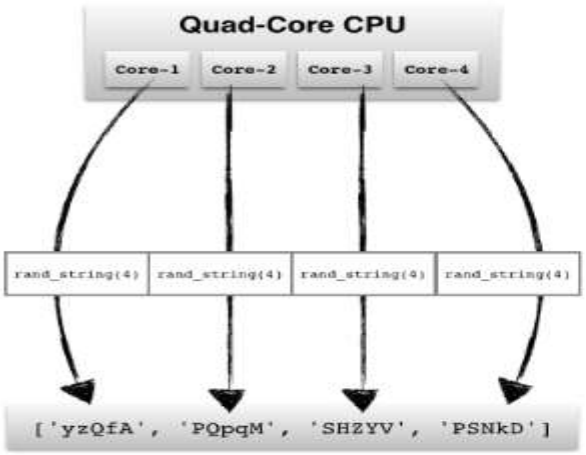
Process

- A running program
  - Keeps track of current instruction and data
- Single-core processor: only one process actually runs at a time
  - Many processes “active” at once – OS goes from one to another via a context switch

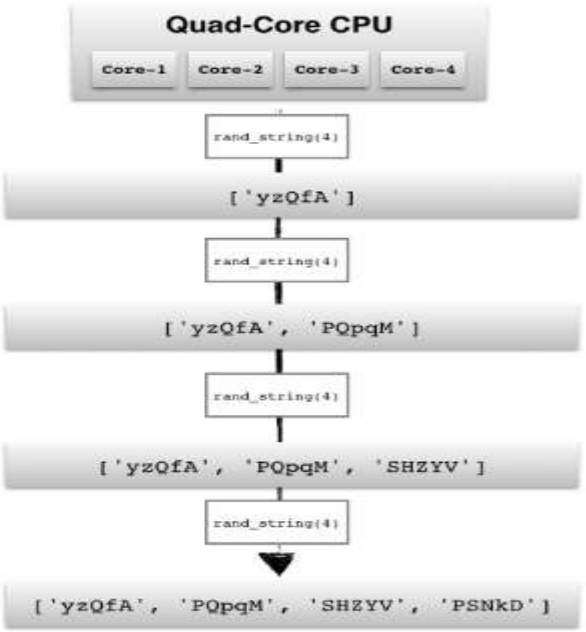
프로세싱 처리 이해

병렬 프로세싱은 여러 core CPU를 동시에 작업을 실행하는 방식

[parallel processing]



[serial processing]



Multiprocessing

Multiprocessing:

using multiple processes with separate memory spaces for concurrency or parallelism

대칭형 멀티프로세싱(symmetric) :

시스템 프로세스와 응용프로그램 프로세스가 가용 cpu를 모두 이용해 실행

비대칭형 멀티프로세싱(Asymmetric) :

프로세스를 특정 CPU에 지정해 수행. 일단 시작되면 다른 CPU와 상관없이 해당 CPU만 동작(로드 분산에 제약)

Thread 란

Thread 란

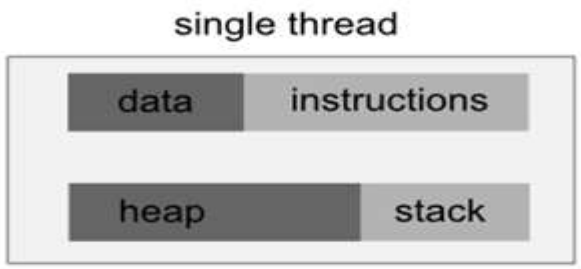
스레드(thread)는 어떠한 프로그램 내에서, 특히 프로세스 내에서 실행되는 흐름의 단위를 말한다.

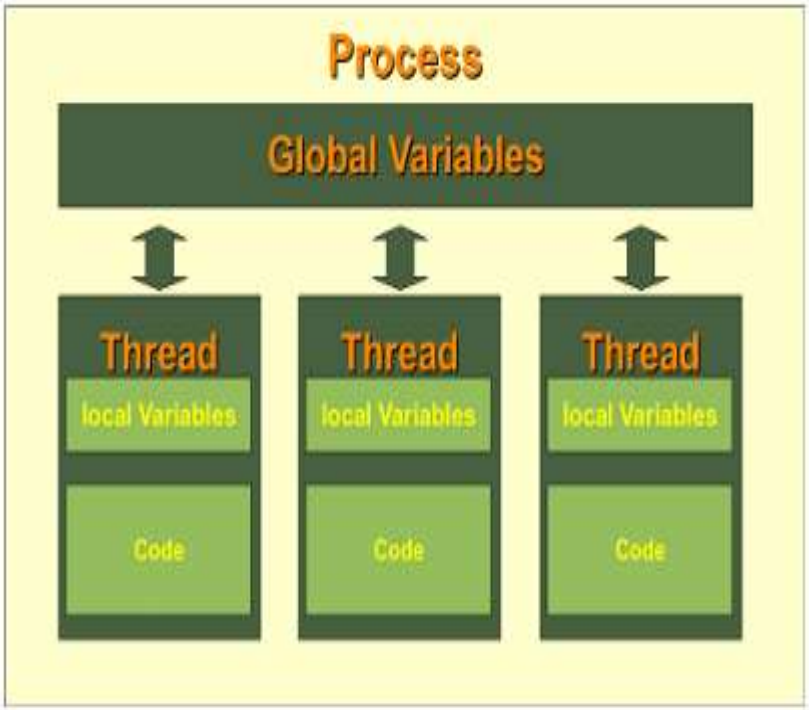
일반적으로 한 프로그램은 하나의 스레드를 가지고 있지만, 프로그램 환경에 따라 둘 이상의 스레드를 동시에 실행할 수 있다.

이러한 실행 방식을 멀티스레드(multithread)라고 한다.

Thread 이미지

멀티 스레드 프로그램은 여러 개의 CPU가 컴퓨터 시스템에 빠르게 실행할 수 있음

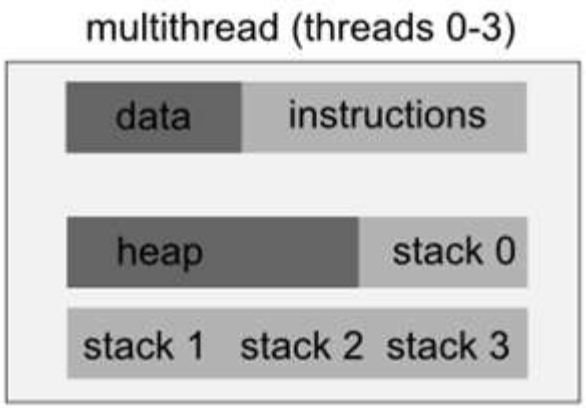


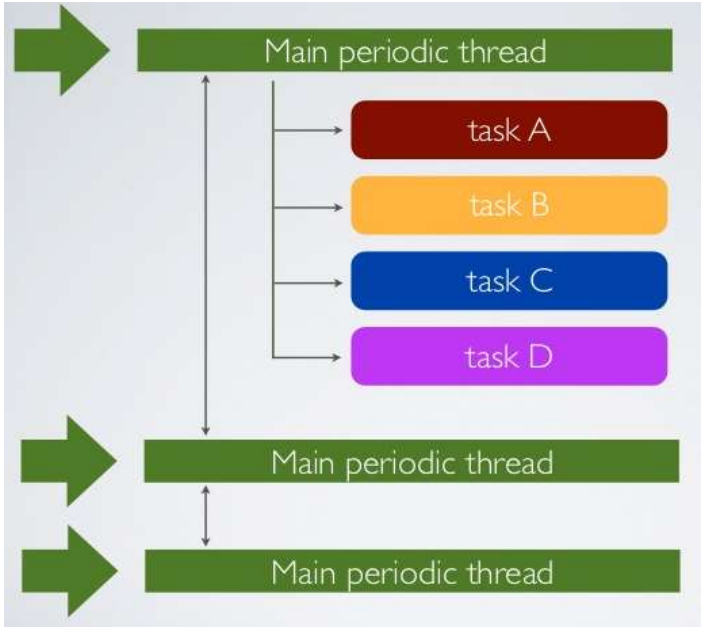


Multithreading

Multithreading:

using multiple threads in the same process for concurrency or parallelism





스레드 종류

Thread의 종류 : 사용자

2개의 Thread 종류를 가지면 python 에서 이를 처리하는 모듈을 제공

사용자 레벨 스레드 (User-Level Thread)

커널 영역의 상위에서 지원되며 일반적으로 사용자 레벨의 라이브러리를 통해 구현되며, 라이브러리는 스레드의 생성 및 스케줄링 등에 관한 관리 기능을 제공한다.

동일한 메모리 영역에서 스레드가 생성 및 관리되므로 속도가 빠른 장점이 있는 반면, 여러 개의 사용자 스레드 중 하나의 스레드가 시스템 호출 등으로 중단되면 나머지 모든 스레드 역시 중단되는 단점이 있다.

이는 커널이 프로세스 내부의 스레드를 인식하지 못하며 해당 프로세스를 대기 상태로 전환시키기 때문이다.

Thread의 종류 : 커널

2개의 Thread 종류를 가지면 python 에서 이를 처리하는 모듈을 제공

커널 레벨 스레드 (Kernel-Level Thread)

커널 스레드는 운영체제가 지원하는 스레드 기능으로 구현되며, 커널이 스레드의 생성 및 스케줄링 등을 관리한다.

스레드가 시스템 호출 등으로 중단되더라도, 커널은 프로세스 내의 다른 스레드를 중단시키지 않고 계속 실행시켜준다.

다중처리기 환경에서 커널은 여러 개의 스레드를 각각 다른 처리기에 할당할 수 있다. 다만, 사용자 스레드에 비해 생성 및 관리하는 것이 느리다.

스레드 데이터

Thread 데이터

Thread 데이터도 기본과 특정 데이터로 구분

스레드 기본 데이터

스레드도 프로세스와 마찬가지로 하나의 실행 흐름이므로 실행과 관련된 데이터가 필요하다. 일반적으로 스레드는 자신만의 고유한 스레드 ID, 프로그램 카운터, 레지스터 집합, 스택을 가진다. 코드, 데이터, 파일 등 기타 자원은 프로세스 내의 다른 스레드와 공유한다.

스레드 특정 데이터

위의 기본 데이터 외에도 하나의 스레드에만 연관된 데이터가 필요한 경우가 있는데, 이런 데이터를 스레드 특정 데이터(Thread-Specific Data, 줄여서 TSD)라고 한다. 멀티스레드 프로그래밍 환경에서 모든 스레드는 프로세스의 데이터를 공유하고 있지만, 특별한 경우에는 개별 스레드만의 자료 공간이 필요하다. 예를 들어 여러 개의 트랜잭션을 스레드로 처리할 경우, 각각의 트랜잭션 ID를 기억하고 있어야 하는데, 이때 TSD가 필요하다. TSD는 여러 스레드 라이브러리들이 지원하는 기능 중의 하나이다.

스레드 처리 이슈

프로세스 문제

스레드 처리 상의 문제

fork 문제 :

어떤 프로세스 내의 스레드가 fork를 호출하면 모든 스레드를 가진 프로세스를 생성할 것인지, 아니면 fork를 요청한 스레드만 가진 프로세스를 생성할 것인지 하는 문제이다.

exec 문제 :

fork를 통해 모든 스레드를 복제하고 난 후, exec를 수행한다면 모든 스레드들이 초기화된다. 그렇다면 교체될 스레드를 복제하는 작업은 필요가 없기 때문에 애초에 fork를 요청한 스레드만을 복제했어야 한다.

한편, fork를 한 후에 exec를 수행하지 않는다면 모든 스레드를 복제할 필요가 있는 경우도 있다

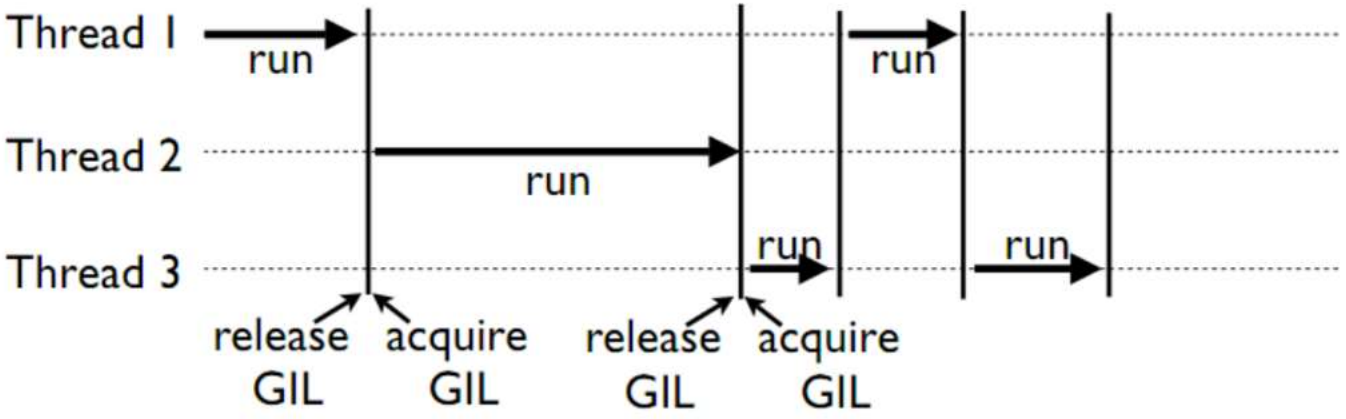
GLOBAL INTERPRETER LOCK 이해하기

Moon Yong Joon

Global Interpreter Lock

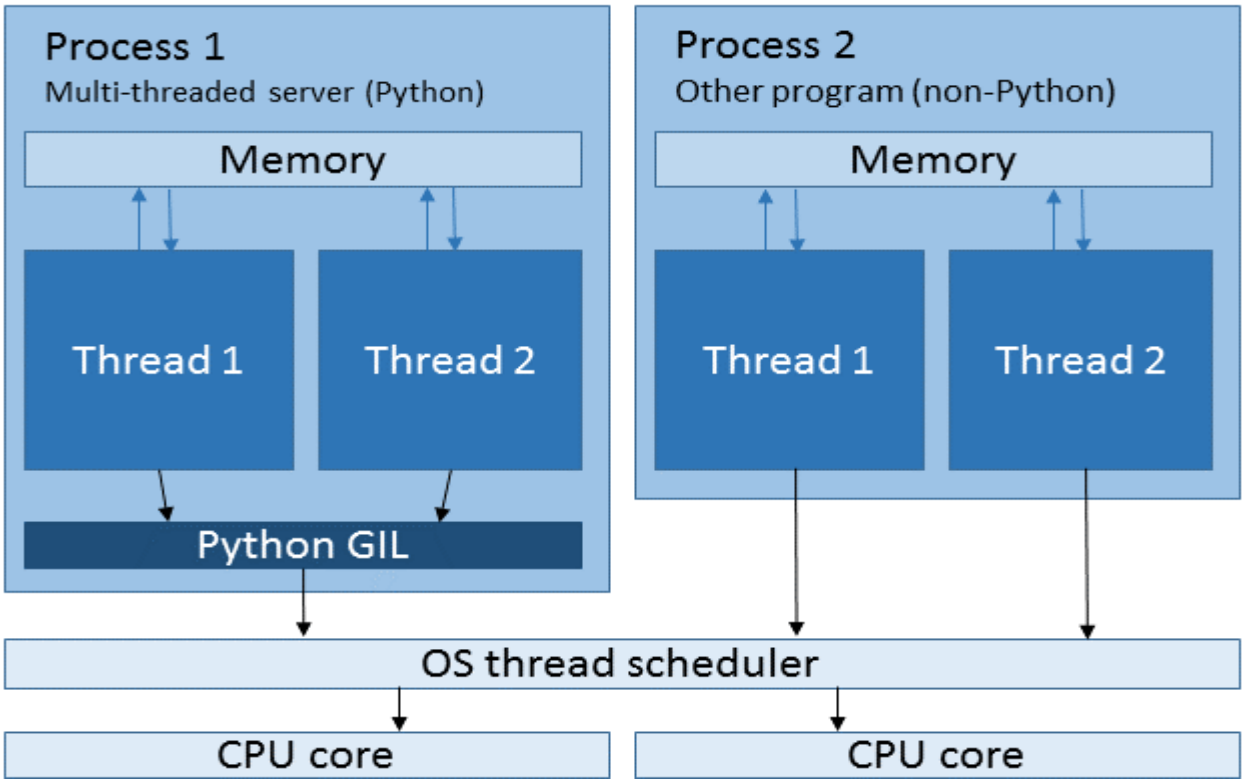
GIL : Global Interpreter Lock

하나의 파이썬 인터프리터에서는 하나의 작업만 실행이 가능한 것을 말하며, 하나의 인터프리터가 실행이 될때, global 변수로 제어 시 한번에 하나의 스레드만이 인터프리터 내부 global 변수에 접근가능하도록 해놓은 것(Cpython 엔진 기본처리)



Python vs. Others

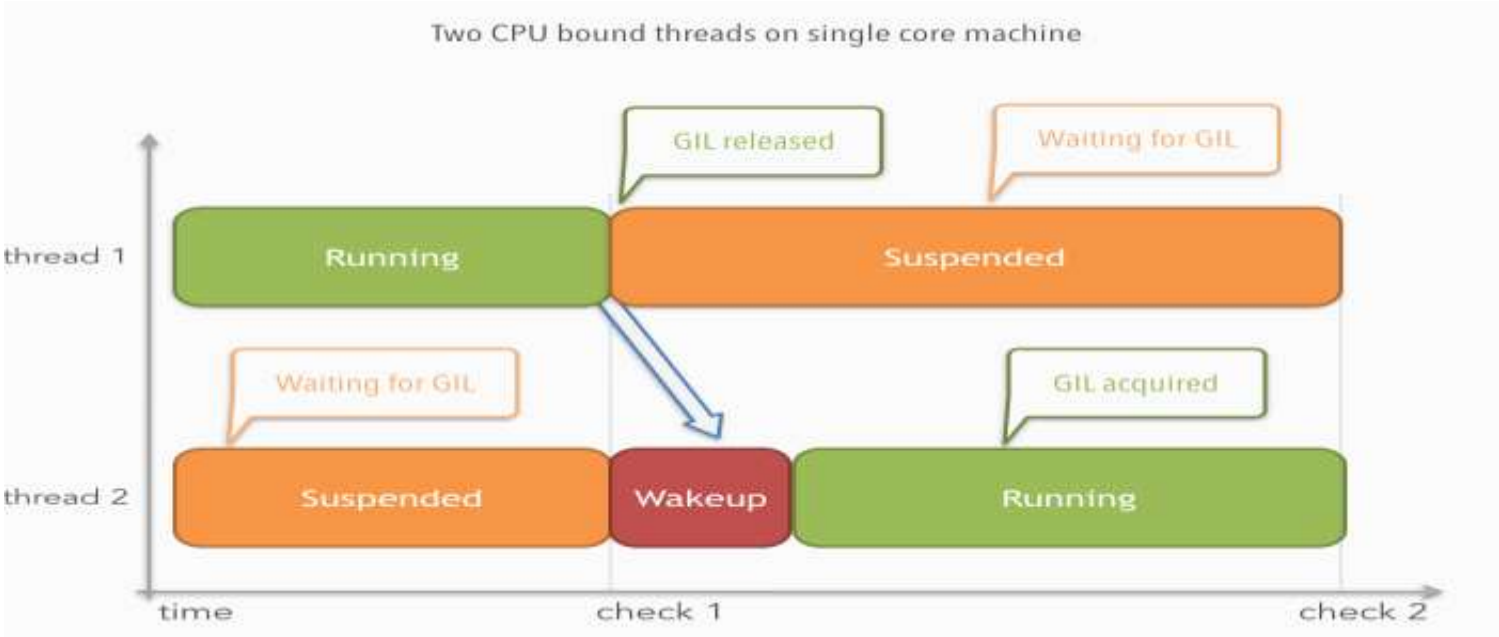
Python process 와 Others process 비교



Lock 스위칭

GIL : Global Interpreter Lock

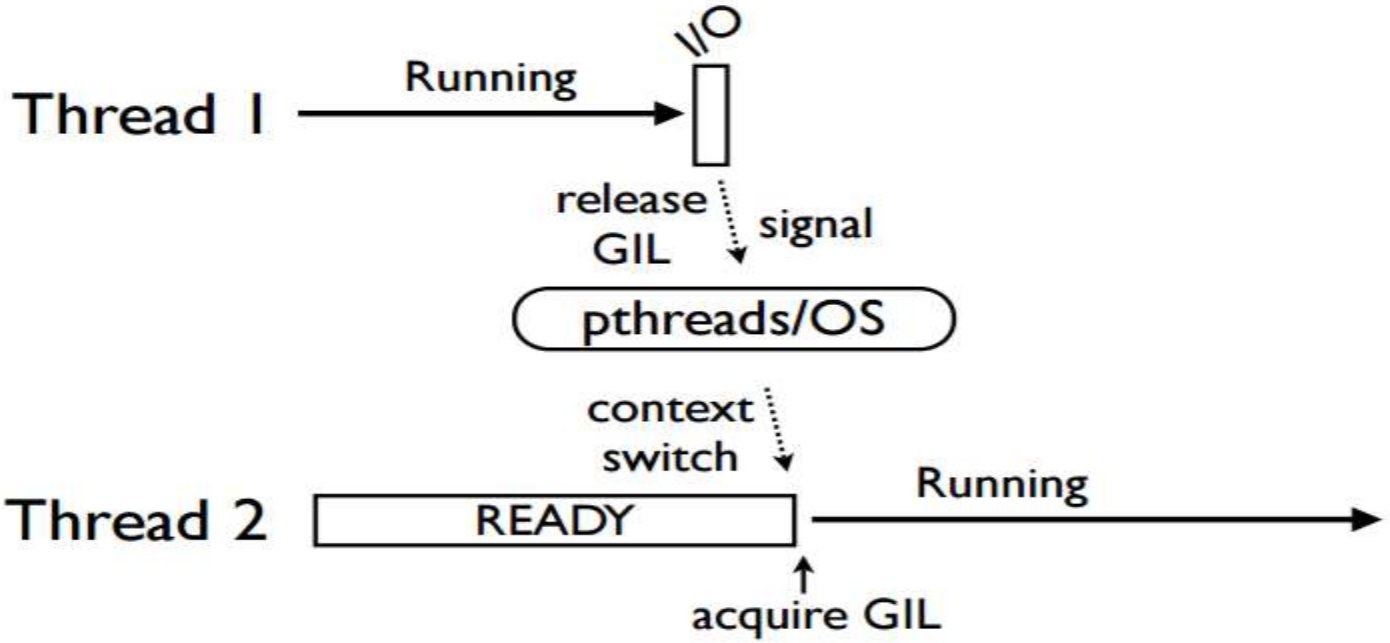
GIL 기준으로 처리하므로 GIL release가 없으면 다른 스레드는 실행이 되지 않음



참조문서 : http://www.dabeaz.com/python/UnderstandingGIL.pdf

Lock switching

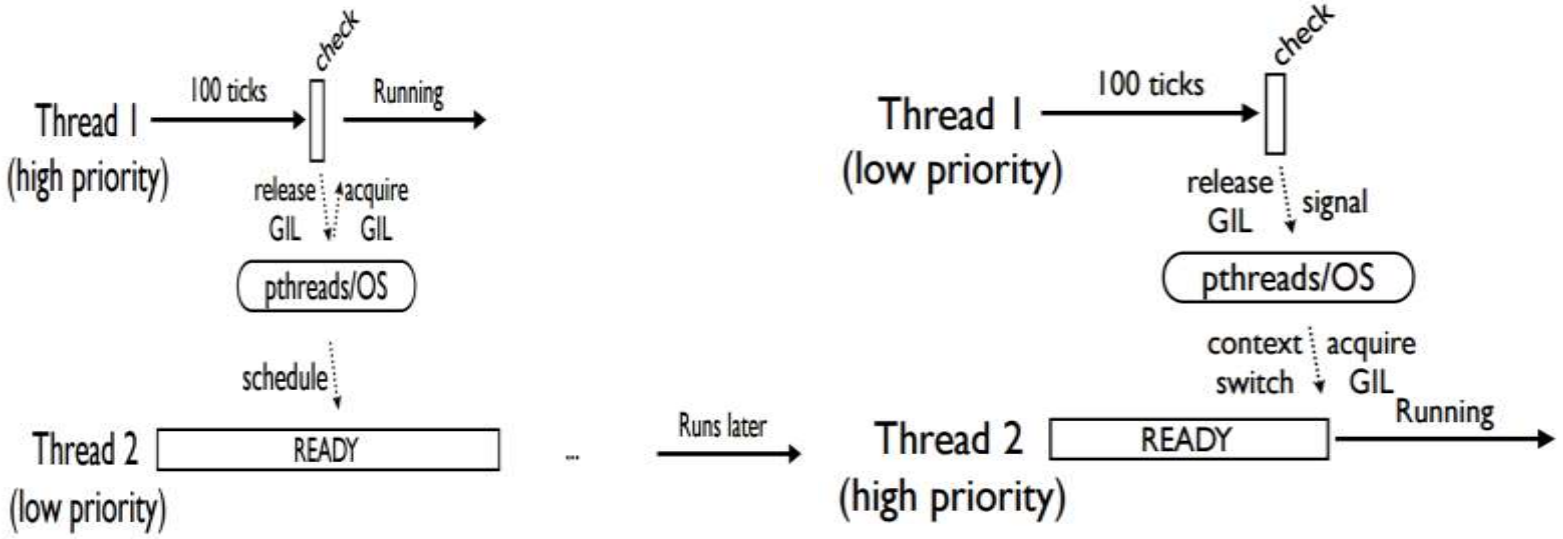
GIL은 execute만 lock이 걸린다. I/O작업을 진행하면 lock이 해제



참조문서 : http://www.dabeaz.com/python/UnderstandingGIL.pdf

High priority 처리

Thread는 기본 100tick 에서 check를 하지만 high priority 일 경우의 Thread를 처리 함

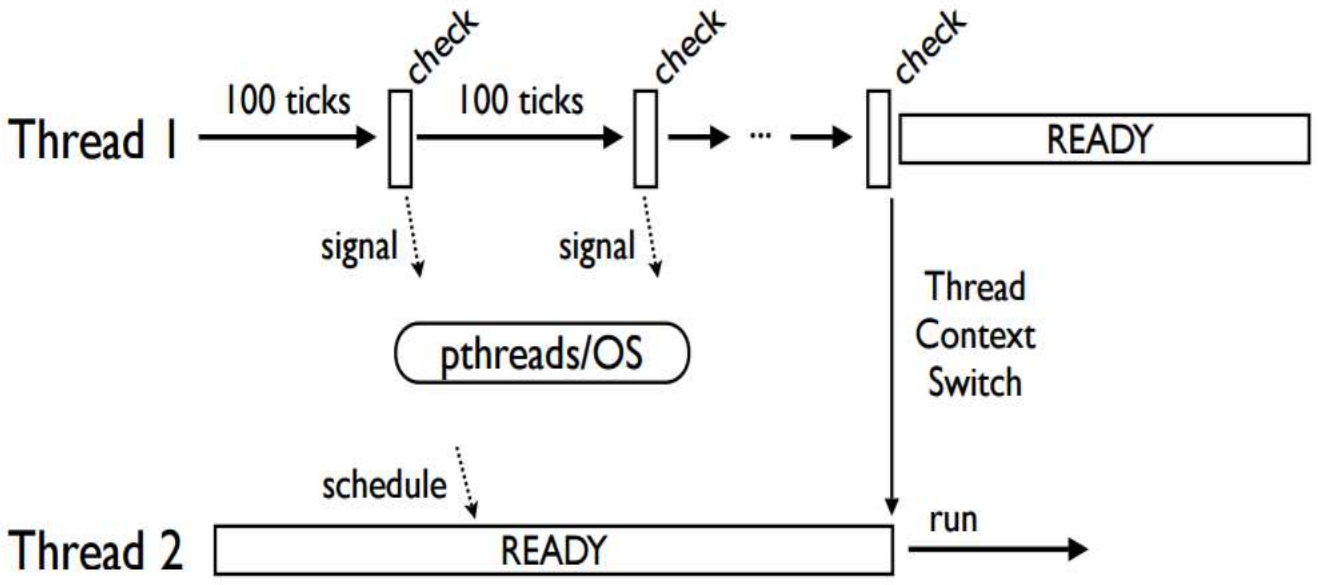


참조문서 : http://www.dabeaz.com/python/UnderstandingGIL.pdf

GIL 성능 비교

GIL 내의 thread 처리 흐름

GIL 은 기본 single thread 처리이므로 함수(thread)를 처리하고 다음 함수(thread)를 처리



singlethread 처리 : 예시

GIL 은 기본 single thread 처리이므로 함수를 하나 처리하면 실행 시간이 나옴

```
import time

def countdown(n):
    start = time.time()
    print("countdown thread ", threading.current_thread())
    while n > 0:
        n -= 1
    end = time.time()
    print(" exec ", float(end) - float(start))
    return n

COUNT = 100000000 # 100 million
countdown(COUNT)
```

```
countdown thread  <_MainThread(MainThread, started 1224)>
 exec  6.154295921325684
```

Mutlithread 처리 : 예시

Multi threading 처리를 했지만 single thread보다 거의 2배 걸리는 이유는 GIL 으로 성능을 내지 못함

```
import threading
import time

def countdown(n):
    start = time.time()
    print("countdown thread ", threading.current_thread())
    while n > 0:
        n -= 1
    end = time.time()
    print(" exec ", float(end) - float(start))
    return n

COUNT = 100000000 # 100 million

t1 = threading.Thread(target=countdown,args=(COUNT//2,))
t2 = threading.Thread(target=countdown,args=(COUNT//2,))
t1.start()
t2.start()
t1.join()
t2.join()
```

```
countdown thread  <Thread(Thread-31, started 1872)>
countdown thread  <Thread(Thread-32, started 1344)>
 exec  exec  6.226718187332153
 6.204715967178345
```

2. THREADING 모듈

Moon Yong Joon

THREADING THREAD CLASS 이해하기

Moon Yong Joon

Thread class 속성

Thread class

threading.Thread에는 다음과 같은 속성이 존재

```
: from threading import Thread
for i in dir(Thread) :
    if i.startswith("_") :
        pass
    else :
        print(i)
```

```
daemon
getName
ident
isAlive
isDaemon
is_alive
join
name
run
setDaemon
setName
start
```

Thread class 변수

threading.Thread는 name, ident(thread id), daemon 변수를 가지고 있음

```
from threading import Thread
def myfunc(data):
    print(data)

thread1 = Thread(target=myfunc, args=("Hello,",))
thread1.start()
print(thread1.name)
print(thread1.daemon)
print(thread1.ident)
thread1.join()
```

```
Hello,Thread-79
False
7568
```

Daemon thread

Daemon Thread

데몬은 주 프로그램이 실행 중일 때만 유용하며, 데몬이 아닌 다른 스레드가 종료되면 종료 할 수 있습니다.

데몬 스레드가 없으면 프로그램을 완전히 종료하기 전에 스레드를 추적하고 종료하도록 지시해야 합니다.

데몬 스레드를 사용하면 스레드를 쉽게 중단 할 수 없거나 스레드가 데이터를 손실 또는 손상시키지 않고 작업을 중단시킬 수 있는 서비스에 유용합니다.

Daemon Thread 생성

threading.Thread는 daemon 변수에 True로 세팅하고 daemon 처리

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-9s) %(message)s',)

def n():
    logging.debug('Starting')
    logging.debug('Exiting')

def d():
    logging.debug('Starting')
    time.sleep(5)
    logging.debug('Exiting')
```

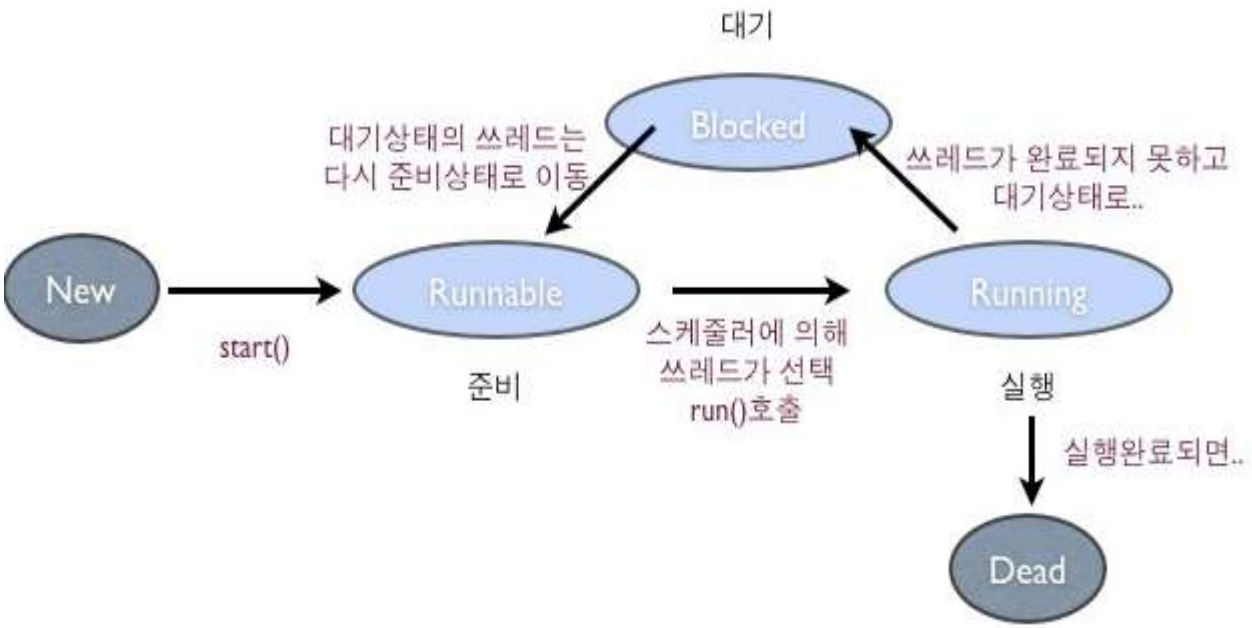
```
if __name__ == '__main__':

    t = threading.Thread(name='non-daemon', target=n)

    d = threading.Thread(name='daemon', target=d)
    d.daemon = True

    d.start()
    t.start()
    d.join()
    t.join()
```

```
(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
(daemon    ) Exiting
```

Thread class의 메소드

Thread 처리 흐름

스레드(thread)도 처리 흐름을 가지면 흐름에 맞도록 파이썬 Thread 내부 class의 메소드가 정의되어 있음.



Thread class의 메소드

threading.Thread의 객체에서 사용할 수 있는 메소드

- thread.start() : thread를 start 메소드로 실행시켜야 함
- thread.join() : thread가 종료될 때까지 기다렸다가 멈춤
- Thread.run() : subclass로 정의시 실제 실행할 메소드 정의
- getName(), setName() : thread name 조회 및 세팅
- is_alive() : thread가 현재 활동하는지 상태 조회
- isDaemon(), setDaemon() : daemon 상태 확인 및 변경

Thread class의 메소드

threading.Thread의 객체가 생기면 메소드로 실행 및 정보 조회/갱신이 가능

```
import threading

def worker():
    """thread worker function"""
    print('Worker')
    return

threads = []
for i in range(5):
    t = threading.Thread(target=worker)
    threads.append(t)
    t.start()
print("threading count ",threading.active_count())
print(threads)
print(threads[0].__dict__)

for th in threads:
    th.join()
```

```
Worker
Worker
Worker
Worker
Worker
threading count  5
[<Thread(Thread-52, stopped 4504)>, <Thread(Thread-53, stopped 15064)>,
topped 14932)>, <Thread(Thread-56, stopped 14356)>]
{'_stderr': <ipykernel.iostream.OutStream object at 0x000000004D90E10>,
k': None, '_name': 'Thread-52', '_started': <threading.Event object at (
```

Thread instance의 변수

Thread 인스턴스 변수

threading.Thread로 처리시 세부 항목을 직접 인스턴스 변수에 세팅해서 처리되어야 함. 특히 subclass로 정의시 함수나 파라미터는 인스턴스 변수에 세팅이 되도록 처리하고 이를 run 메소드 내부에서 실행처리 필요

```
_stderr
_name
_target
_daemonic
_initialized
_ident
_is_stopped
_kwargs
_tstate_lock
_started
_args
```

Thread 인스턴스 변수

threading.Thread로 생성된 인스턴스 객체 내부에 있는 인스턴스 변수를 확인

```
import threading

destination_name = 'name'
destination_config = "config"

class Destination:
    def run(self, name, config):
        print('In thread', name, config)

destination = Destination()
thread = threading.Thread(target=destination.run,
    args=(destination_name, destination_config))
print(thread.__dict__)

thread.start()
```

{'_stderr': <ipykernel.iostream.OutStream object at 0x0000000004D90A90>,

'_name': 'Thread-13',

'_target': <bound method Destination.run of <__main__.Destination object at 0x00000000052E3F60>>,

'_daemonic': False,

'_initialized': True,

'_ident': None,

'_is_stopped': False,

'_kwargs': {},

'_tstate_lock': None,

'_started': <threading.Event object at 0x00000000052E3EF0>,

'_args': ('name', 'config')}

Subclass 내부 변수를 이용 처리

subclass를 만들고 target에 함수, args, kwars에 함수의 인자를 전달하면 run 메소드 내에서 전달된 것을 _target인 내부 속성을 이용해서 처리

```
import sys
import threading

class DestinationThread(threading.Thread):

    def run(self):
        if sys.version_info[0] == 2:
            print("version 2")
            self._Thread__target(*self._Thread__args, **self._Thread__kwargs)
        else: # assuming v3
            print("version 3")
            self._target(*self._args, **self._kwargs)

def func(a, k):
    print("func(): a=%s, k=%s" % (a, k))

thread = DestinationThread(target=func, args=(1,), kwargs={"k": 2})
thread.start()
thread.join()
```

```
version 3
func(): a=1, k=2
```

THREADING TASK 사용하기

Moon Yong Joon

Task 구성하기

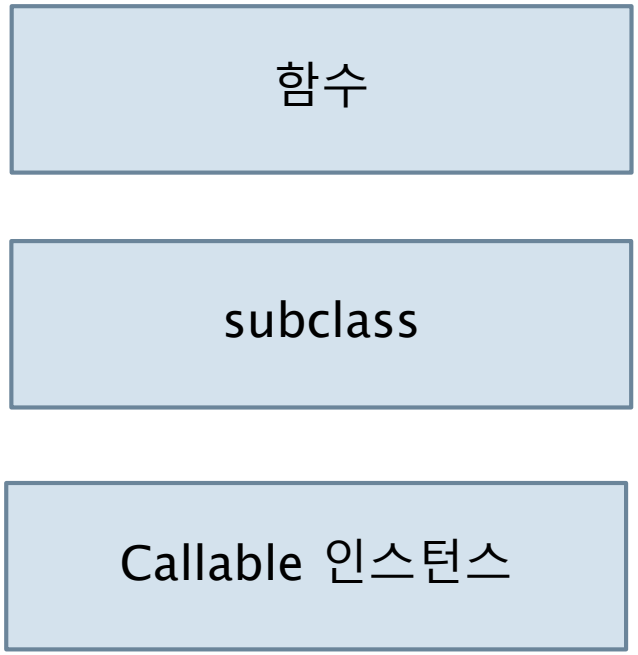
Task

태스크(tasks)란 비동기적으로 수행되는 계산을 말한다.

태스크의 상태는 생성, (실행자에게) 제출, 시작, 완료로 변해간다. 또한 취소하거나 끼어드는 것도 가능하다.

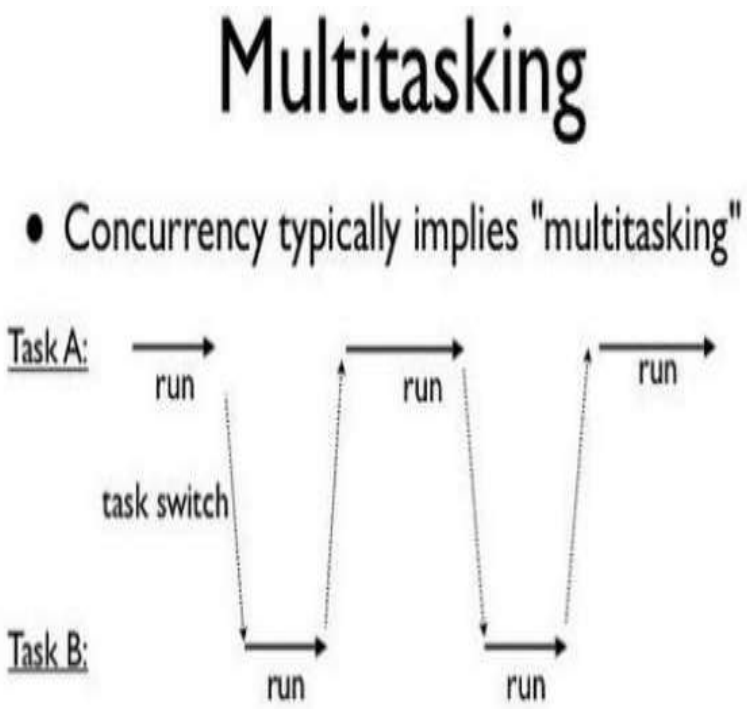
Task 종류

파이썬의 메소드 또는 함수로 대응되기 때문이다.



Task 분리 및 할당

파일을 읽고 라인별로 작업을 thread에 배정




```python
import threading
d = {}
fd = open('fil.txt')

def f(data):
    print(" f exec ", threading.current_thread())
    d[threading.current_thread()] = data

t = []
for l in fd:
    t.append(threading.Thread(target=f, args=(l,)))

for t1 in t:
    t1.start()

for t2 in t:
    t2.join()
print(d)
```

Task 분리 및 할당

파일을 읽고 두 라인에 대해 dict 타입에 처리된 결과를 저장

```
%%writefile fil.txt
aaa,123
bbb,456
```

```
Overwriting fil.txt
```

```
f exec <Thread(Thread-62, started 11016)>
 f exec <Thread(Thread-63, started 10200)>

{<Thread(Thread-63, stopped 10200)>: 'bbb,456',
<Thread(Thread-62, stopped 11016)>: 'aaa,123\n'}
```

Thread 처리 성능

Thread 처리 성능 점검

timeit 모듈을 이용해서 thread 처리 처리 시간을 점검

```
from threading import Thread
import timeit
def myfunc(data):
    print(data)

thread1 = Thread(target=myfunc, args=("Hello,",))

setup = """
from threading import Thread

thread1 = Thread(target=myfunc, args=("Hello,",))
"""

print(timeit.repeat(setup, "from __main__ import myfunc",number=100000))
thread1.start()
print(thread1.name)
print(thread1.daemon)
print(thread1.ident)
thread1.join()
```

Thread 처리 성능 점검 확인

timeit 모듈을 이용해서 thread 처리 처리 시간이 3개가 default로 처리

```
[1.3309866744082228, 1.0158320668998613, 0.9735126647005927]
Hello,
Thread-600092
False
8772
```

Thread 인자 전달

함수에 인자 전달

Thread에 처리되는 함수에 대한 인자를 전달하려면 함수 파라미터와 args에 인자를 tuple로 처리

```
from threading import Thread

def myfunc(data):
    print(data)

thread1 = Thread(target=myfunc, args=("Hello,",))
thread2 = Thread(target=myfunc, args=(" World!",))

thread1.start()
thread2.start()
thread1.join()
thread2.join()
```

```
Hello,
 World!
```

Subclass.run에 인자 전달

Thread에 대한 subclass를 정의하고 thread 생성하고 target에 run메소드를 주고 run파라미터에 args에 인자를 전달

```
import threading

destination_name = 'name'
destination_config = "config"

class Destination:
    def run(self, name, config):
        print('In thread', name, config)

destination = Destination()
thread = threading.Thread(target=destination.run,
    args=(destination_name, destination_config))

thread.start()
```

```
In thread name config
```

THREADING
생성 함수/클래스 정의하기

Moon Yong Joon

함수로 전달하기

함수로 전달

threading.Thread의 객체가 생기면 메소드로 실행 및 정보 조회/갱신이 가능

```
import threading

def worker():
    """thread worker function"""
    print('Worker')
    return

threads = []
for i in range(5):
    t = threading.Thread(target=worker)
    threads.append(t)
    t.start()
print("threading count ",threading.active_count())
print(threads)
print(threads[0].__dict__)

for th in threads:
    th.join()
```

함수로 전달 처리 결과

threading.Thread의 객체가 생기면 메소드로 실행 및 정보 조회/갱신이 가능

```
Worker
Worker
Worker
Worker
Worker
threading count  5
[<Thread(Thread-52, stopped 4504)>, <Thread(Thread-53, stopped 15064)>,
topped 14932)>, <Thread(Thread-56, stopped 14356)>]
{'_stderr': <ipykernel.iostream.OutStream object at 0x0000000004D90E10>,
k': None, '_name': 'Thread-52', '_started': <threading.Event object at 0
```

Subclass : override

사용자 정의 thread 생성

사용자 정의 클래스에서 threading.Thread 클래스를 상속받고 처리 run 메소드를 오버라이딩

thread.run : 메소드를 사용자 정의 클래스에 오버라이딩을 처리

start 메소드를 실행하면 run 메소드가 작동됨

Subclass 정의: run override

Thread에 대한 subclass를 정의하고 run method를 오버라이딩

```
import threading


class MyThread(threading.Thread):

    def run(self):
        print('running', self.name)
        return

for i in range(5):
    t = MyThread()
    t.start()
```

```
running Thread-36
running Thread-37
running Thread-38
running Thread-39
running Thread-40
```

Subclass : super class 이용

Subclass 정의: super class

Thread에 대한 subclass를 정의하지만 Thread class의 __init__에 세팅하고 즉시 실행해서 thread 생성

```
import threading

count = 0
class Thread(threading.Thread):
    def __init__(self, t, *args):
        threading.Thread.__init__(self, target=t, args=args)
        self.start()

def hello_there():
    global count
    while count < 5:
        print("hello_there ", threading.current_thread())
        count += 1

def main():
    hello = Thread(hello_there)

if __name__ == '__main__':
    main()
```

```
hello_there  <Thread(Thread-22, started 1528)>
hello_there  <Thread(Thread-22, started 1528)>
hello_there  <Thread(Thread-22, started 1528)>
hello_there  <Thread(Thread-22, started 1528)>
hello_there  <Thread(Thread-22, started 1528)>
```

Subclass 정의: super class

Thread 정의 후 __init__에서 start 메소드 처리하면 run메소드(오버로딩)가 처리

```
import random
import time
from threading import Thread

class MyThread(Thread):
    def __init__(self, name):
        """Initialize the thread"""
        Thread.__init__(self)
        self.name = name
        self.start()

    def run(self):
        """Run the thread"""
        amount = random.randint(3, 15)
        time.sleep(amount)
        msg = "%s has finished!" % self.name
        print(msg)

# 함수
def create_threads():
    for i in range(5):
        name = "Thread #%s" % (i+1)
        my_thread = MyThread(name=name)


if __name__ == "__main__":
    create_threads()
```

```
Thread #3 has finished!
Thread #1 has finished!
Thread #5 has finished!
Thread #4 has finished!
Thread #2 has finished!
```

Callable로 전달하기

callable 객체 처리

Callable class로 정의하고 __call__에서 실행하도록 처리

```
import threading
import time

def worker(*args):
    print(threading.currentThread().getName(), 'Starting')
    time.sleep(2)
    print(threading.currentThread().getName(), 'Exiting')

class Add:
    def __init__(self, func, *args):
        self.func = func
        self.args = args

    def __call__(self):
        return self.func(*self.args)



t = threading.Thread(name='my_service', target=Add(worker,None))
t.start()
```

```
my_service Starting
my_service Exiting
```

THREADING 모듈 함수 이해하기

Moon Yong Joon

주요 함수

주요 함수

threading 모듈의 주요함수

```
import threading as th


# 호출자의 스레드 컨트롤 스레드 객체의 수
t = th.currentThread()
print(t)
print(th.current_thread())

# 활성 스레드 객체의 수
print(th.activeCount())
print(th.active_count())

#  현재 활성화 된 모든 스레드 객체의리스트
print(th.enumerate())
print(th.main_thread())
```

```
<_MainThread(MainThread, started 19256)>
<_MainThread(MainThread, started 19256)>
5
5
[<_MainThread(MainThread, started 19256)>, <Thread(Thread-4, starte
>, <ParentPollerWindows(Thread-3, started daemon 17088)>, <HistoryS
<_MainThread(MainThread, started 19256)>
```

main_thread()

현재 작동 중인 환경에서 main thread 조회

```
import threading

print("threading count ",threading.active_count())

print("Main thread ", threading.main_thread())
```

```
threading count  5
Main thread  <_MainThread(MainThread, started 15316)>
```

active_count()

현재 작동 중인 thread 개수를 확인하기

```
import threading
import time

def worker():
    print(threading.currentThread().getName(), 'Starting')
    time.sleep(2)
    print(threading.currentThread().getName(), 'Exiting')

def my_service():
    print(threading.currentThread().getName(), 'Starting')
    time.sleep(3)
    print(threading.currentThread().getName(), 'Exiting')

t = threading.Thread(name='my_service', target=my_service)
w = threading.Thread(name='worker', target=worker)
w2 = threading.Thread(target=worker) # use default name

w.start()
w2.start()
t.start()

print("threading count ",threading.active_count())
w.join()
w2.join()
t.join()
```

```
worker Starting
Thread-7 Starting
my_service Starting
threading count  8
Thread-7worker Exiting
 Exiting
my_service Exiting
```

currentThread()

현재 작동 중인 thread를 조회

```
import threading

print("threading count ",threading.active_count())

print("Main thread ", threading.main_thread())

# 현행 하나의 thread만 가져옴
print(threading.currentThread())
```

```
threading count  5
Main thread  <_MainThread(MainThread, started 15316)>
<_MainThread(MainThread, started 15316)>
```

enumerate()

현재 작동 중인 thread 전체를 조회해서 list로 결과를 조회

```
import threading

print("threading count ",threading.active_count())

print("Main thread ", threading.main_thread())



# 프로세스 내의 모든 thread만 가져옴
a = threading.enumerate()
for i in a :
    print(i)
```

```
threading count  5
Main thread  <_MainThread(MainThread, started 15316)>
<ParentPollerWindows(Thread-3, started daemon 6720)>
<HistorySavingThread(IPythonHistorySavingThread, started 9256)>
<Heartbeat(Thread-5, started daemon 17972)>
<Thread(Thread-4, started daemon 16968)>
<_MainThread(MainThread, started 15316)>
```

THREAD 관리모듈과 실행 모듈 분리 후 처리

Moon Yong Joon

모듈분리

Thread 관리 모듈

thread 처리용 subclass 생성

__init__ 초기화 메소드를 정의해서 Thread 클래스의 __init__으로 초기화

```
%%writefile work.py
import threading

class Worker(threading.Thread):
    # Our workers constructor, note the super() method which is vital if we want this
    # to function properly
    def __init__(self):
        super(Worker, self).__init__()

    def run(self):
        for i in range(10):
            print(i)
```

```
Writing work.py
```

Thread 실행 모듈

Thread를 실행하기 위한 메인 모듈 작성

```
%%writefile work_thread.py
import threading
from work import Worker

def main():
    # This initializes ''thread1'' as an instance of our Worker Thread
    thread1 = Worker()
    # This is the code needed to run our newly created thread
    thread1.start()

if __name__ == "__main__":
    main()
```

```
Overwriting work_thread.py
```

```
!python work_thread.py
```

```
0
1
2
3
4
5
6
7
8
9
```

Subcall 초기값 배정

Thread 관리 모듈

thread 처리용 subclass 생성

__init__ 초기화 메소드를 정의해서 Thread 클래스의 __init__으로 초기화

```
%%writefile work.py
import threading

class Worker(threading.Thread):
    # Our workers constructor, note the super() method which is vital if we want this
    # to function properly
    def __init__(self, num):
        super(Worker, self).__init__()
        self.num = num

    def run(self):
        print(" worker num %s " % (self.num))
        for i in range(10):
            print(i)
```

```
Overwriting work.py
```

Thread 실행 모듈

Thread를 실행하기 위한 메인 모듈 작성

```
%%writefile work_thread.py
import threading
from work import Worker

def main():
    # This initializes ''thread1'' as an instance of our Worker Thread
    thread1 = Worker(1)
    thread2 = Worker(2)
    thread3 = Worker(3)
    # This is the code needed to run our newly created thread
    thread1.start()
    thread2.start()
    thread3.start()

if __name__ == "__main__":
    main()
```

```
Overwriting work_thread.py
```

```
!python work_thread.py
```

```
 worker num 1
0
1
2
3
4
5
6
7
8
9
 worker num 2
0
1
2
3
4
5
6
7
8
9
 worker num 3
0
1
2
3
4
5
6
7
8
9
```

THREADING 함수로 THREAD를 생성하기

Moon Yong Joon

함수를 Thread로 생성

Thread 함수로 생성

_start_new_threadthread()로 thread를 생성하면 Thread class로 생성과의 차이점은 자동 실행됨

thread._start_new_thread(func, args, kwargs=None)

- func = thread 실행 함수
- args = func에 넘겨줄 인수
- kwargs = 키워드 인수

Thread 함수로 생성 예시

_start_new_threadthread()로 thread는 자동 실행 됨

```
import threading, time

result_values= []

# thread에서 실행될 함수
def counter(id):
    for i in range(5):
        print('id %s --> %s' % (id, i))

        result_values.append((id,i))
        time.sleep(0.1)

# thread 5개 실행
for i in range(5):
    threading._start_new_thread(counter, (i,))

# thread가 다 돌 때까지 대기
time.sleep(1)
print(result_values)
print( 'Exiting')
```

```
id 2 --> 0id 0 --> 0

id 3 --> 0
id 1 --> 0
id 4 --> 0
id 0 --> 1
id 2 --> 1
id 1 --> 1
id 3 --> 1
id 4 --> 1
id 0 --> 2
id 2 --> 2
id 3 --> 2
id 1 --> 2
id 4 --> 2
id 0 --> 3
id 2 --> 3
id 3 --> 3
id 1 --> 3
id 4 --> 3
id 0 --> 4
id 2 --> 4
id 4 --> 4
id 3 --> 4
id 1 --> 4
[(0, 0), (2, 0), (3, 0), (1, 0), (4, 0),
 (2, 3), (3, 3), (1, 3), (4, 3), (0, 4),
Exiting
```

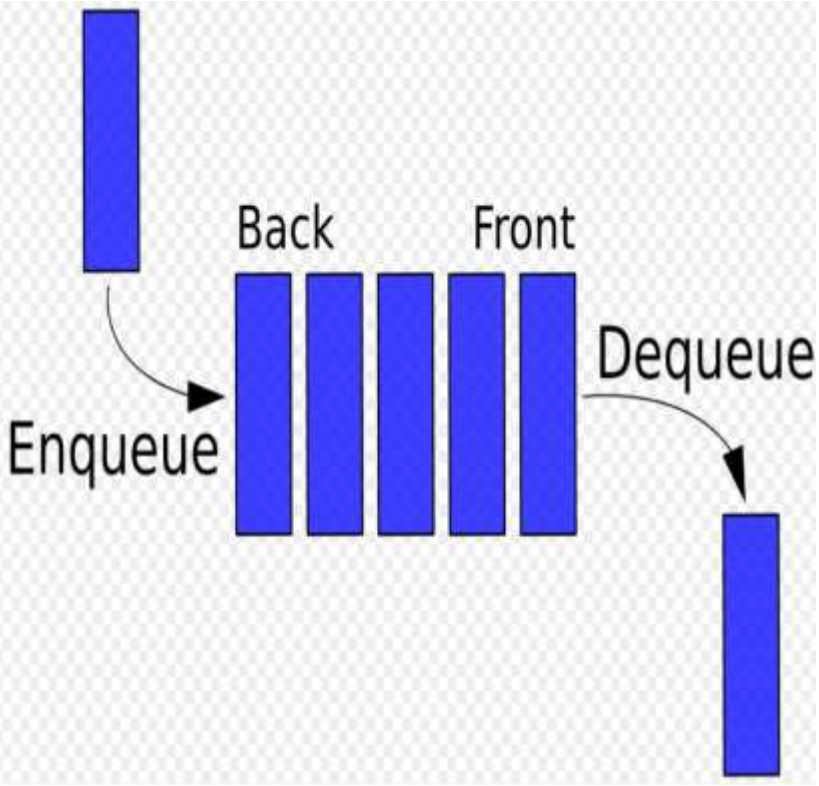
THREADING간 COMMUNICATION

Moon Yong Joon

Queue 모듈 이해하기

Queue : put/get 함수 정의

Queue에 대한 처리 기준을 만듬




```
: from queue import Queue

q = Queue()
q.put("Hello")
print(q.queue)
q.put("world")
print(q.queue)

print("queue size ",q.qsize())
print( q.get() + ' ' + q.get())
print(q.queue)

print(q.empty())
```

```
deque(['Hello'])
deque(['Hello', 'world'])
queue size  2
Hello world
deque([])
True
```

Share memory 메시지 교환

Globale 변수를 통한 공유

특정 값에 대해 공유해서 그 값을 기준으로 thread로 변경(GIL이라 interrupt가 없으면 계속 실행)

```
import threading
#shared memory
x= 0

def f1(num) :
    print(" f1 thread", threading.current_thread())
    global x
    while x < num :
        print(x)
        x += 1
    print(x)

def f2() :
    print(" f2 thread", threading.current_thread())
    global x
    while x > 0 :
        print(x)
        x -= 1
    print(x)

th1 = threading.Thread(target=f1, args=(10,))
th2 = threading.Thread(target=f2)
th1.start()
th2.start()
th1.join()
th2.join()
```

실행결과

특정 값에 대해 공유해서 그 값을 기준으로 thread로 변경

```
 f1 thread <Thread(Thread-8, started 1268)>
0
1
2
3
4
5
6
7
8
9
10
 f2 thread <Thread(Thread-9, started 2572)>
10
9
8
7
6
5
4
3
2
1
0
```

Queue 메시지 교환 1

queue 통한 메시지 전달

특정 값에 대해 공유해서 그 값을 기준으로 thread로 변경 (GIL이라 interrupt가 없으면 계속 실행)

```
import threading
import queue

q = queue.Queue()
def f1(num) :
    print(" f1 thread", threading.current_thread())
    x = 0
    while x < num :
        print(x)
        x += 1
    q.put(x)
    print(x)

def f2() :
    print(" f2 thread", threading.current_thread())
    x = int(q.get())
    while x > 0 :
        print(x)
        x -= 1
    print(x)

th1 = threading.Thread(target=f1, args=(10,))
th2 = threading.Thread(target=f2)
th1.start()
th2.start()
th1.join()
th2.join()
```

실행결과

특정 값에 대해 공유해서 그 값을 기준으로 thread로 변경

```
 f1 thread <Thread(Thread-6, started 10724)>
0
1
2
3
4
5
6
7
8
9
10
 f2 thread <Thread(Thread-7, started 1128)>
10
9
8
7
6
5
4
3
2
1
0
```

Queue 메시지 교환 2

Queue : Put/get 함수 정의

Queue에 대한 처리 기준을 만듬

```
import threading
from queue import Queue
import time

q = Queue()
# The worker thread pulls an item from the queue and processes it
def worker():
    print(" get thread ", threading.current_thread())
    while True:
        item = q.get()
        print(" get queue", item)
        q.task_done()

def queue_put() :
    print(" put thread ", threading.current_thread())
    for item in range(4):
        print(" put queue", item)
        q.put(item)
```

Thread : 메시지 교환 정의

thread에 Queue put/get를 지정해서 처리

```
# Create the queue and thread pool.
threads = []
for i in range(2):
    t = threading.Thread(target=worker)
    threads.append(t)

for i in range(2):
    t = threading.Thread(target=queue_put)
    threads.append(t)

print(threads)
for t in threads :
    t.start()
    if q.full() == True :
        print(t,q.queue)

q.join()       # block until all tasks are done
```

실행 결과

thread에 Queue put하는 부분을 먼저 처리하고 thread get을 처리

```
[<Thread(Thread-63, initial)>, <Thread(Thread-64, initial)>, <Thread(Thread-65, initial)>, <Thread(Thread-66, initial)>]
 get thread  <Thread(Thread-63, started 4072)>
 get thread  <Thread(Thread-64, started 5188)>
 put thread  <Thread(Thread-65, started 14316)>
 put queue 0
 put queue 1
 put queue 2
 put queue 3
 get queue 0
 get queue 1
 get queue 2
 get queue 3
 put thread  <Thread(Thread-66, started 14992)>
 put queue 0
 put queue 1
 put queue 2
 put queue 3
 get queue 0
 get queue 1
 get queue 2
 get queue 3
```

THREADING
동기화 처리

Moon Yong Joon

동기화 분류

동기화 종류

5가지의 동기화 처리 방식이 존재

- Lock : 일반적으로 공유 데이터 구조를 보호하기 위해 사용되는 상호 배타 잠금.

- RLock. : 코드 기반의 잠금 기능이나 방법 또는 모니터를 구현하는 데 유용합니다 재진입 상호 배제 잠금.

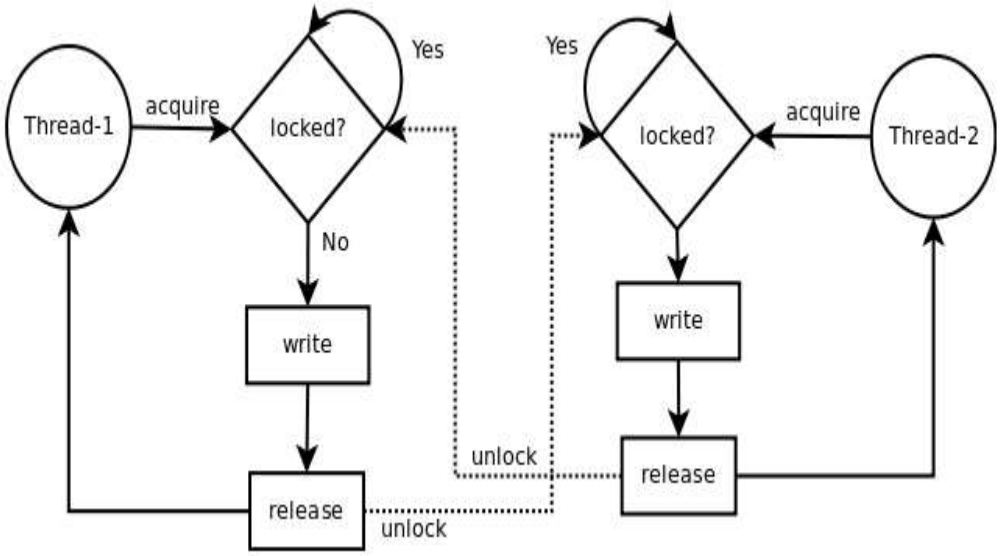
- Event : 목적은 하나 이상의 스레드가 발생하는 약간의 "이벤트"를 기다릴 수있다. 장벽을 구현하기 위해 또는 일부 작업의 완료를 신호하는 데 사용됩니다.

- Condition : 조건 변수입니다. 스레드 사이에 신호를 전송하는 데 사용됩니다. 생산자 - 소비자 문제 예를 들어, 생산자는 데이터를 사용할 수 있는 사용자에게 신호를 보내도록 조건 변수를 사용한다.

- Semaphore. : 정수 카운터에 기초하여 프리미티브를 상위 레벨의 동기화; 세마포어를 획득하면 카운터를 감소시키고 세마포어를 해제하는 카운터를 증가시킨다. 카운터가 0이고 스레드가 획득하려고하면 다른 스레드가 세마포어를 해제 할 때까지 차단합니다.

Lock 처리 방식

공유메모리에 대해 locking하고 스레드가 실행되다 해제시 다른 스레드가 실행되도록 함



THREADING
LOCK 동기화 처리

Moon Yong Joon

interrupt 처리

Interrupt 없이 스레드 실행

Interrupt 없이 스레드 실행하면 단일 스레드처럼 순차처리

```python
import threading
import time

total = 0
def counter():
    global total
    print(" exec thread", threading.current_thread())

    for i in range(0,10):
        print(i)
        total += i

thread1 = threading.Thread(target = counter)
thread2 = threading.Thread(target = counter)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print(total)
```

```
 exec thread <Thread(Thread-14, started 6748)>
0
1
2
3
4
5
6
7
8
9
 exec thread <Thread(Thread-15, started 16472)>
0
1
2
3
4
5
6
7
8
9
90
```

Interrupt 스레드 실행

Interrupt 스레드 실행하면 스레드간의 병행 처리

```
import threading
import time

total = 0
def counter():
    global total

    for i in range(0,10):
        print(threading.current_thread(),i)
        time.sleep(0.1)
        total += i

thread1 = threading.Thread(target = counter)
thread2 = threading.Thread(target = counter)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print(" total ", total)
```

```
<Thread(Thread-28, started 3212)> 0
<Thread(Thread-29, started 5228)> 0
<Thread(Thread-28, started 3212)> 1
<Thread(Thread-29, started 5228)> 1
<Thread(Thread-28, started 3212)> 2
<Thread(Thread-29, started 5228)> 2
<Thread(Thread-28, started 3212)> 3
<Thread(Thread-29, started 5228)> 3
<Thread(Thread-28, started 3212)> 4
<Thread(Thread-29, started 5228)> 4
<Thread(Thread-28, started 3212)> 5
<Thread(Thread-29, started 5228)> 5
<Thread(Thread-28, started 3212)> 6
<Thread(Thread-29, started 5228)> 6
<Thread(Thread-28, started 3212)> 7
<Thread(Thread-29, started 5228)> 7
<Thread(Thread-28, started 3212)> 8
<Thread(Thread-29, started 5228)> 8
<Thread(Thread-28, started 3212)> 9
<Thread(Thread-29, started 5228)> 9
 total  90
```

Lock 이해하기

Lock을 release 하지 않는 경우

Lock을 생성한 후 release를 안하면 다른 스레드가 사용할 수 없음

```
import threading

total = 0
lock = threading.Lock()

def update_total(amount):
    """
    Updates the total by the given amount
    """
    global total
    lock.acquire()
    try:
        total += amount
    finally:
        pass
        #lock.release()
    print (threading.current_thread(),total)

if __name__ == '__main__':
    for i in range(10):
        my_thread = threading.Thread(
            target=update_total, args=(5,))
        my_thread.start()
```

```
<Thread(Thread-42, started 7080)> 5
```

Lock을 release 할 경우

Lock을 생성한 후 release하면 다른 스레드가 사용할 수 있음

```
import threading

total = 0
lock = threading.Lock()

def update_total(amount):
    """
    Updates the total by the given amount
    """
    global total
    lock.acquire()
    try:
        total += amount
    finally:
        lock.release()
    print (threading.current_thread(),total)

if __name__ == '__main__':
    for i in range(10):
        my_thread = threading.Thread(
            target=update_total, args=(5,))
        my_thread.start()
```

```
<Thread(Thread-52, started 16132)> 5
<Thread(Thread-53, started 1056)> 10
<Thread(Thread-54, started 19232)> 15
<Thread(Thread-55, started 18076)> 20
<Thread(Thread-56, started 4372)> 25
<Thread(Thread-57, started 18928)> 30
<Thread(Thread-58, started 18736)> 35
<Thread(Thread-59, started 13072)> 40
<Thread(Thread-60, started 19408)> 45
<Thread(Thread-61, started 9808)> 50
```

Lock 이해하기 : with 문 사용

With 문 사용

with문을 사용하면 aquire/release가 자동으로 처리 됨

```python
import threading

total = 0
lock = threading.Lock()

def update_total(amount):
    """
    Updates the total by the given amount
    """
    global total
    with lock:
        total += amount
    print (threading.current_thread(),total)


if __name__ == '__main__':
    for i in range(10):
        my_thread = threading.Thread(
            target=update_total, args=(5,))
        my_thread.start()
```

```
<Thread(Thread-100, started 18684)> 5
<Thread(Thread-101, started 8504)> 10
<Thread(Thread-102, started 18636)> 15
<Thread(Thread-103, started 13100)> 20
<Thread(Thread-104, started 16696)> 25
<Thread(Thread-105, started 7080)> 30
<Thread(Thread-106, started 860)> 35
<Thread(Thread-107, started 2768)> 40
<Thread(Thread-108, started 4072)> 45
<Thread(Thread-109, started 7684)> 50
```

Lock 처리 : subclass

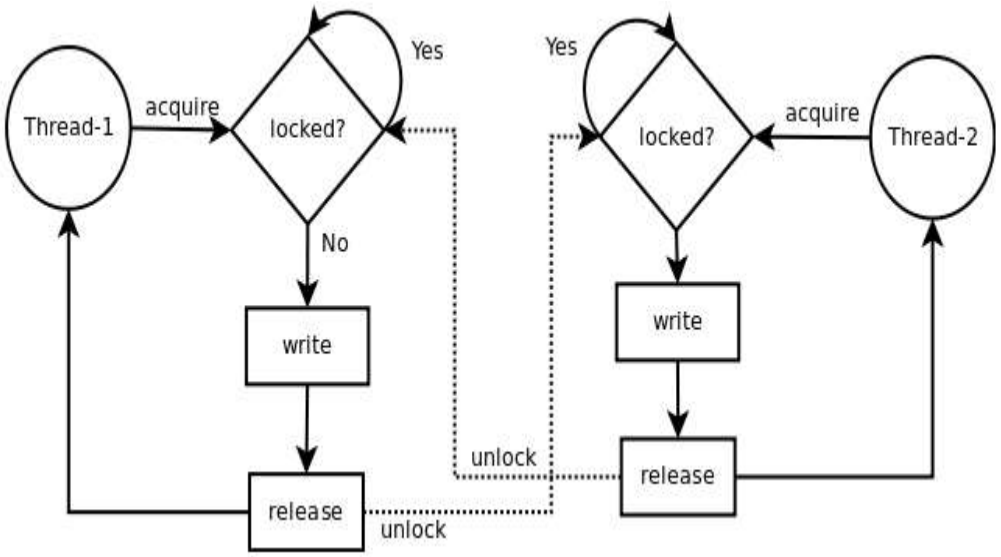
Lock Class 작성

공유 데이터 TOTAL 에 대해 Lock을 생성하고 스레드가 종료시까지 Locking 처리

```
import threading

TOTAL = 0
MY_LOCK = threading.Lock()

class CountThread(threading.Thread):
    def run(self):
        global TOTAL
        for i in range(10):
            MY_LOCK.acquire()
            TOTAL = TOTAL + 1
            print("exex", threading.current_thread()._tstate_lock, TOTAL)
            MY_LOCK.release()
        print('%s\n' % (TOTAL))

a = CountThread()
b = CountThread()
a.start()
b.start()
```

Lock Class 실행

하나의 thread가 locking을 처리하고 끝나면 해체해서 다른 스레드 처리시 locking해서 처리




```
exex <locked _thread.lock object at 0x0000000004E45C60> 1
exex <locked _thread.lock object at 0x0000000004E45C60> 2
exex <locked _thread.lock object at 0x0000000004E45C60> 3
exex <locked _thread.lock object at 0x0000000004E45C60> 4
exex <locked _thread.lock object at 0x0000000004E45C60> 5
exex <locked _thread.lock object at 0x0000000004E45C60> 6
exex <locked _thread.lock object at 0x0000000004E45C60> 7
exex <locked _thread.lock object at 0x0000000004E45C60> 8
exex <locked _thread.lock object at 0x0000000004E45C60> 9
exex <locked _thread.lock object at 0x0000000004E45C60> 10
10

exex <locked _thread.lock object at 0x0000000004E45B20> 11
exex <locked _thread.lock object at 0x0000000004E45B20> 12
exex <locked _thread.lock object at 0x0000000004E45B20> 13
exex <locked _thread.lock object at 0x0000000004E45B20> 14
exex <locked _thread.lock object at 0x0000000004E45B20> 15
exex <locked _thread.lock object at 0x0000000004E45B20> 16
exex <locked _thread.lock object at 0x0000000004E45B20> 17
exex <locked _thread.lock object at 0x0000000004E45B20> 18
exex <locked _thread.lock object at 0x0000000004E45B20> 19
exex <locked _thread.lock object at 0x0000000004E45B20> 20
20
```

Lock 처리 : decorator

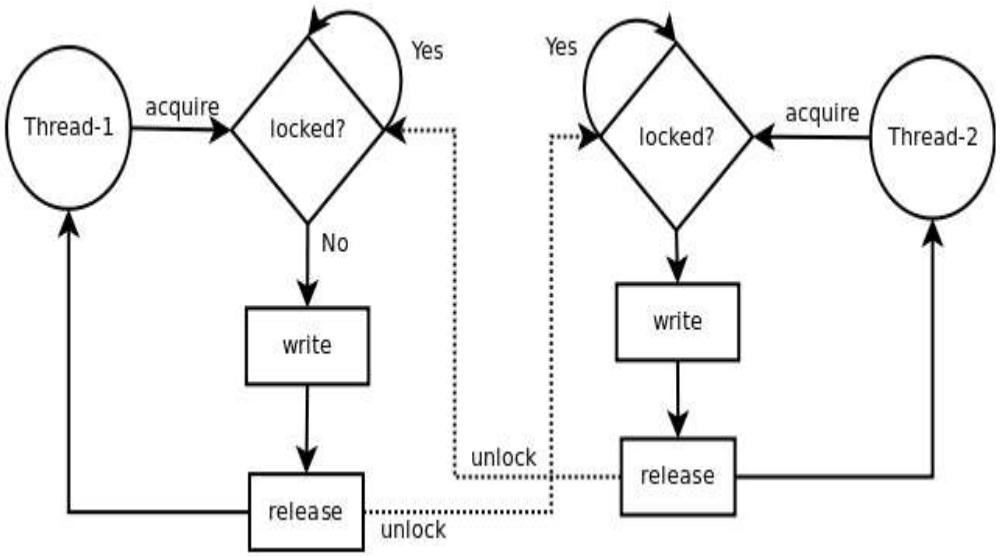
Lock Class 작성

Lock을 생성하고 interrupt가 발생시 Lock이 해제되어 처리됨

```
import threading
import time

total = 0
def synchronized(func):
    func.__lock__ = threading.Lock()
    def synced_func(*args, **kws):
        with func.__lock__:
            return func(*args, **kws)
    return synced_func

@synchronized
def count():
    global total
    curr = total + 1
    time.sleep(0.1)
    print(threading.current_thread(), curr)
    total = curr


def counter():
    print(" exec thread", threading.current_thread())
    for i in range(0,10):
        count()

thread1 = threading.Thread(target = counter)
thread2 = threading.Thread(target = counter)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print(total)
```

Lock Class 실행

두개의 thread가 lock를 얻고 time.sleep() 동안 lock을 해제하고 처리




```
 exec thread <Thread(Thread-20, started 19304)>
 exec thread <Thread(Thread-21, started 15996)>
<Thread(Thread-20, started 19304)> 1
<Thread(Thread-20, started 19304)> 2
<Thread(Thread-21, started 15996)> 3
<Thread(Thread-21, started 15996)> 4
<Thread(Thread-21, started 15996)> 5
<Thread(Thread-21, started 15996)> 6
<Thread(Thread-21, started 15996)> 7
<Thread(Thread-21, started 15996)> 8
<Thread(Thread-21, started 15996)> 9
<Thread(Thread-20, started 19304)> 10
<Thread(Thread-20, started 19304)> 11
<Thread(Thread-20, started 19304)> 12
<Thread(Thread-21, started 15996)> 13
<Thread(Thread-21, started 15996)> 14
<Thread(Thread-21, started 15996)> 15
<Thread(Thread-20, started 19304)> 16
<Thread(Thread-20, started 19304)> 17
<Thread(Thread-20, started 19304)> 18
<Thread(Thread-20, started 19304)> 19
<Thread(Thread-20, started 19304)> 20
20
```

THREADING
RLOCK 동기화 처리

Moon Yong Joon

재진입: reentrant

RLock처리 : 스레드 없이 실행

별도의 스레드가 없어도 RIock을 가지고 실제 재진입되어 처리됨

```
import threading

total = 0
lock = threading.RLock()

def do_something():
    with lock:
        print(lock)
    print('repease',lock)
    return "Done doing something"

def do_something_else():
    with lock:
        print(lock)
    print('repease',lock)
    return "Finished something else"

def main():
    with lock:
        result_one = do_something()
        result_two = do_something_else()
    print (result_one)
    print (result_two)

main()
```

```
<locked _thread.RLock object owner=11188 count=2 at 0x0000000004E55990>
repease <locked _thread.RLock object owner=11188 count=1 at 0x0000000004E55990>
<locked _thread.RLock object owner=11188 count=2 at 0x0000000004E55990>
repease <locked _thread.RLock object owner=11188 count=1 at 0x0000000004E55990>
Done doing something
Finished something else
```

RLock처리 : 스레드 생성 처리

반드시 별도의 스레드 객체를 생성시켜 실행해야 함

```
import threading

total = 0
lock = threading.RLock()

def do_something():
    with lock:
        print(lock)
    print('repease',lock)
    return "Done doing something"

def do_something_else():
    with lock:
        print(lock)
    print('repease',lock)
    return "Finished something else"

def main():
    result_one = do_something()
    result_two = do_something_else()
    print (result_one)
    print (result_two)

for i in range(2):
    my_thread = threading.Thread(target=main)
    my_thread.start()
```

별도의 count를 관리해서 acquire/release시마다 증감이 발생

```
<locked _thread.RLock object owner=11892 count=1 at 0x0000000005056EE0>
repease <unlocked _thread.RLock object owner=0 count=0 at 0x0000000005056EE0>
<locked _thread.RLock object owner=11892 count=1 at 0x0000000005056EE0>
repease <unlocked _thread.RLock object owner=0 count=0 at 0x0000000005056EE0>
Done doing something
Finished something else
<locked _thread.RLock object owner=8868 count=1 at 0x0000000005056EE0>
repease <unlocked _thread.RLock object owner=0 count=0 at 0x0000000005056EE0>
<locked _thread.RLock object owner=8868 count=1 at 0x0000000005056EE0>
repease <unlocked _thread.RLock object owner=0 count=0 at 0x0000000005056EE0>
Done doing something
Finished something else
```

Lock/RLock 비교

Lock처리

반드시 별도의 스레드 객체를 생성시켜야 Lock이 실행됨

```
import threading

total = 0
lock = threading.Lock()

def do_something():
    with lock:
        print(lock)
    print('repease',lock)
    return "Done doing something"

def do_something_else():
    with lock:
        print(lock)
    print('repease',lock)
    return "Finished something else"

def main():
    result_one = do_something()
    result_two = do_something_else()
    print (result_one)
    print (result_two)

for i in range(2):
    my_thread = threading.Thread(target=main)
    my_thread.start()
```

Lock 객체는 사용되는 count가 미존재

```
<locked _thread.lock object at 0x0000000004D7DFA8>
repease <unlocked _thread.lock object at 0x0000000004D7DFA8>
<locked _thread.lock object at 0x0000000004D7DFA8>
repease <unlocked _thread.lock object at 0x0000000004D7DFA8>
Done doing something
Finished something else
<locked _thread.lock object at 0x0000000004D7DFA8>
repease <unlocked _thread.lock object at 0x0000000004D7DFA8>
<locked _thread.lock object at 0x0000000004D7DFA8>
repease <unlocked _thread.lock object at 0x0000000004D7DFA8>
Done doing something
Finished something else
```

THREADING SEMAPHORE 동기화 처리

Moon Yong Joon

Semaphore 이해

semaphore

자원이 소비되면 감소하고 자원이 해제되면 증가되어 자원의 사용 가능여부가 판단할 수 있는 동기화 방식

```
import threading

sem = threading.Semaphore(3)

print(sem)
print(sem._value)

print(sem.acquire())
print(sem._value)
print(sem.acquire())
print(sem._value)
print(sem.acquire())
print(sem._value)


print(sem.release())
print(sem._value)
print(sem.release())
print(sem._value)
print(sem.release())
print(sem._value)
```

```
<threading.Semaphore object at 0x00000000050AE240>
3
True
2
True
1
True
0
None
1
None
2
None
3
```

Semaphore 문제점

자원이 해제되면 증가되는데 초기에 설정한 것보다 더 큰 수가 될 수가 있음

```
import threading

sem = threading.Semaphore(3)

print(sem)
print(sem._value)

print(sem.acquire())
print(sem._value)
print(sem.acquire())
print(sem._value)
print(sem.acquire())
print(sem._value)

print(sem.release())
print(sem._value)
print(sem.release())
print(sem._value)
print(sem.release())
print(sem._value)
print(sem.release())
print(sem._value)
```

```
<threading.Semaphore object at 0x000000004E68860>
3
True
2
True
1
True
0
None
1
None
2
None
3
None
4
```

BoundedSemaphore

자원이 해제되면 증가되는데 초기에 설정한 것보다 더 큰 수가 될 경우 오류처리

```
import threading

sem = threading.BoundedSemaphore(3)

print(sem)
print(sem._value)

print(sem.acquire())
print(sem._value)
print(sem.acquire())
print(sem._value)
print(sem.acquire())
print(sem._value)

print(sem.release())
print(sem._value)
print(sem.release())
print(sem._value)
print(sem.release())
print(sem._value)
print(sem.release())
print(sem._value)
```

```
<threading.BoundedSemaphore object at 0x000000000508E518>
3
True
2
True
1
True
0
None
1
None
2
None
3

---------------------------------------------------------------
ValueError                                Traceback (most
<ipython-input-3-d4004b4c6020> in <module>()
     19 print(sem.release())
     20 print(sem._value)
---> 21 print(sem.release())
     22 print(sem._value)

C:\Program Files\Anaconda3\lib\threading.py in release(sel
    478         with self._cond:
    479             if self._value >= self._initial_value:
--> 480                 raise ValueError("Semaphore releas
    481             self._value += 1
    482             self._cond.notify()

ValueError: Semaphore released too many times
```

Semaphore 이해

Semaphore 제한 확인

자원이 소비되면 감소하는 경우만 확인할 경우
차원이 소비되면 더 생기지 않음

```
# 100개의 쓰레드 중에 3개의 쓰레드만 작업이 가능하다.

import threading

# 세마포 객체 생성, 3개의 쓰레드로 제한
sem = threading.Semaphore(3)

class RestrictedArea(threading.Thread):
    def run(self):
        msg = 'Threading Semaphore TEST : %s' % self.getName()
        # 3개의 쓰레드만이 존재할수 있는 영역
        sem.acquire()
        print(sem._value,msg)
        #sem.release()

threads = []

for i in range(100):
    threads.append(RestrictedArea())

for th in threads:
    th.start()              # 쓰레드 시작

for th in threads:
    th.join()               # 종료대기

print('Finish All Threading ')
```

```
2 Threading Semaphore TEST : Thread-207
1 Threading Semaphore TEST : Thread-208
0 Threading Semaphore TEST : Thread-209
```

Semaphore 처리

자원이 소비되면 감소와 증가가 생기면서 처리

```python
# 100개의 쓰레드 중에 3개의 쓰레드만 작업이 가능하다.

import threading

# 세마포 객체 생성, 3개의 쓰레드로 제한
sem = threading.Semaphore(3)

class RestrictedArea(threading.Thread):
    def run(self):
        msg = 'Threading Semaphore TEST : %s' % self.getName()
        # 3개의 쓰레드만이 존재할수 있는 영역
        sem.acquire()
        print(sem._value,msg)
        sem.release()

threads = []

for i in range(10):
    threads.append(RestrictedArea())

for th in threads:
    th.start()          # 쓰레드 시작

for th in threads:
    th.join()           # 종료대기

print('Finish All Threading ')
```

```
2 Threading Semaphore TEST : Thread-6
2 Threading Semaphore TEST : Thread-7
2 Threading Semaphore TEST : Thread-8
2 Threading Semaphore TEST : Thread-9
2 Threading Semaphore TEST : Thread-10
2 Threading Semaphore TEST : Thread-11
2 Threading Semaphore TEST : Thread-12
2 Threading Semaphore TEST : Thread-13
2 Threading Semaphore TEST : Thread-14
2 Threading Semaphore TEST : Thread-15
Finish All Threading
```

THREADING CONDITION 동기화 처리

Moon Yong Joon

동기화: Condition

Condition Class

조건 변수가 항상 잠금 일종의 연결되어 있기 때문에, 이것은 공유 자원에 연결된다.

잠금이 전달 될 수 있거나 하나가 기본적으로 생성됩니다. Lock은 Condition object의 일부입니다 : 그래서, 상태 객체는 자원 업데이트 될 때까지 스레드를 대기 할 수있다.

wait 메소드 : 통지 할 때까지 또는 시간 초과가 발생할 때까지 기다립니다. 이 메서드를 호출 할 때 호출 스레드가 잠금을 획득하지 않은 경우, 런타임 오류가 발생합니다.

notifyAll 메소드 : 대기 중의 모든 스레드에를 캐우며, 이 메서드를 호출 할 때 호출 스레드가 잠금을 획득하지 않은 경우 RuntimeError에가 발생합니다.

Condition Class 예시 1


```
import logging
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s (%(threadName)-2s) %(message)s',
                    )

def consumer(cond):
    """wait for the condition and use the resource"""
    logging.debug('Starting consumer thread')
    t = threading.currentThread()
    print(" consumer ", t)
    with cond:
        cond.wait()
        logging.debug('Resource is available to consumer')

def producer(cond):
    """set up the resource to be used by the consumer"""

    t = threading.currentThread()
    logging.debug('Starting producer thread')
    print(" producer ", t)
    with cond:
        logging.debug('Making resource available')
        cond.notifyAll()
```

Condition Class 예시 2


```
condition = threading.Condition()
c1 = threading.Thread(name='c1', target=consumer, args=(condition,))
c2 = threading.Thread(name='c2', target=consumer, args=(condition,))
p = threading.Thread(name='p', target=producer, args=(condition,))


c1.start()
time.sleep(2)
c2.start()
time.sleep(2)
p.start()
```

```
2016-11-07 15:49:50,261 (c1) Starting consumer thread
 consumer  <Thread(c1, started 9296)>
2016-11-07 15:49:52,261 (c2) Starting consumer thread
 consumer  <Thread(c2, started 16424)>
2016-11-07 15:49:54,263 (p ) Starting producer thread
2016-11-07 15:49:54,264 (p ) Making resource available
2016-11-07 15:49:54,265 (c2) Resource is available to consumer
 producer  <Thread(p, started 6860)>
2016-11-07 15:49:54,267 (c1) Resource is available to consumer
```

3. MULTI PROCESSING 모듈

Moon Yong Joon

PROCESS CLASS

Moon Yong Joon

Process class

Process 객체 생성 방법

Process class는 target에는 실행함수, args, kwargs에는 실행함수 전달인자를 지정

Procss(target=함수명, name=None, args=함수인자, kwargs=함수인자, *, daemon=None)

```
%%writefile process_test1.py


from multiprocessing import Process, current_process

def doubler(number):

    print('number{0} process id: {1}'.format(
        number, current_process()))


if __name__ == '__main__':
    numbers = [5, 10, 15, 20, 25]
    procs = []

    for index, number in enumerate(numbers):
        proc = Process(target=doubler, args=(number,))
        procs.append(proc)
        proc.start()

    for proc in procs:
        proc.join()
```

Process 객체 : args

함수(파라미터)를 args에 전달하려면 args=(파라미터)로 동일한 개수를 입력해서 처리

Target 함수 doubler(number) 와 target=doubler, args=(number,) 를 매칭해야 함

```
from multiprocessing import Process, current_process

def doubler(number):

    print('number{0} process id: {1}'.format(
        number, current_process()))
```

```
if __name__ == '__main__':
    numbers = [5, 10, 15, 20, 25]
    procs = []

    for index, number in enumerate(numbers):
        proc = Process(target=doubler, args=(number,))
        procs.append(proc)
        proc.start()

    for proc in procs:
        proc.join()
```

Process class 변수

Process class 변수 1

Process class에 정의된 변수를 출력

```
%%writefile mp_sample1.py
import multiprocessing as mp
import os
import time

def func(x) :
    print("func processing")

def main() :
    for i in range(1) :
        p = mp.Process(target=func, args=(i,))
        p.start()

        print("process authkey ",p.authkey)
        print("process daemon ",p.daemon) # Return whether process is a daemon
        print("process name ",p.name)
        print("process ident ",p.ident) #Return identifier (PID) of process or `None` if it has yet to start
        print("process pid ",p.pid)
        # Return a file descriptor (Unix) or handle (Windows) suitable for waiting for process termination.
        print("process sentinel ",p.sentinel)

    time.sleep(2)
    p.join()

if __name__ == "__main__" :
    main()
```

```
Overwriting mp_sample1.py
```

Process class 변수 2

authkey, daemon, name, ident, pid, sentinel 에 대한 변수값 출력

```
!python mp_sample1.py
```

```
func processing
process authkey  b'i\xb8\xd8V\xab]\xd8Im\x87\x88\x7f\xab\xfb\xac2;\xed.\xfc\xe3c\xab;\xe5\n\x13\x93f\x00\x90\xb4'
process daemon  False
process name  Process-1
process ident  17708
process pid  17708
process sentinel  196
```

Process class 변수 할당

변수 할당 모듈 생성

Process class 내의 변수 name에 사용자 정의 및 default 할당

```
%%writefile process_variance.py
import multiprocessing
import time

def worker():
    name = multiprocessing.current_process().name
    print(name, 'Starting')
    time.sleep(2)
    print( name, 'Exiting')

def my_service():
    name = multiprocessing.current_process().name
    print(name, 'Starting')
    time.sleep(3)
    print( name, 'Exiting')

if __name__ == '__main__':
    service = multiprocessing.Process(name='my_service', target=my_service)
    worker_1 = multiprocessing.Process(name='worker 1', target=worker)
    worker_2 = multiprocessing.Process(target=worker) # use default name

    worker_1.start()
    worker_2.start()
    service.start()
```

```
Overwriting process_variance.py
```

변수 할당 모듈 실행

Process class 내의 변수 name에 사용자 정의 및 default 처리에 대한 실행

```
!python process_variance.py
```

```
Process-3 Starting
Process-3 Exiting
worker 1 Starting
worker 1 Exiting
my_service Starting
my_service Exiting
```

Process instance 내부

Process instance 변수 1

Process class로 process 인스턴스를 생성하면 아래의 변수들이 생김

```
%%writefile process_test3.py
import multiprocessing
import time

def worker(*args,**kwargs):
    proc = multiprocessing.current_process()
    print(proc, args,kwargs)
    time.sleep(2)
    print(proc.__dict__ )

if __name__ == '__main__':

    worker_1 = multiprocessing.Process(name='worker 1',
                                       target=worker,
                                       args=(1,2,3),
                                       kwargs={'a':1,'b':2})

    worker_1.start()

    worker_1.join(4)
```

Process instance 변수 2

Process class의 속성 이름으로 조회하면 실제 process 인스턴스의 값을 조회

```
: !python process_test3.py
```

```
<Process(worker 1, started)> (1, 2, 3) {'a': 1, 'b': 2}

{
  '_popen': None,
  '_name': 'worker 1',
  '_target': <function worker at 0x000000000107F378>,
  '_kwargs': {'a': 1, 'b': 2},
  '_parent_pid': 12800,
  '_args': (1, 2, 3),
  '_config': {
              'authkey':
                '\xca\xa2\x02\x19\xaaG\x88\xc8\x91c\xa2\xb9\xd2\xe9fK\x88\xc7\xeco^\xce\x13z\xc1\xc6\xfd\x11\xd6\xa89\xe0',
             'semprefix': '/mp‘
           },
  '_identity': (1,)
}
```

PROCESS
생성 및 실행

Moon Yong Joon

Process 하나 생성

Process : 한 개 정의 및 실행

프로세스를 정의하고 함수를 프로세스에 전달해서 실행

```
%%writefile process_test1.py
from multiprocessing import Process

def say_hello(name='world'):
    print("Hello, %s" % name)

if __name__ == "__main__" :
    p = Process(target=say_hello, name="say_hello")

    p.start()
    print(p.name)
    print(p.pid)
    p.join()
```

```
Overwriting process_test1.py
```

```
!python process_test1.py
```

```
Hello, world
say_hello
2928
```

Process 여러개 생성

Process : 여러 개 정의 및 실행

Process class를 생성해서 프로세스를 3개 만들고 처리

```
%%writefile mp_sample.py
import multiprocessing as mp
import os
import time

def func(x) :

    print(" process id ", os.getpid())
    print(" parents process id",os.getppid())

def main() :
    for i in range(3) :
        p = mp.Process(target=func, args=(i,))
        print(p)
        p.start()
    time.sleep(2)
    p.join()

if __name__ == "__main__" :
    main()
```

```
Overwriting mp_sample.py
```

```
!python mp_sample.py
```

```
 process id  17136
 parents process id 14040
 process id  17560
 parents process id 14040
 process id  10996
 parents process id 14040
<Process(Process-1, initial)>
<Process(Process-2, initial)>
<Process(Process-3, initial)>
```

PROCESS DAEMON

Moon Yong Joon

Process instance변수

daemon processing 모듈 정의

daemon process는 지속적으로 작업을 하므로 항상 실행되어야 함

```
%%writefile process_daemon.py
import multiprocessing
import time, sys

def daemon():
    p = multiprocessing.current_process()
    print('Starting:', p.name, p.pid)
    sys.stdout.flush() # 정의하지 않으면 출력이 안됨
    time.sleep(2)
    print('Exiting :', p.name, p.pid)
    sys.stdout.flush()

def non_daemon():
    p = multiprocessing.current_process()
    print('Starting:', p.name, p.pid)
    print('Exiting :', p.name, p.pid)

if __name__ == '__main__':
    d = multiprocessing.Process(name='daemon', target=daemon)
    d.daemon = True

    n = multiprocessing.Process(name='non-daemon', target=non_daemon)
    n.daemon = False

    d.start()
    time.sleep(1)
    n.start()
    print("daemon processing status ",d.is_alive())
```

```
Overwriting process_daemon.py
```

daemon processing 실행

daemon process를 실행해서 is_alive 메소드로 확인하면 계속 활동 중

```
!python process_daemon.py
```

```
Starting: daemon 5856
daemon processing status  True
Starting: non-daemon 16220
Exiting : non-daemon 16220
```

PROCESS TERMINATE

Moon Yong Joon

Process 종료하기

terminamte 메소드

process를 termiante 메소드로 먼저 중단시킴

```
%%writefile process_terminate.py
import multiprocessing
import time

def slow_worker():
    print('Starting worker')
    time.sleep(0.1)
    print('Finished worker')

if __name__ == '__main__':
    p = multiprocessing.Process(target=slow_worker)
    print('BEFORE:', p, p.is_alive())

    p.start()
    print('DURING:', p, p.is_alive())

    p.terminate()
    print('TERMINATED:', p, p.is_alive())

    p.join()
    print('JOINED:', p, p.is_alive())
```

```
Writing process_terminate.py
```

```
!python process_terminate.py
```

```
BEFORE: <Process(Process-1, initial)> False
DURING: <Process(Process-1, started)> True
TERMINATED: <Process(Process-1, started)> True
JOINED: <Process(Process-1, stopped[SIGTERM])> False
```

Daemon Process 종료하기

terminamte 메소드

daemon process를 termiante 메소드로 중단되지 않음

```
%%writefile process_daemon1.py
import multiprocessing
import time, sys, os, signal

def daemon():
    p = multiprocessing.current_process()
    print('Starting:', p.name, p.pid)
    sys.stdout.flush() # 정의하지 않으면 출력이 안됨
    time.sleep(2)
    print('Exiting :', p.name, p.pid)
    sys.stdout.flush()

if __name__ == '__main__':
    d = multiprocessing.Process(name='daemon', target=daemon)
    d.daemon = True

    d.start()
    time.sleep(1)
    print('DURING:', d, d.is_alive())

    d.terminate()   # daemon은 죽지 않음
    print('TERMINATED:', d, d.is_alive())
    |
    os.kill(d.pid, signal.CTRL_C_EVENT)
    print('Kill :', d, d.is_alive())

    sys.exit() # daemon은 현재 빠져나가야 함
    print('sys exit :', d, d.is_alive())
```

```
Overwriting process_daemon1.py
```

sys.exit()으로 전체를 종료

PROCESS
함수

Moon Yong Joon

현재 프로세스 조회

current_process 1

Pool 내의 처리가능한 process만 생성하고 map 메소드로 함수 실행

```
%%writefile process_test7.py

import multiprocessing
import time

def myfunction(i):

    print(i, " excute process ",multiprocessing.current_process())
    time.sleep(0.5)
    return i*i

if __name__ == "__main__":
    print("__name__")
    p = multiprocessing.Pool(2)
    result = p.map(myfunction, range(10))
    print(" result ", result)
```

```
Overwriting process_test7.py
```

current_process 2

정의된 내용을 실행하면 각 process가 주어진 함수에 각 인자를 받으면서 실행

```
!python process_test7.py
```

```
__name__
 result  [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
0  excute process  <SpawnProcess(SpawnPoolWorker-2, started daemon)>
1  excute process  <SpawnProcess(SpawnPoolWorker-2, started daemon)>
4  excute process  <SpawnProcess(SpawnPoolWorker-2, started daemon)>
5  excute process  <SpawnProcess(SpawnPoolWorker-2, started daemon)>
2  excute process  <SpawnProcess(SpawnPoolWorker-1, started daemon)>
3  excute process  <SpawnProcess(SpawnPoolWorker-1, started daemon)>
6  excute process  <SpawnProcess(SpawnPoolWorker-1, started daemon)>
7  excute process  <SpawnProcess(SpawnPoolWorker-1, started daemon)>
8  excute process  <SpawnProcess(SpawnPoolWorker-1, started daemon)>
9  excute process  <SpawnProcess(SpawnPoolWorker-1, started daemon)>
```

모듈 분리(실행함수, PROCESS 처리) 처리

Moon Yong Joon

Process와 함수 모듈 분리

실행 함수를 별도 모듈로 정의

Process로 전달될 target 함수를 별도 모듈로 정의

```
%%writefile import_worker.py
def worker():
    """worker function"""
    print('Worker')
    return
```

```
Overwriting import_worker.py
```

Process 처리 모듈 생성/실행

Process 정의시 targer에 imort한 함수를 전달하고 생성하고 start메소드로 실행

```
%%writefile process_main.py
import multiprocessing
import import_worker

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=import_worker.worker)
        jobs.append(p)
        p.start()
```

```
Overwriting process_main.py
```

```
!python process_main.py
```

```
Worker
Worker
Worker
Worker
Worker
```

PROCESS 재사용 POOL 처리

Moon Yong Joon

Pool구조

Pool class

Pool class를 processes 개수를 넣고 많들면 precess pool object를 리턴

```
Pool(processes=None, initializer=None, initargs=(),
maxtasksperchild=None)
```

pool 처리

Pool 처리

- Multiprocessing has the Pool object. This supports the up-front creation of a number of processes and a number of methods of passing work to the workers.
- Pool.apply() - this is a clone of builtin apply() function.
  - Pool.apply_async() - which can call a callback for you when the result is available.
- Pool.map() - again, a parallel clone of the built in function.
  - Pool.map_async() method, which can also get a callback to ring up when the results are done.

pool 내의 메소드 처리

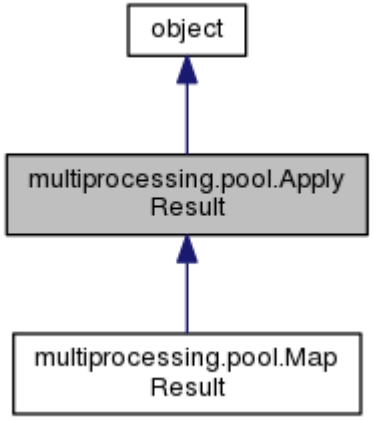
Pool 내의 4개의 메소드가 존재하며 이 메소드 별도 처리 기준이 다름

	Multi-args	Concurrence	Blocking	Ordered-results
map	no	yes	yes	yes
apply	yes	no	yes	no
map_async	no	yes	no	yes
apply_async	yes	yes	no	no

Pool.map 실행

map 메소드

map 메소드는 실행함수에 대한 인자를 하나만 처리(iterable로 제공)가 가능하며 결과는 list로 제공함




```
from multiprocessing import Pool
help(Pool().map)
```

```
Help on method map in module multiprocessing.pool:

map(func, iterable, chunksize=None) method of multiprocessing.pool.Pool instance
    Apply `func` to each element in `iterable`, collecting the results
    in a list that is returned.
```

map : 작성

task (함수와 data)로 분리하고 각 프로세스에 할당

```
%%writefile process_npsqrt.py

from multiprocessing import Pool, current_process
import numpy

def sqrt(x):
    print(" process", current_process())
    return numpy.sqrt(x)

if __name__ == '__main__':
    pool = Pool(5)
    roots1 = pool.map(sqrt, range(6))
    print(roots1)
    roots2 = pool.map(sqrt, range(6,12))
    print(roots2)
```

```
Overwriting process_npsqrt.py
```

map : 실행

프로세스를 확인해보면 하나의 프로세스에서 처리 함

```
!python process_npsqrt.py
```

```
[0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0, 2.23606797749979]
[2.449489742783178, 2.6457513110645907, 2.8284271247461903, 3.0, 3.1622776601683795, 3.3166247903554]
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
```

map 동시성 처리 가능

프로세스를 확인해보면 하나의 프로세스에서 처리 함

```
%%writefile pool_map1.py
from multiprocessing import Pool, current_process
import numpy as np

def worker(x):
    print(current_process())
    return np.add(x,x)

if __name__ == "__main__" :

    pool = Pool(processes=4)
    # map
    results = pool.map(worker, [1, 2, 3])
    print(results)
    pool.close()
```

```
Overwriting pool_map1.py
```

```
!python pool_map1.py
```

```
[2, 4, 6]
<SpawnProcess(SpawnPoolWorker-1, started daemon)>
<SpawnProcess(SpawnPoolWorker-4, started daemon)>
<SpawnProcess(SpawnPoolWorker-4, started daemon)>
```

Pool.map_async 실행

process Pool 생성

Pool 내의 처리가능한 process만 생성하고 map_async 메소드로 함수 실행

```
%%writefile process_mapasync.py
import multiprocessing
import time

def square(x):
    # This is is reeeeally slow way to square numbers.
    print(" excute process ",multiprocessing.current_process())
    time.sleep(0.5)
    return x**2

if __name__ == "__main__" :
    print("Creating pool with 3 workers")
    pool = multiprocessing.Pool(processes=3)
    print("Invoking map_async(square, [11, 12, 13, 14, 15, 16])")
    result = pool.map_async(square, [11, 12, 13, 14, 15, 16])
    print( "Waiting for result...")
    start_time = time.time()
    print("Result: %r (%.2f secs)" % (result.get(), time.time() - start_time))
    pool.close()
    pool.join()
```

```
Writing process_mapasync.py
```

process Pool 처리결과

정의된 내용을 실행하면 각 processsquffh 전부 사용하고 결과값을 합쳐서 표시

```
!python process_mapasync.py
```

```
 excute process  <SpawnProcess(SpawnPoolWorker-2, started daemon)>
 excute process  <SpawnProcess(SpawnPoolWorker-2, started daemon)>
 excute process  <SpawnProcess(SpawnPoolWorker-1, started daemon)>
 excute process  <SpawnProcess(SpawnPoolWorker-1, started daemon)>
 excute process  <SpawnProcess(SpawnPoolWorker-3, started daemon)>
 excute process  <SpawnProcess(SpawnPoolWorker-3, started daemon)>
Creating pool with 3 workers
Invoking map_async(square, [11, 12, 13, 14, 15, 16])
Waiting for result...
Result: [121, 144, 169, 196, 225, 256] (1.23 secs)
```

callback 인자 사용

Pool 내의 처리가능한 process만 생성하고 map_async 메소드에 callback도 정의하고 함수 실행

```
%%writefile pool_map1.py
from multiprocessing import Pool, current_process
import numpy as np

def worker(x):
    print(current_process())
    return np.add(x,x)

results_1 = []
def result_back(result) :
    results_1.append(result)

if __name__ == "__main__" :

    pool = Pool(processes=4)
    # map
    result = pool.map_async(worker, [1, 2, 3], callback=result_back)

    print(result.get())
    print(results_1)
    pool.close()
    pool.join()
```

```
Overwriting pool_map1.py
```

```
!python pool_map1.py
```

```
<SpawnProcess(SpawnPoolWorker-1, started daemon)>
<SpawnProcess(SpawnPoolWorker-1, started daemon)>
<SpawnProcess(SpawnPoolWorker-1, started daemon)>
[2, 4, 6]
[[2, 4, 6]]
```

Pool.apply 실행

Pool.apply()

Pool 내의 apply_aync() 처리결과 처리

```
%%writefile process_apply.py
import multiprocessing
import time

def square(x):
    # This is is reeeeally slow way to square numbers.
    print(" excute process ",multiprocessing.current_process())
    time.sleep(0.5)
    return x**2

if __name__ == "__main__" :
    print("Creating pool with 3 workers")
    pool = multiprocessing.Pool(processes=3)
    print(pool._pool)
    print("Invoking apply(square, 3)")
    print("Result: %s" % (pool.apply(square, [3]),))
    print("Result: %s" % (pool.apply(square, [9]),))
    pool.close()
    pool.join()
```

```
Overwriting process_apply.py
```

Pool.apply() 실행

Pool 내의 apply() 처리결과 처리

```
!python process_apply.py
```

```
 excute process  <SpawnProcess(SpawnPoolWorker-2, started daemon)>
 excute process  <SpawnProcess(SpawnPoolWorker-3, started daemon)>
Creating pool with 3 workers
[<SpawnProcess(SpawnPoolWorker-1, started daemon)>, <SpawnProcess(SpawnPoolWorker-2, started daemon)>, <SpawnProcess(SpawnPoolWo
rker-3, started daemon)>]
Invoking apply(square, 3)
Result: 9
Result: 81
```

Pool.apply_async :callback

process Pool 생성

Pool 내의 처리가능한 process만 생성하고 apply_async 메소드로 함수 실행

```
%%writefile process_test9.py

import multiprocessing as mp
import time

def foo_pool(x):
    time.sleep(2)
    return x*x

result_list = []
def log_result(result):
    # This is called whenever foo_pool(i) returns a result.
    # result_list is modified only by the main process, not the pool workers.
    result_list.append(result)

def apply_async_with_callback():
    pool = mp.Pool()
    for i in range(10):
        pool.apply_async(foo_pool, args = (i, ), callback = log_result)
    pool.close()
    pool.join()
    print(result_list)

if __name__ == '__main__':
    apply_async_with_callback()
```

```
Writing process_test9.py
```

process Pool 처리결과

정의된 내용을 실행하면 각 process가 주어진 함수에 각 인자를 받으면서 실행

```
!python process_test9.py
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

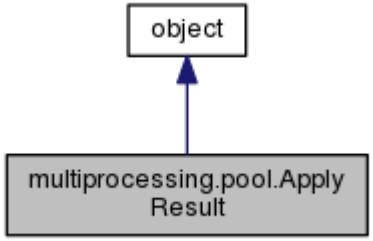
Pool.apply_async : get

apply_async : 작성

pool.apply_async 은 proxy 객체로 전달하므로 결과값.get()으로 처리

Pool.apply() - this is a clone of builtin apply() function.

- Pool.apply_async() - which can call a callback for you when the result is available.




```
%%writefile process_npsqrt1.py

from multiprocessing import Pool, current_process
import numpy

def sqrt(x):
    print(" process", current_process())
    return numpy.sqrt(x)


if __name__ == '__main__':

    pool = Pool()
    results = [pool.apply_async(sqrt, (x,)) for x in range(6)]
    print(results)
    roots = [r.get() for r in results]
    print(roots)
```

apply_async : 실행

프로세스를 확인해보면 하나의 프로세스에서 처리 함

```
!python process_npsqrt1.py
```

```
[<multiprocessing.pool.ApplyResult object at 0x0000000003742908>, <multiprocessing.pool.ApplyResult object at 0x00000000037429E8
>, <multiprocessing.pool.ApplyResult object at 0x0000000003742AC8>, <multiprocessing.pool.ApplyResult object at 0x0000000003742B
A8>, <multiprocessing.pool.ApplyResult object at 0x0000000003742C88>, <multiprocessing.pool.ApplyResult object at 0x000000000374
2D68>]
[0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0, 2.2360679774997898]
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
 process <SpawnProcess(SpawnPoolWorker-4, started daemon)>
```

Pool.apply_async()

Pool 내의 apply_async() 처리결과를 get 메소드로 가져오기

```
%%writefile process_applyasync.py
import multiprocessing
import time

def square(x):
    # This is is reeeeally slow way to square numbers.
    print(" excute process ",multiprocessing.current_process())
    time.sleep(0.5)
    return x**2

if __name__ == "__main__" :
    print("Creating pool with 3 workers")
    pool = multiprocessing.Pool(processes=3)
    print("Invoking apply_async(square, 4)")
    result = pool.apply_async(square, [4])
    print("Waiting for result...")
    start_time = time.time()
    print("Result: %s (%.2f secs)" % (result.get(), time.time() - start_time))
    pool.close()
    pool.join()
```

```
Writing process_applyasync.py
```

process Pool 처리결과

정의된 내용을 실행하면 각 process가 주어진 함수에 각 인자를 받으면서 실행

```
!python process_applyasync.py
```

```
 excute process  <SpawnProcess(SpawnPoolWorker-2, started daemon)>
Creating pool with 3 workers
Invoking apply_async(square, 4)
Waiting for result...
Result: 16 (0.69 secs)
```

PROCESS COMMUNICATION SHARED MEMORY 처리

Moon Yong Joon

Value

Value 객체를 통해 공유

process 간 메모리 처리를 위해 Value 객체로 값 생성

```
%%writefile multi_test.py
import multiprocessing

def f1(x) :
    print(" f1 process", multiprocessing.current_process())
    num = 10
    while x.value < num :
        print(x.value)
        x.value += 1
    print(x.value)

def f2(x) :
    print(" f2 process", multiprocessing.current_process())

    while x.value > 0 :
        print(x.value)
        x.value -= 1
    print(x.value)

if __name__ == '__main__' :
    #shared memory
    x = multiprocessing.Value('i', 0)
    th1 = multiprocessing.Process(target=f1, args=(x,))
    th2 = multiprocessing.Process(target=f2, args=(x,))
    th1.start()
    th2.start()
    th1.join()
    th2.join()
    print(x.value)
```

실행결과

process 간 메모리 처리를 위해 Value 객체로 값 처리

```
!python multi_test.py
 f1 process <Process(Process-1, started)>
0
1
2
3
4
5
6
7
8
9
10
 f2 process <Process(Process-2, started)>
10
9
8
7
6
5
4
3
2
1
0
0
```

PROCESS COMMUNICATION QUEUE 처리

Moon Yong Joon

Queue 구조

Queue는 deque로 데이터 보관

Process로 전달될 queue는 deque로 저장되어 저장순서대로 데이터를 꺼냄

```
from multiprocessing import Queue

q = Queue()

q.put('Why hello there!')
q.put(['a', 1, {'b': 'c'}])

print(q._buffer)

print(q.get())# Returns 'Why hello there!'
print(q.get())# Returns ['a', 1, {'b': 'c'}]
```

```
deque(['Why hello there!', ['a', 1, {'b': 'c'}]])
Why hello there!
['a', 1, {'b': 'c'}]
```

단일 process에서 put/get

Queue: 동시 메시지 처리

Process를 정의해서 queue에 메시지를 전송하고 이를 받아서 출력

```
%%writefile process_test3.py
from multiprocessing import Process, Queue, current_process
import random

process_str = []
def rand_num(queue):
    num = random.random()
    p = current_process()
    queue.put([p.name,num])
    process_str.append(p)
    print(p, num)

if __name__ == "__main__":
    queue = Queue()
    processes = [Process(target=rand_num, args=(queue,)) for x in range(4)]

    for p in processes:
        p.start()

    for p in processes:
        print(queue.get())


    for p in processes:
        p.join()
```

Queue: 동시 메시지 처리

Process를 정의해서 queue에 메시지를 전송하고 이를 받아서 출력

```
!python process_test3.py
```

```
<Process(Process-4, started)> 0.2522448010584196
<Process(Process-2, started)> 0.34353831133644364
<Process(Process-1, started)> 0.5899377989836242
<Process(Process-3, started)> 0.9579919151283043
['Process-4', 0.2522448010584196]
['Process-2', 0.34353831133644364]
['Process-1', 0.5899377989836242]
['Process-3', 0.9579919151283043]
```

Process간 메시지 교환

Queue: process간 메시지교환

Process를 정의해서 queue에 메시지를 전송하고 이를 받아와서 처리

```
%%writefile process_test2.py
from multiprocessing import  Process, Queue

q = Queue()

def put_queue(q,msg):
    q.put(msg)
    print("put , %s" % q._buffer)

def get_queue(q):
    msg = q.get()
    print("queue buffer , %s" % q._buffer)
    print("get, %s" % msg)

if __name__ == "__main__" :
    msg = "Why hello there!"
    p = Process(target=put_queue, name="put_queue", args=(q,msg))
    g = Process(target=get_queue, name="get_queue", args=(q,))

    p.start()
    g.start()
    p.join()
    g.join()
```

```
!python process_test2.py
```

```
put , deque(['Why hello there!'])
queue buffer , deque([])
get, Why hello there!
```