파이썬 동시성 프로그래밍

파이썬 동시성 프로그래밍

명료하고 훌륭한 동시성 파이썬 코드 작성하고 개념 이해하기

엘리엇 포브스 지음

이창화 옮김

Packt>

에이콘

지은이 소개

엘리엇 포브스 Elliot Forbes

2년간 J.P. 모건체이스앤컴퍼니JPMorgan Chase에서 풀타임 소프트웨어 엔지니어로 일했다. 대학 생활 동안 웹 솔루션 개발 프리랜서로 일했으며, 2015년 스코틀랜드의 스트라스클라이드 대학교를 졸업했다.

Go 언어, Node.js, 자바 등 다양한 언어를 다루며 기업용 동시성 시스템 개발에 많은 시간을 보냈다. 이러한 경험들이 이 책을 집필한 계기가 됐다.

런던의 바클리스 투자 은행Barclays Investment Bank에서 인턴으로 있었으며, 지난 3년간 여러 소프트웨어 개발 웹사이트를 운영했다.

기술 감수자 소개

니콜라우스 그래드올Nikolaus Gradwohl

1976년 오스트리아의 비엔나에서 태어나 자이로 기어루스Gyro Gearloose[1] 같은 발명가를 꿈꾸는 사람이었다. 아타리Atari[2]를 처음 가졌을 때, 컴퓨터 프로그래머가 되기로 마음먹었다. 8비트 마이크로컨트롤러부터 메인프레임 프로그램을 만드는 데 많은 시간을 보냈으며, 시간이 날 때는 종종 프로그래밍 언어나 운영체제 관련 공부를 한다.

저서로는 「Processing 2: Creative Coding Hotshot」(Packt, 2013)이 있으며, http://www.local-guru.net/에서 그의 정보를 볼 수 있다.

1 디즈니 만화 영화에 나오는 발명가 캐릭터 – 옮긴이

2 1970년대 최초의 8비트 콘솔 게임기 – 옮긴이

옮긴이 소개

이창화(leechanghwa.knu@gmail.com)

경북대학교에서 기계공학 및 컴퓨터공학을 전공하고 있으며, 여러 방면의 공학 기술과 학문에 관심이 많다. 대학 입학 전 프로그래밍에 관심을 갖기 시작한 후부터 C, 파이썬, 웹 언어, 오픈소스 임베디드 개발을 하게 됐다. 최근에는 머신 러닝과 딥러닝에 관련해 공부하고 있으며, 관련 도서를 찾고 읽는 데 푹 빠져 있다. 회사에 연연하지 않고 원하는 일과 연구에 몰입할 수 있는 라이프를 추구한다. 옮긴 책으로 『파이썬을 이용한 데이터 분석』(에이콘, 2018)이 있다.

옮긴이의 말

IEEE, PYPL, Tiobe를 참고하면 파이썬 언어의 사용 빈도 순위는 모두 5위 안에 들 만큼 전 세계의 많은 사람이 사용하고 있다. 그만큼 쉬운 문법 구조와 사용 및 편리함을 나타내고 있다.

컴퓨터 및 전자 관련 전공으로 입학하면 가장 먼저 배우는 것이 '컴퓨터 구조'다. 폰 노이만의 현대식 컴퓨터 개발부터 시작해, 오늘날 운영체제의 원리까지 길고 긴 여정이 시작된다. 그중 가장 기본적인 컴퓨터 구조 및 원리에서 동시성과 병렬화를 빼놓을 수 없다. 이러한 개념을 적용하는 이유는 바로 프로그램의 속도 때문이다. 더 빠르고 효율적으로 연산을 수행하고자 적용하는데, 이제 막 입문한 사람들에게 쉽지만은 않다.

이 책은 스레드, 프로세스, 실행자, 풀, 이벤트 기반 및 리액트 프로그래밍 같은 컴퓨터 과학의 개념과 함께 이를 파이썬 프로그래밍으로 풀어본다. 어려운 개념과 설명을 쉽게 예제 형식으로 풀었으며, 실제로 이러한 개념을 적용할 때 사용되는 다양한 라이브러리를 적극적으로 이용한다.

먼저 동시성과 병렬화의 기본적인 개념을 소개한다. 컴퓨터 아키텍처와 관련된 부분도 쉽게 설명하고 있다. 다음으로 스레드와 관련해 설명한다. 큐 자료 구조와 접목한 스레드 프로그램을 작성해본다. 6장에서는 파이썬 프로그램의 디버깅과 벤치마킹을 해본다. 7장과 8장에서는 퓨처 객체, `ProcessPoolExecutor`, 멀티프로세싱에 대해 살펴본 후, 9장에서 `asyncio` 라이브러리를 활용해 이벤트 기반 프로그래밍을 배워본다. 10장에서는 RxPY 라이브러리를 이용해 리액트 프로그래밍을 알아보고, 11장에서는 GPU를 활용해 본다.

입문 및 예제에 초점이 맞춰진 책이다 보니, 깊은 개념에 대한 설명이 조금 부족할 수 있다. 각주 및 참고 박스를 통해 최대한 친절하게 전달하고자 노력했다. 대부분 윈도우 운영체제 환경에서 정상적으로 실행되지만, 스레드나 프로세스를 다루는 예제에서는 리눅스 환경을 이용하거나 각주를 참고하자.

최근에는 초등학교부터 코딩 열풍이 분다고 한다. 사교육 시장에서는 단순한 암기와 주입식 코딩 교육이 많다고 한다. 가장 중요한 것은 흥미다. 내가 좋아하는 부분에 흥미를 가지고 즐겁게 프로그래밍을 배우고, 모르는 부분은 스스로 찾아보고 생각하는 과정이 가장 중요하다. 때로는 정교한 알고리즘을 설계해야 하며, 좋은 라이브러리가 있을 때는 이를 적재적소에 활용할 필요도 있다. 컴퓨터 과학을 배우기 시작했고 기본적인 파이썬 언어를 다룰 줄 안다면 이 책을 적극 추천한다.

끝으로, 이 책의 번역과 편집에 도움을 준 에이콘출판사 관계자분들과 가족들에게 큰 감사를 전한다.

이창화

차례

파이썬은 간단한 문법으로 다양한 하이 레벨 및 로우 레벨 라이브러리와 프레임워크를 제공하는 사용자 중심의 언어다. 가이드를 통해 코드를 연습하고 최적화해보며, 더 좋은 파이썬 코드를 작성하는 방법을 배워보자. 작은 예제들이 모여 개념을 이루고, 이를 바탕으로 좋은 애플리케이션을 작성할 수 있을 것이다.

이 책 전반에 걸쳐 효율적이고 탄탄한 동시성 애플리케이션을 작성하는 방법을 다룬다. 동시성 코드를 작성하는 데 길잡이 역할을 하는 예제들을 살펴보고, 멀티 프로세스 및 멀티 코어 시스템에서 실행 속도를 높이고 안정화하는 방법도 배워본다.

이 책의 구성

1장, '시작하기!' 스레드와 프로세스를 소개한다. 동시성 애플리케이션을 구성하는 데 있어 파이썬의 한계점도 살펴본다.

2장, '병렬화' 동시성과 병렬화의 차이점을 다양한 관점으로 접근해본다. 이 두 개념을 어떻게 CPU에 활용하고, 나아가 컴퓨터 시스템 디자인과 동시성 및 병렬화 프로그래밍이 무엇인지 배운다.

3장, '스레드 라이프' 파이썬 내장 스레드 라이브러리에 대해 자세히 다루고, 여러 가지 스레드 타입도 살펴본다. 멀티스레딩 모델과 사용자 스레드를 로우 레벨의 커널 스레드로 만드는 방법도 다룬다.

4장, '스레드 간의 동기화' 동시성 파이썬 애플리케이션에 영향을 미치는 요인을 살펴본다. 데드락의 개념과 '철학자의 저녁식사' 문제가 시스템에 어떤 영향을 주는지 알아본다.

5장, '스레드 간의 **통신**' 멀티스레드 시스템에서 통신을 구현하는 여러 메커니즘을 다룬다. 파이썬에 내장된 스레드 세이프 큐에 대해 자세히 배워본다.

6장, '디버깅과 벤치마킹' 제품 출시에 앞서 동시성 시스템에 버그가 생기지 않았는지 확인하는 기술을 여러 측면에서 살펴본다. 코드상 논리 구조에 문제가 없는지 테스트하는 방법도 알아본다.

7장, '실행자와 풀' 스레드 풀, 프로세스 풀, 퓨처 객체에 대한 모든 것을 다룬다. 스레드와 프로세스 풀을 인스턴스화하는 방법과 그 장점에 대해 살펴본다.

8장, '멀티프로세싱' 멀티프로세싱의 개념과 이를 시스템에서 어떻게 활용하는지 배운다. 생성부터 종료까지 스레드 라이프에 대해 살펴본다.

9장, '이벤트 기반 프로그래밍' 이벤트 기반 프로그래밍의 개념을 살펴보며, asyncio 라이브러리와 이벤트 기반 시스템이 어떻게 사용되는지 다룬다.

10장, '리액트 프로그래밍' 리액트 프로그래밍의 핵심 원리를 깨우친다. 리액트 프로그래밍과 이벤트 기반 프로그래밍의 주된 차이점을 살펴보고, RxPY 파이썬 라이브러리를 소개한다.

11장, 'GPU 사용하기' 데이터 사이언티스트가 실제 GPU를 활용하는 부분을 래퍼 라이브러리 활용법과 함께 소개한다.

12장, '솔루션 선택하기' 지금까지 다루지 못한 라이브러리를 간단히 소개한다. 파이썬 프로그램에서 어떤 라이브러리와 개념을 적재적소에 사용할지 그 과정을 배워본다.

준비 사항

이 책을 읽기 전에 다음 소프트웨어를 시스템에 설치하자.

- BeautifulSoup
- RxPY

- Anaconda
- Theano
- PyOpenCL

이 책의 대상 독자

이 책은 동시성 프로그래밍을 시작해볼 파이썬 개발자들을 대상으로 하며, 기본적인 파이썬 지식을 알고 있다는 가정하에 집필됐으니 참고하자.

편집 규약

이 책에서는 정보의 유형에 따라서 텍스트의 스타일이 바뀐다. 각 스타일은 다음과 같은 의미를 지닌다.

문장 속에서 코드는 다음과 같이 표기한다.

"`main` 함수에서는 `for` 반복문이 시작된다."

코드 블록은 다음과 같이 표기한다.

```
import urllib.request
import time
t0 = time.time()
req = urllib.request.urlopen('http://www.example.com')
pageHtml = req.read()
t1 = time.time()
print("Total Time To Fetch Page: {} Seconds".format(t1-t0))
```

코드의 특정 부분을 강조할 때는 굵은 글씨체로 표현한다.

```
import urllib.request
import time
t0 = time.time()
req = urllib.request.urlopen('http://www.example.com')
pageHtml = req.read()
t1 = time.time()
print("Total Time To Fetch Page: {} Seconds".format(t1-t0))
```

모든 명령행 입출력은 다음과 같이 기술한다.

```
pip install rx
```

새로운 용어나 중요한 단어, 그리고 메뉴나 대화상자처럼 컴퓨터 화면에 표시되는 단어는 다음과 같이 고딕체로 표기한다.

"**프로그램 카운터**PC, program counter로부터 명령받아, 다음 실행 위치를 알아본다."

주의를 요하거나 중요한 메시지는 이와 같이 나타낸다.

팁이나 유용한 요령은 이와 같이 나타낸다.

독자 의견

독자 여러분의 의견은 언제든지 환영한다. 이 책을 어떻게 생각하는지 부담 없이 이야기해준다면 좋겠다. 더 유익한 책을 만드는 데 있어 독자의 의견은 무엇보다 중요하다.

일반적인 의견은 이 책의 제목을 메일 제목으로 해서 feedback@packtpub.com으로 보내면 된다.

특정 분야의 책을 쓰거나 기여하는 데 관심이 있다면 www.packtpub.com/authors에 있는 저자 가이드를 참조하기 바란다.

고객 지원

팩트출판사의 구매자가 된 독자에게 도움이 되는 몇 가지를 제공하고자 한다.

예제 코드 다운로드

http://www.packtpub.com에 회원 가입해 팩트출판사의 도서를 구매한 모든 독자는 책에 등장하는 예제 코드 파일을 직접 내려받을 수 있다. 다른 곳에서 도서를 구매한 독자는 http://www.packtpub.com/support에 접속해 등록하면 이메일로 직접 받아볼 수 있다.

에이콘출판사의 도서정보 페이지 http://www.acornpub.co.kr/book/concurrency-python에서도 예제 코드를 내려받을 수 있다.

이 책에 수록된 코드는 깃허브에도 올려져 있고, 주소는 https://github.com/PacktPublishing/Learning-Concurrency-in-Python이다. https://github.com/PacktPublishing/에는 다른 책의 코드와 동영상도 올라와 있으니 확인해보길 바란다.

오탈자

내용을 정확하게 전달하려고 최선을 다했지만, 실수가 있을 수 있다. 팩트출판사의 책에서 텍스트나 코드상의 문제를 발견해서 알려준다면, 매우 감사하게 생각할 것이다. 그러한 참여를 통해 다른 독자에게 도움을 주고, 다음 버전에서 책을 더 완성도 있게 만들 수 있다. 오자를 발견한다면 http://www.packtpub.com/submit-errata에서 Errata

Submission Form 링크를 통해 구체적인 내용을 알려주기 바란다. 보내준 내용이 확인되면 웹사이트에 그 내용이 올라가거나, 해당 서적의 정오표 섹션에 그 내용이 추가될 것이다.

https://www.packtpub.com/books/content/support를 방문해 검색창에 해당 타이틀을 입력하면 지금까지의 정오표를 확인할 수 있다. 한국어판은 에이콘출판사의 도서정보 페이지 http://www.acornpub.co.kr/book/concurrency-python에서 찾아볼 수 있다.

저작권 침해

인터넷에서의 저작권 침해는 모든 매체에서 벌어지고 있는 심각한 문제다. 팩트출판사에서는 저작권과 사용권 문제를 아주 심각하게 인식하고 있다. 어떤 형태로든 팩트출판사 서적의 불법 복제물을 인터넷에서 발견한다면 적절한 조치를 취할 수 있게 해당 주소나 사이트명을 알려주길 부탁한다.

의심되는 불법 복제물의 링크를 copyright@packtpub.com으로 보내주기 바란다.

저자와 더 좋은 책을 위한 팩트출판사의 노력을 배려하는 마음에 깊은 감사의 마음을 전한다.

질문

이 책에 관련된 질문이 있다면 questions@packtpub.com으로 문의하기 바란다. 온 힘을 다해 질문에 답해드리겠다. 한국어판에 관한 질문은 이 책의 옮긴이나 에이콘출판사 편집 팀(editor@acornpub.co.kr)으로 문의할 수 있다.

01

시작하기

"최근 10년 동안, 많은 사람이 단일 컴퓨터는 한계에 도달했고 다수의 컴퓨터를 상호 연결해야만 진정한 발전을 이룰 수 있다고 주장한다."

– 진 암달(Gene Amdahl, 1922~2015)

수많은 소프트웨어가 만들어지면서, 애플리케이션의 성능을 향상하는 동시성 프로그래밍concurrent programming이 주목받기 시작했다. 이전의 단일 스레드single thread 형태의 동시성 개념 애플리케이션과 달리, 하드웨어의 성능이 높아지면서 과거에는 풀지 못했던 문제를 풀 수 있게 됐다.

동시성을 적용하면 여러 요청을 동시에 처리하고, 백엔드 작업이 완료될 때까지 기다리는 대신 프론트엔드를 업데이트하는 등 애플리케이션의 성능을 높일 수 있다. 프로그램 내에서 충돌이 일어났는지 작동 중인지 알 수 있는 방법이 전혀 없는 비반응형 프로그램

의 시대는 지나갔다.

하지만 이러한 프로그램의 성능을 높이는 데는 많은 비용이 든다. 동시성 형태로 시스템을 구현한다면, 일반적으로 코드가 복잡해질 뿐만 아니라 새로운 코드에서 버그로 인한 위험성도 뒤따르게 된다. 이를 막으려면, 우선 동시성의 핵심 기술과 개념을 깊이 이해해 새로운 오류로부터 안전한 애플리케이션을 구현할 수 있어야 한다.

1장에서는 동시성 소프트웨어 시스템을 구현하기 전에 프로그래머가 알아야 할 기본적인 지식을 다룬다. 어떤 주제가 있는지 살펴보자.

- 동시성 개념의 역사
- 스레드와 멀티스레드
- 프로세스와 멀티프로세스
- 이벤트 기반, 반응형, GPU 기반 프로그래밍의 기초
- 프로그램을 바탕으로 동시성의 효과 알아보기
- 동시성 시스템 구현 시 파이썬의 한계

동시성 개념의 역사

'동시성concurrency'은 일찍이 철도와 전신 분야에서 유래됐다.[1] 기본적으로, 다수의 기차가 한 철로에서 아무런 사고 없이 안전하게 목적지에 도착해야 한다.

1960년대 에츠허르 데이크스트라Edsger W. Dijkstra(1930~2002)의 상호 배제 문제mutual exclusion problem[2]에 관한 첫 논문으로, 학회에서는 동시성 컴퓨팅 이론이 등장하기 시작했다. 그 후 신호기, 상호 배제, 교착 상태deadlock 및 데이크스트라 최단 경로 알고리즘Dijkstra's Shortest Path Algorithm의 발표로, 기본적인 동시성 개념을 정의했다.

---

1 아직까지도 왜 수기 신호(semaphore)가 사용되는지 생각해보자. 세마포어라는 개념이 동시성에서는 어떻게 사용되는지를 4장에서 다루고 있다. – 옮긴이

2 여러 개의 병행 프로세스가 공통인 변수 및 자원을 액세스할 때는 하나의 프로세스에 한정해야 한다. – 옮긴이

컴퓨터 공학에서 동시성은 연산 분야의 연구에 비해 아직 신생 분야여서 앞으로 연구할 주제가 많이 남아 있다. 현재는 기대되는 유망 분야 중 하나이며, 학자나 프로그래밍 언어 디자이너, 개발자 같은 여러 영역에서 주목받고 있다.

고수준의 동시성 도입과 더 나은 언어 지원은 소프트웨어 아키텍처로서 동시성 솔루션을 구현하는 방법을 획기적으로 개선했다. 수년간 이러한 일은 어려웠는데, 새로운 동시성 API와 우수한 프레임워크 및 언어의 도입으로 개발자인 우리에게 좀 더 쉬워졌다.

프로그래밍 언어 디자이너는 오류가 없는 동시성 프로그램을 구현할 뿐만 아니라, 해당 언어로 작성하기도 쉬운 프로그램을 디자인하고자 한다. 구글의 고Go와 러스트Rust, 파이썬Python 같은 언어는 크게 발전했고, 현재 여러 작동 환경에서 최상의 성능을 뽑아내고 있다.

스레드와 멀티스레드

이번 절에서는 스레드thread가 무엇이며, 프로그램의 실행 속도를 높이고자 여러 개의 스레드를 어떻게 이용하는지 살펴본다.

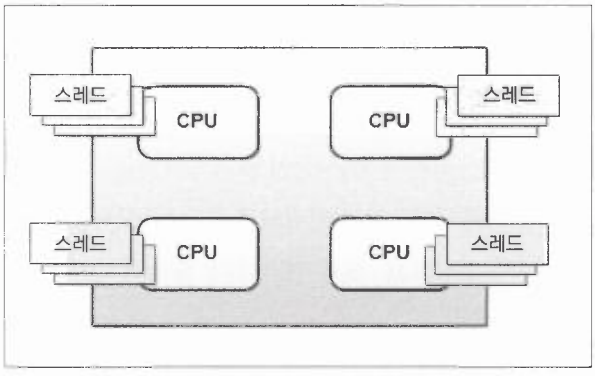
스레드란?

스레드란 운영체제에서 작동되는 스케줄링될 수 있는 인스트럭션instruction의 순차적인 흐름이다. 일반적으로 이러한 스레드는 프로세스에 속해 있으며, 프로그램 카운터program counter, 스택stack, 레지스터register를 비롯해 식별자로 구성된다. 스레드는 프로세서가 시간을 할당할 수 있는 최소 단위의 실행이라고 할 수 있다.

스레드는 공유 자원 사이에서 상호작용이 가능하며, 다수의 스레드끼리 통신도 가능하다. 메모리 공유도 가능하며, 그 밖의 메모리 주소에서 읽기 및 쓰기도 가능하지만 문제가 있다. 2개의 스레드가 메모리를 공유하고 스레드의 실행 순서가 따로 정의되지 않는

다면, 이상한 값이 도출되거나 시스템 충돌 문제를 야기할 수 있다. 이는 4장에서 자세히 살펴볼 경합 조건race condition이라는 개념 때문이다.

다음 그림은 여러 스레드가 각기 다른 CPU에 있는 구조를 보여준다.



스레드의 종류

일반적인 운영체제에서, 스레드는 크게 두 가지로 구분할 수 있다.

- 사용자 레벨 스레드: 다양한 작업에서 생성, 실행, 종료가 이뤄지는 스레드
- 커널 레벨 스레드: 운영체제의 하위 레벨에서 실행되는 스레드

파이썬은 사용자 레벨에서 동작하기에, 이 책에서는 사용자 레벨 스레드에 관한 내용에 초점을 맞추겠다.

멀티스레딩이란?

사람들이 멀티스레드 프로세서에 대해 이야기하면, 일반적으로 동시에 여러 스레드가 작동되는 단일 프로세서를 생각하게 된다. 이러한 방식은 여러 스레드 사이에서 빠르게 컨텍스트context를 바꾸는 단일 코어를 활용할 수 있다. 컨텍스트를 바꾼다는 건, 매우 짧은 시간에 여러 스레드가 (실제로는 아니지만) 병렬로 작동된다는 뜻이다.

멀티스레딩을 이해하려면, 사무실에서 일어나는 상황을 생각해보면 된다. 단일 스레드 프로그램은 한 사람이 모든 작업을 차례대로 처리한다고 보면 된다. 하지만 작업자가 문서 작업에 어려움이 있어 멈추면, 다른 작업을 더 이상 할 수 없게 된다. 즉, 문제 대처가 되지 않으며, 결론적으로는 회사의 비용 증가로 이어진다.

멀티스레딩에서는 한 명의 작업자가 시간을 쪼개어 여러 작업을 진행하는 훌륭한 멀티태스커multitasker가 된다고 보면 된다. 문서 작업을 하다가 작업이 막히면, 다른 작업을 진행한다. 특정 작업이 막혔을 때 컨텍스트를 바꿔 짧은 시간에 많은 작업을 할 수 있어, 회사 비용은 감소하게 된다.

이번 예제에서는 한 명의 작업자 또는 하나의 프로세싱 코어가 여전히 한계로 작용한다는 점을 주목하자. 회사에서 많은 업무를 병렬로 처리하고 싶으면 더 많은 직원을 구해야 하고, 파이썬의 경우에는 프로세스가 더 필요하다.

스레딩의 장점을 살펴보자.

- 다수의 스레드는 입출력 범위I/O bound[3]가 막혔을 때 속도를 획기적으로 높여준다.
- 프로세서와 비교했을 때, 메모리를 조금 차지한다.
- 스레드는 자원을 공유하기에, 스레드 간의 통신이 쉽다.

물론 다음과 같은 단점도 있다.

- CPython 스레드는 GILglobal interpreter lock[4]로 인해 사용에 제약이 따른다(다음 장에서 자세히 살펴본다).
- 스레드 간의 통신이 쉬워진 반면, 경합 조건이 발생하지 않도록 주의해서 코드를 작성해야 한다.
- 다수의 스레드 간에 컨텍스트를 바꾸는 데 수많은 계산이 필요하다. 다수의 스레드를 추가함으로써 전반적인 프로그램 성능이 저하되는 것을 볼 수 있다.

---

3 연산은 간단하고 입출력이 많을 때 속도가 떨어지는 현상 – 옮긴이

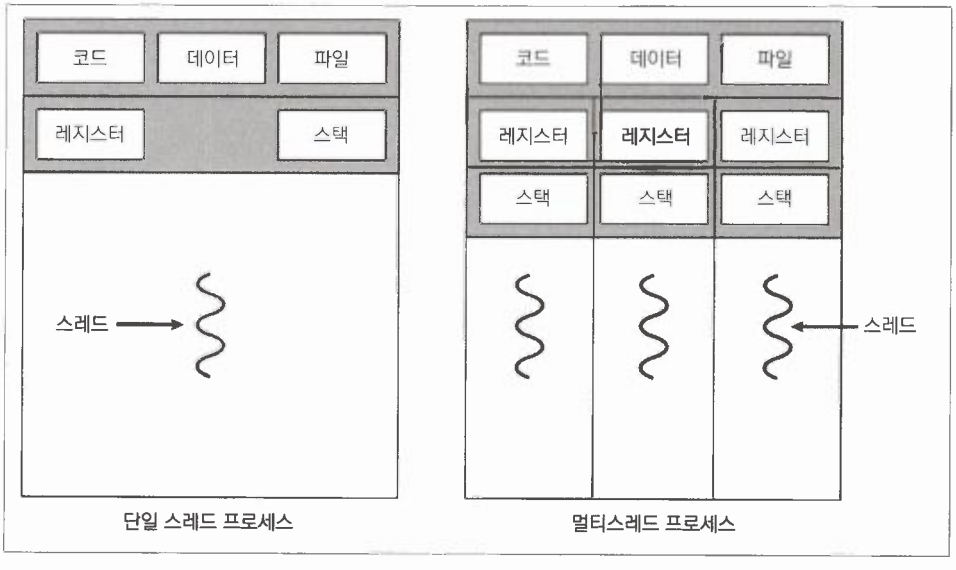
4 한 번에 하나의 스레드만 수행할 수 있는 락 – 옮긴이

프로세스

프로세스는 스레드와 비슷한 개념이지만(프로세스는 스레드가 할 수 있는 거의 모든 것이 가능하다), 한 가지 중요한 부분은 단일 CPU 코어에 국한되지 않는다는 것이다. 앞서 살펴본 비유에서 사무실이 더 커진다는 것(4 코어 CPU라고 생각하자)은, 2명의 판매부 팀원과 2명의 직원을 고용해 총 4명의 사람이 병렬적으로 업무를 처리한다고 보면 된다. 프로세스는 또한 멀티스레드 형태의 단일 작업자만큼 동일한 여러 업무를 처리할 수 있다.

이러한 프로세스는 하나의 주 스레드를 갖고 있지만, 각 스레드마다 자체의 레지스터와 스택을 포함한 다수의 서브 스레드를 만들 수 있다. 비로소 원하는 멀티스레드가 구현된 것이다. 모든 프로세스는 컴퓨터가 프로그램을 실행하는 데 필요한 모든 자원을 제공받는다.

다음 그림에서 양쪽 모두가 프로세스에 관한 것이다. 왼쪽 프로세서는 하나의 스레드만 있는데, 이 스레드를 주 스레드라고 한다. 오른쪽의 프로세스는 여러 스레드를 갖고 있으며, 각 스레드마다 자신의 레지스터와 스택으로 구성된다.



단일 스레드 프로세스 멀티스레드 프로세스

프로세스를 이용해, CPU 제약이 있거나, 더 많은 성능을 필요로 하는 특정 프로그램의 실행 속도를 높일 수 있다. 그러나 멀티프로세스로 인해 크로스 프로세스 통신cross-process communication과 관련된 문제가 발생하고, **프로세스 간 통신**IPC, inter-process communication에서 많은 시간을 낭비해 성능이 저하되지 않도록 주의해야 한다.

프로세스의 속성

운영체제에서 생성된 유닉스UNIX 프로세스는 일반적으로 다음과 같이 구성된다.

- 프로세스 ID, 프로세스 그룹 ID, 사용자 ID, 그룹 ID
- 환경
- 작업 디렉토리
- 프로그램 인스트럭션
- 레지스터
- 스택
- 힙heap
- 파일 기술자file descriptor[5]
- 신호 동작
- 공유 라이브러리
- 프로세스 간 통신 도구(메시지 큐, 파이프, 세마포어, 공유 메모리)

프로세스의 장점도 살펴보자.

- 멀티 코어 프로세서를 사용해 프로세스의 성능을 높일 수 있다.
- CPU가 많이 필요한 작업에서는 멀티스레드보다 유용하다.
- 멀티프로세스를 이용해 GIL의 한계를 피할 수 있다.
- 프로세스의 충돌은 전체 프로그램에 영향을 주지 않는다.

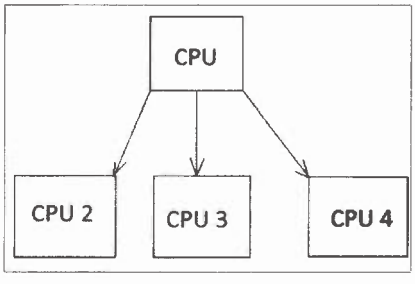
5 운영체제에서 파일을 사용할 때 각 파일에 대한 정보를 유지하는 기억 장치의 한 영역 및 정보 – 옮긴이

프로세스의 단점은 무엇일까?

- 프로세스 간 공유 자원이 없다(IPC 형태로 구현해야 한다).
- 많은 메모리가 필요하다.

멀티프로세싱

파이썬에서는 멀티스레드 또는 멀티프로세스 중 하나를 선택해 코드를 실행할 수 있어, 단일 스레드 실행보다 성능을 향상할 수 있다. 단일 코어의 처리 능력을 알아보기 전, 멀티프로세싱에 대한 이해와 컴퓨터에서 사용 가능한 모든 CPU 코어를 활용하는 부분에 대해 알아보자. 오늘날의 컴퓨터는 많은 CPU와 코어를 갖고 있는데, 하나의 코어만 사용하도록 제한하면 컴퓨터의 나머지 부분은 쉽게 유휴 상태가 된다. 여기서 목표는 하드웨어의 성능을 최대한 뽑아내는 것이며, 효율적인 비용으로 빠르게 문제를 해결하는 것이다.



파이썬의 멀티프로세싱 모듈을 사용하면, 모든 코어와 CPU를 사용할 수 있는데, CPU 집약적 문제에서는 좋은 성능을 낼 수 있다. '위의 그림은 하나의 CPU 코어가 다른 코어에게 어떻게 작업을 지시하는지 보여준다.

파이썬 2.6 이하 버전의 경우, 다음 코드에서 CPU 코어의 개수를 살펴볼 수 있다.

```
# 먼저 multiprocessing 모듈을 불러온다.
import multiprocessing
```

```
# 다음으로, multiprocessing.cpu_count()에서
# CPU의 스레드 개수를 정숫값 형태로 반환받는다.
multiprocessing.cpu_count()
```

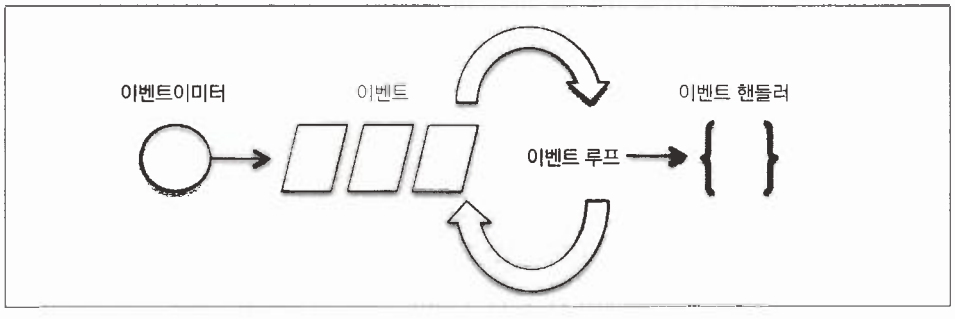
멀티프로세싱은 하드웨어를 좀 더 활용할 수 있을 뿐만 아니라, GIL이 CPython에서 문제되는 부분을 피할 수 있다.

멀티프로세스의 한 가지 본질적인 문제는 공유된 상태가 없고, 통신도 부족하다는 점이다. 그러므로 IPC 형태로 전달돼야 하고, 이것은 성능에 영향을 미칠 수도 있다. 그러나 공유 상태가 많지 않다는 건 코드에서 내부적 경합 조건에 신경을 쓰지 않아도 된다는 뜻이다.

이벤트 기반 프로그래밍

이벤트 기반 프로그래밍event-driven programming은 핸드폰을 열거나, 컴퓨터 작업을 할 때 등 일상생활에서 자주 볼 수 있다. 이러한 기기들은 이벤트 기반 방식으로 작동하는데, 컴퓨터의 아이콘을 클릭하면 운영체제가 이를 이벤트로 생각하고 관련된 부분을 실행한다.

모든 상호작용은 이벤트 과정으로 표현할 수 있으며, 일반적으로 콜백callback이 일어난다. 자바스크립트를 다뤄봤다면, 콜백의 개념 및 콜백 디자인 패턴에 대해 잘 알고 있을 것이다. 자바스크립트에서 콜백을 두드러지게 사용할 때는 RESTful HTTP 요청을 수행할 때이며, 이 작업이 성공적으로 완료되면 HTTP 응답을 받고 콜백을 사용한다.



앞의 그림을 살펴보면, 이벤트 기반 프로그램이 작동되는 모습을 간략히 볼 수 있다. 왼쪽의 **이벤트이미터**EventEmitters는 **이벤트**Event를 발생시키고, 프로그램의 **이벤트 루프**Event Loop는 이벤트를 가져와서 미리 정의된 **이벤트 핸들러**Event Handler와 매치시킨다. 그러면 매치된 핸들러가 이벤트를 처리하기 위해 수행된다.

콜백은 비동기 방식의 프로그램에서 사용된다. 예컨대, 구글에 입사 지원을 하고 이메일 주소를 전달하면 구글은 이메일로 연락을 할 것이다. 근본적으로, 이는 이메일을 주는 것 대신 콜백을 등록하는 것과 같다. 다만 콜백이 발생할 때마다 어떤 특정 부분의 코드가 실행되는 상황이 다를 뿐이다.

터틀

터틀turtle은 파이썬에서 사용하는 그래픽 모듈이며, 프로그램에 관심 있는 학생들이 사용하기 좋다. 복잡한 그래픽 프로그래밍은 터틀이 처리하고, 학생들은 오로지 관심 있는 부분만 배우는 데 집중할 수 있다.

또한 터틀은 이벤트 기반 프로그램을 테스트해보기에도 좋은 도구다. 테스트에 필요한 이벤트 핸들러 및 대기 상태listener도 갖추고 있다.

```
import turtle
turtle.setup(500,500)
window = turtle.Screen()
window.title("Event Handling 101")
window.bgcolor("lightblue")
nathan = turtle.Turtle()
def moveForward():
  nathan.forward(50)
def moveLeft():
  nathan.left(30)
def moveRight():
  nathan.right(30)
def start():
```

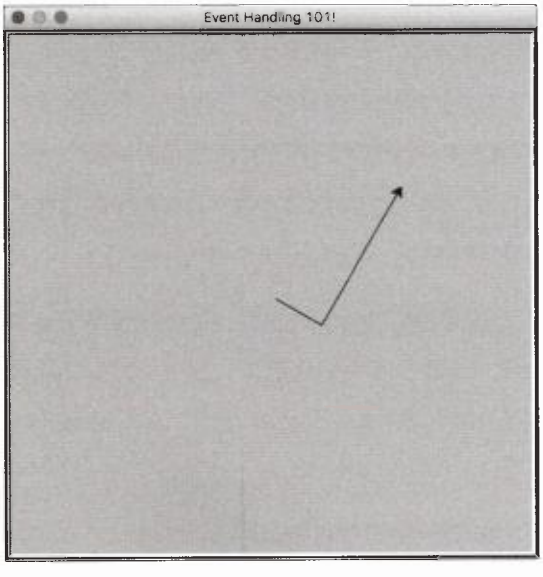
```
  window.onkey(moveForward, "Up")
  window.onkey(moveLeft, "Left")
  window.onkey(moveRight, "Right")
  window.listen()
  window.mainloop()
if __name__ == '__main__':
  start()
```

코드에 관한 설명

위 코드의 첫 번째 줄에서, 터틀 그래픽 모듈을 불러온다. 제목은 'Event Handling 101'로 하고, 배경색을 밝은 파란색으로 설정해 기본적인 창을 구성한다.

초기 구성을 마친 후, 다음 3개의 이벤트 핸들러를 정의한다.

- `moveForward`: 터틀을 50유닛만큼 앞으로 이동시킬 때
- `moveLeft/moveRight`: 터틀의 방향을 각기 30도 회전시킬 때



3개의 핸들러를 정의한 후, onkey 메소드를 사용해 위, 왼쪽, 오른쪽 버튼을 누르면 해당 이벤트 핸들러가 작동하게 한다.

핸들러 구성을 마쳤다면, 대기 상태로 만든다. 대기 상태에서 특정 버튼이 눌리면, 이벤트 핸들러 함수가 작동될 것이다. 마지막으로, 코드가 실행되면 중간에 화살표가 화면에 나타나고, 화살키로 움직일 수 있다.

반응형 프로그래밍

반응형 프로그램은 이벤트 기반 방식과 매우 비슷하지만, 이벤트 발생이 아닌 데이터에 초점을 둔다는 점에서 다르다. 자세히 말해, 데이터의 흐름을 다루면서 특정 데이터 변화에 반응하게 된다.

ReactiveX(RxPY)

RxPY는 파이썬에서 ReactiveX[6] 프레임워크와 비슷한 기능을 한다. 앵귤러Angular 2 및 이후 버전으로 프로그래밍을 해봤다면, HTTP 서비스를 사용할 때 이를 사용해봤을 것이다. 이 프레임워크는 옵저버 패턴observer pattern[7]과 반복자 패턴iterator pattern[8], 함수형 프로그래밍의 결합이다. 각기 다른 데이터의 흐름을 이용해 특정 이벤트에 반응하는 옵저버를 생성한다. 옵저버가 작동되면, 해당되는 코드가 실행된다.

데이터 센터data center를 살펴보면 반응형 프로그래밍이 어떻게 활용되는지 알 수 있다. 데이터 센터에 수천 개의 서버랙server rack[9]이 있다고 하자. 모든 서버가 끊임없이 수십만 번의 계산을 해야 한다. 데이터 센터에서 가장 중요한 부분은 빽빽하게 구성된 서버랙이 적

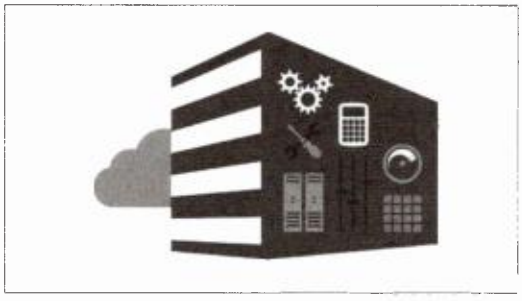
6 비동기 및 이벤트 기반 프로그램 작성에 필요한 라이브러리 – 옮긴이

7 한 객체의 상태가 바뀌면 해당 객체에 의존하는 객체들에게 연락이 가고 자동으로 내용이 갱신되는 일대다 방식의 관계 – 옮긴이

8 객체의 내부 구조를 노출하지 않고 원소를 반복자를 이용해 접근하고 순회하는 방식 – 옮긴이

9 서버를 모아 쌓아두는 보관 장비 – 옮긴이

절한 온도로 유지되는 것이다. 데이터 센터의 각 부분마다 여러 개의 온도 센서를 설치해 온도 상승이 일어나지 않게 하며, 중앙 컴퓨터는 끊임없이 센서 데이터를 수신받는다.



중앙 관제소에서는 RxPY 프로그램을 구성해 실시간 온도 정보를 관찰한다. 이러한 옵저버는 아무런 이벤트 없이 대기 상태를 유지할지, 아니면 반응을 할지가 조건을 바탕으로 정의된다.

다음은 데이터 센터의 특정 부분 온도가 올라가면 이벤트가 발생하는 예제다. 해당 이벤트가 발생하면 자동으로 반응하고, 특정 부분의 냉각 시스템이 작동해 온도가 정상적으로 내려온다.

```
import rx
from rx import Observable, Observer
# on_next 메소드, on_error 메소드, on_completed 메소드가 있는
# 사용자 옵저버를 정의한다.
class temperatureObserver(Observer):
  # 온돗값을 읽을 때마다 해당 메소드가 호출된다.
  def on_next(self, x):
    print("Temperature is: %s degrees centigrade" % x)
    if (x > 6):
      print("Warning: Temperate Is Exceeding Recommended Limit")
    if (x == 9):
      print("DataCenter is shutting down. Temperature is too high")
  # 에러 메시지를 받으면, 해당 부분에서 핸들링한다.
  def on_error(self, e):
```

```
    print("Error: %s" % e)
  # 스트림이 종료됐을 때 호출된다.
  def on_completed(self):
    print("All Temps Read")
# 가상의 온돗값을 만든다.
xs = Observable.from_iterable(range(10))
# 해당 온돗값을 전달한다.
d = xs.subscribe(temperatureObserver())
```

코드에 관한 설명

위에서 처음 두 줄은 실행에 필요한 rx 모듈을 불러오고, Observable과 Observer를 모두 불러온다.

그 후, temperatureObserver 클래스를 불러와 옵저버를 확장한다. 이 클래스는 함수 3개를 포함한다.

- on_next: 옵저버가 새로운 것을 볼 때 호출된다.
- on_error: 에러 핸들러 함수가 작동되며, 에러가 보이면 호출된다.
- on_completed: 모든 데이터 입력이 종료되면 호출된다.

on_next 함수에서는 현재 온도를 출력하며, 전달된 온돗값이 제한선보다 아래인지 확인한다. 온도가 조건에 하나라도 부합하면, 어떤 조건이 발생했는지 나타내는 에러를 출력한다.

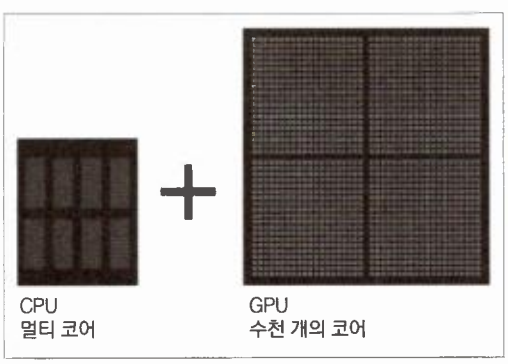
클래스 선언 후, Observable.from_iterable()을 사용해 10개의 각기 다른 값을 포함한 관찰 가능 속성[observable]을 생성한다. 마지막으로, 코드의 마지막 줄에서는 이전에 생성한 가짜 식별자에 temperatureObserver 클래스의 인스턴스를 등록한다.

GPU 프로그래밍

GPU는 고해상도 랜더링 및 비디오 게임에 사용되는 걸로 알려져 있다. GPU는 1초에 수십만 번의 연산을 고속으로 처리하면서 게임의 3D 모델을 구현하고, 60FPS[10]의 부드러운 화면을 보여준다.

일반적으로, GPU는 수백만 연산을 병렬로 하는 작업에서 매우 좋은 성능을 보인다. 그렇다면 왜 GPU의 성능이 이렇게 좋은데 CPU 대신 GPU를 사용하지 않는 것일까? GPU는 그래픽 처리에 좋은 성능을 보이는 반면, 운영체제 및 일반적인 연산을 수행하는 복잡한 사항을 다루기에는 부적합하게 디자인됐기 때문이다. CPU는 더 적은 코어로 이뤄져 있지만, 태스크task의 컨텍스트를 빠른 속도로 바꿀 수 있도록 디자인됐다. GPU에서 이와 동일한 작업을 한다면, 컴퓨터의 전반적인 성능이 하락할 것이다.

그러면 어떻게 고성능의 그래픽 카드를 그래픽 프로그래밍에 활용할 수 있을까? 바로 PyCUDA와 OpenCL, Theano 같은 라이브러리를 이용하는 것이다. 이 라이브러리는 GPU를 활용하기 위한 복잡한 로우 레벨 코드를 감추어 그래픽 API로 추상화한 것이다. GPU상의 수천 개의 코어를 간단히 구성해, 복잡한 연산을 수행하는 프로그램에도 활용한다.



10 프레임, 즉 화면이 바뀌는 속도를 초 단위로 나타내는 단위 – 옮긴이

이러한 **그래픽 프로세싱 유닛**GPU, Graphics Processing Unit은 스크립팅 언어scripting language 자체로 수행하기 힘든 부분을 할 수 있다. 다시 말해, 높은 병렬성을 보이며 최대의 성능을 끌어낸다. 파이썬을 활용해 두 마리 토끼를 모두 잡을 수 있는 것이다. 즉, 사용하기 쉬운 언어인 파이썬으로 성능이 높은 프로그램을 만들 수 있다.

GPU를 이용한 여러 라이브러리를 살펴보자.

PyCUDA

PyCUDA는 파이썬에서 엔비디아Nvidia의 CUDA 병렬 컴퓨팅 API를 사용하기 위한 라이브러리다. 동일한 CUDA API를 제공하는 여타 프레임워크와 달리 많은 이점이 있다. PyCUDA는 높은 내부 속도 및 CUDA 드라이버 API의 완벽한 제어가 가능하며, 많은 공식 지원 문서가 제공된다.

그러나 PyCUDA는 엔비디아 기반의 그래픽 카드만 이용해야 API 사용이 가능하다는 단점도 있다. 그렇지만 엔비디아 그래픽 카드가 아니라도 동일한 작업을 할 수 있는 대안이 있다.

OpenCL

OpenCL은 PyCUDA의 대안으로 나온 것인데, 엔비디아 그래픽 카드 외의 그래픽 카드를 사용해 동일한 구현이 가능하며 폭넓은 호환성 때문에 개인적으로 추천한다. OpenCL은 원래 애플사가 CPU, GPU, 디지털 신호 프로세서, FPGAfiled-programmble gate arrays[11] 등 프로세서 및 하드웨어 가속기와 같은 호환성을 장점으로 가져온 것이다.

현재 파이썬뿐만 아니라 자바와 닷넷.NET에도 서드파티third-party API가 있어, 컴퓨터의 처리 능력을 최대한 활용하고자 한다면 좋은 방법이다.

---

11 각종 게이트로 이뤄져 있어 프로그래밍이 가능한 IC 칩 – 옮긴이

Theano

Theano는 C 언어로 구현됐으며, 엄청난 양의 데이터를 연산하고자 할 때 GPU를 활용해 높은 처리 속도를 보이는 라이브러리다.

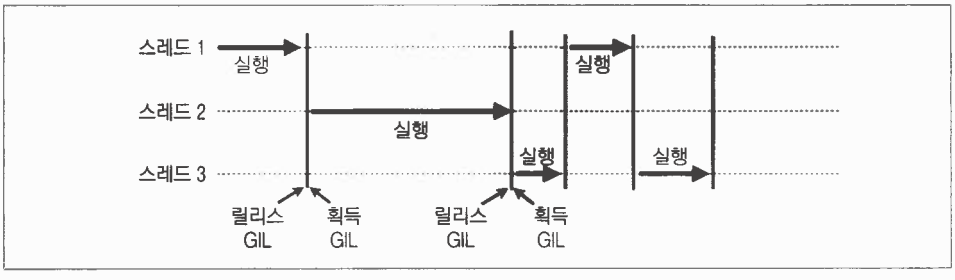
그렇지만 파이썬이 Theano가 작동되는 공간에서 중간 매개 역할을 한다는 것이 여타 프로그래밍 스타일과 다른 점이다.

Theano 공식 사이트(http://deeplearning.net/software/theano/)를 확인해보자.

파이썬의 한계

이 장의 도입부에서 파이썬에 있는 GIL Global Interpreter Lock의 한계에 대해 말했는데, 과연 무슨 뜻일까?

먼저, GIL이 하는 역할을 알아보자. GIL이란 병렬 파이썬 코드를 실행할 때 여러 스레드의 사용을 제한하는 상호 배제 락mutual exclusion lock[12]이다. 한 번에 1개의 스레드만 유지하는 락이며, 스레드를 자체 코드에서 실행하려면 자체 코드를 실행하기 전에 락을 먼저 점유해야 한다. 이로 인해 락이 되어 있는 동안에는 모든 실행이 불가능하다.



12 여러 개의 병렬 프로세스가 공통의 변수 및 자원에 액세스할 때 임의의 시점에서 하나의 프로세스만 액세스하도록 제어하는 것 – 옮긴이

앞의 그림에서, GIL에 의해 멀티스레드가 제한되는 모습을 볼 수 있다. 각 스레드는 다음 작업을 진행하기 전에 GIL을 기다리고 받아야 하며, 작업을 완료하기 전에 GIL을 해제해야 한다. 이는 무작위 라운드 로빈round-robin[13] 방식을 이용하며, 어떤 스레드가 락을 먼저 점유할지 알 수 없다.

이러한 부분이 왜 중요할까? GIL은 파이썬의 한 부분으로서 오랫동안 자리잡았는데, 그동안 그 유용성에 있어 많은 문제가 제기됐다. 하지만 이는 좋은 취지로 구현됐으며 스레드가 없는 파이썬 메모리 관리를 방지하고자 구현됐다. 특정 멀티프로세서 시스템 구성의 약점 또한 막을 수 있다.

파이썬을 개발한 귀도 반 로섬Guido van Rossum(1956~)은 GIL을 제거한 버전과 그 장점을 공개했다(http://www.artima.com/weblogs/viewpost.jsp?thread=214235 참조). 그는 GIL이 없는 파이썬을 만드는 데 반대하지 않을 것이라 했다. 하지만 단일 스레드 애플리케이션의 성능에 부정적인 영향을 미치지 않을 경우에 한해 코드 통합을 고려할 것이라고 했다.

GIL을 제거하려는 노력은 몇 번 있었지만, 안정적인 스레드를 보장하는 락의 추가가 2배 이상의 성능 감소로 이어졌다. 즉, 2개 이상의 CPU보다 1개의 CPU를 통한 작업이 더 불리하다는 것이다. 그러나 GIL을 사용하지 않는 NumPy 같은 라이브러리가 있는데, GIL을 완전히 배제하고 작업하는 것은 이후에 좀 더 자세히 살펴본다.

Jython, IronPython 같은 형태의 파이썬이 있다는 사실도 중요한데, 모두 GIL 형태가 없으며 멀티프로세서 시스템에 적합하다. Jython과 IronPython 모두 각기 다른 가상 머신에서 동작해, 개별적인 런타임 환경을 이용할 수 있다.

Jython

Jython은 자바 플랫폼에서 작동 가능한 파이썬 형태다. 스크립트를 자바 언어로 사용 가능하며, 많은 데이터 세트를 다룰 때 파이썬의 일반적인 구현 형태인 CPython의 성능을

13 스케줄링의 한 방법으로, 다중 처리에서 사이클릭(cyclic) 방식으로 작업을 처리한다. – 옮긴이

능가한다. 그렇지만 대부분의 경우 CPython의 단일 코어 및 멀티 코어 성능이 Jython을 능가한다.

Jython은 자바 라이브러리 및 프레임워크를 불러와 작업하는 등, 파이썬 코드에 부분적으로 자바가 필요할 때 유용하다.

IronPython

IronPython은 Jython의 닷넷.NET 형태로, 마이크로소프트 닷넷 프레임워크의 상위에서 동작한다. 다시 말해, 닷넷 애플리케이션을 보완하는 방식으로 사용 가능하다. 많은 닷넷 개발자는 닷넷 애플리케이션을 빠르고 쉬운 파이썬 코드로 작성할 수 있다는 점에 매료되어 자주 사용한다.

왜 파이썬이어야 하는가?

보다시피 파이썬이 동시성 프로그램을 만드는 데 있어 다소 제한이 보이는데, 그런데도 왜 사용할까? 짧게 말하자면, 최고의 작업 능력을 보일 뿐만 아니라 복잡한 컴퓨터 연산 작업을 신속하게 처리하는 데 있어 더할 나위 없이 좋은 언어이기 때문이다. 프로그래밍 경험이 없는 많은 이들도 쉽게 이해 가능하고 공부할 수 있는 직관적인 언어이기 때문이기도 하다.

파이썬 언어는 머신 러닝 및 정량 분석 분야 같은 흥미로운 분야에서 일하는 데이터 사이언티스트 및 수학자가 많이 사용하고 있다.

파이썬 2, 3 환경 모두 이러한 상황에 알맞은 수많은 라이브러리를 제공하고 있으며, 파이썬의 한계를 뛰어넘어, 프로그램이 필요하는 사항을 정확히 수행하는 효율적인 소프트웨어를 제작할 수 있다.

지금까지 파이썬의 한계와 함께 스레드와 프로세스가 무엇인지 살펴봤으며, 이제는 프로그램의 속도를 높이기 위해 멀티스레딩을 활용하는 방법을 살펴볼 것이다.

동시에 그림 다운로드하기

여러 이미지 및 파일을 스레드를 사용해 다운로드하는 멀티스레딩 예제를 만들어본다. 이는 입출력 속성을 차단하므로 훌륭한 멀티스레딩 예제 중 하나다.

성능 향상을 강조하고자 링크에 접속할 때마다 새로운 이미지가 보이는 무료 API http://lorempixel.com/400/200/sports/에서 10개의 그림을 다운로드한다. 그런 다음, 각 그림을 임시 폴더에 저장해서 나중에 사용한다.

이 예제에 사용된 전체 코드는 깃허브GitHub 저장소 https://github.com/elliotforbes/Concurrency-With-Python에서 볼 수 있다.

순차적으로 다운로드하기

먼저, 성능 개선을 측정할 기본 구성을 만들어보자. 그 전에 다음과 같이 10개의 그림을 차례대로 다운로드하는 프로그램을 작성해본다.

```
import urllib.request
def downloadImage(imagePath, fileName):
  print("Downloading Image from ", imagePath)
  urllib.request.urlretrieve(imagePath, fileName)
def main():
  for i in range(10):
    imageName = "temp/image-" + str(i) + ".jpg"
    downloadImage("http://lorempixel.com/400/200/sports", imageName)
if __name__ == '__main__':
  main()
```

코드에 관한 설명

위 코드에서는 `urllib.request`를 불러오는 것부터 시작한다. 이는 해당 그림을 다운로드하는 HTTP 요청을 수행하기 위한 모듈이다. 그 후, `imagePath`와 `fileName` 파라미터

를 가진 downloadImage 함수를 정의한다. imagePath는 다운로드 이미지 경로 URL을 나타낸다. fileName은 로컬에 파일을 저장할 때 지정할 파일 이름이다.

main 함수에서는 for 반복문이 시작된다. for 반복문에서는 temp/ 디렉토리상의 imageName이 현재 반복자가 str(i)로 지정되고, .jpg 파일 확장자가 붙여진다. 다음으로 downloadImage 함수가 호출되어, 새로 생성된 imageName으로 무작위 그림을 제공하는 lorempixel 사이트를 거친다.

코드가 실행되면, temp 디렉토리에 차례대로 10개의 개별 그림이 채워지는 모습을 볼 수 있다.

동시에 다운로드하기

이제 기본적인 구성을 살펴봤으니, 그림을 한꺼번에 동시에 다운로드하는 프로그램을 작성해보자. 다음 장에서 스레드를 구성하는 부분을 배워볼 테니, 코드가 어려워도 걱정하지 말자. 여기서 배워야 할 부분은 동시성 형태로 프로그램을 구성해보니 성능 개선이 있었음을 알아차리는 것이다.

```
import threading
import urllib.request
import time
def downloadImage(imagePath, fileName):
  print("Downloading Image from ", imagePath)
  urllib.request.urlretrieve(imagePath, fileName)
  print("Completed Download")
def executeThread(i):
  imageName = "temp/image-" + str(i) + ".jpg"
  downloadImage("http://lorempixel.com/400/200/sports", imageName)
def main():
  t0 = time.time()
  # 모든 스레드 형태를 저장하는 배열을 생성한다.
  threads = []
  # 10개의 스레드를 생성하고, 배열에 추가하며, 실행해본다.
```

```
  for i in range(10):
    thread = threading.Thread(target=executeThread, args=(i,))
    threads.append(thread)
    thread.start()
  # 배열 내의 모든 스레드는 전체 종료 시간을 기록하기 전
  # 모든 실행을 마친다.
  for i in threads:
    i.join()
  # 전체 실행 시간을 계산한다.
  t1 = time.time()
  totalTime = t1 - t0
  print("Total Execution Time {}".format(totalTime))
if __name__ == '__main__':
  main()
```

코드에 관한 설명

기존의 프로그램에서 새로 수정된 부분은 스레딩 모듈이 추가된 것으로, 멀티스레드 애플리케이션을 구현해주는 모듈이다. 그 후, 경로명을 확장해 파일 이름 변수를 정의하고, executeThread 함수 내에서 downloadImage 함수를 호출한다.

먼저 main 함수 내에서는 빈 배열의 스레드를 생성하고, 스레드 배열에 추가하는 방식으로 새로운 스레드 객체를 생성하고, 스레드를 시작해보자.

마지막으로, 스레드 배열 i를 호출해 배열을 반복하며 각 스레드 안의 join 메소드를 호출한다. 이는 모든 스레드가 그림 다운로드가 완료되기 전까지 나머지 코드의 실행이 정지됨을 보장한다.

컴퓨터에서 실행해보면, 순간적으로 10개의 각기 다른 그림이 다운로드된다. 다운로드가 완료되면, 메시지와 함께 해당 temp 폴더가 그림으로 채워진 모습을 볼 수 있다.

위 코드 모두 동일한 urllib.request 라이브러리를 이용해 동일한 작업을 진행했는데, 전체 실행 시간을 살펴보면 동시에 그림 10개를 내려받는 코드에서 10배 정도 실행 시간이 단축됐다.

멀티프로세싱으로 소인수 찾기

자, 지금까지 그림 다운로드 부분에 있어 어떻게 성능을 높이는지 알아봤는데, 숫자 처리에 있어서 성능 향상은 어떻게 이뤄질까? 바로 적절하게 멀티프로세싱을 이용하는 것이다.

이 예제에서는 20,000에서 100,000,000 사이에서 10,000개의 무작위 수를 골라 소인수를 찾아본다. 작업이 완료되기만 하면 실행 순서에 따로 신경 쓸 필요가 없으며, 프로세스 간 메모리 공유도 없다.

순차적으로 소인수 구하기

한 번 더, 순차 방식으로 작동하는 코드를 작성해 잘 동작되는지 실행해보자.

```
import time
import random
def calculatePrimeFactors(n):
  primfac = []
  d = 2
  while d*d <= n:
    while (n % d) == 0:
      primfac.append(d) # 인수를 반복적으로 구한다고 해보자.
      n //= d
    d += 1
  if n > 1:
    primfac.append(n)
  return primfac
def main():
  print("Starting number crunching")
  t0 = time.time()
  for i in range(10000):
    rand = random.randint(20000, 100000000)
    print(calculatePrimeFactors(rand))
  t1 = time.time()
```

```
  totalTime = t1 - t0
  print("Execution Time: {}".format(totalTime))
if __name__ == '__main__':
  main()
```

코드에 관한 설명

위에서 두 번째 줄까지는 필요한 시간 및 무작위 관련 모듈을 불러오는 부분이다. 그 후, n을 입력으로 하는 calculatePrimeFactors 함수를 정의한다. 이 함수는 주어진 수의 모든 소인수를 배열에 추가해 결괏값이 반환된다.

다음으로 main 함수를 정의해, 시작 시간 및 10,000개의 수를 무작위 randint를 이용해 생성한다. 생성된 수를 calculatePrimeFactors 함수에 전달하고, 결과를 출력해보자. 마지막으로, for 반복문의 종료 시간을 계산하고 출력해본다.

각자 컴퓨터에서 실행해보면 소인수 배열이 각기 다른 무작위 수 10,000개로 출력되며, 전체 실행 시간 또한 마찬가지로 볼 수 있다. 내 맥북에서는 약 3.6초의 실행 시간이 소요됐다.

동시에 소인수 구하기

이제는 멀티프로세스를 사용해 프로그램의 성능을 높여보자.

작업을 진행하고자, 소인수 분해할 10,000개의 무작위 수를 생성하는 대신 executeProc 함수를 정의해 1,000개의 무작위 수를 생성해보자. 10개의 프로세스를 생성해 각 함수를 10번 실행하는데, 전체 계산하는 횟수는 순차 방식으로 할 때와 정확히 일치한다.

```
import time
import random
from multiprocessing import Process
# 주어진 'n'에 관한 모든 소인수를 구한다.
```

```
def calculatePrimeFactors(n):
  primfac = []
  d = 2
  while d*d <= n:
    while (n % d) == 0:
      primfac.append(d) # 인수를 반복적으로 구한다고 해보자.
      n //= d
    d += 1
  if n > 1:
    primfac.append(n)
  return primfac
# 10,000번의 계산 작업을
# 1,000번씩 10개로 쪼개어 처리한다.
def executeProc():
  for i in range(1000):
    rand = random.randint(20000, 100000000)
    print(calculatePrimeFactors(rand))
def main():
  print("Starting number crunching")
  t0 = time.time()
  procs = []
  # 프로세스를 생성하고 실행한다.
  for i in range(10):
    proc = Process(target=executeProc, args=())
    procs.append(proc)
    proc.start()
  # 모든 프로세스가 종료할 때까지 대기하고자,
  # 다시 한번, .join() 메소드를 사용한다.
  for proc in procs:
    proc.join()
  t1 = time.time()
  totalTime = t1 - t0
  # 10번의 procs에 소요된 시간을 출력한다.
  print("Execution Time: {}".format(totalTime))
if __name__ == '__main__':
  main()
```

코드에 관한 설명

이번 코드는 이전 코드와 동일한 기능을 한다. 하지만 세 번째 줄을 살펴보자. 여기서 멀티프로세싱 모듈로부터 프로세스를 불러왔다. 다음의 calculatePrimeFactors 메소드는 그대로다.

다음으로 10,000번의 반복을 수행하는 for 문을 지운다. 그런 다음 executeProc 함수를 대신 넣고, for 루프의 횟수를 1,000으로 한다.

main 함수 내에서 procs라는 빈 배열을 생성한다. 10개의 개별 프로세스를 생성한 후, executeProc 함수 반환값을 받게 하고 args는 비운다. 새로 생성된 프로세스를 procs 배열에 추가하고, proc.start() 하는 프로세스로 시작해본다.

10개의 개별 프로세스를 생성한 후, 현재 procs 배열에 있는 프로세스를 따라 반복한 다음 조인join한다. 이는 모든 프로세스가 전체 실행 시간을 계산하기 전에 끝내기 위해서다.

실행해보면 10,000개의 출력이 콘솔 창에 보이며, 순차적인 방식과 비교해 실행 시간이 대폭 적어졌음을 볼 수 있다. 참고로 내 컴퓨터에서 순차적 프로그램은 3.9초가 소요됐으나, 멀티프로세싱 버전에서는 1.9초가 걸렸다.

지금까지 애플리케이션상에서 멀티프로세싱을 어떻게 구현하는지 간단하게 알아봤다. 이후에는 어떻게 풀pool을 생성하고 이그제큐터executor를 활용하는지 살펴볼 것이다. 이후의 핵심은 멀티 코어를 활용해 CPU 기반 작업의 성능을 높이는 것이다.

요약

동시성 프로그래밍에 관한 기본적인 이해 과정을 거쳤다. 스레드와 프로세스에 대해 알고, 동시성 애플리케이션을 구현할 때 파이썬을 이용하는 데 있어 한계와 어려움도 이해해야 한다. 마지막으로, 애플리케이션에 각기 다른 형태의 동시성을 구현함으로써 얻는 성능 개선을 직접 배워봤다.

모든 애플리케이션에 동시성을 적용하고 지속적으로 성능을 향상할 수 있는 왕도는 없다. 특정 방식의 동시성 프로그래밍이 때에 따라서 다른 어떤 방식보다 더 훌륭할 수도 있다. 따라서 이후의 장들에서는 여러 메커니즘과 이를 언제 활용할지 살펴본다.

2장에서는 동시성과 병렬화 개념을 더 깊게 알아보며, 이 둘의 차이도 비교해본다. 동시성 시스템에 방해가 되는 요소를 찾아보고, 여러 가지 형태의 컴퓨터 시스템 아키텍처도 배우며, 성능 향상을 어떻게 이루는지 살펴본다.

02

병렬화

동시성과 병렬화 개념은 서로 혼동되기 마련이다. 하지만 실제로는 구분되니, 소프트웨어의 병렬 실행 대신 동시성을 가진 디자인을 하고자 한다면 소프트웨어의 실제 특성에 주의를 기울여야 한다.

이로 인해, 이 두 개념을 정확하게 알고 그 차이점을 이해해야 한다. 이 둘의 차이점을 이해함으로써 파이썬으로 고성능 소프트웨어를 설계하는 데 있어 많은 도움이 된다.

2장에서 다루는 내용은 다음과 같다.

- 동시성의 정의 및 애플리케이션에 미치는 영향
- 병렬화의 정의 및 동시성과의 차이점
- 여러 형태의 컴퓨터 시스템 아키텍처 및 동시성과 병렬화 효과적으로 이용하기
- 컴퓨터 메모리 구조

동시성에 대한 이해

근본적으로 동시성이란 동시에 여러 작업을 하는 것을 뜻하지만, 병렬화와는 구분된다. 이는 애플리케이션의 성능과 실행 속도 또한 높여준다.

동시성을 가장 잘 이해하려면 한 사람이 여러 작업을 하고 있고 그 작업을 빠르게 바꾸며 진행하는 모습을 상상해보면 된다. 프로그램을 동시에 작업하고 있다고 생각해보고, 동시에 지원 요청을 하고 있다고 해보자. 그는 프로그램을 작성하는 데 집중할 것이며, 버그를 수정하고 지원 문제를 다루고자 빠르게 컨텍스트를 전환할 것이다. 지원 작업이 완료되면, 한 번 더 컨텍스트 스위치가 일어나고, 빠르게 프로그램 작성으로 넘어간다.

그러나 컴퓨팅 성능에 있어 조심해야 하고 프로그램 작성 시 참고해야 할 부분이 두 가지 있다. 이 두 가지의 차이점을 알아야 하는데, CPU 기반의 문제에 동시성을 적용하면 프로그램 성능이 감소하는 모습을 볼 수 있을 것이다. 동시성이 필요한 작업에 병렬화를 적용할 때도 마찬가지로 성능 감소 현상을 볼 수 있다.

동시성 시스템의 특징

수많은 동시성 시스템에서 다음과 같은 특징을 볼 수 있다.

- 다양한 구성: 여러 프로세서와 스레드가 각자의 작업에 모두 임하는 것을 뜻한다. 동시에 작동되는 다수의 스레드가 있는 프로세스를 여러 개 갖고 있다.
- 자원 공유: 메모리, 디스크 등의 자원 구성이 프로그램의 실행에 활용돼야 한다.
- 규칙: 모든 동시성 시스템이 락을 획득하고, 메모리에 접근하고, 상태를 변경하는 등의 조건을 따라야 한다. 이러한 조건은 동시성 시스템의 작동에 매우 중요하고, 어길 경우 프로그램에 치명적일 수 있다.

I/O 문제

I/O 문제란 간단히 말해 입력 및 출력 정보를 처리하는 데 많은 시간이 소모되는 장애를 뜻한다.

일반적으로, I/O를 다량으로 사용하는 애플리케이션에서 볼 수 있다. 사용자의 기본 웹 브라우저를 많은 I/O를 사용하는 애플리케이션으로 만들 수 있다. 보통 화면에 렌더링이 되는 것과는 대조적으로, 브라우저상에서 스타일 시트, 스크립트, 로드될 HTML 페이지를 마치기 위한 네트워크 요청에 많은 시간이 소비된다.

요청되는 데이터양이 처리되는 데이터양보다 느리다면, I/O 문제가 발생한다.

이러한 애플리케이션의 속도를 높이기 위해서는 I/O 자체의 속도를 높이거나, 더 빠른 하드웨어를 이용해 이러한 I/O 요청을 다뤄야 한다.

I/O 문제에 해당하는 좋은 예제는 바로 웹 크롤러web crawler다. 웹 크롤러는 웹을 탐색하면서, 웹 페이지의 인덱싱으로 구글 엔진 등에서 키워드에 해당하는 상위 10개의 결과를 보여주는 탐색 순위 알고리즘에 사용된다.

이제 다음과 같이 페이지를 요청하고 소요 시간을 측정하는 예제를 작성해보자.

```
import urllib.request
import time

t0 = time.time()
req = urllib.request.urlopen('http://www.example.com')
pageHtml = req.read()
t1 = time.time()
print("Total Time To Fetch Page: {} Seconds".format(t1-t0))
```

코드 오류가 난다면, 먼저 urllib.request와 time 모듈을 불러온다. 그 후 시작 시간을 기록하고 웹 페이지를 요청하며, 종료 시간을 기록하고, 시간차를 출력한다.

이제 코드에 살을 보태고, 다른 페이지에 링크를 걸어 인덱스가 가능하게 한다. 다음과 같이 BeautifulSoup 라이브러리를 사용해 쉽게 해본다.

```
import urllib.request
import time
from bs4 import BeautifulSoup

t0 = time.time()
req = urllib.request.urlopen('http://www.example.com')
t1 = time.time()
print("Total Time To Fetch Page: {} Seconds".format(t1-t0))
soup = BeautifulSoup(req.read(), "html.parser")

for link in soup.find_all('a'):
  print(link.get('href'))

t2 = time.time()
print("Total Execeution Time: {} Seconds".format(t2-t1))
```

프로그램을 실행해보면, 터미널에서 다음과 같은 결과가 나온다.

```
Total Time To Fetch Page: 0.318880558013916 Seconds
http://www.iana.org/domains/example
Total Execeution Time: 0.005640983581542969 Seconds
```

위 결과를 살펴보면, 페이지를 추출하는 데 약 0.2초가 소요됐음을 볼 수 있다. 그렇다면 여기서 수백만 개의 웹 페이지를 웹 크롤러로 실행한다면, 대략 현재 소요된 시간의 수백만 배의 시간이 걸릴 것이다.

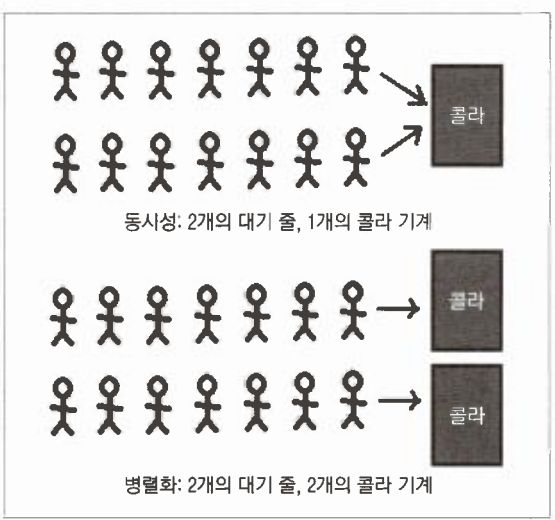
이러한 문제의 근본적인 원인은 프로그램에서 보이는 I/O 문제다. 네트워크 요청과 나머지 크롤링을 위한 페이지 파싱parsing의 많은 시간을 대기 상태로 있다.

병렬화 이해하기

1장에서 파이썬 멀티프로세싱 능력과 하드웨어에서 많은 프로세싱 코어가 갖는 장점을 살리는 방법을 다뤘다. 하지만 프로그램이 병렬적으로 동작된다는 것은 무슨 뜻일까?

여러 작업을 동시에 처리하는 구성을 취하는 동시성과 달리, 병렬화parallelism란 여러 작업을 동시에 실행하고 계산한다는 뜻이다. 병렬화를 제대로 알려면 이 둘을 구분해야 하는데, 동시에 코드를 실행할 멀티프로세서가 필요할 것이다.

이해를 돕고자 콜라를 먹기 위해 사람들이 차례대로 대기하고 있다고 생각해보자. 20명씩 2줄의 사람들이 차례대로 한 대의 콜라 기계에 대기한다면, 이것은 동시성 문제다. 이제 이 사람들에게 1대의 콜라 기계를 더 설치해 사용하게 한다면 병렬적인 발생이 일어났다고 본다. 여기서 각 콜라 기계는 프로세싱 코어를 나타내며, 작업을 동시에 처리하는 병렬 처리 형태를 잘 보여준다.



▲ 출처: https://github.com/montanaflynn/programming-articles/blob/master/articles/parallelism-and-concurrency-need-different-tools.md

병렬화의 효과를 가장 잘 설명하는 실질적인 예시는 바로 컴퓨터 그래픽 카드다. 그래픽 카드는 보통 독립적으로 동시에 계산하는 프로세싱 코어 개수가 수천 개다. 고사양 PC 게임이 부드럽게 동작될 수 있는 이유는 이러한 그래픽 카드에 수많은 병렬 코어가 있기 때문이다.

CPU 제약 문제

일반적으로, CPU 제약 문제는 I/O 제약 문제와 그 형태가 완전 반대다. 이는 많은 숫자를 처리하고, 복잡한 계산을 수행하는 애플리케이션에서 자주 볼 수 있다. 이러한 프로그램은 실행 비율이 CPU 속도에 제약되어 있어, 고속의 CPU라면 프로그램 속도가 바로 향상되는 것을 볼 수 있다.

처리하는 데이터가 요청되고 있는 데이터보다 많다면, CPU 제약 문제가 발생한다.

1장에서 10,000개의 무작위 수 소인수 분해를 할 때 CPU 제약 프로그램을 다루는 방법을 살펴봤는데, CPU에 과도한 연산량이 치중됐다. 그 후 CPU를 더 활용할 수 있는 방법으로 소인수 분해 프로그램을 동일하게 구현했는데, 프로그램 실행에서 바로 속도가 향상됐다.

CPU상에서 어떻게 작동될까?

동시성과 병렬화의 차이점을 이해하는 것도 중요하지만, 소프트웨어가 동작하는 시스템을 이해하는 일도 중요하다. 다양한 아키텍처 스타일과 로우 레벨 메커니즘을 살펴보면 소프트웨어 디자인을 결정하는 데 많은 도움이 된다.

단일 코어 CPU

단일 코어 프로세서는 주어진 시간에 활용할 수 있는 스레드가 1개다. 하지만 애플리케이션이 중지되거나 응답하지 않는 상황에서 프로세서는 초당 수천 번의 실행을 여러 스레드 간에 스위칭해야 한다. 이런 스레드 간 스위칭을 '컨텍스트 스위치context switch'라고 하며, 특정 시간에 스레드 간에 필요한 정보를 저장하고 나중에 그 밖의 부분에 한 번 더 저장한다.

일정하게 스레드에 저장하고 불러오는 메커니즘을 통해 주어진 시간에 여러 스레드에서 처리가 가능하고, 한 번에 많은 작업을 할 수 있다. 실제로 주어진 시간에 한 작업만 수행하지만, 그 속도를 인간이 인식하기에는 너무 빠르다.

파이썬 기반 멀티스레드 애플리케이션은 컨텍스트 스위치가 복잡한 계산 형태다. 안타깝게도 이를 피할 방법이 없는데, 오늘날 수많은 운영체제 디자인이 이러한 컨텍스트 스위치 형태로 최적화되고 있어 큰 불편을 느끼진 않는다.

단일 코어 CPU의 장점은 다음과 같다.

- 여러 코어들 간의 복잡한 통신 프로토콜이 필요 없다.
- 적은 전력 소모를 보여, 사물인터넷IoT 제품에 적합하다.

하지만 단점도 존재한다.

- 처리 속도에 한계가 있어, 무거운 애플리케이션의 경우 느리고 동작이 멈추기도 한다.
- 단일 코어 CPU의 작동 제한 속도에 따라 방열 문제가 발생한다.

클록 속도

컴퓨터에서 동작하는 단일 코어 애플리케이션의 가장 큰 제한은 CPU 클록clock 속도다. 클록 속도를 얘기할 때는 초당 몇 번의 클록 사이클clock cycle이 일어나는지 알아봐야 한다.

지난 10년 동안, 많은 제조사가 무어의 법칙Moore's law(트랜지스터 하나에 집적되는 실리콘이 2년마다 두 배씩 늘어난다는 법칙)을 능가하고자 했다.

2년마다 트랜지스터가 2배가 된다는 것은 단일 CPU 클록 속도를 급격히 증가시켰으며, 수 MHz CPU에서 인텔의 i7 6700k 프로세서와 같이 4~5GHz의 속도를 갖게 됐다.

그렇지만 트랜지스터가 나노미터 단위로 작아짐에 따라 기술적으로 더 이상 작아질 수 없게 됐고, 양자 터널 효과effects of quantum tunneling[1]가 필요해졌다. 이러한 기술적 한계로, 높은 연산 속도를 높이고자 그 밖의 방법을 찾게 됐다.

여기서 마르텔리 범용성 모델Martelli's Model of Scalability이 등장했다.

마르텔리 범용성 모델

알렉스 마르텔리Alex Martelli(1955~)의 저서인 『Python Cookbook』(O'Reilly Media, 2005)에서 범용성 모델이 등장했는데, 레이먼드 헤팅거Raymond Hettinger[2]는 '동시성 알아보기'라는 주제로 PyCon Russia 2016에서 강연을 했다. 이 모델은 코어 형태에 따라 프로그램 및 문제를 세 가지로 나눴다.

- 1 코어: 단일 스레드 및 단일 프로세스 프로그램
- 2~8 코어: 멀티스레드 및 멀티프로세싱 프로그램
- 9+ 코어: 분산 컴퓨팅

첫 번째 부분인 단일 코어, 단일 스레드 카테고리는 단일 코어 CPU의 일정한 속도 증가로 인해 발생하는 문제를 다룰 수 있고, 그 결과 두 번째 카테고리는 더욱 사용 빈도가 줄어든다. 결국, 2~8 코어 시스템의 동작 속도는 한계에 다다를 것이며, 다량의 CPU 시스템 및 분산 컴퓨팅 같은 방법을 고려해봐야 한다.

---

1 DNA 굵기의 15분의 1 정도인 전자가 터널을 오가며 정보를 저장하게 된다. 이는 양자 컴퓨터의 기초 기술이다. – 옮긴이

2 프리랜서 파이썬 개발자다. https://github.com/rhettinger – 옮긴이

주어진 연산을 빨리 하려면, 많은 전력이 필요하며 분산 컴퓨팅을 고려해봐야 하고 병렬적인 방법으로 문제를 해결하기 위해 많은 컴퓨터와 프로그램 인스턴스가 작동돼야 한다. 수백만의 요청을 다루는 거대한 기업용 시스템의 경우 이 카테고리의 대표 주자다. 일반적으로, 여러 곳에 위치한 수십, 수백 개의 고성능 서버 같은 기업용 시스템에서 찾아볼 수 있다.

시분할(작업 스케줄러)

운영체제의 가장 중요한 부분 중 하나인 작업 스케줄러task scheduler를 알아보자. 이는 오케스트라의 지휘자처럼 모든 작업을 매우 정확한 시간과 규칙에 알맞게 지시한다. 모든 작업이 완료될 때까지 실행돼야 하는 규칙도 있지만, 언제 어디에서 작업이 실행되는지는 결정하지 않는다. 다시 말해, 첫 번째로 실행되는 프로그램이 먼저 끝나지 않을 수도 있다는 말이다. 이러한 비결정적 형태는 동시성 프로그램 작성을 어렵게 만든다.

다음 코드에서 비결정적인 요소를 알아보자.

```
import threading
import time
import random

counter = 1

def workerA():
  global counter
  while counter < 1000:
    counter += 1
    print("Worker A is incrementing counter to {}".format(counter))
    sleepTime = random.randint(0,1)
    time.sleep(sleepTime)

def workerB():
  global counter
```

```
  while counter > -1000:
    counter -= 1
    print("Worker B is decrementing counter to {}".format(counter))
    sleepTime = random.randint(0,1)
    time.sleep(sleepTime)

def main():
  t0 = time.time()
  thread1 = threading.Thread(target=workerA)
  thread2 = threading.Thread(target=workerB)

  thread1.start()
  thread2.start()

  thread1.join()
  thread2.join()

  t1 = time.time()

  print("Execution Time {}".format(t1-t0))

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

위 코드에서는 counter 변수가 1,000까지 증가하고 −1,000까지 감소하는 함수로 이뤄진 2개의 파이썬 스레드를 볼 수 있다. 단일 코어 프로세서에서 작업 B가 실행되기 전에 작업 A가 완료될 가능성이 보이는데, 이는 작업 B 입장에서도 마찬가지다. 그러나 작업 스케줄러가 작업 A와 작업 B 사이를 계속적으로 무한히 반복하며 끝내지 않을 가능성도 존재한다.

더군다나, 위 코드에서 동기화 없이 공유 자원에 접근하려는 멀티스레드 문제도 보인다. 카운터에 어떤 값이 들어갈지 정확한 방법도 없어, 프로그램의 신뢰도는 낮아 보인다.

멀티 코어 프로세서

지금까지 단일 코어 프로세서의 작동 방식을 살펴봤는데, 멀티 코어 프로세서도 알아보자. 멀티 코어 프로세서는 여러 개의 독립적인 유닛, 즉 '코어'가 있다. 각 코어에는 저장된 인스트럭션을 처리하기 위한 요소가 있다. 이러한 코어는 다음과 같은 프로세스 형태의 사이클을 따른다.

- 인출 단계fetch : 프로그램 메모리에서 인스트럭션을 인출하는 단계다. 프로그램 카운터PC, program counter 로부터 명령받아, 다음 실행 위치를 알아본다.
- 해독 단계decode : 코어는 인출된 인스트럭션을 CPU의 여러 곳에서 작동하는 신호 형태로 변환한다.
- 실행 단계execute : 마지막으로, 실행 단계를 수행한다. 인출되고 해독된 인스트럭션을 실행하고, 실행 결과는 CPU 레지스터에 저장된다.

멀티 코어는 '인출 → 해독 → 실행' 사이클을 독립적으로 각각 수행함으로써 많은 장점을 갖는다. 이러한 아키텍처는 병렬 실행을 수행하는 고성능 프로그램에 적용된다.

멀티 코어 프로세서의 장점을 살펴보자.

- 단일 코어가 지닌 성능 한계가 멀티 코어에서는 없다.
- 우수하게 디자인된 애플리케이션을 멀티 코어상에서 빠른 속도로 실행할 수 있다.

그러나 단점도 존재한다.

- 일반적인 단일 코어 프로세서에 비해 많은 전력이 소모된다.
- 크로스 코어 통신cross-core communication 외에 많은 방법이 있으니, 이 장 후반부에서 자세히 살펴보자.

시스템 아키텍처 스타일

프로그램을 디자인할 때, 다양한 사용 형태 범위에 따라 적합한 메모리 아키텍처 스타일이 있다. 어떤 메모리 아키텍처는 병렬 컴퓨팅 및 계산에, 어떤 것은 일반적인 가정용 컴퓨터에 적합할 수 있다.

이때 다음과 같이 1972년 마이클 플린Michael Flynn(1934~)이 제안한 분류체계를 따른다. 이 분류체계는 컴퓨터 아키텍처를 다음 네 가지로 정의하고 있다.

- SISD: 단일 인스트럭션 스트림, 단일 데이터 스트림
- SIMD: 단일 인스트럭션 스트림, 복수 데이터 스트림
- MISD: 복수 인스트럭션 스트림, 단일 데이터 스트림
- MIMD: 복수 인스트럭션 스트림, 복수 인스트럭션 스트림

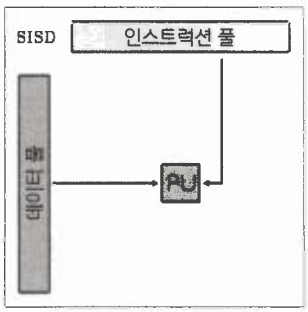
각 아키텍처에 대해 자세히 알아보자.

SISD

단일 인스트럭션 스트림, 단일 데이터 스트림은 단일 프로세서 시스템에서 주로 사용된다. 이러한 시스템은 하나의 데이터 스트림 입력과 이를 실행할 하나의 단일 프로세싱 유닛으로 구성된다.

일반적으로, 이러한 아키텍처는 폰 노이만Von Neumann(1903~1957) 기계에서 나타나며, 멀티 코어 프로세서가 일반화되기 전, 가정용 컴퓨터에 적용됐다. 모든 작업은 단일 코어 프로세서 형태로 처리된다. 그러나 인스트럭션 병렬화 및 데이터 병렬화 같은 작업 수행이 안 되고, 시스템에 많은 부하를 주는 그래픽 프로세싱 또한 불가능하다.

단일 프로세서 시스템을 나타내는 다음 그림을 살펴보자. 단일 프로세서 유닛으로 처리되는 1개의 데이터 소스를 볼 수 있다.



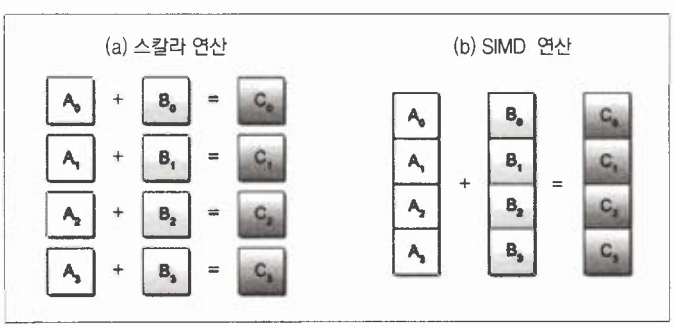
▲ 출처: wikipedia.org

해당 아키텍처의 장단점은 단일 코어 프로세서 부분에서 다뤘으니 참고하자.

인텔의 펜티엄 4가 단일 프로세서에 해당되겠다.

SIMD

SIMD[single instruction stream, multiple data streams] 아키텍처에서 복수 데이터 스트림 형태는 많은 멀티미디어를 처리하기에 알맞은 방식이다. 벡터를 다루기에, 3D 그래픽 작업 같은 부분에 이용된다. 예컨대, `[10, 15, 20, 25]`와 `[20, 15, 10, 5]`라는 배열 2개가 있다고 하자. SIMD 아키텍처에서 한 번의 연산으로 벡터들을 더해 `[30, 30, 30, 30]`을 얻을 수 있다. 스칼라 아키텍처에서 이뤄진다면, 네 번의 덧셈 연산을 다음과 같이 수행해야 한다.

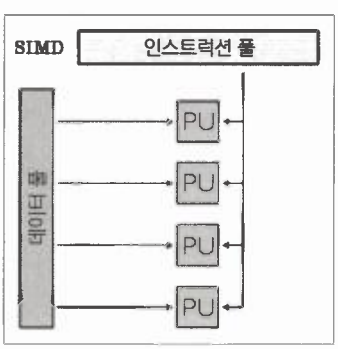


▲ 출처: wikipedia.org

이러한 아키텍처 방식은 그래픽 프로세싱 장치에서 찾아볼 수 있다. OpenGL 그래픽 프로그래밍에서 VAO[Vertex Array Object][3]라는 객체를 갖는데, 이는 일반적으로 게임상의 3D 객체를 표현하는 여러 개의 버텍스 버퍼 객체[Vertex Buffer Object][4]를 갖고 있다. 따라서 게임상의 캐릭터가 화면상에서 부드럽게 움직이기 위해 버텍스 버퍼 객체의 모든 엘리먼트는 매우 빠르게 연산이 수행된다.

SIMD 아키텍처의 장점을 살펴보자. 각 VAO 내부의 모든 엘리먼트를 지나면서 채워지면, 회전 행렬[rotation matrix]을 이용해 곱셈을 하기 위한 준비를 한다. 그 후, 벡터가 아닌 아키텍처보다 효율적으로 모든 엘리먼트에 동일한 작업을 빠르게 진행한다.

다음 그림은 SIMD 아키텍처의 전체적인 도식을 나타낸다. 여러 개의 벡터를 나타내는 데이터 스트림이 있고, 주어진 시간에 하나의 명령만 수행하는 프로세싱 유닛도 여러 개 존재한다. 그래픽 카드의 경우 이러한 프로세싱 유닛이 수백 개 정도 된다.



▲ 출처: wikipedia.org

SIMD의 장점을 살펴보자.

- 하나의 명령을 통해 동일한 연산을 다수의 엘리먼트에서 수행할 수 있다.
- 최근 그래픽 카드상의 코어 수가 증가하면서 처리량 또한 증가하고 있다.

---

3 OpenGL에서 제공하는 객체로, 버텍스 정보를 클라이언트(CPU에서 처리되는 부분)에서 서버(GPU)로 필요한 상태 정보를 전달한다. – 옮긴이

4 각 버텍스의 속성은 개별 VBO에 저장하고 이는 곧 하나의 VAO로 묶을 수 있다. – 옮긴이

SIMD 아키텍처를 이용하는 자세한 내용은 11장에서 다룬다.

MISD

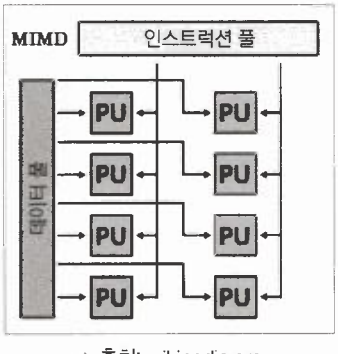
복수 인스트럭션 스트림, 단일 데이터 스트림인 MISD는 현재 출시된 관련 제품이 없어서, 잘 알려진 아키텍처는 아니다. MISD 아키텍처 스타일이 문제 수행에 적합한지 아직 명확하지 않기 때문이다.

현재 MISD를 사용하는 제품은 알려져 있지 않다.

MIMD

복수 인스트럭션 스트림, 복수 데이터 스트림인 MIMD는 오늘날 넓게 보아 멀티 코어 프로세서를 말한다. 프로세서를 구성하는 각 코어는 독립적 및 병렬적으로 작동 가능하다. SIMD와 비교했을 때 MIMD 기반의 컴퓨터는 복수 데이터 세트의 단일 연산과 반대로, 병렬적으로 복수 데이터 세트를 여러 연산으로 구별해 동작할 수 있다.

독립적으로 동작하는 여러 개의 입력 데이터 스트림으로 이뤄진 다수의 프로세싱 유닛을 나타내는 다음 그림을 살펴보자.



▲ 출처: wikipedia.org

일반적인 멀티 코어 프로세서가 MIMD 아키텍처를 이용한다.

컴퓨터 메모리 아키텍처 스타일

동시성과 병렬화라는 개념을 소개하면서, 차후 고려하고 다뤄볼 새로운 문제에 봉착했다. 가장 큰 문제는 데이터에 접근하는 속도다. 데이터에 충분히 빠르게 접근할 수 없을 때 프로그램에서 병목 현상이 발생할 것이고, 아무리 디자인이 훌륭한 프로그램이라도 성능 향상이 힘들다.

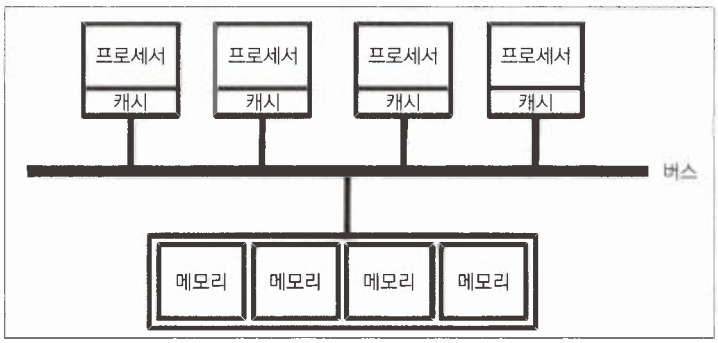
컴퓨터 프로그래머는 문제를 해결할 수많은 병렬 솔루션을 찾으려고 계속 노력한다. 성능을 높일 수 있는 방법 중 하나는 모든 코어가 프로세서상에서 접근할 수 있는 단일 물리 주소 공간을 제공하는 것이다. 프로그래머의 입장에서, 이는 복잡한 부분을 없애 코드상의 스레드 부분에 좀 더 신경 쓸 수 있게 한다.

여러 상황에서 사용되는 각기 다른 아키텍처 스타일이 존재한다. 이 중 시스템 프로그래머가 사용하는 아키텍처 스타일 두 가지가 있는데, 바로 UMA 형식과 NUMA이다.

UMA

UMA Uniform Memory Access (균일 메모리 접근) 아키텍처 스타일은 프로세싱 코어 개수에 상관없이 동일한 방식으로 공유 메모리 공간을 사용하는 것을 말한다. 쉽게 말해, 코어의 위치와 메모리와의 거리에 상관없이 동일한 시간에 직접 메모리에 접근할 수 있다. 이러한 스타일을 흔히 SMP Symmetric Shared-Memory Multiprocessors (대칭형 공유 메모리 멀티프로세서) 방식이라고도 한다.

UMA 스타일을 다음 그림으로 이해해보자. 각 프로세서는 버스를 통해 인터페이싱하며 모든 메모리에 접근하게 된다. 시스템은 또한 버스 대역폭에 부담을 준다. 따라서 NUMA 아키텍처를 사용한다면, 동일한 방법으로 그 크기를 조정할 수 없다.



▲ 출처: https://software.intel.com/en-us/articles/optimizing-applications-for-numa

UMA의 장점은 다음과 같다.

- 모든 RAM 접근은 정확한 시간에 일어난다.
- 캐시cache가 일정하다.
- 하드웨어 디자인이 간단하다.

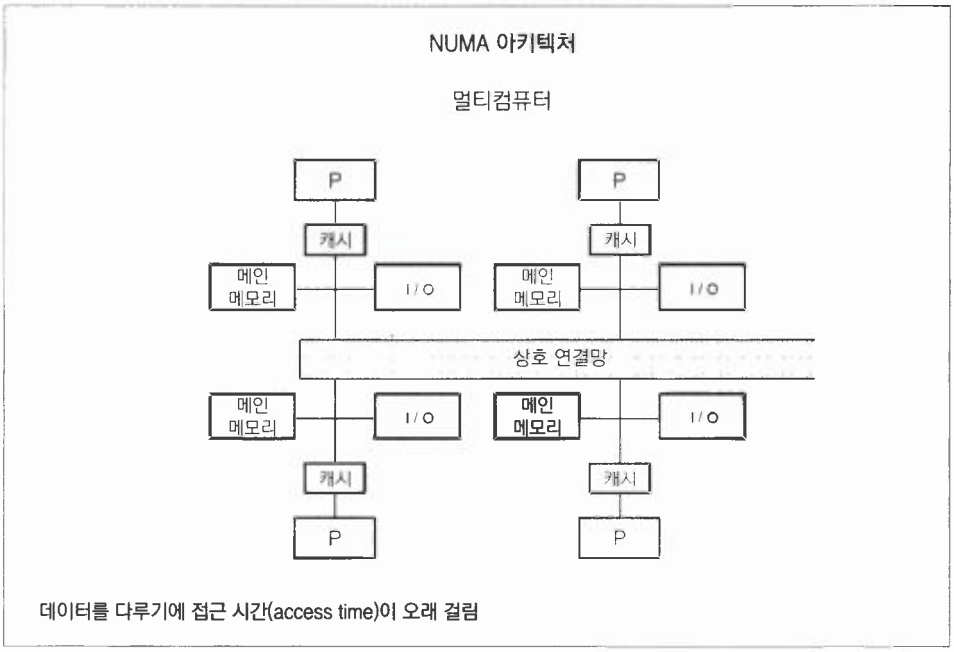
하지만 단점도 존재한다.

- UMA 시스템은 모든 시스템 접근 메모리에서 1개의 메모리 버스만 이용하므로, 스케일링scaling 문제가 나타난다.

NUMA

NUMANon-Uniform Memory Access(불균일 메모리 접근)란 요청되는 프로세스에 따라 메모리 접근이 더 빠를 수 있는 아키텍처 스타일이다(이는 메모리를 기준으로 프로세서의 위치 때문이다).

다음 그림은 수많은 프로세서가 NUMA 형태로 어떻게 연결되어 있는지를 보여준다. 각 프로세서는 캐시, 메인 메모리, 독립적인 입출력을 갖고 있는데, 모두 상호 연결망interconnection network으로 연결됐다.



▲ 출처: https://virtualizationdeepdive.wordpress.com/deep-dive-numa-vnuma/

NUMA 형태의 장점을 살펴보자.

- NUMA 컴퓨터는 균일 메모리 접근 방식과 비교해 스케일링이 쉽다.

반면, 단점도 있다.

- 비결정적 메모리 접근 시간non-deterministic memory access times은 로컬 메모리일 경우 접근 시간이 매우 빠르지만, 외부 메모리일 경우 시간이 오래 걸린다.
- 프로세서는 그 외 프로세서에서 생긴 변화를 봐야 하는데, 주변 프로세서의 개수에 따라 그 시간이 증가할 수도 있다.

요약

2장에서는 동시성과 병렬화의 차이점과 함께 다양한 주제를 배웠다. 동시성과 병렬화가 다양한 형태로 CPU에 미치는 영향과, 나아가 컴퓨터 시스템 디자인과 동시성 및 병렬 프로그래밍과 관련되는 부분까지 알아봤다.

이제 소프트웨어에 영향을 주는 두 가지 문제를 알아보고, 이를 해결할 수 있는 방법도 생각해야 한다. 여러 형태로 시스템 아키텍처가 사용되는 부분도 살펴보고, 소프트웨어 디자인에 어떤 영향을 미치는지도 알아야 한다.

3장에서는 스레드의 사이클을 더 알아보고, 컴퓨터에서 어떻게 동작되는지 살펴보자.

03

스레드 라이프

2장에서 동시성과 병렬화 및 멀티스레드 파이썬 애플리케이션을 자세히 알아봤다. 이제 스레드를 통한 작업 수행으로 넘어가 보자.

3장에서는 스레드 라이프thread life에 대해 알아본다. 다음과 같은 주제를 다룬다.

- 스레드의 상태
- 여러 형태의 스레드(윈도우와 POSIX)
- 자체 스레드를 이용하는 가장 좋은 방법
- 스레드로 작업을 쉽게 하는 방법
- 스레드와 수많은 멀티스레딩 모델을 종료하는 방법

파이썬에서의 스레드

스레드 라이프를 자세히 살펴보기 전, 실제 경우의 예시를 통해 설명하는 것이 더 좋다고 본다. 하지만 먼저 threading.py에서 볼 수 있는 파이썬 스레드 클래스 정의를 살펴보자.

이 파일에서 스레드 클래스의 정의를 살펴볼 수 있다. 다음과 같이 생성자constructor 함수를 갖고 있다.

```
# 파이썬 스레드 클래스 생성자
def __init__(self, group=None, target=None, name=None,
  args=(), kwargs=None, verbose=None):
```

위 생성자에는 5개의 인자가 있는데, 다음과 같다.

- group: 이후의 확장성을 고려해 남겨두는 파라미터다.
- target: run() 메소드에 의해 호출되는 객체다. 이 파라미터를 거치지 않으면, None으로 세팅되며, 시작되지 않는다.
- name: 스레드의 이름
- args: 타깃을 실행하기 위한 튜플 인자다. 기본적으로 () 형태다.
- kwargs: 기본 클래스 생성자를 실행하는 키워드 인자 딕셔너리다.

스레드 상태

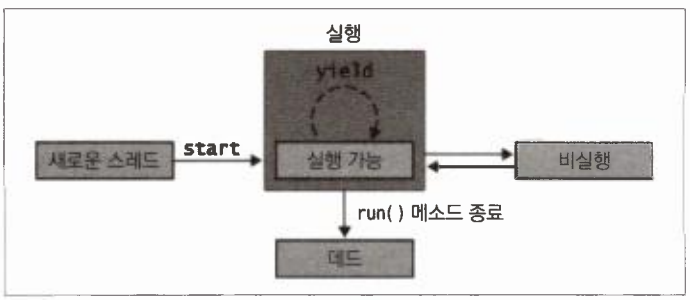
스레드는 5개의 상태가 있는데, 실행running, 비실행not-running, 실행 가능runnable, 시작starting, 종료ended다. 일반적으로 스레드를 생성한다고 해서 자원을 바로 할당받지는 않는다. 초기화되지 않았기에 아무런 상태가 아니며, 시작될 수도 정지될 수도 있다.

- 새로운 스레드New Thread: 새로운 스레드 상태는 스레드가 시작되지도 자원을 할당받지도 않은 상태다. 객체 인스턴스만을 나타낸다.
- **실행 가능**Runnable: 스레드 실행 전의 준비 상태로, 필요한 모든 자원을 갖고 있으며 작업 스케줄러는 실행에 필요한 스케줄링을 준비하고 있다.

- **실행**Running : 스레드는 작업 스케줄러에 의해 선택된 주어진 작업을 실행한다. 이후, 스레드가 종료된다면 데드 상태 또는 비실행 상태로 넘어간다.
- **비실행**Not-running : 스레드가 정지된 상태다. 장시간의 입출력 요청 반응에 대기하는 등의 여러 이유로 발생한다. 그 외 스레드의 실행이 완료될 때까지 일부러 차단되기도 한다.
- **데드**Dead : 여러 이유로 스레드는 데드 상태가 된다. 크게 보면, 자연스럽게 종료되는 경우와 그렇지 않은 경우가 있다. 후자의 경우 심각한 문제로 이어지는 경우가 있으니, 마지막에 살펴본다.

상태 플로우 차트

다음 그림은 스레드에서 나타나는 다섯 가지 상태와 서로 간의 연결을 나타낸다.



스레드 상태 예제

그렇다면 여러 가지 스레드 상태를 파이썬 코드로는 어떻게 나타낼까? 다음 코드를 살펴보자.

```
import threading
import time
# 스레드를 간단히 실행해보자.
def threadWorker():
```

```
    # 스레드가 '실행 가능'에서 '실행' 상태로 시작되는 부분이다.
    print("My Thread has entered the 'Running' State")
    # time.sleep() 메소드를 호출하면, 스레드는 비실행 상태가 된다.
    # 해당 스레드는 더 이상 아무런 작업이 없다.
    time.sleep(10)
    # 스레드 작업이 완료되고 종료된다.
    print("My Thread is terminating")

# 이 시점에서 스레드는 아무런 상태가 없다.
# 어떤 시스템 자원으로부터 할당받지도 않는다.
myThread = threading.Thread(target=threadWorker)
# myThread.start()를 호출하면, 파이썬은 스레드를 실행하고자
# 필요한 자원을 할당하고, 스레드 실행 메소드를 호출한다.
# '시작'에서 '실행 가능' 상태가 된다.
myThread.start()

# 다음 메소드가 호출되면 스레드는 조인되며, '데드' 상태가 된다.
# 의도대로 작업은 종료된다.
myThread.join()
print("My Thead has entered a 'Dead' state")
```

코드에 관한 설명

threadWorker라는 함수를 정의했는데, 이는 생성될 스레드의 타깃을 지정한다. threadWorker 함수는 현재 상태를 출력하고, time.sleep(10)을 호출해 10초간 대기한다.

threadWorker를 정의한 후, '새로운 스레드' 객체를 생성한다.

```
myThread = threading.Thread(target=threadWorker)
```

이때 스레드 객체는 현재 '새로운 스레드' 상태이며, 실행에 필요한 시스템 자원을 아직 할당받지 못한 상태다. 이는 다음 함수를 호출해야만 할당받는다.

```
myThread.start()
```

다음으로 스레드는 자원을 할당받으며, run 함수가 호출된다. 스레드는 이제 '실행 가능' 상태에 접어든다. 현재 상태가 출력되며, `time.sleep(10)` 함수에 의해 10초간 실행이 중지된다. 10초 동안 스레드는 '비실행' 상태로 간주되며, 그 밖의 스레드는 해당 스레드 대신 실행되고자 스케줄링될 것이다.

10초간 대기 후, 스레드는 종료되고 마침내 '데드' 상태가 된다. 더 이상 할당 자원이 필요하지 않고, 가비지 컬렉터garbage collector[1]가 메모리를 정리한다.

여러 형태의 스레드

파이썬은 로우 레벨 스레딩 API의 복잡함을 없애, 훨씬 복잡한 시스템을 구성하는 데 집중하게 한다. 그뿐 아니라, 코드가 실행될 운영체제에 따라 POSIX 또는 윈도우에 영향을 주는 운용 가능한 코드를 작성하게 한다.

하지만 POSIX 스레드 또는 윈도우 스레드를 통해 무엇을 말하는 걸까?

POSIX 스레드

POSIX 스레드[2]에 대해 말할 때는 IEEE POSIX 1003.1c 스탠더드에 따라 구현된 스레드를 살펴본다. 이는 IEEE 협회의 트레이드마크로 등록되어, 유닉스UNIX 시스템상의 수많은 하드웨어 스레드 구현의 표준을 정하고자 개발됐다. 일반적으로, 이를 따르는 모든 스레드를 POSIX 스레드 및 PThreads라고 한다.

윈도우 스레드

윈도우 스레드의 경우, 마이크로소프트가 그 외 스레드와 차별화해 자체적으로 로우 레벨 스레드를 구현한 형태다. POSIX 스레드와 비교했을 때 많은 차이가 있지만, API가

---

1 시스템에서 더 이상 사용하지 않는 동적 할당된 메모리 블록 및 객체를 찾아 자동으로 다시 사용 가능한 자원으로 회수하는 프로그램 – 옮긴이

2 병렬적으로 작동하는 소프트웨어의 작성을 위해 제공되는 표준 API – 옮긴이

간단하며 전반적으로 가독성 높은 형태를 띠고 있다.

스레드를 시작하는 방법

이 절에서는 스레드와 프로세스를 시작하는 다양한 방법을 살펴본다.

스레드 시작하기

파이썬에서는 스레드 시작 방법을 다양하게 지원한다. 비교적 간단하게 멀티스레드 작업을 구성하고, 간단한 함수로 이를 정의해보자.

다음 예제에서, 무작위 시간 간격 동안 유휴 상태에 들어가는 간단한 함수를 작성해보자. 이는 간단한 함수성 형태를 띠며, 함수의 캡슐화encapsulating 도 구현해 `threading.Thread` 객체를 타깃으로 하는 간단한 함수를 거친다.

```
import threading
import time
import random
def executeThread(i):
  print("Thread {} started".format(i))
  sleepTime = random.randint(1,10)
  time.sleep(sleepTime)
  print("Thread {} finished executing".format(i))
for i in range(10):
  thread = threading.Thread(target=executeThread, args=(i,))
  thread.start()
  print("Active Threads:" , threading.enumerate())
```

스레드 클래스로부터 상속

단일 함수보다 많은 코드가 필요한 상황에서, 실제로 스레드 네이티브native 클래스에서 디렉토리를 상속받는 클래스를 정의할 수 있다.

이는 단일 함수에서 복잡한 코드 형태에 적합하고, 따라서 여러 함수로 쪼개야 한다. 스레드를 다룰 때 전반적인 유연성이 증가되지만, 해당 클래스 내에서 스레드를 다루는 부분을 고려해야 한다.

네이티브 파이썬 스레드 클래스의 하위 클래스인 '새로운 스레드'를 정의하고자, 다음 세 부분을 살펴보자.

- 스레드 클래스 안에 클래스를 정의한다.
- 스레드를 초기화하고자 생성자 내의 `Thread.__init__(self)`를 호출한다.
- 스레드가 시작될 때 호출되는 `run()` 함수를 정의한다.

```
from threading import Thread
class myWorkerThread(Thread):
  def __init__(self):
  print("Hello world")
  Thread.__init__(self)
  def run(self):
  print("Thread is now running")
myThread = myWorkerThread()
print("Created my Thread Object")
myThread.start()
print("Started my thread")
myThread.join()
print("My Thread finished")
```

코드에 관한 설명

위 코드에서 `myWorkerThread`라는 간단한 클래스를 정의했는데, 스레드 클래스로부터 상속받았다. 생성자 내에서 필수인 `Thread.__init__(self)` 함수를 호출했다.

그 후 `myThread.start()`를 시작할 때 호출되는 `run` 함수를 정의한다. 해당 `run` 함수 내에서 간단하게 `print` 함수를 통해 콘솔 창에 현재 상태를 출력하고, 스레드가 종료됨을 볼 수 있다.

포킹

프로세스 포킹forking이란 주어진 프로세스를 모방하는 것을 말한다. 다시 말해, 프로세스를 포킹하면 효과적으로 복제당한 프로세스의 자식 프로세스로서 동작된다.

이로써 생성된 프로세스는 자체 주소 공간과 함께 부모의 데이터와 프로세스 내의 코드 실행까지 정확하게 복사된다. 그 후, 복제된 프로세스는 자체의 **프로**세스 식별자PID, Process IDentifier[3]를 받아, 복제된 대상인 부모 프로세스와는 독립적으로 작동한다.

그렇다면 왜 기존의 프로세스를 복제할까? 웹사이트 호스팅을 해보면 대부분 아파치Apache 웹 서버를 이용했을 것이다. 아파치는 여러 개의 서버 프로세스를 생성하는 유틸리티 프로그램이다. 각각의 독립적인 프로세스는 해당 주소 공간 내의 요청을 다룰 수 있다. 따라서 프로세스가 충돌하고 종료되는 등의 영향 없이, 동시에 실행될 때와 같은 상황에서 새로운 요청에 계속적으로 대응할 수 있다.

예제

예제 코드를 살펴보자.

```
import os
def child():
  print ("We are in the child process with PID= %d"%os.getpid())
def parent():
  print ("We are in the parent process with PID= %d"%os.getpid())
  newRef=os.getpid()
  if newRef==0:
    child()
  else:
    print ("We are in the parent process and our child process has
PID=%d"%newRef)
parent()
```

3 유닉스 커널 같은 운영체제에서 각 프로세스나 서비스를 식별하고자 할당하는 고유 번호 – 옮긴이

코드에 관한 설명

앞의 코드에서는 os 파이썬 모듈을 불러온다. 그 후, child와 parent라는 2개의 함수를 호출한다. 두 함수 모두 PID를 출력한다.

parent 함수에서는 현재 실행되는 프로세스를 포킹하고자 os.fork()[4] 메소드를 호출하기 전 PID를 출력한다. 이는 완전히 새로운 프로세스를 생성해 새로운 PID를 건네받는다. 다음으로 child 함수를 호출하는데, 현재 PID를 출력한다. 보다시피, 해당 PID는 코드 실행 초기에 출력됐던 기존 PID와는 다르다.

이렇게 각 PID는 프로세스가 포킹돼 완전 새로운 프로세스가 생성됐음을 보여준다.

스레드 데몬화

먼저 데몬 스레드daemon thread가 무엇인지부터 알아보자. 데몬 스레드란 종료점이 정의되지 않은 '실질적인' 스레드다. 이는 프로그램이 종료될 때까지 계속 작동한다.

필요성에 대한 의문이 생길 것이다. 예컨대, 서비스 요청을 여러 애플리케이션의 인스턴스로 전송하는 로드 밸런서load balancer가 있다고 하자. 이러한 요청이 어디로 전송될지 로드 밸런서에 알려주는 레지스트리 서비스가 있는데, 그렇다면 레지스트리 서비스는 어떻게 현재 인스턴스의 상태를 알까? 일반적으로, 이러한 상황에서는 심장박동이나 일정한 간격의 패킷 형태로 레지스트리 서비스에게 전달해야 한다.

이러한 예제는 애플리케이션에서 데몬 스레드가 가장 빈번히 사용되는 경우다. 심장박동 신호로 보내는 것이 컴퓨터에서는 데몬 스레드에서 서비스 레지스트리로 보내는 경우이며, 이는 애플리케이션이 시작될 때 같이 시작된다. 해당 데몬 스레드는 프로그램의 백그라운드에 위치해 주기적으로 변경 없이 갱신값을 전달한다. 데몬 스레드는 인스턴스가 종료되는 것을 걱정할 필요 없이 알아서 관리한다.

---

4 리눅스 시스템에서만 동작한다. – 옮긴이

예제

예제 코드를 살펴보자.

```
import threading
import time
def standardThread( ):
  print("Starting my Standard Thread")
  time.sleep(20)
  print("Ending my standard thread")
def daemonThread( ):
  while True:
    print("Sending Out Heartbeat Signal")
    time.sleep(2)
if __name__ == '__main__':
  standardThread = threading.Thread(target=standardThread)
  daemonThread = threading.Thread(target=daemonThread)
  daemonThread.setDaemon(True)
  daemonThread.start( )

  standardThread.start( )
```

코드에 관한 설명

위 코드에서는 타깃으로 동작하는 daemonThread 및 데몬이 아닌 일반 스레드 함수를 정의한다. standardThread 함수는 현재 상태를 간단히 출력하고 20초간 유휴한다.

daemonThread 함수는 영구적인 while 반복문에 들어가며, 2초마다 Sending Out Heartbeat Signal이라고 출력한다. 이는 모든 경우에 그다음 문장을 실행함을 의미한다.

main 함수는 일반 스레드와 데몬 스레드를 생성하는데, 모두 start( ) 메소드로 시작한다. daemonThread 객체의 setDaemon 함수도 사용한다. 이 함수는 원하는 스레드 객체를 데몬 플래그[flag]를 통해 데몬 스레드로 만드니 참고하자.

파이썬에서 스레드 다루기

이 절에서는 파이썬 프로그램 내에서 여러 스레드를 생성하고 다루는 방법을 살펴본다.

스레드 시작하기

먼저 살펴볼 부분은 한 번에 모든 스레드를 시작하는 방법이다. for 반복문을 이용해 여러 가지 스레드 객체를 생성하고, 동일한 반복문 내에서 시작해본다. 다음 예제에서는 정수를 취하는 함수를 정의하고, 해당 정수 시간 동안 유휴 상태를 갖고, 시작과 종료 상태를 출력해본다.

그 후, 10번의 for 문 반복으로 executeThread를 타깃으로 지정한 10개의 개별 스레드 객체를 생성한다. 바로 스레드 객체를 실행하고, 현재 실행 중인 스레드를 출력한다.

예제

예제 코드를 살펴보자.

```
import threading
import time
import random
def executeThread(i):
  print("Thread {} started".format(i))
  sleepTime = random.randint(1,10)
  time.sleep(sleepTime)
  print("Thread {} finished executing".format(i))
for i in range(10):
  thread = threading.Thread(target=executeThread, args=(i,))
  thread.start()
  print("Active Threads:" , threading.enumerate())
```

코드에 관한 설명

i를 파라미터로 하는 executeThread 함수를 정의한 것을 볼 수 있다. 해당 함수에서 time.sleep() 함수 호출을 통해 1에서 10까지 무작위로 생성된 정숫값을 받아 유휴 상태가 된다.

다음으로 1부터 10까지 반복되는 for 문을 선언해 스레드 인자가 i일 때 스레드 객체가 생성되고 시작되게 한다. 코드를 실행하면 다음과 같이 출력될 것이다.

```
$ python3.6 00_startingThread.py
Thread 0 started
Active Threads: [<_MainThread(MainThread, started 140735793988544)>,
<Thread(Thread-1, started 123145335930880)>]
Thread 1 started
Active Threads: [<_MainThread(MainThread, started 140735793988544)>,
<Thread(Thread-1, started 123145335930880)>, <Thread(Thread-2, started
123145341186048)>]
Thread 2 started
Active Threads: [<_MainThread(MainThread, started 140735793988544)>,
<Thread(Thread-1, started 123145335930880)>, <Thread(Thread-2, started
123145341186048)>, <Thread(Thread-3, started 123145346441216)>]
```

스레드를 이용해 프로그램 속도 낮추기

스레드로 작업을 이어나갈 때, 수백 개의 스레드를 특정 문제에 모두 쏟아붓는다고 프로그램의 성능이 향상되는 것은 아니다. 수백, 수천 개의 스레드를 모두 이용한다면, 성능은 급격히 떨어진다고 본다.

1장에서 집중적인 계산을 하는 소인수 분해 프로그램의 속도를 향상하고자 멀티프로세스를 사용했다. 내 컴퓨터에서 멀티프로세스를 사용하면 많게는 50~100%의 성능 향상을 볼 수 있었지만, 멀티프로세서가 아닌 멀티스레드일 때 이러한 현상이 나타났다. 다음 코드를 살펴보자.

예제

```
import time
import random
import threading
def calculatePrimeFactors(n):
  primfac = []
  d = 2
  while d*d <= n:
    while (n % d) == 0:
      primfac.append(d)
      n //= d
    d += 1
  if n > 1:
    primfac.append(n)
  return primfac
def executeProc():
  for i in range(1000):
    rand = random.randint(20000, 100000000)
    print(calculatePrimeFactors(rand))
def main():
  print("Starting number crunching")
  t0 = time.time()
  threads = []
  for i in range(10):
    thread = threading.Thread(target=executeProc)
    threads.append(thread)
    thread.start()
  for thread in threads:
    thread.join()
  t1 = time.time()
  totalTime = t1 - t0
  print("Execution Time: {}".format(totalTime))
if __name__ == '__main__':
  main()
```

코드에 관한 설명

앞의 예제는 1장에서 살펴본 '순차 소인수 분해'와 동일하다. 하지만 main 함수 내에서 10개의 프로세스를 정의하고 조인하는 방법이 아닌, 10개의 스레드를 정의했다.

프로그램을 실행하면, 단일 스레드와 비교해 멀티프로세싱 및 멀티스레드의 전반적인 성능이 급격히 감소함을 볼 수 있다. 결과를 살펴보자.

단일 스레드	3.69초
멀티프로세싱	1.98초
멀티스레드	3.95초

위 표와 같이, 멀티스레드로 문제를 수행하면 단일 스레드에 비해 약 7%의 성능 감소가 일어남을 볼 수 있으며 멀티프로세서와는 2배 정도 차이가 난다.

현재 실행 중인 모든 스레드의 개수 구하기

애플리케이션의 상태를 나타내고자, 현재 실행 중인 파이썬 프로그램의 활성화된 스레드 개수를 나열해보자. 고맙게도, 파이썬 내장 threading 모듈에서는 실행 중인 스레드의 개수를 구하는 함수를 제공한다.

예제

```
import threading
import time
import random
def myThread(i):
  print("Thread {}: started".format(i))
  time.sleep(random.randint(1,5))
  print("Thread {}: finished".format(i))
def main():
  for i in range(random.randint(2,50)):
```

```
    thread = threading.Thread(target=myThread, args=(i,))
    thread.start()
  time.sleep(4)
  print("Total Number of Active Threads: {}".format(threading.active_count()))
if __name__ == '__main__':
  main()
```

코드에 관한 설명

위 코드를 살펴보면, 2부터 50까지의 무작위 개수의 스레드를 실행해 무작위 시간 동안 유휴 상태에 들어간다. 주어진 스레드를 모두 실행하면, 4초 동안 유휴 상태를 거치고, threading.active_count()를 호출해 print 문으로 결괏값을 출력한다.

현재 스레드 나타내기

현재 어떤 스레드가 동작되는지 보고 싶다면, threading.current_thread() 함수를 사용한다.

예제

```
import threading
import time
def threadTarget():
  print("Current Thread: {}".format(threading.current_thread()))
threads = []
for i in range(10):
  thread = threading.Thread(target=threadTarget)
  thread.start()
  threads.append(thread)
for thread in threads:
  thread.join()
```

코드에 관한 설명

threadTarget이라는 함수는 현재 스레드를 출력한다. 다음으로 스레드를 넣은 빈 배열을 생성하고, 스레드 객체 10개를 해당 배열에 넣는다. 그 후, 스레드를 서로 조인해 프로그램이 갑자기 종료되지 않게 한다. 프로그램의 콘솔 출력은 다음과 같다.

```
$ python3.6 10_gettingCurrentThread.py
Current Thread: <Thread(Thread-1, started 123145429614592)>
Current Thread: <Thread(Thread-2, started 123145429614592)>
Current Thread: <Thread(Thread-3, started 123145434869760)>
Current Thread: <Thread(Thread-4, started 123145429614592)>
Current Thread: <Thread(Thread-5, started 123145434869760)>
Current Thread: <Thread(Thread-6, started 123145429614592)>
Current Thread: <Thread(Thread-7, started 123145434869760)>
Current Thread: <Thread(Thread-8, started 123145429614592)>
Current Thread: <Thread(Thread-9, started 123145434869760)>
Current Thread: <Thread(Thread-10, started 123145429614592)>
```

메인 스레드

모든 파이썬 프로그램에는 최소 1개의 스레드가 있는데, 이를 메인 스레드라고 한다. 파이썬에서는 메인 스레드 객체가 검색하는 부분을 불문하고 main_thread() 함수를 적절히 호출할 수 있다.

예제

```
import threading
import time
def myChildThread():
  print("Child Thread Starting")
  time.sleep(5)
  print("Current Thread ----------")
```

```
  print(threading.current_thread())
  print("-------------------------")
  print("Main Thread -------------")
  print(threading.main_thread())
  print("-------------------------")
  print("Child Thread Ending")
child = threading.Thread(target=myChildThread)
child.start()
child.join()
```

코드에 관한 설명

위 코드에서는 myChildTread라는 함수를 간단히 정의하고 있다. 설명하고자 생성된 스레드 객체의 타깃이 될 것이다. 해당 함수에서는 현재 스레드와 메인 스레드를 차례대로 출력한다.

다음으로 스레드 객체 생성으로 넘어가, 새로 생성된 스레드를 시작하고 조인한다. 다음 출력을 참고하자.

```
$ python3.6 15_mainThread.py
Child Thread Starting
Current Thread ----------
<Thread(Thread-1, started 123145387503616)>
-------------------------
Main Thread -------------
<_MainThread(MainThread, started 140735793988544)>
-------------------------
Child Thread Ending
```

보다시피 자식 스레드 객체가 먼저 출력되고, MainThread 객체 레퍼런스가 다음으로 출력된다.

모든 스레드 열거하기

현재 활성화된 스레드를 검색하고자 모든 스레드를 열거해야 할 때가 있다. 하지만 주어진 애플리케이션의 어떤 스레드가 작동되는지 그 위치를 놓칠 수도 있다.

다행히도 파이썬은 기본적으로 모든 스레드의 검색을 지원하며, 필요한 정보를 얻을 수 있고, 적절하게 종료할 수도 있다. 다음 예제 코드를 살펴보자.

예제

```
import threading
import time
import random
def myThread(i):
  print("Thread {}: started".format(i))
  time.sleep(random.randint(1,5))
  print("Thread {}: finished".format(i))
def main():
  for i in range(4):
    thread = threading.Thread(target=myThread, args=(i,))
    thread.start()
  print("Enumerating: {}".format(threading.enumerate()))
if __name__ == '__main__':
  main()
```

코드에 관한 설명

위 코드에서는 myThread라는 간단한 함수를 통해 추후 생성할 스레드의 타깃으로 지정한다. 해당 함수는 스레드가 시작될 때 출력되고, 1에서 5 사이의 무작위 시간 동안 유휴하고 출력문과 함께 종료된다.

다음으로 4개의 스레드 객체를 각각 생성하는 main 함수를 정의해 시작해본다. 스레드 생성과 시작이 끝났다면, threading.enumerate()의 결괏값을 출력한다.

```
$ python3.6 07_enumerateThreads.py
Thread 0: started
Thread 1: started
Thread 2: started
Thread 3: started
Enumerating: [<_MainThread(MainThread, started 140735793988544)>,
<Thread(Thread-1, started 123145554595840)>, <Thread(Thread-2,
started 123145559851008)>, <Thread(Thread-3, started
123145565106176)>, <Thread(Thread-4, started 123145570361344)>]
Thread 2: finished
Thread 3: finished
Thread 0: finished
Thread 1: finished
```

스레드 확인하기

개발자는 특정 상황에서는 각 스레드를 구별할 수 있어야 한다. 애플리케이션이 수백 개의 스레드로 이뤄진 경우, 프로그램의 디버깅이나 문제를 확인하는 데 많은 도움을 준다.

대규모 시스템에서 스레드가 그 밖의 작업을 진행하고 있다면, 역할별로 그룹화하는 것도 좋다. 예컨대, 수입 주가 변화를 입력하고 가격이 어떻게 이뤄질지 예측하는 애플리케이션이 있다고 하자. 스레드를 두 가지로 그룹화해, 하나는 변화를 입력하고 나머지는 필요한 연산을 수행하게 한다.

입력과 계산을 수행하는 스레드에 적절하게 명명함으로써 작업에 많은 도움이 된다.

예제

이 예제에서는 간단하게 Thread-x(x는 특정 숫자)라고 명명해본다.

```
import threading
import time
def myThread( ):
```

```
  print("Thread {} starting".format(threading.currentThread().getName()))
  time.sleep(10)
  print("Thread {} ending".format(threading.currentThread().getName()))
for i in range(4):
  threadName = "Thread-" + str(i)
  thread = threading.Thread(name=threadName, target=myThread)
  thread.start()
print("{}".format(threading.enumerate()))
```

코드에 관한 설명

myThread 함수 정의를 먼저 알아보자. 해당 함수는 현재 스레드의 이름을 검색하고자 threading.currentThread().getName()을 사용하며, 스레드 실행이 시작되고 종료될 때 출력한다.

다음으로 for 문에서 4개의 이름 파라미터를 갖는 스레드 객체를 생성해 "Thread-" + str(i)라고 정의하며, myThread 함수도 스레드 실행의 타깃으로 한다.

마지막으로, 현재 활성화돼 실행 중인 모든 스레드를 출력한다.

```
$ python3.6 11_identifyingThreads.py
Thread Thread-0 starting
Thread Thread-1 starting
Thread Thread-2 starting
Thread Thread-3 starting
[<_MainThread(MainThread, started 140735793988544)>,
<Thread(Thread-0, started 123145368256512)>, <Thread(Thread-1,
started 123145373511680)>, <Thread(Thread-2, started
123145378766848)>, <Thread(Thread-3, started 123145384022016)>]
Thread Thread-0 ending
Thread Thread-2 ending
Thread Thread-3 ending
Thread Thread-1 ending
```

스레드 종료하기

스레드를 종료하는 것은 보통 일반적이지 않다고 여겨지지만, 개인적으로 생각하기에 알아야 된다고 본다. 파이썬은 스레드를 종료하는 내장 스레드 함수를 지원하지 않으니 주의해야 한다. 종료하려는 스레드는 입출력에 필요한 중요한 시스템 자원에 연결돼 있거나, 자식 스레드가 있을 수도 있다. 자식 스레드를 종료하지 않고 부모 스레드를 종료하면, 자식 스레드는 고아orphan 스레드가 된다.

스레드를 종료하는 방법

스레드를 종료하는 메커니즘을 구현하려면, 올바른 방법으로 스레드를 종료해야 한다.

하지만 차선책을 살펴보자. 스레드에는 종료에 필요한 내장 메커니즘을 지원하지 않으니, 프로세스에서 이러한 작업을 진행해야 한다. 프로세스는 스레드의 확장된 버전이므로, 이상적인 방법은 아니더라도 프로그램이 적절하게 종료되는 상황에서는 자체 스레드 종료 프로그램을 구현하는 것보다 좋은 해결책이다.

예제

```
from multiprocessing import Process
import time
def myWorker():
  t1 = time.time()
  print("Process started at: {}".format(t1))
  time.sleep(20)
myProcess = Process(target=myWorker)
print("Process {}".format(myProcess))
myProcess.start()
print("Terminating Process...")
myProcess.terminate()
myProcess.join()
print("Process Terminated: {}".format(myProcess))
```

앞의 코드에서 myWorker() 함수는 시작된 시각을 출력하며, 20초간 유휴 상태를 갖는다. 다음으로, 프로세스 형태인 myProcess를 선언해 myWorker 함수를 실행 타깃으로 정한다.

프로세스를 시작해 terminate 메소드로 즉시 종료한다. 프로그램이 갑자기 종료됐다는 사실이 출력되며, 실제로 myProcess 프로세스는 20초 미만으로 실행된다.

출력

```
$ python3.6 09_killThread.py
Process <Process(Process-1, initial)>
Terminating Process...
Process Terminated: <Process(Process-1, stopped[SIGTERM])>
```

고아 프로세스

고아 프로세스orphan process란 부모 프로세스가 없는 스레드를 말한다. 이는 시스템 자원만 차지하고 아무런 역할을 하지 못한 채, 현재 활성화된 스레드를 열거해서만 종료할 수 있다.

운영체제는 어떻게 스레드를 다룰까?

지금까지 스레드의 라이프 사이클을 살펴봤는데, 이러한 스레드는 각자 컴퓨터에서 어떻게 작동할까? 멀티스레딩 모델과 파이썬 스레드의 시스템 스레드 매핑을 이해하는 것은 고성능 소프트웨어를 디자인하는 입장에서 중요하다.

프로세스 생성과 스레드 생성

보다시피, 프로세스는 스레드보다 복잡한 형태다(한 프로세스에서 멀티스레드가 작동하기 때문이다). 프로세스는 각각의 GIL 인스턴스 형태를 띠므로, 일반적인 스레드보다 CPU 기반 작업을 더 수행할 수 있다.

하지만 CPU 기반 문제뿐만 아니라 많은 자원이 소모된다는 사실을 알아야 한다. 많은 자원이 소모된다는 건, 빠르게 실행하고 종료하기가 힘들다는 뜻이다. 다음 예제에서는 멀티스레드의 실행이 성능에 미치는 영향을 살펴보고, 멀티프로세서와 비교해보자.

예제

```
import threading
from multiprocessing import Process
import time
import os
def MyTask( ):
  print("Starting")
  time.sleep(2)
t0 = time.time( )
threads = []
for i in range(10):
  thread = threading.Thread(target=MyTask)
  thread.start( )
  threads.append(thread)
t1 = time.time( )
print("Total Time for Creating 10 Threads: {} seconds".format(t1-t0))
for thread in threads:
  thread.join( )
t2 = time.time( )
procs = []
for i in range(10):
  process = Process(target=MyTask)
  process.start( )
  procs.append(process)
```

```
t3 = time.time()
print("Total Time for Creating 10 Processes: {} seconds".format(t3-t2))
for proc in procs:
  proc.join()
```

코드에 관한 설명

추후 생성될 스레드와 프로세스 모두 타깃으로 하는 MyTask 함수를 정의한다.

t0 변수에 시작 시간을 먼저 저장하고, 스레드가 호출되는 빈 배열을 생성해 생성된 모든 스레드 객체를 저장해보자. 시간을 기록하기 전, 해당 스레드를 생성하고 실행해 생성과 실행에 소모된 전체 시간을 계산한다.

다음으로 동일한 생성과 시작 프로세스를 거치지만, 이번에는 스레드가 아닌 프로세스 방식이다. 다시 소요 시간을 측정하고 계산한다. 내 컴퓨터에서 실행했을 때, 생성과 시작에 걸린 시간은 약 10배 정도였다. 프로세스에서의 시간이 스레드보다 10배 이상 걸렸음을 볼 수 있다. 내 컴퓨터에서의 출력값은 다음과 같다.

```
[20:08:07] ~/Projects/Python/Chapter 03 master
$ python3.6 13_forkVsCreate.py
Total Time for Creating 10 Threads: 0.0017189979553222656 seconds
Total Time for Creating 10 Processes: 0.02233409881591797 seconds
```

몇 개 안 되는 간단한 예제의 경우 시간 소요가 적지만, 수백 수천 개의 프로세스나 스레드가 대규모 서버 랙 등에 있다면 성능에 어떤 영향을 미칠지 생각해보자.

이를 해결하려면 모든 프로세스 및 스레드를 시작할 때 풀pool에 저장해 생성에 많은 부하가 발생하지 않도록 차후의 인스트럭션에 대기하고 있어야 한다. 스레드 풀 및 프로세스 풀은 7장에서 자세히 살펴본다.

멀티스레딩 모델

1장의 첫 절을 살펴보면 동시성을 간략히 소개하는데, 한 하드웨어 내 두 종류의 스레드를 말하고 있다. 바로 사용자 스레드와 커널 스레드인데, 서로 어떻게 매핑되는지 다양한 방법을 알아본다. 크게 세 가지로 나눌 수 있다.

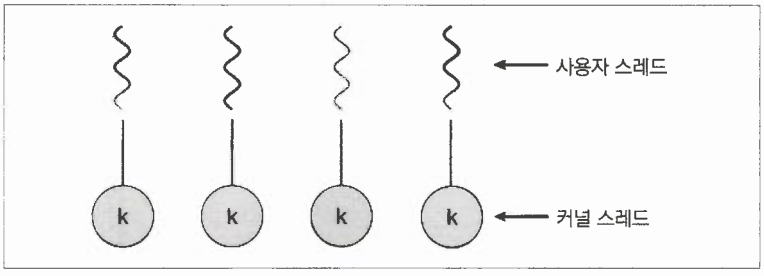
- 1개의 사용자 스레드와 1개의 커널 스레드
- 다수의 사용자 스레드와 1개의 커널 스레드
- 다수의 사용자 스레드와 다수의 커널 스레드

파이썬 내에서는 일반적으로 1개의 사용자 스레드와 1개의 커널 스레드 매핑 방식을 채택하고 있으며, 멀티스레드 기반 애플리케이션 내 생성된 모든 스레드는 많은 양의 자원을 차지한다.

하지만 파이썬에는 단일 스레드로 작동할 때, 멀티스레드와 같은 함수성 프로그래밍을 구현하는 모듈을 지원하고 있다. asyncio 모듈이 이 부분에서 좋은 예제이니, 추후 9장에서 자세히 살펴보겠다.

일대일 스레드 매핑

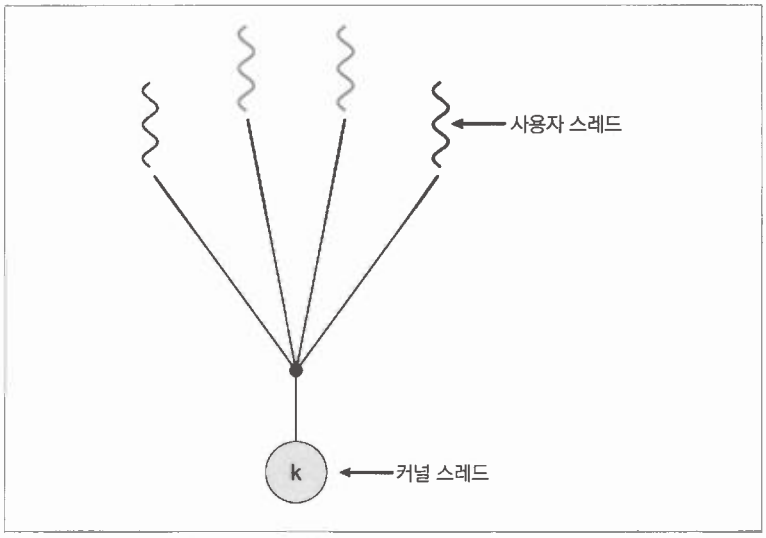
1개의 사용자 레벨 스레드가 직접 1개의 커널 레벨 스레드와 매핑된다. 일대일 매핑은 커널 스레드의 생성과 유지에 많은 부하가 걸리지만, 그 외 사용자 스레드로부터 방해를 받지 않는다는 장점이 있다.



▲ 출처: http://www2.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

다대일

다대일 매핑에서 여러 개의 사용자 스레드가 1개의 커널 스레드와 매핑됐다. 사용자 스레드를 효율적으로 사용할 수 있지만, 사용자 스레드가 정지되면 동일한 커널 스레드에 연결된 그 외 스레드도 정지된다.

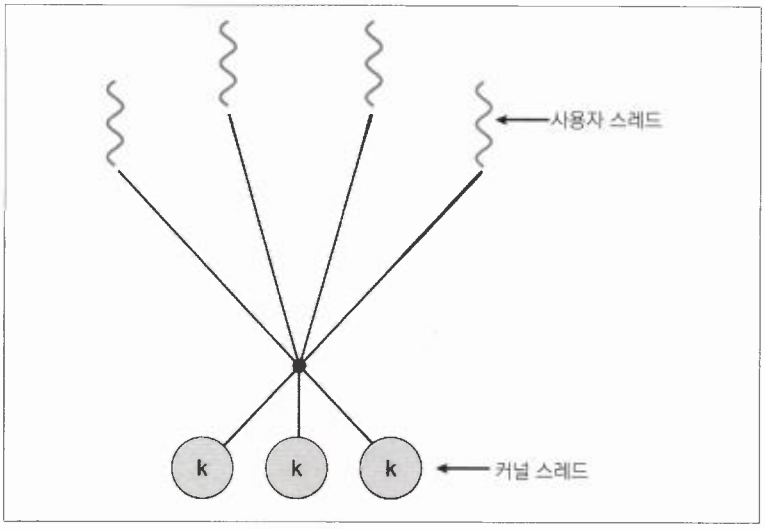


▲ 출처: http://www2.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

다대다

다대다 방식은 다수의 사용자 스레드가 여러 개의 커널 스레드와 매핑된 형태다. 이는 위 두 모델의 단점을 해결하고자 만든 방식이다.

개별적인 사용자 스레드는 단일 및 복수의 커널 스레드 모두의 결합 형태로 매핑이 가능하다. 프로그래머로서 특정 사용자 스레드와 커널 스레드의 매핑을 결정할 수 있고, 멀티 스레드 환경에서 전반적으로 높은 성능 향상을 보장한다.



▲ 출처: http://www2.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

요약

3장에서는 파이썬 내장 스레딩 라이브러리를 자세히 살펴봤다. 스레드를 사용해 효율적으로 작업할 수 있는 방법을 자세하게 배웠으며, 파이썬 스레딩 API에서 제공하는 이점도 살펴봤다.

여러 가지 스레드 종류와 이를 서로 비교해봤다. 또한 멀티스레딩 모델의 여러 가지 형태를 자세히 다루고, 사용자 레벨 및 커널 레벨 스레드를 연결하는 다양한 방법까지 알아봤다.

4장에서는 파이썬이 제공하는 동시성의 가장 핵심적인 부분을 살펴본다. 이러한 기초를 다룸으로써 실제 소프트웨어를 다루는 데 있어 스레드에 문제가 없는 방법으로 디자인이 가능하다.

04

스레드 간 동기화

지금까지 스레드를 이용한 작업과 여러 가지 메커니즘으로 스레드를 생성하는 방법을 알아봤다. 4장에서는 멀티스레드를 사용할 때의 기본적인 동기화 프리미티브primitive를 살펴보자.

애플리케이션의 성능을 높이고자 스레드를 추가하는 것만으로 충분하지 않다. 경합 조건 같은 복잡한 경우를 고려하고, 마찬가지로 코드도 적절하게 작성돼야 한다.

4장에서 다루는 내용은 다음과 같다.

- 스레드 간에 데이터를 동기화하는 방법
- 경합 조건의 정의와 방지책
- 교착상태로 인한 시스템 손상과 그 해결책
- 파이썬에서 제공하는 동기화의 기본적인 사항

파이썬 개발 시 쉽고 버그가 없는 프로그램을 개발하기 위한 여러 툴도 소개한다.

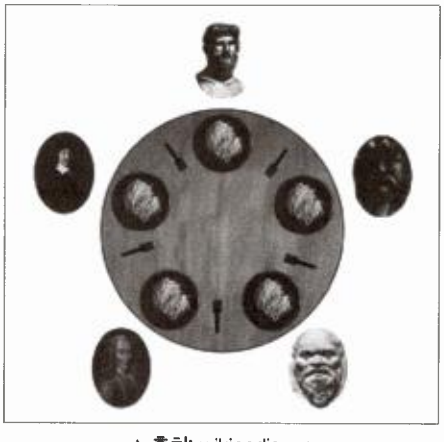
스레드 간 동기화

이제는 파이썬에서 스레드가 무엇이며 어떻게 적절히 생성하고 종료하는지 알지만, 지금부터는 동시성 프로그램을 구현할 때의 복잡성에 대해 알아야 한다. 하지만 프로그램 흐름에 대한 타협 없이 어떻게 온전한 멀티스레딩을 구현할 수 있을까? 이 장에서는 멀티스레드 프로그램에서 발생하는 기본적인 문제를 살펴보겠다.

동기화 문제를 다루기 전에, 프리미티브 문제에 대해 먼저 알아보자. 이는 동시성 시스템을 디자인할 때 교착상태deadlock를 발생시킨다. 교착상태를 가장 잘 설명해줄 수 있는, 철학자의 저녁식사 문제를 살펴보겠다.

철학자의 저녁식사

철학자의 저녁식사 문제는 동시성 소프트웨어 시스템에서 마주하는 문제를 잘 설명해주는 예시다. 원래 1장에서 소개한 에츠허르 데이크스트라Edsger Dijkstra(1930~2002)에 의해 세상에 알려졌지만, 토니 호어Tony Hoare(1934~)에 의해 공식화됐다.

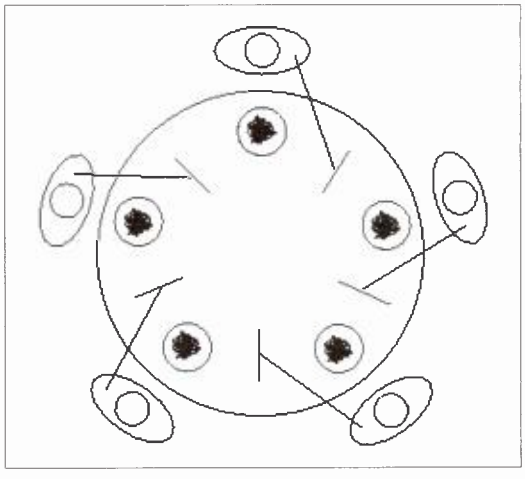


▲ 출처: wikipedia.org

철학자의 저녁식사 문제에서는 5명의 철학자가 둥근 탁자에 둘러앉아 스파게티를 먹는다. 각 접시 사이에는 철학자들이 사용할 5개의 포크가 놓여 있다. 하지만 어떤 이유에서든, 철학자들은 2개의 포크로 식사를 하고자 한다.

여기서 철학자는 식사 중이거나 생각 중이며, 음식을 먹으려면 먼저 왼쪽이나 오른쪽의 포크를 선택해야 한다. 하지만 다른 철학자가 포크를 집는 순간, 그 철학자가 음식을 다 먹고 포크를 놓을 때까지 기다려야 한다.

그런데 5명의 철학자 모두가 각자 왼쪽에 있는 포크를 사용할 때 문제가 발생한다.



▲ 출처: http://www.cs.fsu.edu/~baker/realtime/restricted/notes/philos.html

위의 그림에서 보다시피 문제가 발생했다. 5명의 철학자 모두가 자신의 왼쪽 포크를 집었는데, 오른쪽 포크를 이용할 때까지 모두 생각 중인 상태다. 어떤 철학자도 자신의 포크를 놓지 못하기 때문에 저녁식사 자리는 교착상태에 들어가며, 더 이상의 행동은 없다.

이는 동기화 프리미티브(락)에 의지하는 동시성 시스템을 디자인할 때 발생하는 문제를 나타낸다. 여기서 포크는 시스템 자원이고, 각 철학자는 프로세스를 나타낸다.

예제

다음 예제에서 파이썬의 R락RLocks을 이용해 저녁식사 문제를 구현해보자. R락은 포크를 나타낸다. Philosopher 클래스와 생성자를 다음과 같이 정의하자.

```
class Philosopher(threading.Thread):
def __init__(self, leftFork, rightFork):
  print("Our Philosopher Has Sat Down At the Table")
  threading.Thread.__init__(self)
  self.leftFork = leftFork
  self.rightFork = rightFork
```

Philosopher 클래스는 파이썬 내장 스레드 클래스로부터 상속받아 생성자 함수 내의 왼쪽 및 오른쪽 포크를 취한다. 차후 시작하는 스레드를 초기화한다. 이를 정의한 후, 다음과 같이 스레드의 run 함수를 정의한다.

```
def run(self):
  print("Philosopher: {} hasstarted thinking".format(threading.current_thread()))
  whileTrue:
    time.sleep(random.randint(1,5))
    print("Philosopher {} has finished thinking".format(
threading.current_thread()))
    self.leftFork.acquire()
    time.sleep(random.randint(1,5))
    try:
      print("Philosopher {} has acquired the left fork".format(
threading.current_thread()))
      self.rightFork.acquire()
      try:
        print("Philosopher {} has attained both forks, currently eating".format(
threading.current_thread()))
      finally:
        self.rightFork.release()
        print("Philosopher {} has released the right fork".format(
threading.current_thread()))
```

```
    finally:
      self.leftFork.release()
      print("Philosopher {} has released the left fork".format(
threading.current_thread()))
```

run 함수에서는 1에서 5초 사이의 무작위 시간 동안 생각한다. 철학자의 생각 상태가 끝나고, 왼쪽 포크를 집고 다시 무작위 시간 동안 생각하고, 콘솔 창에 출력한다.

대기 상태를 마치고, 식사 상태에 돌입하고자 오른쪽 포크를 선택하게 한다. 왼쪽 및 오른쪽 포크를 모두 놓기 전 모든 식사가 끝난다.

출력

위의 파이썬 프로그램을 실행하면 철학자들은 포크를 놓기 전에 음식을 먹으려 한다. 하지만 매우 빠르게 모든 포크가 철학자에게 가고, 오른쪽 포크를 가지려는 상황에 빠진다.

다음 출력으로 철학자의 저녁식사에서 생각, 식사, 포크를 집고 놓는 것을 볼 수 있다. 하지만 시간이 어느 정도 지나면 결국 모든 철학자가 왼쪽 포크 모두를 가져, 다음 상황으로 나아갈 수 없음을 볼 수 있다.

```
Marx has started thinking
Russell has started thinking
Aristotle has finished thinking
Marx has finished thinking
Aristotle has acquired the left fork
Aristotle has attained both forks, currently eating
Aristotle has released the right fork
Aristotle has released the left fork
Aristotle has finished thinking
Russell has finished thinking
Kant has finished thinking
Spinoza has finished thinking
Aristotle has acquired the left fork
Marx has acquired the left fork
```

```
Russell has acquired the left fork
Kant has acquired the left fork
Spinoza has acquired the left fork
```

경합 조건

교착상태에 대해 살펴봤으니, 이제 경합 조건race condition도 알아보자. 경합 조건도 마찬가지로 수많은 동시성 프로그래밍에서 나타나는 골칫거리 문제다.

경합 조건의 정의는 다음과 같다.

> 경합 조건 및 경합 위험race hazard이라고 하는 것은 전기적, 소프트웨어적 및 시스템의 출력이 제어할 수 없는 이벤트의 연속 및 타이밍에 의지하는 상태를 말한다.

간단한 용어로 파헤쳐보자. 경합 조건을 가장 잘 보여주는 비유는 은행계좌에 돈을 예금하고 인출하는 은행 애플리케이션이다.

계좌에는 2,000파운드가 있으며, 보너스로 5,000파운드를 받았다고 한다. 그날 동시에 1,000파운드의 렌트비를 내야 한다고 하자(경합 조건이 일어날 빌미를 제공했다).

은행 프로그램에는 2개의 프로세스가 있는데, 하나는 인출을 담당하는 '프로세스 A'이고 다른 하나는 입금을 담당하는 '프로세스 B'이다. 입금을 담당하는 프로세스 B가 계좌잔고를 2,000파운드로 입력했다. 프로세스 B가 거래를 시작한 후 곧바로 프로세스 A가 렌트비를 인출하려고 한다면, 계좌잔고는 2,000파운드부터 시작한다. 프로세스 B는 그 후 거래를 완료하고, 5,000파운드 보너스를 더하고, 총잔고를 7,000파운드로 할 것이다.

하지만 프로세스 A가 계좌잔고 2,000파운드부터 거래를 시작할 때 자신도 모르게 최종 계좌잔고가 1,000파운드로 바뀌면, 보너스가 없어질 수 있다. 이것이 바로 이번 프로그램에서 발생한 경합 조건으로, 심각한 위험이 도사리고 있다.

프로세스 실행 과정

좀 더 자세히 살펴보자. 다음 표는 프로세스 A와 프로세스 B의 이상적인 실행 흐름도다.

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

하지만 계좌잔고에 적절한 동기화 메커니즘을 구현하지 않았기에, 프로세스 A와 프로세스 B는 실제로 다음 실행 과정을 거치며 오류를 범한다.

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

해결책

그렇다면, 차후의 내 보너스가 없어질 수도 있는 매우 중요한 상황에서 문제를 어떻게 해결할 수 있을까? 비교적 간단한 예제에서는 계좌를 읽는 코드와 거래에 필요한 코드를 서로 묶으면 된다(이후에 좀 더 자세히 살펴본다).

계좌잔고를 읽고 갱신하는 코드를 감쌈으로써 프로세스 A는 읽기 및 갱신하는 동안 락을 하고, 프로세스 B 또한 마찬가지다. 이는 일종의 결정 가능한 프로그램으로 바꾸며, 초기의 경합 조건도 해결 가능하다. 하지만 결정 가능한 프로그램은 본질적으로 단일 스레드 및 멀티스레드 상황의 성능에 영향을 줄 수 있는 코드여야 한다.

위험 영역

공유 자원을 수정하고 이에 접근하는 모든 코드의 위험 영역critical section에 대해 알아보자. 이러한 위험 영역은 어떤 경우에서든 한 번에 1개의 프로세스만 실행할 수 있다. 예기치 못한 에러를 찾고자 할 때는 위험 영역을 넣어 테스트한다.

예컨대, 앞서의 은행계좌 프로그램을 작성한다고 하자. 프로그램 코드를 계좌 열기, 위험 영역과 계좌를 갱신하는 영역 형태로 분류해야 한다.

경합 조건에 이어 위험 영역에서의 동시성 실행을 알아봤다. 코드를 이해함으로써 위험 영역을 알게 되고, 추후 살펴볼 프리미티브를 이용한 영역을 정확히 보호할 수 있다.

파일시스템

경합 조건은 프로그램뿐만 아니라 파일시스템에도 영향을 준다. 이는 2개의 프로세스가 동시에 파일시스템상의 파일을 변경할 때 발생한다. 파일 자체에 적절한 동기화 제어가 없다면, 파일의 충돌로 프로세스 간 작성이 계속돼 파일을 사용할 수 없을 것이다.

생명과 직결되는 시스템

경합 조건이 소프트웨어에 영향을 준 최악의 예는 Therac-25 방사선 치료기 제어 소프트웨어다. 이는 불행히도, 치료 중인 환자 3명의 목숨을 앗아갔다.

오늘날, 의료기기처럼 생명과 직결되는 시스템life-critical system에 사용되는 소프트웨어는 매우 중요하다. 하지만 때론 소프트웨어 자체에 영향을 받지 않고자 매번 확인해야 하는 경고 차원의 역할도 한다.

▲ 출처: wikipedia.org

공유 자원과 데이터 경합

동시성 애플리케이션을 구현할 때 주의해야 할 부분 중 하나는 경합 조건이다. 이러한 경합 조건은 애플리케이션에 치명적인 영향을 주며, 수정하기 힘든 버그로 이어질 수 있다. 이러한 문제를 막고자 경합 조건이 어떻게 발생하는지 이해하고, 4장에서 다루는 동기화 프리미티브를 사용해 이를 막는 방법을 알아야 한다.

동기화 및 기본적인 프리미티브에 대해 알아보는 것은 파이썬에서 스레드 및 고성능 프로그램을 만들 때 특히 중요하다. 다행히, 파이썬은 다양한 동시성 상황에 유용한 스레딩 모듈 내의 여러 가지 동기화 프리미티브를 지원한다.

이번 절에서는 다양한 동기화 프리미티브에 대해 알아보고, 이에 해당하는 예제를 프로그램 내에서 어떻게 사용하는지 살펴본다. 마지막에는, 올바른 스레드 방식으로 자원에 접근하는 동시성 파이썬 프로그램을 구현해본다.

Join 메소드

중요한 기업용 시스템을 개발할 때는 실행 순서를 꼭 숙지해야 한다. 다행히, 파이썬의 스레드 객체는 `join` 메소드로 이를 다루도록 지원하고 있다.

join 메소드는 스레드가 종료될 때까지 부모 스레드가 더 진행되지 않게 막아준다. 이는 자연적으로 종료 상태가 되거나, 처리되지 않은 예외 경우를 발생시킨다. 다음 예제를 살펴보자.

```
import threading
import time
def ourThread(i):
  print("Thread {} Started".format(i))
  time.sleep(i*2)
  print("Thread {} Finished".format(i))
def main():
  thread1 = threading.Thread(target=ourThread, args=(1,))
  thread1.start()
  print("Is thread 1 Finished?")
  thread2 = threading.Thread(target=ourThread, args=(2,))
  thread2.start()
  thread2.join()
  print("Thread 2 definitely finished")
if __name__ == '__main__':
  main()
```

코드에 관한 설명

위의 예제에서는 join 메소드를 이용한 결정 가능한 스레드 프로그램의 흐름이 어떻게 되는지 볼 수 있다.

myThread라는 간단한 함수에 1개의 파라미터를 정의한다. 앞선 모든 함수가 그렇듯이 시작 시 출력 후 파라미터의 2배의 시간 동안 유휴 상태에 들어가며, 종료 시 한 번 더 출력한다.

main 함수에는 2개의 스레드를 정의해 하나는 thread1 및 1을 인자로 한다. 다음으로 스레드를 실행하고, print 문을 실행한다. 여기서 중요한 부분은 thread1이 완료되기 전에 출력문을 실행하는 것이다.

다음으로 두 번째 스레드 객체인 thread2를 생성하고, 이번에는 2를 인자로 한다. 여기서 다른 점은 스레드를 시작한 후 thread2.join( )을 호출하는 것이다. thread2를 호출함으로써 print 문을 실행하고, thread2가 종료된 후 Thread 2 definitely finished가 출력된다.

모아보기

join 메소드는 유용하며 빠르고 확실하게 코드상의 순서를 보장하는데, 멀티스레드 코드로 만듦으로써 얻는 이득에 대해 확실히 이해해보자.

위 예제의 thread2 객체를 생각해보자. 멀티스레딩을 통해 무엇을 얻었는가? 이는 꽤 간단한 프로그램인데, 시작한 후 바로 조인한 부분은 thread2가 완료될 때까지 첫 번째 스레드를 묶고 있는다. 본질적으로, 멀티스레드 애플리케이션은 thread2가 실행하고 있는 동안 단일 스레드처럼 보이는 것이다.

락

락lock이란 다수의 스레드 실행으로 공유 자원에 접근할 때 필요한 메커니즘이다. 이를 이해하려면 1개의 화장실과 여러 명의 룸메이트를 생각하면 된다. 한 명이 화장실에서 샤워를 하려면 다른 사람이 볼 수 없게 문을 잠가야 한다.

파이썬 락은 화장실 문을 잠그는 동기화 프리미티브다. 이는 '락' 및 '언락' 상태가 있으며, '언락' 상태에서만 락을 요청할 수 있다.

예제

2장에서 다음 코드를 살펴봤다.

```
import threading
import time
import random
```

```
counter = 1
def workerA( ):
  global counter
  while counter < 1000:
    counter += 1
    print("Worker A is incrementing counter to {}".format(counter))
    sleepTime = random.randint(0,1)
    time.sleep(sleepTime)
def workerB( ):
  global counter
  while counter > -1000:
    counter -= 1
    print("Worker B is decrementing counter to {}".format(counter))
    sleepTime = random.randint(0,1)
    time.sleep(sleepTime)
def main( ):
  t0 = time.time( )
  thread1 = threading.Thread(target=workerA)
  thread2 = threading.Thread(target=workerB)
  thread1.start( )
  thread2.start( )
  thread1.join( )
  thread2.join( )
  t1 = time.time( )
  print("Execution Time {}".format(t1-t0))
if __name__=='__main__':
  main( )
```

위 예제에서는 카운터를 증감하고자 2개의 스레드를 일정하게 사용하고 있다. 락을 추가해 카운터가 안전한 상태로 스레드에 접근하게 한다.

```
Import threading
Import time
Import random
counter = 1
lock = threading.Lock( )
```

```
def workerA():
  global counter
  lock.acquire()
  try:
    while counter < 1000:
      counter += 1
      print("Worker A is incrementing counter to {}".format(counter))
      sleepTime = random.randint(0,1)
      time.sleep(sleepTime)
  finally:
    lock.release()
def workerB():
  global counter
  lock.acquire()
  try:
    while counter > -1000:
      counter -= 1
      print("Worker B is decrementing counter to {}".format(counter))
      sleepTime = random.randint(0,1)
      time.sleep(sleepTime)
  finally:
    lock.release()
def main():
  t0 = time.time()
  thread1 = threading.Thread(target=workerA)
  thread2 = threading.Thread(target=workerB)
  thread1.start()
  thread2.start()
  thread1.join()
  thread2.join()
  t1 = time.time()
  print("Execution Time {}".format(t1-t0))
if __name__=='__main__':
  main()
```

코드에 관한 설명

앞의 코드에서는 2개의 함수 내에 while 반복문을 모두 넣은 간단한 락 프리미티브를 구현했다. 스레드가 시작되면, 두 함수 모두 락을 요청하고자 경합해 각자의 목표에 도달하게 되며, 다른 스레드 사용 없이 카운터를 1,000 및 −1,000을 기준으로 증감한다. 이는 1개의 스레드가 목표를 완료하고 다른 스레드가 락을 요청할 수 있고, 카운터를 증감할 수 있을 때 실행된다.

앞의 코드는 설명을 진행하고자 천천히 실행되는데, while 문 내의 time.sleep( ) 호출을 제거하면 정상적으로 실행된다.

R락

리엔트런트[1] 락reentrant-locks, 즉 R락RLocks이란 일반적인 락 프리미티브에서 작동하는 동기화 프리미티브인데, 스레드가 이미 갖고 있다면 여러 번 요청된다.

예컨대, 스레드 1이 R락을 요청할 때마다 R락 프리미티브 내의 카운터는 1씩 증가된다. 스레드 2가 실행되고 R락을 요청하면, 요청하기 전 R락의 카운터는 0으로 감소될 때까지 대기 상태가 된다. 스레드 2는 이러한 0 조건이 충족될 때까지 블로킹 상태가 된다.

그렇다면, 왜 이런 방식이 유용할까? 예컨대, 다른 클래스 메소드에 접근하는 클래스 내의 메소드에 스레드 세이프thread-safe 접근을 이용할 때 쓸모가 있다.

예제

다음 코드를 살펴보자.

```
import threading
import time
class myWorker( ):
```

1 한 번 로드하면 여러 가지 메인 루틴에서 자유롭게 사용할 수 있는 구조 – 옮긴이

```
def __init__(self):
  self.a=1
  self.b=2
  self.Rlock=threading.RLock()
def modifyA(self):
  with self.Rlock:
    print("Modifying A : RLockAcquired:
    {}".format(self.Rlock._is_owned()))
    print("{}".format(self.Rlock))
    self.a = self.a + 1
    time.sleep(5)
def modifyB(self):
  with self.Rlock:
    print("Modifying B : RLock Acquired:
    {}".format(self.Rlock._is_owned()))
    print("{}".format(self.Rlock))
    self.b = self.b - 1
    time.sleep(5)
def modifyBoth(self):
  with self.Rlock:
    print("Rlock acquired, modifying A and B")
    print("{}".format(self.Rlock))
    self.modifyA()
    self.modifyB()
  print("{}".format(self.Rlock))
workerA = myWorker()
workerA.modifyBoth()
```

코드에 관한 설명

위 코드에서는 단일 스레드 프로그램 내의 R락을 작동하는 방법에 대해 살펴봤다. myWorker 클래스를 정의하는데, 이는 4개의 함수로 이뤄진 R락 및 a와 b 변수를 초기화하는 생성자다.

다음으로, a와 b 변수를 변경하는 2개의 함수를 각각 정의한다. 이는 모두 with 문을 사용해 R락 클래스를 요청하고, 내부 변수에 변경이 필요하다면 실행한다.

마지막으로, modifyBoth 함수에서 modifyA와 modifyB 함수를 호출하기 전, 초기 R락 요청을 실행한다.

각 단계마다 R락의 상태를 출력한다. modifyBoth 함수 내에서 R락이 요청된 후, 해당 요소는 메인 스레드로 가며, 카운터는 1로 증가한다. 다음으로 modifyA를 호출하면 R락 카운터는 다시 1 증가되며, modifyA가 R락을 해제하기 전 모든 연산이 수행된다. modifyA가 R락을 해제하면, modifyB 함수에 의해 다시 2로 증가되기 전 카운터가 1 감소된다.

끝으로, modifyB는 실행을 끝내고, R락을 해제하며, modifyBoth 함수도 마찬가지로 실행된다. R락 객체의 모든 출력을 마친 후, 소유자는 0으로 세팅되며, 카운터 또한 0이 된다. 이론적으로는 또 다른 스레드가 해당 락을 가져갈 부분이다.

출력

출력은 다음과 같다.

```
$ python3.6 04_rlocks.py
Rlock acquired, modifying A and B
<locked _thread.RLock object owner=140735793988544 count=1 at 0x10296e6f0>
Modifying A : RLock Acquired: True
<locked _thread.RLock object owner=140735793988544 count=2 at 0x10296e6f0>
<locked _thread.RLock object owner=140735793988544 count=1 at 0x10296e6f0>
Modifying B : RLock Acquired: True
<locked _thread.RLock object owner=140735793988544 count=2 at 0x10296e6f0>
<unlocked _thread.RLock object owner=0 count=0 at 0x10296e6f0>
```

R락과 일반적인 락

기존의 락 프리미티브를 이용해 동일한 프로그램을 실행하고 있다면, modifyA() 함수를 실행하는 부분에 절대 도달하지 않을 것이다. 기본적으로 이 프로그램은 교착상태에 들어가는데, 이는 스레드가 더 나아갈 수 있는 릴리스 메커니즘을 구현하지 않았기 때문이다.

다음 코드를 살펴보자.

```
import threading
import time
class myWorker( ):
def__init__(self):
  self.a=1
  self.b=2
  self.lock = threading.Lock( )
def modifyA(self):
  with self.lock:
    print("Modifying A : RLock Acquired: {}".format(self.lock._is_owned( )))
    print("{}".format(self.lock))
    self.a = self.a + 1
    time.sleep(5)
def modifyB(self):
  with self.lock:
    print("Modifying B : Lock Acquired: {}".format(self.lock._is_owned( )))
    print("{}".format(self.lock))
    self.b = self.b - 1
    time.sleep(5)
def modifyBoth(self):
  with self.lock:
    print("lock acquired, modifying A and B")
    print("{}".format(self.lock))
    self.modifyA( )
    print("{}".format(self.lock))
    self.modifyB( )
  print("{}".format(self.lock))
workerA=myWorker( )
workerA.modifyBoth( )
```

R락은 코드상 릴리스 락 로직 및 복잡한 요청 구현 없이 재귀적인 방식으로 스레드를 안정적으로 실행하고 있다. 쉽게 코드를 작성할 수 있게 해주며, 유지 관리도 쉽다.

컨디션

컨디션condition이란 다른 스레드의 신호를 기다리는 동기화 프리미티브다. 예컨대, 해당 스레드가 실행을 마쳐야지만 현재 스레드가 나머지 계산을 수행할 수 있다.

정의

파이썬 내장 라이브러리의 condition 객체 정의를 살펴보자. 기본적인 프리미티브에 대해 이해하고 세세한 부분까지 파고들어, 시간이 된다면 전체 코드를 살펴보는 것도 추천한다.

```
def Condition(*args, **kwargs):
  """Factory function that returns a new condition variable object.
  A condition variable allows one or more threads to wait until they are
  Notified by another thread.
  If the lock argument is given and not None, it must be a Lock or RLock
  object, and it is used as the underlying lock. Otherwise, a new RLock object
  is created and used as the underlying lock.
```

컨디션의 장점을 보려면 프로듀서producer 및 컨슈머consumer를 적용해보자. 프로듀서의 경우 메시지를 큐queue에게 전달하고 그 밖의 스레드에게 알리는 역할을 하며, 컨슈머의 경우 큐에서 대기해온 메시지가 존재한다.

예제

이 예제에서는 스레드 클래스에서 상속받은 2개의 클래스를 생성해본다. 이 클래스는 각각 발행자publisher 및 구독자subscriber 클래스가 된다. 발행자는 정수를 정수 배열에 전달하고, 구독자에게 배열로부터 넘겨받은 새로운 정수가 있다는 걸 알린다.

Publisher

Publisher 클래스는 정수의 기준을 취하는 생성자와 컨디션 프리미티브인 2개의 함수를 정의한다.

run 함수는 0과 1000 사이의 정수를 무한 반복해 무작위로 계속 생성한다. 정수가 생성되면 컨디션을 요청하고, 이를 정수 배열에 추가한다.

배열에 추가한 후, 구독자에게 새로운 배열 요소를 먼저 알리고 컨디션을 릴리스한다.

```
class Publisher(threading.Thread):
  def __init__(self, integers, condition):
    self.condition = condition
    self.integers = integers
    threading.Thread.__init__(self)
def run(self):
  while True:
    integer = random.randint(0,1000)
    self.condition.acquire()
    print("Condition Acquired by Publisher: {}".format(self.name))
    self.integers.append(integer)
    self.condition.notify()
    print("Condition Released by Publisher: {}".format(self.name))
    self.condition.release()
    time.sleep(1)
```

Subscribe

Subscribe 클래스도 마찬가지로 생성자와 run 함수가 있다. 생성자는 소비될 정수 배열의 기준과 컨디션 동기화 프리미티브를 취한다.

run 함수 내부에서는 컨디션 요청을 계속적으로 한다. 해당 락을 요청하고자 하면, 스레드가 이를 요청했다는 부분을 출력하고 정수 배열에서 첫 번째 정수를 '팝pop'한다. 성공적으로 끝나면, 컨디션 프리미티브를 릴리스하고, 다시 한번 컨디션을 재요청한다.

```
class Subscriber(threading.Thread):
  def __init__(self, integers, condition):
    self.integers = integers
    self.condition = condition
```

```
    threading.Thread.__init__(self)
  def run(self):
    while True:
      self.condition.acquire()
      print("Condition Acquired by Consumer: {}".format(self.name))
      while True:
        if self.integers:
          integer = self.integers.pop()
          print("{} Popped from list by Consumer: {}".format(integer, self.name))
          break
        print("Condition Wait by {}".format(self.name))
        self.condition.wait()
      print("Consumer {} Releasing Condition".format(self.name))
      self.condition.release()
```

시작하기

main 함수에서 메시지 큐 역할을 하는 정수 배열을 먼저 선언한다. 그 후 컨디션 프리미티브를 선언한다.

마지막으로, 1개의 발행자와 2개의 구독자를 정의한다. 다음으로 모두 실행해 스레드와 조인하며, 스레드가 실행하기 전까지 종료되지 않게 한다.

```
def main():
  integers = []
  condition = threading.Condition()
  # 발행자
  pub1 = Publisher(integers, condition)
  pub1.start()
  # 구독자
  sub1 = Subscriber(integers, condition)
  sub2 = Subscriber(integers, condition)
  sub1.start()
  sub2.start()
  # 스레드 조인
  pub1.join()
```

```
  consumer1.join()
  consumer2.join()
if __name__=='__main__':
  main()
```

결과

프로그램을 실행하면 다음과 같은 출력을 볼 수 있다. 발행자가 컨디션을 요청하면, 배열에 숫자를 추가하며 컨디션에게 알리고 릴리스한다.

```
$ python3.6 03_pubSub.py
Condition Acquired by Publisher: Thread-1
Publisher Thread-1 appending to array: 108
Condition Released by Publisher: Thread-1
Condition Acquired by Consumer: Thread-2
108 Popped from list by Consumer: Thread-2
Consumer Thread-2 Releasing Condition
Condition Acquired by Consumer: Thread-2
Condition Wait by Thread-2
Condition Acquired by Consumer: Thread-3
Condition Wait by Thread-3
Condition Acquired by Publisher: Thread-1
Publisher Thread-1 appending to array: 563
...
```

컨디션을 알릴 때, 2개의 구독자가 이 컨디션을 먼저 요청할 경우 둘 사이에 경합이 일어난다. 1개가 싸움에서 이기면, 배열에서 숫자를 '팝'하게 된다.

세마포어

1장에서 동시성의 역사와 함께 데이크스트라라는 인물도 살펴봤다. 그는 철도 시스템으로부터 세마포어semaphores라는 개념을 끌어와 복잡한 동시성 시스템에 적용한 인물이다.

세마포어란 요청 및 릴리스 호출이 있을 때마다 증감되는 내부 카운터다. 초기화된 카운터는 다른 세팅이 없으면 기본적으로 1이다. 세마포어 카운터는 음수가 될 수 없다.

세마포어를 통해 코드를 블로킹한다면 세마포어 값은 2가 된다. 한 스레드가 세마포어를 요청하면, 세마포어 값은 1로 감소된다. 그 외 스레드가 세마포어를 요청하면, 값은 0으로 감소된다. 이때 또 다른 스레드가 요청해도 세마포어는 기존의 스레드 2개가 릴리스될 때까지 요청을 거절하며, 카운터는 계속 0이 된다.

클래스 정의

파이썬 세마포어 객체의 정의 부분은 다음과 같다.

```
class _Semaphore(_Verbose):
  # Tim Peters' 세마포어 클래스 이후이나 동일하지 않다(최댓값은 아니다).
  def __init__(self, value=1, verbose=None):
    if value < 0:
      raise ValueError("semaphore initial value must be >= 0")
    _Verbose.__init__(self, verbose)
    self.__cond = Condition(Lock())
    self.__value = value
```

semaphore 클래스의 생성자 함수에서 별도의 세팅이 없으면 기본적으로 1의 값을 취한다.

코드상의 주석에서 세마포어를 다음과 같이 정의한다.

> 세마포어는 release() 호출 횟수에서 acquire() 호출 횟수를 빼고 초깃값을 더한 값에 따라 카운터를 관리한다. acquire() 메소드는 카운터를 양수 범위에서 반환될 때까지 블로킹한다. 아무런 값이 없으면, 기본값은 1이다.

예제

다음 예제는 스탠퍼드 컴퓨터공학에서 온 동시성 코드다. 이 예제는 4개의 각 스레드가 티켓이 매진될 때까지 모든 티켓 배당을 최대로 판매하도록 구성된 '티켓 판매 프로그램'이다.

TicketSeller 클래스

먼저 TicketSeller 클래스를 구현해보자. 이 클래스는 자체 내부 카운터로 몇 개의 티켓이 판매됐는지 파악한다. 생성자에서 스레드를 초기화하고 세마포어의 기준을 정의한다. run 함수에서는 판매 가능한 티켓의 수가 0 이하이면 세마포어를 요청하고, 0보다 크면 ticketSeller가 판매한 티켓 숫자를 증가시키며, ticketsAvailable은 1 감소시킨다. 그 후, 세마포어를 릴리스하고 과정을 출력한다.

```
Class TicketSeller(threading.Thread):
ticketsSold = 0
def __init__(self, semaphore):
  threading.Thread.__init__(self)
  self.sem = semaphore
  print("Ticket Seller Started Work")
def run(self):
  global ticketsAvailable
  running = True
  while running:
    self.randomDelay()
    self.sem.acquire()
    if(ticketsAvailable <= 0):
      running = False
    else:
      self.ticketsSold = self.ticketsSold + 1
      ticketsAvailable = ticketsAvailable - 1
      print("{} Sold One ({} left)".format(self.getName(), ticketsAvailable))
    self.sem.release()
  print("Ticket Seller {} Sold {} tickets in total".format(self.getName(),
```

```
self.ticketsSold))
def randomDelay(self):
  time.sleep(random.randint(0,1))
```

위 코드에서는 TicketSeller 클래스를 정의한다. 이 클래스는 전역 세마포어 객체의 레퍼런스를 갖는 생성자를 구성하며, 스레드는 초기화된다. run 함수에는 0에서 1초 사이에 블로킹하는 while 반복문을 정의해 세마포어를 요청한다. 세마포어 요청을 성공했다면 판매할 티켓이 있는지 확인한다. 판매할 티켓이 있다면 ticketsSold 개수를 증가시키고, ticketsAvailable은 감소시킨 후, 콘솔 창에 결과를 출력한다.

이제까지 TicketSeller 클래스를 정의했는데, 다음과 같이 TicketSeller의 모든 인스턴스를 지나는 세마포어 객체를 먼저 생성해야 한다.

```
# 세마포어 프리미티브
Semaphore = threading.Semaphore()
# 티켓 배당
ticketsAvailable = 10
# 배열 생성
sellers = []
for i in range(4):
  seller = TicketSeller(semaphore)
  seller.start()
  sellers.append(seller)
# 모든 스레드 조인
For seller in sellers:
  seller.join()
```

출력

위의 프로그램을 실행하면, 다음과 같이 출력된다. 특정 실행에서는 4개의 스레드 사이에 균등하게 판매 개수가 배분됐을 수 있다. 이 중 한 스레드가 어느 정도의 시간 동안 블로킹한다면, 또 다른 스레드는 세마포어를 요청하고 티켓을 판매할 것이다.

```
$ python3.6 06_semaphores.py
Ticket Seller Started Work
Thread-1 Sold One (9 left)
Ticket Seller Started Work
Ticket Seller Started Work
Ticket Seller Started Work
Thread-1 Sold One (8 left)
Thread-3 Sold One (7 left)
Thread-3 Sold One (6 left)
Thread-4 Sold One (5 left)
Thread-2 Sold One (4 left)
Thread-1 Sold One (3 left)
Thread-4 Sold One (2 left)
Thread-2 Sold One (1 left)
Thread-3 Sold One (0 left)
Ticket Seller Thread-4 Sold 2 tickets in total
TicketSeller Thread-1 Sold 3 tickets in total
Ticket Seller Thread-2 Sold 2 tickets in total
Ticket Seller Thread-3 Sold 3 tickets in total
```

스레드 경합

위 예제에서 run 함수 내의 self.randomDelay에 의해 주석 처리된 블로킹 스레드 없이 실행된다면, 스레드는 계속 세마포어를 요청해 모든 티켓이 팔릴 것이다. 세마포어를 차지한 스레드가 다른 스레드보다 먼저 락을 재요청해 우선순위에 있기 때문이다.

한정된 세마포어

한정된 세마포어와 일반적인 세마포어를 비교하며 알아보자.

한정된 세마포어는 현재 값이 초깃값을 초과하는지 확인한다. 만약 초과하지 않는다면 ValueError가 발생한다. 대부분의 상황에서 세마포어는 한정된 능력으로부터 자원을 보호한다.

세마포어가 여러 번 릴리스된다면 버그가 생길 수 있다. 값이 주어지지 않는다면 기본적으로 1이다.

일반적으로, 한정된 세마포어는 수많은 사람들이 한곳에 집중되거나, 특정 작업을 한 번에 수행하는 등의 자원 소비를 막고자 웹 서버와 데이터베이스에 주로 구현된다.

보통 일반적인 세마포어와는 반대로 한정된 세마포어가 주로 사용된다. 앞의 코드에서 `threading.BoundedSemaphore(4)`로 수정하고 실행하면, 해결되지 않던 간단한 오류는 해결될 수도 있다(나머지는 동일하게 동작된다).

이벤트

이벤트는 동시적으로 실행되는 여러 스레드 사이의 간단한 통신 형태에 유용하다. 일반적으로, 한 스레드가 다른 스레드의 신호를 받아들일 때 이벤트가 발생한다.

이벤트는 참, 거짓이 가능한 내부 플래그로 구성된 객체가 필수적으로 존재한다. 스레드는 이벤트 객체의 상태를 계속 확인하며, 언제 어떤 방식으로 플래그 상태를 바꿀지 정한다.

파이썬 내장 함수에는 스레드를 완벽히 종료하는 메커니즘이 없다고 3장에서 언급했다. 하지만 이벤트 객체를 활용해, 이벤트 객체가 해제 상태일 때 스레드가 실행되게 할 수 있다. SIGKILL 시그널[2]이 송출된 곳에는 유용하지 않지만, 프로그램 종료 및 스레드를 해제하기 전 대기하는 상황에서는 유용하다.

이벤트에는 수정 및 활용 가능한 4개의 퍼블릭public 함수가 존재한다.

- `isSet()`: 이벤트가 세팅됐는지 확인한다.
- `set()`: 이벤트를 세팅한다.
- `clear()`: 이벤트 객체를 리셋한다.
- `wait()`: 내부 플래그가 참이 될 때까지 블로킹한다.

2 프로세스를 무조건 강제 종료하는 리눅스 커널 시그널 – 옮긴이

예제

다음 예제에서는 이벤트 객체를 바탕으로 자식 스레드를 다루는 방법을 살펴보고, 효과적인 종료 방법을 알아본다.

```
import threading
import time
def myThread(myEvent):
  while not myEvent.is_set():
    print("Waiting for Event to be set")
    time.sleep(1)
  print("myEvent has been set")

def main():
  myEvent = threading.Event()
  thread1 = threading.Thread(target=myThread, args=(myEvent,))
  thread1.start()
  time.sleep(10)
  myEvent.set()
if __name__=='__main__':
  main()
```

코드에 관한 설명

위 코드에서는 myThread 함수를 정의했는데, 이 함수의 while 반복문은 이벤트 객체 함수가 해제되는 동안만 실행된다. 반복문 내에서는 1초 간격으로 이벤트를 대기하고 있다는 문장이 출력된다.

main 함수에서는 모든 자식 객체를 지니는 이벤트 객체를 정의한다. 따라서 myEvent = threading.Event()를 호출하고, 이벤트 객체를 새로 인스턴스화한다.

myEvent 객체 내의 스레드 객체도 인스턴스화하고 실행한다. myEvent 신호를 세팅하기 전에 10초 동안 대기해 자식 스레드 실행을 마친다.

배리어

배리어[barrier]란 버전 3 파이썬 언어에서 소개된 동기화 프리미티브로, 컨디션 및 세마포어의 복잡한 조합으로 해결된 문제를 처리한다.

이러한 배리어는 모든 스레드 내의 작업이 스레드 그룹에 의해 생성됐는지 보장하는 중요한 부분이다.

복잡하고 불필요하게 보일 수 있으나, 특정 상황에서는 꼭 필요하며 코드 가독성도 높인다.

예제

다음 예제에서는 배리어를 활용해 모든 스레드가 원하는 실행 포인트에 도달하기까지 해당 스레드의 실행을 막아보자.

```
import threading
import time
import random
class myThread(threading.Thread):
  def __init__(self, barrier):
    threading.Thread.__init__(self)
    self.barrier = barrier
  def run(self):
    print("Thread {} working on something".format(threading.current_thread()))
    time.sleep(random.randint(1,10))
    print("Thread {} is joining {} waiting on Barrier".format(threading.current_
thread(), self.barrier.n_waiting))
    self.barrier.wait()
    print("Barrier has been lifted, continuing with work")
barrier = threading.Barrier(4)
threads = []
for i in range(4):
  thread = myThread(barrier)
  thread.start()
  threads.append(thread)
```

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

코드에 관한 설명

위 코드를 살펴보면, threading.Thread로부터 상속받은 myThread 사용자 클래스를 정의한다. 이 클래스에서 __init__ 함수와 run 함수를 정의한다. __init__ 함수는 배리어 객체를 가져 추후 기준을 정할 수 있게 한다.

run 함수는 1에서 10초 사이의 무작위 시간 동안 동작하는 스레드를 실행하며, 배리어에서는 대기한다.

클래스 정의를 마치면, barrier = threading.Barrier(4) 호출로 배리어 객체를 생성한다. 인자로 들어가는 4는 배리어가 해제되기 전, 배리어에서 대기하는 스레드의 개수를 나타낸다. 그 후 4개의 스레드를 각각 정의해 모두 조인한다.

출력

위의 코드를 실행해보면, 다음과 같이 출력이 나타난다.

4개의 스레드가 하나씩 무작위로 배리어 객체 내에 대기하는 상태로 작동됨을 볼 수 있다. 네 번째 스레드가 대기 상태에 들어가면 프로그램이 종료되기 직전이며, 모든 4개의 스레드는 마지막 출력문을 실행하고 배리어는 해제된다.

```
$ python3.6 08_barriers.py
Thread <myThread(Thread-1, started 123145344643072)> working on something
Thread <myThread(Thread-2, started 123145349898240)> working on something
Thread <myThread(Thread-3, started 123145355153408)> working on something
Thread <myThread(Thread-4, started 123145360408576)>working on something
Thread <myThread(Thread-1, started 123145344643072)> is joining 0
waiting on Barrier
Thread <myThread(Thread-3, started 123145355153408)> is joining 1
waiting on Barrier
```

```
Thread <myThread(Thread-2, started 123145349898240)> is joining 2
waiting on Barrier
Thread <myThread(Thread-4, started 123145360408576)> is joining 3
waiting on Barrier
Barrier has been lifted, continuing with work
Barrier has been lifted, continuing with work
Barrier has been lifted, continuing with work
Barrier has been lifted, continuing with work
```

요약

4장에서는 동시성 파이썬 애플리케이션에 영향을 주는 핵심 요소에 대해 살펴봤다. 교착 상태와 철학자의 저녁식사 문제를 파고들며, 소프트웨어에 어떤 영향을 주는지 알아봤다.

이제 파이썬 동기화 프리미티브와 이를 이용하는 방법을 명확히 이해해야 한다. 5장에서는 멀티스레드 및 멀티프로세스 애플리케이션 간 통신을 구현하는 방법을 자세히 살펴보겠다.

05

스레드 간의 통신

통신은 동시성 시스템에서 중요한 부분이다. 적절한 통신 메커니즘 구현 없이는 동시성 및 병렬화의 어떤 성능 효과도 얻기 힘들다. 스레드 및 프로세스 간에 통신을 할 때 가장 큰 문제가 되는데, 그 전에 모든 상황과 옵션을 이해하는 것이 필수다.

5장에서는 자체 통신 메커니즘을 구현하는 다양한 방법을 배우고, 이러한 메커니즘이 언제 어디서 사용되는지 알아보자.

5장에서 다루는 내용은 다음과 같다.

- 파이썬의 기본적인 자료 구조와 스레드 세이프 방식으로 상호작용하기
- 큐를 사용하는 스레드 세이프 통신과 queue 객체를 효과적으로 사용하기
- 더블 앤드 큐와 기존의 큐에 대한 이해
- 새로운 개념을 활용하고 본인만의 멀티스레드 웹사이트 크롤러Crawler 만들기

기본적인 자료 구조

파이썬의 기존 자료 구조는 기본적으로 다양한 스레드 세이프thread-safe 형태를 띠고 있다. 하지만 스레드의 안정성을 보장하고자 이러한 자료 구조에 접근을 통제하는 락 메커니즘을 정의해보자.

세트

내가 파이썬에서 멀티스레드 간의 통신을 작업할 때, 스레드 세이프 방식으로 set 클래스를 이용하는 가장 좋은 방법은 set 클래스를 확장하는 것이다. 더군다나 내가 원하는 방식의 락 메커니즘을 구현할 수 있었다.

클래스 확장

파이썬을 이용한다면 클래스를 확장하는 부분은 어렵지 않다. 기존의 파이썬 set 클래스로부터 상속받는 LockedSet 클래스 객체를 정의한다. 해당 클래스의 생성자에서 lock 객체를 생성하고, 스레드 세이프 상호작용을 위해 다음 함수에서 사용해본다.

다음으로 add, remove, contains 함수를 정의한다. 이들은 한 가지 예외 형태로 super 클래스 함수에 의지한다. 각 함수 내의 모든 상호작용이 주어진 시간에 1개의 스레드만 실행되도록 생성자를 초기화해, 스레드 세이프를 구현한다.

현재의 set 클래스를 그 외 파이썬 프리미티브에 확장하는 부분도 동일한 테크닉을 사용한다. 이로써 최소한의 작업으로 해당 클래스의 함수를 이용할 수 있다.

다음 예제는 스택 오버플로Stack Overflow의 문제에서 가져왔다(http://stackoverflow.com/a/13618333/2903188). 다음 코드는 이 장에서 다루는 중요한 메소드를 잘 설명하고 있다.

```
class LockedSet(set):
  """A set where add(), remove(), and 'in' operator are thread-safe"""
```

```
def __init__(self, *args, **kwargs):
  self._lock = Lock()
  super(LockedSet, self).__init__(*args, **kwargs)

def add(self, elem):
  with self._lock:
    super(LockedSet, self).add(elem)

def remove(self, elem):
  with self._lock:
    super(LockedSet, self).remove(elem)

def __contains__(self, elem):
  with self._lock:
    super(LockedSet, self).__contains__(elem)
```

이렇게 클래스를 확장하고 스레드 세이프 로직을 추가하는 형태는 대부분의 파이썬 프리미티브에서 행해진다. 이는 클래스에 요소를 효과적으로 활용하는 방법이지만, 스레드 안정성 구현이 올바른지 확인해볼 필요가 있다.

연습: 기타 프리미티브 확장

여기서는 그 외 클래스를 확장하고 스레드 세이프 구성을 구현하자. 리스트 내의 값이 1 증가할 때 스레드 안정성을 제공하고자 리스트List 프리미티브를 확장하는 방법을 살펴보자. 이는 다양한 파이썬 프리미티브에 동일한 형태로 어떻게 적용할 수 있는지 보여준다.

데코레이터

현재 파이썬 프리미티브를 확장하는 게 가장 유용할지 모르지만, 스레드 세이프 통신을 위해서라면 그 밖의 기술적인 방법도 찾아봐야겠다.

그중에서 중요한 방법으로 데코레이터decorator를 활용하는 것이 있다. 이러한 메커니즘은 locked_method를 취하는 decorator 메소드를 정의한다. 이 decorator 메소드는 내부에

새로운 메소드를 정의하고, self._lock 요청이 있을 때만 기존 메소드를 호출한다.

비효율적으로 보이지만, 이는 코드 내부상의 오류 부분을 스레드 세이프 형태로 바꿔, 경합 조건 걱정 없이 호출될 수 있게 한다.

다음 예제에서 메소드 내를 거치며 경합 조건을 막으면서 반환하는 decorator 메소드를 구현해보자.

```
def lock(method):

  def newmethod(self, *args, **kwargs):
    with self._lock:
      return method(self, *args, **kwargs)
  return newmethod

class DecoratorLockedSet(set):
  def __init__(self, *args, **kwargs):
    self._lock = Lock()
    super(DecoratorLockedSet, self).__init__(*args, **kwargs)

  @locked_method
  def add(self, *args, **kwargs):
    return super(DecoratorLockedSet, self).add(elem)

  @locked_method
  def remove(self, *args, **kwargs):
    return super(DecoratorLockedSet, self).remove(elem)
```

클래스 데코레이터

클래스 데코레이션은 이전 예제에서 한 발짝 더 나아가, 메소드 1개만 보호하는 걸 넘어 클래스 내의 모든 함수를 보호해 스레드 세이프 방식으로 모두 호출한다.

다음 예제에서 어떻게 클래스 데코레이터 함수를 구현하는지 살펴보자. 처음의 lock_class 함수는 메소드 리스트와 lockfactory를 인자로 해, 데코레이터로 명시된 메소드 이름과 lockFactory를 취하는 lambda 함수를 반환한다.

이는 make_threadsafe를 호출해 클래스를 거치는 인스턴스를 초기화하고, self._lock = lockfactory()라는 새로운 생성자도 정의한다. 이 make_threadsafe 함수는 methodnames의 모든 메소드를 돌며 lock_method 함수로 각 메소드를 락한다.

이는 스레드 안정성을 전체 클래스에 구현하는 가장 간단하고 확실한 방법이며, 원하는 함수를 락할 수 있다.

```
from threading import Lock

def lock_class(methodnames, lockfactory):
  return lambda cls: make_threadsafe(cls, methodnames, lockfactory)

def lock_method(method):
  if getattr(method, '__is_locked', False):
    raise TypeError("Method %r is already locked!" %method)

def locked_method(self, *arg, **kwarg):
  with self._lock:
    return method(self, *arg, **kwarg)
  locked_method.__name__ = '%s(%s)' %
('lock_method', method.__name__)
  locked_method.__is_locked = True
  return locked_method

def make_threadsafe(cls, methodnames, lockfactory):
  init = cls.__init__
  def newinit(self, *arg, **kwarg):
    init(self, *arg, **kwarg)
    self._lock = lockfactory()
  cls.__init__ = newinit
  for methodname in methodnames:
    oldmethod = getattr(cls, methodname)
```

```
    newmethod = lock_method(oldmethod)
    setattr(cls, methodname, newmethod)
  return cls

@lock_class(['add','remove'], Lock)
class ClassDecoratorLockedSet(set):
  @lock_method # 더블 락 메소드라면, TypeError가 발생한다.
  def lockedMethod(self):
    print("This section of our code would be thread safe")
```

리스트

기본적으로 리스트[list]는 스레드에는 안전하나, 이는 리스트에 접근할 때만 그러하다. 리스트 구조에서 나타난 데이터는 사실 보호되지 않으며, 데이터를 안전하게 변경하려면 적절한 락 메커니즘을 구현해 여러 스레드가 실행될 경우 내부적인 경합 조건이 발생하지 않게 해야 한다. 이는 모든 스레드 세이프 컨테이너에서 그러하다.

append() 함수는 리스트 자료 구조에서 스레드 세이프 메소드 중 하나다. 따라서 리스트는 메모리 저장에 있어 임시로 활용하는 가장 빠르고 간단한 구조가 됐다. 하지만 동시성 방식으로 리스트 내의 데이터를 변경하고자 하면, 경합 조건 부작용이 일어날 가능성이 크다.

예컨대, 이러한 부작용 중 하나는 또 다른 스레드와 동시에 리스트 내의 두 번째 요소를 갱신하는 것이다. 하나가 읽히면 스레드가 실행되는 동시에 차례대로 값이 쓰이는데, 부정확하게 변경된 문제를 볼 수 있다.

멀티스레드 애플리케이션에서 리스트를 활용하고자 하면, 이전 절의 세트[set]를 확장한 것과 같은 방식으로 클래스를 확장해야 한다.

큐

큐queue는 여러 가지 형태로 제공되고 있다. 파이썬에서는 내장 queue 모듈에서 일반적인 큐, LifoQueue, PriorityQueue라는 세 가지 형태의 큐를 제공한다.

기본적으로 큐는 파이썬에서 스레드에 안전하며, 이는 애플리케이션에서 큐를 활용할 때 복잡한 락 메커니즘을 구현할 필요가 없다는 뜻이다. 이는 다양한 스레드와 프로세스가 통신하는 통신 매체를 빠르고 간편하게 구현해주는 강력한 무기다.

FIFO 큐

FIFOfirst in first out 큐는 파이썬에서 제공하는 기본적인 큐 구현 형태다. 슈퍼마켓에서 첫 번째 사람이 먼저 계산하고, 두 번째 사람이 차례대로 계산하는 큐 메커니즘을 따른다.

▲ 출처: http://javaworldwide.blogspot.co.uk/2015/07/implementing-blocking-queue-in-java.html

이러한 메커니즘으로 고객의 요청을 공평하게 처리할 수 있고, 대기하는 사람 수에 따라 얼마만큼의 처리 시간이 소요될지 가늠할 수도 있다.

예제

이번 예제에서는 queue.Queue() 객체를 활용해 FIFO 기반의 큐를 구현해보자.

```
import threading
import queue
import random
import time
```

```
def mySubscriber(queue):
  while not queue.empty():
    item = queue.get()
    if item is None:
      break
    print("{} removed {} from the queue".format(threading.current_thread(), item))
    queue.task_done()
    time.sleep(1)

myQueue = queue.Queue()
for i in range(10):
  myQueue.put(i)

print("Queue Populated")

threads = []
for i in range(4):
  thread = threading.Thread(target=mySubscriber, args=(myQueue,))
  thread.start()
  threads.append(thread)

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

코드에 관한 설명

위 예제에서는 파이썬 모듈의 큐를 불러온다. 다음으로 mySubscriber 함수를 정의해 큐에서 사용되는 멀티스레드의 타깃으로 운용한다.

다음의 mySubscriber 함수 선언에서 myQueue = queue.Queue()를 호출해 큐를 선언하며, 0부터 9까지 숫자를 집어넣는다.

마지막으로, 스레드 세이프 큐에서 사용하고자 여러 스레드를 선언하고 인스턴스화한다. 이 스레드를 시작해, 새로 선언한 queue 객체의 인자로 취해 차례대로 조인한다.

출력

코드를 실행하면 4개의 스레드가 처음 큐에 넣었던 순서대로 요소를 큐에서 인출하는 모습을 볼 수 있다. 따라서 큐에서 첫 번째로 위치한 0이 먼저 빠져나온다.

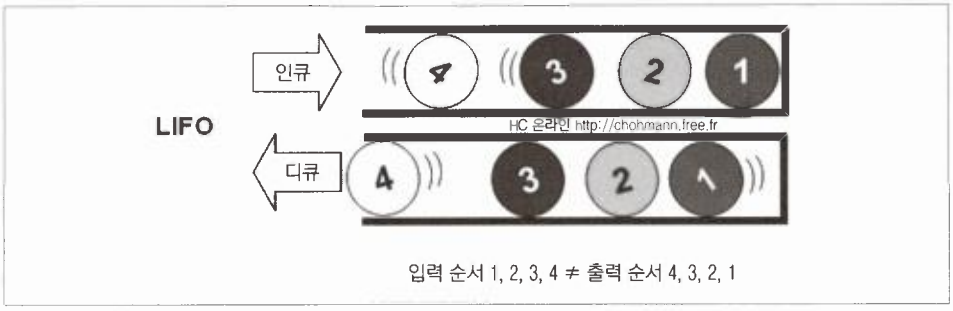
```
$ python3.6 00_queues.py
Queue Populated
<Thread(Thread-1, started 123145445732352)> removed 0 from the queue
<Thread(Thread-2, started 123145450987520)> removed 1 from the queue
<Thread(Thread-3, started 123145456242688)> removed 2 from the queue
<Thread(Thread-4, started 123145461497856)> removed 3 from the queue
<Thread(Thread-1, started 123145445732352)> removed 4 from the queue
<Thread(Thread-3, started 123145456242688)> removed 5 from the queue
<Thread(Thread-4, started 123145461497856)> removed 6 from the queue
<Thread(Thread-2, started 123145450987520)> removed 7 from the queue
<Thread(Thread-1, started 123145445732352)> removed 8 from the queue
<Thread(Thread-3, started 123145456242688)> removed 9 from the queue
```

LIFO 큐

LIFO[last in first out] 큐는 일반적인 큐와는 정반대로 동작한다. 한 번 더 슈퍼마켓에 비유하면, 맨 마지막 사람이 계산대에 먼저 온 사람보다 일찍 계산하게 된다. 상상할 수 있듯이 실제 상황이라면 대기하고 있는 많은 사람으로부터 불평이 나올 것이다.

사람들이 차례대로 나가기 전에 이미 계속 사람이 들어올 수 있는 상황이기에, 먼저 큐에 들어온 2명의 순서는 계속 일정하다. 실제 세계에서는 이러한 메커니즘이 유용하지 않지만, 프로그래밍에서는 유용하다.

LIFO 큐는 깊이 우선 검색[depth-first search], 깊이 제한 검색[depth-limited search] 같은 인공지능 기반 알고리즘에 부분적으로 사용된다. 순서를 뒤집을 때도 사용되는데, LIFO 큐에 집어넣고 꺼내기만 하면 된다. 다음 그림을 살펴보면 더 확실히 이해할 수 있다.



▲ 출처: http://www.transtutors.com/homework-help/accounting/inventory-valuation-lifo/

예제

다음 예제에서 LifoQueue를 정의해 1부터 10까지 숫자를 넣는다. 다음으로 큐가 빌 때까지 모든 요소를 검색해 빼낸다.

```
import threading
import queue
import random
import time

def mySubscriber(queue):
  while not queue.empty():
  item = queue.get()
  if item is None:
    break
  print("{} removed {} from the queue".format(threading.current_thread(), item))
  queue.task_done()
  time.sleep(1)

myQueue = queue.LifoQueue()
for i in range(10):
  myQueue.put(i)

print("Queue Populated")

threads = []
```

```
for i in range(2):
  thread = threading.Thread(target=mySubscriber, args=(myQueue,))
  thread.start()
  threads.append(thread)

for thread in threads:
  thread.join()

print("Queue is empty")
```

코드에 관한 분석

위 코드는 앞서 일반적인 FIFO 큐를 사용한 것과 많이 다르지 않다. 차이점은 myQueue 객체를 선언할 때 queue.Queue( ) 대신 queue.LifoQueue( )를 사용한 것뿐이다.

출력

프로그램을 실행해보면 처음에 큐를 넣는 순서와 큐에서 빠져나오는 순서가 뒤집혀서 나올 뿐, 나머지는 FIFO 큐 형태와 거의 동일하다.

```
$ python3.6 01_lifoQueues.py
Queue Populated
<Thread(Thread-1, started 123145362374656)> removed 9 from the queue
<Thread(Thread-2, started 123145367629824)> removed 8 from the queue
<Thread(Thread-1, started 123145362374656)> removed 7 from the queue
<Thread(Thread-2, started 123145367629824)> removed 6 from the queue
<Thread(Thread-1, started 123145362374656)> removed 5 from the queue
<Thread(Thread-2, started 123145367629824)> removed 4 from the queue
<Thread(Thread-2, started 123145367629824)> removed 3 from the queue
<Thread(Thread-1, started 123145362374656)> removed 2 from the queue
<Thread(Thread-2, started 123145367629824)> removed 1 from the queue
<Thread(Thread-1, started 123145362374656)> removed 0 from the queue
Queue is empty
```

PriorityQueue

다시 비유하자면, 공항 보안 구역에서 일반 승객보다 더 중요한 파일럿, 승무원 등이 있다고 하자. 이러한 상황에서 이들은 대기줄의 맨 앞으로 와 비행기가 이륙하기 전에 미리 도착해 준비하고 있어야 한다.

다시 말해, 큐 메커니즘에 있어 우선순위가 주어진 것이다. 시스템 개발에 있어서도 우선순위 메커니즘을 바탕으로 비교적 중요하지 않은 작업에 무한정 시간을 쏟아 중요한 작업이 멈추지 않게 해야 한다. 이러한 이유로 PriorityQueue 객체가 등장했다.

PriorityQueue는 데이터가 얼마만큼 중요한지 가중치를 두어 큐에 집어넣는다. 일반적인 queue 객체와 달리 튜플을 사용해 priority_number를 첫 번째 값으로 튜플에 넣어 큐에 등록한다.

예제

다음 예제에서 (1,1)부터 (5,5)까지 두 세트의 데이터를 넣을 PriorityQueue를 생성한다. 그런 다음, 큐에 아무것도 없을 때까지 PriorityQueue에서 데이터를 빼내는 구독자를 정의한다.

```
import threading
import queue
import random
import time

def mySubscriber(queue):
  whilenot queue.empty():
    item = queue.get()
    if item is None:
      break
    print("{} removed {} from the queue".format(threading.current_thread(), item))
    queue.task_done()
    time.sleep(1)
```

```
myQueue = queue.PriorityQueue( )

for i in range(5):
  myQueue.put(i, i)

for i in range(5):
  myQueue.put(i, i)

print("Queue Populated")
threads = []
for i in range(2):
  thread = threading.Thread(target=mySubscriber, args=(myQueue,))
  thread.start( )
  threads.append(thread)

for thread in threads:
  thread.join( )

print("Queue is empty")
```

코드에 관한 분석

위 코드도 마찬가지로 이전에 살펴본 코드인데, 차이점은 queue 객체를 초기화하는 방법과 큐에 이를 넣는 방법이 달라졌다는 것뿐이다.

```
myQueue = queue.PriorityQueue( )

for i in range(5):
  myQueue.put(i, i)

for i in range(5):
  myQueue.put(i, i)
```

위 코드는 튜플 형태의 두 세트를 큐에 넣고, 모두 다 priority_number와 data를 갖고 있다.

큐 삽입 순서	튜플 (priority_number,data)	실행 순서
1	(0,0)	1
2	(1,1)	3
3	(2,2)	5
4	(3,3)	7
5	(4,4)	9
6	(0,0)	2
7	(1,1)	4
8	(2,2)	6
9	(3,3)	8
10	(4,4)	10

출력

명령행에서 프로그램을 실행하면 원소를 지운 순서가 큐 원소의 우선순위와 일치함을 볼 수 있다. 2개의 0 원소는 가장 높은 우선순위이므로 먼저 지워지고, 1이 그다음 이어지며, 우선순위 큐의 모든 원소가 없어질 때까지 실행된다.

```
$ python3.6 02_priorityQueue.py
Queue Populated
<Thread(Thread-1, started 123145475166208)> removed 0 from the queue
<Thread(Thread-2, started 123145480421376)> removed 0 from the queue
<Thread(Thread-2, started 123145480421376)> removed 1 from the queue
<Thread(Thread-1, started 123145475166208)> removed 1 from the queue
<Thread(Thread-2, started 123145480421376)> removed 2 from the queue
<Thread(Thread-1, started 123145475166208)> removed 2 from the queue
<Thread(Thread-2, started 123145480421376)> removed 3 from the queue
<Thread(Thread-1, started 123145475166208)> removed 3 from the queue
<Thread(Thread-2, started 123145480421376)> removed 4 from the queue
<Thread(Thread-1, started 123145475166208)> removed 4 from the queue
Queue is empty
```

queue 객체

앞서 언급한 queue 객체에서 이것으로 작업 때 사용하는 다양한 퍼블릭 메소드가 있다.

꽉 찬 큐와 빈 큐

프로그램상에서는 큐의 크기를 지정할 수 있어야 하는데, 무한정 크기가 커진다면 이론적으로는 MemoryErrors가 발생한다. 파이썬 프로그램에서 사용할 수 있는 메모리 크기는 시스템상의 메모리로 인해 제한받는다.

큐의 크기를 제한함으로써 효과적으로 메모리 문제로부터 보호할 수 있다. 예제에서 큐를 생성해 maxsize 인자를 넣어 0으로 설정한다. 다음으로 임의의 숫자를 큐에 넣는 4개의 스레드를 생성해본다.

그런 후, 생성된 스레드를 모두 조인해 가능한 한 많은 원소를 큐에 넣는다.

예제

다음 예제에서 queue 객체가 가득 찰 때까지 계속 삽입하는 발행자를 생성한다.

```
import threading
import queue
import time

def myPublisher(queue):
  while not queue.full():
    queue.put(1)
    print("{} Appended 1 to queue: {}".format(threading.current_thread(),
queue.qsize()))
    time.sleep(1)

myQueue = queue.Queue(maxsize=5)

threads = []
for i in range(4):
```

```
  thread = threading.Thread(target=myPublisher, args=(myQueue,))
  thread.start()
  threads.append(thread)

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

출력

코드를 실행하면, 각 스레드가 큐에 5개의 그 외 원소가 있을 때까지 최소 1개의 원소를 넣는 것을 볼 수 있다. 이때 큐가 꽉 찼다고 여기고, 모든 스레드를 종료한다.

```
$ python3.6 09_fullQueue.py
<Thread(Thread-1, started 123145399971840)> Appended 1 to queue: 1
<Thread(Thread-2, started 123145405227008)> Appended 1 to queue: 2
<Thread(Thread-3, started 123145410482176)> Appended 1 to queue: 3
<Thread(Thread-4, started 123145415737344)> Appended 1 to queue: 4
<Thread(Thread-1, started 123145399971840)> Appended 1 to queue: 5
```

join() 함수

queue 객체의 join() 함수는 현재 스레드 실행을 큐의 모든 원소가 나갈 때까지 블로킹 한다. 이는 실행될 때까지 필요한 사항을 보장해주는 역할을 임시적으로 한다.

예제

다음 예제에서는 큐 객체에서 뽑아내는 구독자를 여러 개 생성한다. 그런 후 큐가 빌 때까지 가져오는 메소드를 호출한다.

```
import threading
import queue
import time
```

```
def mySubscriber(queue):
time.sleep(1)
while not queue.empty():
  item = queue.get()
  if item is None:
    break
  print("{} removed {} from the queue".format(threading.current_thread(), item))
  queue.task_done()

myQueue = queue.Queue()
for i in range(5):
  myQueue.put(i)

print("Queue Populated")

thread = threading.Thread(target=mySubscriber, args=(myQueue,))
thread.start()

print("Not Progressing Till Queue is Empty")
myQueue.join()
print("Queue is now empty")
```

코드에 관한 분석

위 예제에서는 queue 객체를 인자로 취하는 mySubscriber 함수를 정의한다. 이 함수는 1초 동안 유휴하고, 큐가 빈 상태가 아닐 동안 실행하는 while 문을 구현한다. while 반복문에서는 배열의 원소를 검색하고, 해당 원소가 있는지 확인해본다.

원소가 없다면 현재 스레드와 큐로부터 읽은 값을 출력한다. 그런 후 task_done()을 호출해 가져오는 요청을 블로킹하는 작업이 끝났음을 알린다.

출력

프로그램을 실행하면 큐가 빈 상태일 때까지 원소를 빼는 스레드를 볼 수 있다. 여기서 조인 조건이 수행되어 프로그램은 종료된다.

```
$ python3.6 10_queueJoin.py
Queue Populated
Not Progressing Till Queue is Empty
<Thread(Thread-1, started 123145410052096)> removed 0 from the queue
<Thread(Thread-1, started 123145410052096)> removed 1 from the queue
<Thread(Thread-1, started 123145410052096)> removed 2 from the queue
<Thread(Thread-1, started 123145410052096)> removed 3 from the queue
<Thread(Thread-1, started 123145410052096)> removed 4 from the queue
Queue is now empty
```

deque 객체

디큐deque 또는 더블 앤드 큐double-ended queue는 내부 스레드 통신을 안전하게 활용하는 통신 프리미티브다. 여기에는 collections 모듈이 있어, 큐 양쪽에서 삽입하고 빼내는 부분을 제외하고는 일반적인 큐와 함수적인 부분이 일치한다.

예제

```
import collections

doubleEndedQueue = collections.deque('123456')

print("Dequeue: {}".format(doubleEndedQueue))

for item in doubleEndedQueue:
  print("Item {}".format(item))

print("Left Most Element: {}".format(doubleEndedQueue[0]))
print("Right Most Element: {}".format(doubleEndedQueue[-1]))

doubleEndedQueue.remove('1')
print("Removing Element: {}".format(doubleEndedQueue))
```

코드에 관한 분석

앞의 코드에서는 collections 모듈을 불러와 deque 객체를 사용해본다. 새로 보여준 deque 객체에 '123456'을 넣는다는 의미로 collections.deque('123456')을 호출한 해당 객체를 정의한다.

다음으로 deque 객체를 출력해, 입력한 모든 원소 배열의 deque 객체가 보이게 하며, 완벽히 하고자 deque 객체를 반복하면서 배열의 원소들을 출력한다. 마지막으로, 가장 왼쪽 및 가장 오른쪽에 있는 원소를 찾아 출력한다.

출력

프로그램에서 먼저 모든 원소를 반복하기 전 deque를 출력한다. 다음으로 doubleEndedQueue[0]을 호출해 가장 왼쪽의 원소를 검색하고, doubleEndedQueue[-1]로는 가장 오른쪽의 원소를 출력한다.

```
$ python3.6 03_deque.py
Dequeue: deque(['1', '2', '3', '4', '5', '6'])
Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Left Most Element: 1
Right Most Element: 6
Removing Element: deque(['2', '3', '4', '5', '6'])
```

원소 추가하기

특정 상황에서는 deque 객체의 모든 원소를 검색하고 살펴보는 게 유용할 수도 있지만, 일반적으로는 객체들끼리 상호작용이 필요하다. 다음 예제에서 append()와 appendLeft() 함수를 소개해 새로운 원소를 deque 객체의 앞 또는 뒤에 넣어본다.

예제

다음 코드에서 deque 객체의 앞과 뒤에 모두 원소를 추가해본다.

```
import collections

doubleEndedQueue = collections.deque('123456')

print("Deque: {}".format(doubleEndedQueue))

doubleEndedQueue.append('1')
print("Deque: {}".format(doubleEndedQueue))

doubleEndedQueue.appendleft('6')
print("Deque: {}".format(doubleEndedQueue))
```

코드에 관한 분석

예제를 살펴보면 deque 객체인 doubleEndedQueue를 생성한다. 다음으로 해당 deque의 현재 상태를 출력하고, append() 함수를 사용해 큐의 끝에 원소 1을 추가한다. 그 후 deque 객체의 상태를 한 번 더 출력하고, 큐의 오른쪽에 1이 추가됐음을 확인한다. 이젠 appendLeft() 함수를 통해 원소 6을 큐의 앞에 추가한다.

출력

위 프로그램을 실행하면 1에서 6까지의 큐를 볼 수 있다. append(1) 함수를 사용해 1을 추가하면 deque 객체의 끝에 1이 추가되어 출력됨을 볼 수 있다. 다음으로 appendLeft(6) 함수를 호출해 다시 출력해보면 deque 객체의 앞에 6이 추가됐음을 볼 수 있다.

```
$ python3.6 04_addRemoveDeque.py
Deque: deque(['1', '2', '3', '4', '5', '6'])
Deque: deque(['1', '2', '3', '4', '5', '6', '1'])
Deque: deque(['6', '1', '2', '3', '4', '5', '6', '1'])
```

원소 꺼내기

반대로, deque 객체에 넣은 원소를 검색해보고 싶을 때가 있다. 이때 퍼블릭 함수인 pop( ) 및 popleft( )를 사용해본다.

예제

다음 코드에서 pop( )과 popleft( ) 함수를 사용해 큐의 시작과 끝에서 원소를 어떻게 꺼내는지 살펴보자.

```
import collections

doubleEndedQueue = collections.deque('123456')

print("Deque: {}".format(doubleEndedQueue))

# 큐에서 요소를 뽑아낸다.
rightPop = doubleEndedQueue.pop( )
print(rightPop)
print("Deque: {}".format(doubleEndedQueue))

leftPop = doubleEndedQueue.popleft( )
print(leftPop)
print("Deque: {}".format(doubleEndedQueue))
```

코드에 관한 분석

위 코드에서도 마찬가지로 doubleEndedQueue 객체를 선언해 1부터 6까지 값을 넣는다. 현재 큐 상태가 기본적으로 어떠한지 출력해 보여준다.

먼저 rightPop 변수를 선언하고 doubleEndedQueue.pop( )을 호출해 deque 객체의 가장 오른쪽 값을 검색한다. 그 후 꺼낸 원소와 현재 deque 객체의 상태를 출력한다.

마찬가지로, popleft( ) 메소드를 활용해 deque 객체의 가장 첫 번째 값을 검색해본다.

이를 leftPop 변수에 넣어, 동일하게 현재 deque 객체 상태와 함께 출력한다.

출력

프로그램의 출력은 pop( )과 popleft( ) 함수에 의해 보인다. 기존의 deque 객체와 pop( )을 통해 deque 객체의 마지막 값을 꺼내어 콘솔에 출력한다.

다음으로 popleft( )를 호출해 deque 객체의 가장 앞 원소를 꺼내 출력한다.

```
$ python3.6 05_removeDeque.py
Deque: deque(['1', '2', '3', '4', '5', '6'])
6
Deque: deque(['1', '2', '3', '4', '5'])
1
Deque: deque(['2', '3', '4', '5'])
```

원소 삽입하기

deque 객체에 원소를 넣는 것이 불가능하다면, deque 객체는 유용하지 않을 것이다.

예제

다음 예제에서 특정 지점에 원소를 어떻게 삽입하는지 살펴보자.

```
import collections

doubleEndedQueue = collections.deque('123456')

print("Deque: {}".format(doubleEndedQueue))

doubleEndedQueue.insert(5,5)

print("Deque: {}".format(doubleEndedQueue))
```

코드에 관한 분석

코드를 살펴보면 insert(n, n) 함수를 활용해 다섯 번째 위치에 원소 5를 삽입하고 있다.

출력

다음 프로그램의 출력으로 다섯 번째 위치에 원소 5가 있음을 볼 수 있다.

```
$ python3.6 06_insertDeque.py
Deque: deque(['1', '2', '3', '4', '5', '6'])
Deque: deque(['1', '2', '3', '4', '5', 5, '6'])
```

회전

queue 객체는 인자가 양수인지 음수인지에 따라 오른쪽, 왼쪽으로 n번 회전할 수 있다.

예제

다음 예제에서 deque 객체에 대한 메소드 인자에 양수 및 음수를 넣어 양방향 모두 원소를 회전시켜보자.

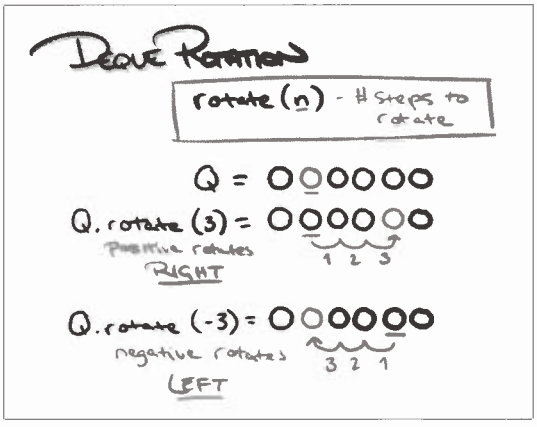
```
import collections

doubleEndedQueue = collections.deque('123456')

print("Deque: {}".format(doubleEndedQueue))

doubleEndedQueue.rotate(3)

print("Deque: {}".format(doubleEndedQueue))

doubleEndedQueue.rotate(-2)

print("Deque {}".format(doubleEndedQueue))
```

코드에 관한 분석

앞의 예제에서 일반적인 deque 객체를 생성하고 1부터 6까지 값을 넣는다. 그 후 객체를 출력하고 오른쪽으로 3칸 회전시켜본다.

회전 후 모든 원소는 오른쪽으로 세 번 움직이는데, 맨 마지막 원소가 앞으로 차례대로 계속 움직인다.

양수 및 음수를 사용해 회전하는 부분은 다음 그림을 참조하자.



▲ 출처: http://www.transtutors.com/homework-help/accounting/inventory-valuation-lifo/

출력

다음 출력에서 볼 수 있듯이, 처음에는 1부터 6까지 순서가 이어진다. 그 후 앞쪽으로 세 번 회전해, 원소가 이동한 것을 볼 수 있다.

다시 두 번 뒤로 회전해 모든 원소가 2칸씩 뒤로 이동했음을 볼 수 있다.

```
$ python3.6 08_rotateDeque.py
Deque: deque(['1', '2', '3', '4', '5', '6'])
Deque: deque(['4', '5', '6', '1', '2', '3'])
Deque deque(['6', '1', '2', '3', '4', '5'])
```

자체적인 스레드 세이프 통신 구조 정의하기

기본적인 통신 프리미티브 대신 스레드 간의 통신을 위해 자체 구성 객체를 구현해야 할 때가 있다.

웹 크롤러 예제

이제 이전 장에서 다룬 동기화 프리미티브와 함께 통신 프리미티브를 다루면서 사용해 보자.

그렇다면 어떻게 이런 프로그램을 만들 수 있을까?

이 절에서는 간단한 멀티스레드 기반 웹 크롤러를 만들어본다.

요구사항

실제 프로젝트처럼 먼저 요구사항을 분석해보자. 다시 말해, 작업해나갈 길잡이가 필요하다. 다음 요구사항을 살펴보자.

- 웹 크롤러는 멀티스레드를 활용한다.
- 웹사이트의 모든 특정 웹 페이지를 크롤링할 수 있어야 한다.
- 404 링크로 응답이 가능해야 한다.
- 명령행에서 도메인 이름을 취한다.
- 주기적인 순회는 허용하지 않는다.

마지막 부분인 주기적인 순회가 중요하다. 프로그램이 계속적으로 상호 링크된 여러 페이지를 크롤링하는 것을 막고자, 크롤링한 페이지를 정확히 알아야 한다. 4장에서 배운 동기화 프리미티브를 여기서 활용해볼 것이다.

디자인

이 프로그램에서는 웹 페이지를 요청하고 크롤링할 웹 페이지에 새로운 링크를 하고자 파싱해야 한다.

이러한 작업은 Crawler 클래스를 호출해 따로 처리한다. 여기에는 생성자 함수, 실행 함수, 보조 processLink 함수를 각각 넣는다.

Crawler 클래스

크롤러를 생성하고자 먼저 Crawler 클래스를 정의한다. 여기에는 크롤링을 수행하고 크롤링할 리스트의 링크를 큐에 넣는 정적 메소드가 내장돼 있다.

상단에서는 HTTPS 요청에 필요한 urllib.request, urllib.parse, ssl, BeautifulSoup 모듈을 불러온다. BeautifulSoup 모듈은 HTML 언어를 파싱하는 작업을 쉽게 수행해준다.

클래스 상단에서는 변수를 선언하고, 크롤링 사이트인 base_url 변수를 만든다. https://tutorialedge.net/ 사이트를 크롤링했다면 이는 base_url로 되며, 해당 도메인에서만 크롤링을 진행한다.

다음으로 myssl 컨텍스트를 선언해 HTTPS 요청의 컨텍스트를 취한다. 마지막으로, errorLinks라고 새로운 설정값을 인스턴스화해 200개 이하의 상태 코드를 넘기는 링크를 넣는다.

```
from urllib.request import Request, urlopen, urljoin, URLError
from urllib.parse import urlparse
import ssl
from bs4 import BeautifulSoup

class Crawler:

  base_url = ''
  myssl = ssl.create_default_context()
  myssl.check_hostname=False
```

```
  myssl.verify_mode=ssl.CERT_NONE
  errorLinks = set()
```

그런 다음 base_url을 설정한 크롤러의 constructor 함수를 선언한다.

다음으로 crawl, enqueueLinks 정적 메소드를 선언한다. 이는 링크 리스트와, queue 객체인 linkToCrawl을 취한다. 이는 해당 링크가 아직 크롤링되지 않았다면 계속 반복하면서, linksToCrawl에 enqueued되지 않았다면 queue 객체에 추가한다.

```
def __init__(self, base_url):
  Crawler.base_url = base_url

@staticmethod
def crawl(thread_name, url, linksToCrawl):
  try:
    link = urljoin(Crawler.base_url, url)

    if (urlparse(link).netloc == 'tutorialedge.net') and (link not in
Crawler.crawledLinks):
      request = Request(link, headers={'User-Agent': 'Mozilla/5.0'})
      response = urlopen(request, context=Crawler.myssl)
      Crawler.crawledLinks.add(link)
      print("Url {} Crawled with Status: {} : {} Crawled In Total".format(
response.geturl(), response.getcode(), len(Crawler.crawledLinks)))
      soup = BeautifulSoup(response.read(), "html.parser")
      Crawler.enqueueLinks(soup.find_all('a'), linksToCrawl)
    except URLError as e:
      print("URL {} threw this error when trying to parse: {}".format(link, e.reason))
      Crawler.errorLinks.add(link)

@staticmethod
def enqueueLinks(links, linksToCrawl):
  for link in links:
    if (urljoin(Crawler.base_url, link.get('href')) not in Crawler.crawledLinks):
      if (urljoin(Crawler.base_url, link.get('href')) not in linksToCrawl):
        linksToCrawl.put(link.get('href'))
```

시작점

다음으로 할 일은 main.py 파일을 구현하는 것이다. 이는 웹 크롤러 프로그램의 시작점이라 할 수 있다.

필요한 모듈과 CheckableQueue를 먼저 불러오는데, 정의는 이후에 한다. 불러온 후, 코드가 본격적으로 실행되기 전에 스레드를 정의한다. 웹 크롤러의 높은 I/O 한계로 인해, 다양한 스레드를 실행하는 것은 멀티 I/O 제약 HTTP 요청을 동시에 수행하게 한다.

다음으로 createCrawlers 함수를 정의하는 데 필수적인 스레드를 작업한다. 그 후, 생성될 모든 스레드를 타깃으로 하는 실행 함수를 정의한다. 해당 함수에서는 끊임없이 반복하며 linksToCrawl 큐의 get( ) 메소드를 호출한다. get( )으로 검색된 요소가 None이면, 스레드를 종료하고, 정적 Crawler.crawl( ) 함수를 호출해 URL을 크롤링하고, current_thread, URL, linksToCrawl 큐를 인자로 한다.

이제 main 함수를 정의한다. 먼저 예제에서 크롤링하고자 하는 URL(https://tutorialedge.net/)을 취하는데, 독자들이 각자 선택해도 좋다. 크롤러를 인스턴스화하고, 그 밖의 웹사이트를 크롤링하는 크롤러를 담는 base_url을 인자로 한다.

마지막으로, main 함수 내에 createCrawlers( ) 함수를 정의하고 linksToCrawl 큐를 조인해 큐의 모든 원소가 진행될 때까지 프로그램 실행이 끝나지 않게 한다.

```
import threading
import queue
from Crawler import *
from CheckableQueue import *

THREAD_COUNT = 20
linksToCrawl = CheckableQueue( )

def createCrawlers( ):
  for i in range(THREAD_COUNT):
    t = threading.Thread(target=run)
    t.daemon = True
    t.start( )
```

```
def run():
  while True:
    url = linksToCrawl.get()
    try:
      if url is None:
        break
      Crawler.crawl(threading.current_thread(), url, linksToCrawl)
    except:
      print("Exception")
    linksToCrawl.task_done()

def main():
  url = input("Website > ")
  Crawler(url)
  linksToCrawl.put(url)
  createCrawlers()
  linksToCrawl.join()
  print("Total Links Crawled: {}".format(len(Crawler.crawledLinks)))
  print("Total Errors: {}".format(len(Crawler.errorLinks)))

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

queue 객체 확장하기

예제에서 queue 객체의 원자성을 활용하고자 했다. 하지만 이를 좀 더 확장해 새로 찾은 링크가 이미 크롤링됐다면, 이후에 더 이상 크롤링될 큐에 넣지 않게 한다.

```
import queue

class CheckableQueue(queue.Queue):
  def __contains__(self, item):
    with self.mutex:
      return item in self.queue

  def __len__(self):
    return len(self.queue)
```

코드에 관한 분석

앞의 코드에서는 queue 모듈을 불러왔는데, queue.Queue로부터 상속받은 CheckableQueue를 정의한다.

다음으로 __contains__ 메소드를 정의해 원소를 정의하고 뮤텍스mutex를 활용한다. 더군다나 해당 큐 안에 이미 지나온 원소가 있는지 확인하고자 큐를 안전하게 순회한다.

__len__ 함수도 정의해, 큐의 길이를 반환받는다. 중요하지는 않지만, 프로그램 실행 중 크롤러가 얼마만큼 작업했는지 알려주는 역할을 한다.

출력

https://tutorialedge.net/을 입력해 크롤러 프로그램을 실행하면, 프로그램은 https://tutorialedge.net/에서 찾는 모든 페이지를 검색하게 된다.

페이지를 찾을 때마다 linksToCrawl CheckableQueue 객체에 추가되며, 스레드는 이를 하나씩 뽑아 인덱스를 붙인다. 다음으로 크롤러의 상태를 크롤링한 URL을 확인하는 요청이 있을 때마다 출력하며, HTTP 상태와 크롤링한 전체 페이지 수도 보여준다.

```
$ python3.6 main.py
Website > https://tutorialedge.net
Url https://tutorialedge.net Crawled with Status: 200 : 1 Crawled In Total
Url https://tutorialedge.net/series/blog/ Crawled with Status: 200 : 2
Crawled In Total
Url
https://tutorialedge.net/post/webdev/difference-between-class-id-selector-c
ss/ Crawled with Status:
200 : 3 Crawled In Total
....
Url https://tutorialedge.net/page/9/ Crawled with Status: 200 : 216 Crawled
In Total
Url https://tutorialedge.net/page/10/ Crawled with Status: 200 : 217
Crawled In Total
Url https://tutorialedge.net/page/11/ Crawled with Status: 200 : 218
```

```
Crawled In Total
Total Links Crawled: 218
Total Errors: 11
```

더 나아가기

이 프로그램은 다양한 방법으로 발전돼온 비교적 간단한 웹사이트 크롤러다. 해당 프로그램을 좀 더 자세히 인덱싱하고 사이트의 상태를 모니터링할 수 있게 하는 등 서비스로 활용 가능한 특성을 갖춘 웹 크롤러로 발전시켜보자.

이를 가장 잘 활용할 수 있는 방법은 API로 묶어 서버에 올림으로써 여러 사이트의 웹 인터페이스를 테스트하는 것이다.

결론

다행히도 이는 사용자에게 이후에 배울 개념과 프리미티브를 활용해 좀 더 복잡한 프로그램을 만들 수 있게 해준다.

이번 예제에서는 주어진 사이트의 모든 링크된 내용의 상태를 볼 수 있는 멀티스레드 기반 웹 크롤러를 구성해봤다. 여기서는 1개의 URL을 시작점으로 페이지 안의 모든 링크를 파싱하며 파싱한 데이터를 처리한다. 웹사이트의 모든 링크된 페이지를 스캔할 때까지 계속한다.

지금까지 살펴본 주제를 정리해보면 다음과 같다.

- I/O 한계 애플리케이션의 성능을 높이는 멀티스레드: 멀티스레드를 활용해 동시적으로 여러 웹 페이지 요청을 할 수 있다.
- 멀티스레드 간의 통신: 이 예제에서는 스레드 세이프 통신을 위해 큐와 세트를 활용했다. 파싱하고자 하는 URL을 저장하고자 queue 객체를, 파싱된 링크를 저장할 때 세트를 이용했다.

연습: 성능 테스트

앞서 살펴본 스레드 통신 성능을 테스트하는 수단으로 '더 나아가기' 절에서 살펴본 웹 크롤러에 함수를 더해보자. 내 경우 스레드 안정성을 확인할 수 있는 좋은 방법은 복잡한 문제를 파고드는 것이다.

별로 흥미롭지 않다면 다양한 애플리케이션을 활용해 제작해보는 것도 추천한다.

웹 서버: 여러 개의 연결을 다룰 수 있는 웹 서버를 만들어보자. 좀 더 많은 파이썬 프레임워크에 대한 이해가 가능하기에, 흥미로울 뿐만 아니라 의미 있는 작업이 될 것이다.

요약

5장에서는 멀티스레드 기반 시스템에서 통신을 구현하는 다양한 메커니즘을 살펴봤다. 파이썬 내장 스레드 세이프 형태의 큐 프리미티브를 자세히 다루고, 어떻게 이러한 프리미티브를 이용해 해결하는지 알아봤다.

마지막 절에서는 3, 4장에서 배운 개념을 끌어와, 주어진 웹사이트의 모든 링크 상태를 확인하는 프로그램을 만들어봤다.

6장에서는 시스템이 생산성 있고 버그가 없는지 확인하고자, 다양한 디버깅 및 벤치마킹 기술을 살펴본다.

06

디버깅과 벤치마킹

프로그래밍이란 단순히 문제를 해결하고 그걸로 끝내버리는 것은 아니다. 대개는 솔루션을 유지하면서 산업이 이뤄지고 수입이 발생한다. 일반적으로 솔루션을 유지하는 것은 디버깅 및 요소를 추가하는 작업으로, 이를 위해서는 파이썬 생태계에서 관련된 툴을 익히는 것이 중요하다.

6장에서는 새로운 요소를 추가하고, 기존의 위험 요소 코드를 리팩토링하고자 다양한 테스트 방식을 살펴본다.

세세한 부분까지 파이썬 애플리케이션에서 이해를 도울 만한 여러 프로그램 툴을 살펴본다. 6장에서 다루는 내용은 다음과 같다.

- 코드의 테스트 방식
- 파이썬 디버거

- Pdb
- `line_profiler` 툴
- `memory_profiler` 툴

마지막 부분에서는 벤치마킹과 프로파일링을 어떻게 하는지와 함께 시스템 테스트의 의미를 살펴본다. 코드를 한 층 업그레이드할 테스트 방법을 살펴보자.

테스트 전략

6장의 제목은 '디버깅과 벤치마킹'이지만, 코드를 디버깅하는 가장 좋은 방법은 가능한 한 많은 코드를 다룰 수 있는 통합 테스트를 구성하는 것이다. 코드 기반에서 왜 테스트를 해야 되는지 그 이유를 지금부터 알아보자.

왜 테스트를 해야 하는가?

이 책의 중반까지 달려왔는데, 정작 테스트를 정의하거나 프로그램이 올바르게 작성됐는지 확인해보지 않았다. 지금까지 내가 보여준 모든 코드를 그대로 받아적기만 했다. 하지만 이를 변경하지 않고도 항상 동일한 결과가 도출될까?

여기서 테스트 전략의 필요성이 느껴진다.

전문적인 소프트웨어 개발에서 버그를 고치는 소프트웨어 테스트는 매우 중요한 작업 중 하나다. 수많은 훌륭한 소프트웨어 개발자는 자신이 구축한 시스템에 필요한 테스트 방법을 고안하고, 실제로도 빠른 수정이 가능하다.

10만 줄의 코드로 구성된 구형 시스템에 구현된 테스트 방법이나 형태가 없다고 하자. 실행 시 막혔을 때 어떻게 이를 테스트할 수 있을까? 구현한 코드 변경 형태가 애플리케이션의 생산성이나 나아가 비용을 줄일 수 있을까? 결론부터 말하면, 이러한 테스트는 불가능하다. 코드를 보는 것조차 힘들고, 수많은 개발자에게 큰 고통이다.

반대로, 10만 줄의 구형 시스템에 새로운 테스트 요소를 개발해 구현했다고 하자. 코드의 특정 부분에 변경이 있으면, 구성원들이 구현한 테스트가 문제를 잡고 기존의 어떤 것에든 영향 없이 생산성을 확실히 높일 수 있을 것이다. 따라서 애자일agile은 개발 팀에서 매우 유용하며, 소프트웨어 시스템을 많이 보완해준다. 생산성과 직결되는 비즈니스 부분에도 도움을 주는데, 지원에 대한 걱정을 덜어준다.

동시성 소프트웨어 시스템 테스트

이 책의 범위를 벗어나지만 그래도 중요한 부분이 있는데, 멀티스레드 프로그램을 구현하기 전 모든 동시성 프로그램을 적절히 테스트해보는 것이다. 버그가 있는 단일 스레드 프로그램인데 멀티스레드로 해야 할 경우, 더 많은 버그가 일어나고 힘들어질 것이다.

모든 소프트웨어 시스템은 최적화가 구현되기 전 정확도가 높아야 한다.

어떤 것을 테스트할까?

이제 왜 테스트를 해야 하는지 살펴보고, 테스트해야 할 것, 하지 말아야 할 것도 알아본다. 내가 종종 사용하는 방법은 코드 검사다. 기본적으로 테스트할 코드가 몇 줄인지 파악하는데, 이를 파이썬 객체로 확인하는 의미 없는 테스트를 작성하는 데 사용자들이 힘쓰고 있는 모습을 봤다. 시스템에 큰 도움이 안 되기에 분명히 피해야 할 작업이다.

코드 검사에 집중하는 대신, 코드의 가장 중요한 부분만을 테스트하는 데 집중하고, 그 후 덜 중요한 부분으로 확장해나가자. 일반적인 프로젝트 환경에서 모든 것을 테스트하는 것은 시장 출시일을 늦추며, 비즈니스 조건에 부합하고자 플랫폼 안정성을 확인하는 테스트가 필요하다.

소프트웨어에 필요한 다양한 방법의 테스트를 진행하기를 추천한다. 의도적으로 로직을 멈추게 해보고, 문제를 발생시켜보자.

단위 테스트

단위 테스트unit test란 코드의 논리적 단일 단위를 테스트하는 프로그래밍적 검증이다. 일반적으로 단위란 코드에서 함수를 의미한다.

파이썬에서 단위 테스트를 작성할 때 가장 먼저 염두에 둬야 할 부분은 파이썬 3.6의 내장 모듈인 unittest 모듈이다.

PyUnit

PyUnit이란 자바에서 JUnit을 파이썬으로 구현한 것이다. 기본적으로 제공되는 단위 테스트 모듈이며, 테스트에 필요한 모든 것이 정의돼 있다.

예제

테스트 프로그램에 먼저 필요한 것은 unittest 모듈을 불러오는 것이다. 간단한 함수를 테스트하는 데 필요한 것이 포함돼 있다.

그 후 simpleFunction을 정의해 인자 하나를 취해 1을 더해 반환한다. 코드는 어렵지 않은데 다른 내용이 함수 출력에 영향을 준다면, 코드 변경 후 실행이 정지되지 않도록 메커니즘을 확인해볼 필요가 있다.

다음에 주어진 함수에서 unittest.TestCase로부터 상속받은 SimpleFunctionTest 클래스를 정의한다. 여기서 setUp과 tearDown 함수를 정의한다. 이는 테스트 전후로 동작된다.

마지막으로, 가장 핵심인 unittest.main()을 호출한다.

```
import unittest

def simpleFunction(x):
  return x + 1

class SimpleFunctionTest(unittest.TestCase):

  def setUp(self):
```

```
    print("This is run before all of our tests have a chance to execute")

  def tearDown(self):
    print("This is executed after all of our tests have completed")

  def test_simple_function(self):
    print("Testing that our function works with positive tests")
    self.assertEqual(simpleFunction(2), 3)
    self.assertEqual(simpleFunction(234135145145432143214321432),
234135145145432143214321433)
    self.assertEqual(simpleFunction(0), 1)

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

출력

위 프로그램을 실행하면 setUp 함수가 먼저 실행되고, 단일 테스트 케이스가 뒤이어 나온 후 tearDown 함수가 실행된다.

그런 다음 테스트 실행 횟수, 실행 시간, 전체 상태를 출력한다.

```
$ python3.6 00_unittest.py
This is run before all of our tests have a chance to execute
Testing that our function works with positive tests
This is executed after all of our tests have completed
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

테스트 확장하기

이전 절에서 간단하게 함수를 테스트해봤다. 코드의 경로는 사용자가 의도한 대로 실행됐지만, 아직 갈 길이 멀다.

현재 상황에서 어떻게 이 코드를 다룰지(문자열 입력, 음수, 객체) 정하지 않았다.

오늘날 안정적이고 생산적인 코드를 작성하는 방법은 모든 코드를 작성하기 전 실패 테스트의 범위를 정의하는 것이다. 이를 테스트 기반test-driven 개발이라고 하며, 버그를 찾고 코드를 변경하는 시간을 획기적으로 줄여준다.

테스트 기반 개발의 숨은 아이디어는 시스템이 어떻게 동작할지 정확하게 테스트하는 실패 테스트를 작성하는 데 있다. 그 후 테스트를 통과할 때까지 코드를 작성하는 것이다.

동시성 코드의 단위 테스트

안타깝게도, 동시성 코드를 단위 테스트할 때 모든 문제를 해결할 수 있는 방법은 없다. 여러 가지 전략을 구성해 나아가고자 하는 방향으로 개발할 시스템을 위해 온 힘을 쏟아부어야 한다.

멀티스레드라면 코드의 모든 시나리오를 테스트할 수 없다. 이상적으로는 다음과 같은 여러 가지 방법을 함께 사용해야 한다.

- 멀티스레드에서 동작하지 않는 코드의 단위 테스트
- 다양한 방법으로 멀티스레드 코드를 살피는 테스트를 작성한다. 가능하다면 특정 코드 영역에서 부하 테스트를 바탕으로 멀티스레드 기반 로직이 안정적인지 확인한다.

물론 많은 방법이 있겠지만, 코드에 대한 충분한 테스트는 시스템의 개발 및 복잡성과 관련된 다양한 학문의 조합이 필요하다.

내 경험상 테스트할 수 없을 정도로 코드가 복잡하다면, 처음 접근했던 시나리오를 떠올리며 최대한 간단한 디자인을 생각해보자. 모든 경우는 아니지만, 시간과 자원이 한정된 상태에서 추천하는 바다.

통합 테스트

단위 테스트가 코드의 단일 단위의 정확성을 나타내는 테스트라면, 통합 테스트integration test는 코드 및 시스템 여러 부분의 정확성을 확인하는 테스트다.

통합 테스트는 기본적으로 더 복잡하지만, 퍼즐의 커다란 조각만큼 시스템을 잘 이해할 수 있다. 단위 테스트와 마찬가지로 통합 테스트는 동기화된 시스템을 실행하는 데 큰 도움을 준다.

통합 테스트는 여기서 마무리하겠지만, 더 자세한 내용을 알고 싶다면 다른 책들을 참고하자. 다양한 통합 테스트 방법을 연구해보고, 코드에 에러가 일어나지 않도록 올바른 통합 테스트를 진행해보자. 디버깅에 대한 부분은 개발자에게 더 쉽게 느껴질 수 있으니 살펴보자.

디버깅

코드 디버깅은 소프트웨어 개발자가 꼭 배워야 하는 기술 중 하나다. 시스템이 복잡할수록 코드 내부의 버그는 급격히 증가하며, 특정 상황에서의 디버깅을 잘 알아야 개발 시간이 단축된다.

다음에 보여줄 기술은 단일 스레드 및 멀티스레드 기반 애플리케이션 모두에서 작동된다. 간단히 하고자 여기서는 단일 스레드 애플리케이션만 보여줄 텐데, 독자들은 다양한 도구를 사용해 멀티스레드 상황에서도 적용해보기 바란다.

단일 스레드에서 작동해보기

웹사이트를 통한 판매량을 엄청나게 올리거나 하룻밤 사이에 백만장자가 될 수 있는 AI 시스템을 만든다고 해보자. 아마 한 달 내내 이 시스템에 몰두해 온 힘을 쏟아붓지만, 매우 느리게 실행될 것이다. 이는 정확한 예측에 초당 수백만 번의 연산이 이뤄지기 때문이다.

이러한 상황에서 이전에 살펴본 예제와 같이 코드 기반의 최적화가 필요하다고 느낄 것이다. 멀티프로세스를 바탕으로 효과적으로 이를 다루고, 속도 증가도 이뤄낼 수 있다.

하지만 아직까지는 다음 시스템의 논리 흐름이 매우 어려움을 볼 수 있다. 멀티스레드 및 프로세스에 따라야 하며, 내부적으로는 시스템에 많은 버그를 일으킨다.

이런 이론적인 이야기는 코드의 복잡성 증가를 설정하기 전 결정론적인 방식으로 애플리케이션이 작동돼야 하며, 이를 최적화해야 함을 보여준다. 단일 스레드 프로그램은 디버깅, 작동 및 논리적 오류를 잡는 데 있어 비결정적인 방식의 멀티스레드로 동작되는 시스템보다 쉽다.

그러나 버그가 나타나기 전 이미 배포 및 실행 중인 시스템을 디버깅하는 상황에서는 불가능하다. 이러한 상황에서는 Pdb 툴을 사용해 버그를 해결해야 한다.

PyCon의 레이먼드 헤팅거Raymond Hettinger는 동시성을 추가하기 전에 단일 스레드 방식으로 작동시켜야 하는 이유를 잘 설명했다. https://www.youtube.com/watch?v=Bv25Dwe84g0&t=640s를 참조하자.

Pdb

파이썬 디버거Python Debugger인 Pdb는 파이썬에 기본적으로 포함된 디버거로, 디버깅에 있어 기본적인 부분에 훌륭한 툴이다. Pdb는 완벽한 제어 및 런타임 도중 변숫값을 확인해볼 수 있게 하며, 단계적으로 코드를 실행하거나 중지점을 지정하는 등의 작업을 가능케 한다.

Pdb를 사용해 프로그램 종료 디버깅을 할 수 있는데, 명령행을 통해 쌍방향 실행이 가능하다. 대화식 명령행을 사용해 코드를 실행하려면, 다음 명령어 목록을 익혀야 하니 참고하자.

- l(리스트list)
- n(다음next)
- c(계속하기continue)

- s(단계step)
- r(반환하기return)
- b(중지하기break)
- python

위 명령어가 기억나지 않는다면, ?를 사용해 Pdb가 실행되는 동안 사용 가능한 명령어 목록을 확인해볼 수 있다. 상당수의 명령어가 중복되니 당황하지 말자.

```
(Pdb) ?
Documented commands (type help <topic>):
========================================
… A table with all of the commands - not shown for brevity
```

대화형 예제

다음 코드를 디버깅하고 특정 변수의 어떤 값이 실행 중인지 본다고 해보자. 이를 위해 코드의 중지점을 지정하고자 다음 줄을 활용한다.

```
import pdb; pdb.set_trace()
```

위와 같이 지정해 프로그램을 실행하면 해당 줄이 지정된 지점까지 실행하다가 멈춘다. 해당 줄에 다다를 때까지 Pdb 대화형 터미널이 시작되고, 프로그램의 현재 상태를 디버깅하고자 앞서 살펴본 다양한 명령어를 활용할 수 있다.

```
from timer import Timer
from urllib.request import Request, urlopen
import ssl

def myFunction():
  # 컨텍스트를 생성해 https 사이트를 크롤링한다.
  myssl = ssl.create_default_context()
```

```
    myssl.check_hostname=False
    myssl.verify_mode=ssl.CERT_NONE
    with Timer() as t:
      import pdb; pdb.set_trace()
      req = Request('https://tutorialedge.net', headers={'User-Agent': 'Mozilla/5.0'})
      response = urlopen(req, context=myssl)

print("Elapsed Time: {} seconds".format(t.elapsed_secs))

myFunction()
```

위 코드를 실행해보자. Pdb를 불러와 set_trace() 함수가 위치한 지점까지 코드가 실행된 후 멈춘다. 위 프로그램에서 열두 번째 줄에 set_trace() 함수를 넣었는데, 열세 번째 줄부터 멈춘 것을 볼 수 있다. 실행 중인 코드를 보고자, l 또는 list 명령어를 통해 현재 실행 중인 코드를 모두 볼 수 있다.

현재 실행하려는 코드 줄은 다음 줄의 -> 표시를 통해 알 수 있다.

```
$ python3.6 04_timeitContext.py
> /Users/elliotforbes/Projects/Python/Chapter
06/04_timeitContext.py(13)myFunction()
-> req = Request('https://tutorialedge.net', headers={'User-Agent':
'Mozilla/5.0'})
(Pdb) l
 8       myssl = ssl.create_default_context();
 9       myssl.check_hostname=False
10       myssl.verify_mode=ssl.CERT_NONE
11       with Timer() as t:
12         import pdb; pdb.set_trace()
13  ->     req = Request('https://tutorialedge.net', headers={'User-Agent':
'Mozilla/5.0'})
14         response = urlopen(req, context=myssl)
15
16       print("Elapsed Time: {} seconds".format(t.elapsed_secs))
17
18
```

이를 확인해보고자 http 응답 객체의 값이 주어졌을 때, n 명령어를 이용해 다음 줄까지 코드 실행이 계속되게 한다. 그 후 response = urlopen(req, context=myssl)을 취한다.

응답 객체의 값을 얻고자 n 명령어를 사용해 마찬가지로 해당 줄을 실행해보고, 응답 객체를 출력하고자 print(response)를 사용해본다. 그런 다음 코드 안에 있는 것처럼 객체를 다룰 수 있으며, 런타임 중 URL의 값이 무엇인지 알고자 geturl() 함수를 호출한다.

```
$ python3.6 04_timeitContext.py
> /Users/elliotforbes/Projects/Python/Chapter
06/04_timeitContext.py(13)myFunction()
-> req = Request('https://tutorialedge.net', headers={'User-Agent':
'Mozilla/5.0'})
(Pdb) n
> /Users/elliotforbes/Projects/Python/Chapter
06/04_timeitContext.py(14)myFunction()
-> response = urlopen(req, context=myssl)
(Pdb) n
> /Users/elliotforbes/Projects/Python/Chapter
06/04_timeitContext.py(16)myFunction()
-> print("Elapsed Time: {} seconds".format(t.elapsed_secs))
(Pdb) print(response)
<http.client.HTTPResponse object at 0x1031e2588>
(Pdb) print(response.geturl())
https://tutorialedge.net
```

내가 파이썬을 배울 때는 원하는 결과를 얻고자 수정한 후에는 바로 애플리케이션을 다시 실행해봤다. 하지만 프로그램이 거대해지고 런타임 시간이 오래 걸리자, 간단한 디버깅만으로는 충분치 않았다.

이러한 훌륭한 디버깅 방법을 배운 후 에러를 찾고 수정하는 시간을 획기적으로 줄일 수 있었으므로, 여러분에게도 Pdb를 활용하기를 권장한다.

공식 문서

더 나아가기 전, 파이썬 3.6의 Pdb 공식 문서를 살펴보자. 여기서 디버깅에 필요한 모든 명령어를 볼 수 있다. https://docs.python.org/3.6/library/pdb.html을 참고하자.

자식 스레드에서 예외 처리하기

멀티스레드 기반 애플리케이션을 작성할 때는 자식 스레드에서 온 예외를 어떻게 다뤄야 할지 중요하게 고민해봐야 한다. 이전 장에서 크로스 스레드 통신을 살펴봤는데, 자식과 부모 스레드 간의 예외를 다루는 논리적인 방법은 이미 여러 형태로 다뤄봤다.

다음 코드에서는 자식 스레드에서 부모 스레드로 넘겨진 예외를 어떻게 다루는지 자세히 살펴본다. sys 모듈을 활용해 예외에 필요한 정보를 추출하고, 스레드 세이프 큐에 넣는다.

```
import sys
import threading
import time
import queue

def myThread(queue):
  while True:
    try:
      time.sleep(2)
      raise Exception("Exception Thrown In Child Thread {}".format(
threading.current_thread()))
    except:
      queue.put(sys.exc_info())

queue = queue.Queue()
myThread = threading.Thread(target=myThread, args=(queue,))
myThread.start()
```

```
while True:
  try:
    exception = queue.get()
    except Queue.Empty:
    pass
  else:
    print(exception)
    break
```

위의 파이썬 프로그램을 실행하면 실행 후 2초 뒤 자식 스레드가 에러를 던지는 것을 볼 수 있는데, 이는 차후 부모 스레드가 읽을 스레드 세이프 큐 객체에 넣어진다. 그 후 부모 스레드 내에서 원하는 방법으로 예외를 다룰 수 있다.

```
[20:22:31] ~/Projects/Python/Chapter 06 master
$ python3.6 07_threadException.py
(<class 'Exception'>, Exception('Exception Thrown In Child Thread
<Thread(Thread-1, started 123145552101376)>',), <traceback object at
0x102320548>)
```

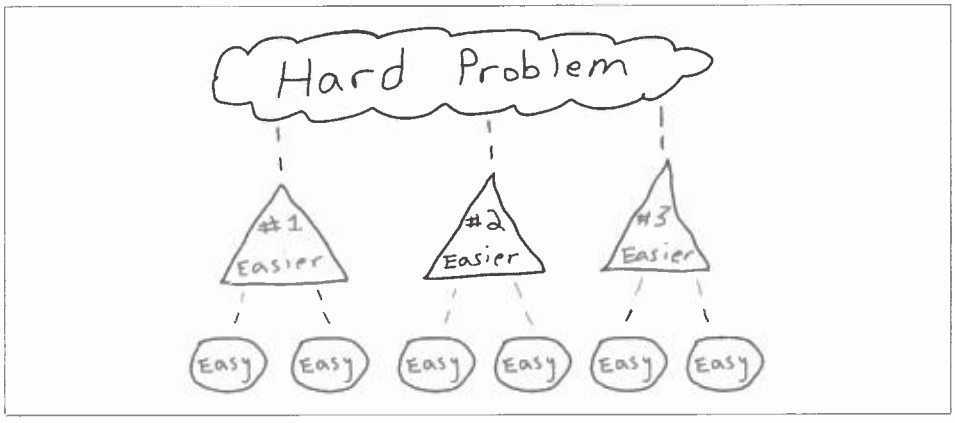
벤치마킹

코드에서 벤치마킹[benchmarking]이란 작업이 얼마나 빨리 완료되는지 그 성능을 측정하는 것을 말한다. 예컨대, 이전에 다룬 웹 크롤러를 벤치마킹하고자 한다면 일반적으로 초당 인덱싱 가능한 페이지 수를 측정한다.

성능 최적화를 진행할 때 전체적인 프로그램의 현재 상태를 나타내는 벤치마킹을 시작하는데, 그 후 마이크로 벤치마킹의 조합을 이용하고, 프로파일링으로 프로그램을 최적화하고 많은 양의 데이터를 처리한다.

여기서 마이크로 벤치마킹이란 애플리케이션을 여러 단계로 쪼개어, 코드의 에러 사항을

개별적인 단계로 벤치마킹하는 것을 말한다. 전체로는 최적화하기 어려운 문제를 여러 부분으로 쪼개어 최적화와 튜닝을 쉽게 할 수 있다.



▲ 출처: https://nickjanetakis.com/blog/breaking-down-problems-is-the-number-1-software-developer-skill

그럼 어떻게 코드를 벤치마킹할 수 있을까? 다행히도 파이썬에서 활용할 수 있는 다양한 방법이 있으니 살펴보자.

timeit 모듈

파이썬의 `timeit` 모듈을 이번 장에서 사용해본다. 기본적으로 파이썬에는 메인 애플리케이션의 작은 파이썬 코드 단위의 성능을 측정하도록 제공하는 `timeit` 모듈이 있다.

`timeit` 모듈은 코드상의 벤치마킹을 하고, 반대로 명령행으로부터 호출해 원하는 시간의 코드 부분을 입력할 수 있는 등 유연한 모듈이다.

timeit 모듈의 공식 문서는 https://docs.python.org/3/library/timeit.html에서 볼 수 있다.

`timeit` 모듈을 시작하는 방법은 명령행 인터페이스를 통해 다뤄본다.

timeit과 time

time 모듈보다 timeit 모듈을 사용할 때 더 많은 장점이 있다. timeit 모듈은 time 모듈에 비해 더 정확한 시간을 측정하고자 개발됐다.

timeit 모듈은 프로그램을 여러 번 실행해 OS에 관련 없는 요인으로부터 영향을 덜 받아 정확한 측정이 가능하게 해 직접적인 컨트롤이 필요 없다.

명령행 예제

명령행에 timeit 모듈을 이용하려면 이 책에서 다룬 코드 프로파일링이 다소 필요하다. 동시성 파이썬 깃허브 저장소에 다양한 예제가 있으니, 지금까지 다룬 다양한 동시성 개념을 폭넓게 볼 수 있다.

코드에 timeit 불러오기

다음 예제에서 2개의 개별적인 함수를 실행하는 데 걸린 시간을 측정하는 데 timeit 모듈을 사용해본다.

```
import timeit
import time

def func1():
  print("Function 1 Executing")
  time.sleep(5)
  print("Function 1 complete")

def func2():
  print("Function 2 executing")
  time.sleep(6)
  print("Function 2 complete")

start_time = timeit.default_timer()
func1()
```

```
print(timeit.default_timer() - start_time)

start_time = timeit.default_timer()
func2()
print(timeit.default_timer() - start_time)
```

위 코드에서 시간 측정을 실행하고, 함수를 실행하고, 측정이 종료되면 두 시간차를 정확히 출력한다.

두 함수에 걸린 시간을 측정하고자 한다면, 굳이 timeit 모듈을 사용할 필요가 없어 보인다.

```
import timeit
import time

def func1():
  print("Function 1 Executing")
  time.sleep(3)
  print("Function 1 complete")

def func2():
  print("Function 2 executing")
  time.sleep(2)
  print("Function 2 complete")

t1 = timeit.Timer("func1()", setup="from __main__ import func1")
times = t1.repeat(repeat=2, number=1)
for t in times:
  print("{} Seconds: ".format(t))

t2 = timeit.Timer("func2()", setup="from __main__ import func2")
times = t2.repeat(repeat=2, number=1)
for t in times:
  print("{} Seconds: ".format(t))
```

앞의 코드에서는 timeit 내에 시간을 측정하고 필요한 함수를 불러오는 2개의 Timer 객체를 실행하고자 인스턴스화한다.

각 Timer 객체에서 .repeat() 메소드를 호출해 repeat = 2 및 number = 1 인자로 몇 번이나 코드가 반복되고 실행되는지 결정한다.

코드를 실행하면 다음과 같은 결과가 나온다.

```
$ python3.6 timeitCode.py
Function 1 Executing
Function 1 complete
Function 1 Executing
Function 1 complete
3.002750840038061 Seconds:
3.0001289139618166 Seconds:
Function 2 executing
Function 2 complete
Function 2 executing
Function 2 complete
2.0005433409824036 Seconds:
2.00145923596574 Seconds:
```

데코레이터 활용하기

하지만 다소 지나친 코드를 수동으로 삽입하거나 불필요한 코드 작성을 피해야 할 때가 있다. 다행히도 파이썬에서는 이런 문제를 해결해준다.

decorator 함수를 정의해 자동으로 timeit.default_timer()에 대한 2개의 호출을 통해 함수의 실행을 감쌀 수 있다. 그 후 이 둘의 차이점을 찾아 콘솔에 출력한다.

```
import random
import timeit
import time
```

```
def timethis(func):

  def function_timer(*args, **kwargs):
    start_time = timeit.default_timer()
    value = func(*args, **kwargs)
    runtime = timeit.default_timer() - start_time
    print("Function {} took {} seconds to run".format(func.__name__, runtime))
    return value
  return function_timer

@timethis
def long_runner():
  for x in range(3):
    sleep_time = random.choice(range(1,3))
    time.sleep(sleep_time)

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

위 코드는 해당 함수와 연결된 모든 함수가 실행되는 데 걸리는 정확한 시간을 출력한다. 이를 실행하면 다음과 같은 출력을 볼 수 있다.

```
$ python3.6 05_timeitDecorator.py
Function long_runner took 5.008787009981461 seconds to run
```

타이밍 컨텍스트 관리자

컨텍스트 관리자란 with 문을 실행할 때 나타나는 런타임 컨텍스트를 정의하는 객체다.

파이썬에서 자체 Timer 컨텍스트 관리자 객체를 정의해 코드에 아무런 영향 없이 특정 부분의 시간을 측정할 때 사용할 수 있다.

여기서 컨텍스트 관리자 객체는 다음과 같다.

```
from timeit import default_timer

class Timer(object):
  def __init__(self, verbose=False):
    self.verbose = verbose
    self.timer = default_timer
  def __enter__(self):
    self.start = default_timer()
    return self
  def __exit__(self, *args):
    end = default_timer()
    self.elapsed_secs = end - self.start
    self.elapsed = self.elapsed_secs * 1000 # 밀리초
    if self.verbose:
      print('elapsed time: %f ms' % self.elapsed)
```

생성자, 입력점, 출력점으로 구성된 Timer 클래스를 정의한다. 입력을 하면 Timer가 경과 시간을 계산할 때까지 실행된다.

그런 다음 아래와 같이 파이썬 프로그램에서 이용할 수 있다.

```
from timer import Timer
from urllib.request import Request, urlopen
import ssl

def myFunction():
  # 컨텍스트를 생성해 https 사이트를 크롤링한다.
  myssl = ssl.create_default_context()
  myssl.check_hostname=False
  myssl.verify_mode=ssl.CERT_NONE
  with Timer() as t:
    req = Request('https://tutorialedge.net', headers={'User-Agent': 'Mozilla/5.0'})
    response = urlopen(req, context=myssl)

print("Elapsed Time: {} seconds".format(t.elapsed))
myFunction()
```

출력

코드를 실행하면, 다음과 같이 요청에 걸리는 전체 경과 시간을 볼 수 있다.

```
$ python3.6 04_timeitContext.py
Elapsed Time: 0.5995572790270671 seconds
```

프로파일링

코드를 프로파일링profiling할 때는 메모리를 얼마나 사용하는지, 그리고 프로그램의 시간 복잡성 및 특정 작업의 사용과 같은 핵심 속성을 측정해야 한다. 이는 프로그래머가 시스템에서 높은 성능을 뽑아낼 때 가장 잘 사용하는 도구 중 하나다.

일반적으로 프로파일링은 동적 프로그램 분석dynamic program analysis이라고 불리는 기술로 측정을 시도하는데, 실제 혹은 가상 프로세서상에서 프로그램을 실행한다. 이러한 기술은 1970년대 초반 IBM/360 및 IBM/370 플랫폼 시대로 거슬러 올라간다.

cProfile

cProfile은 파이썬에 기본적으로 구성된 C 기반 모듈이다. 코드상에 나타나는 다음 항목을 살펴보면서 이해해보자.

- ncalls: 프로그램 실행에서 함수가 호출된 횟수
- tottime: 특정 줄 및 함수가 실행되는 데 걸린 전체 시간
- percall: 전체 시간을 호출 횟수로 나눈 값
- cumtime: 특정 줄 및 함수가 실행되는 데 소비한 누적 시간
- percall: cumtime을 호출 횟수로 나눈 값
- filename:lineno(function): 실제 측정하고자 하는 줄 및 함수

이전 파이썬 예제에서 이러한 속성을 얻고자 cProfile 모듈을 활용해 살펴보자.

간단한 프로파일 예제

이번 예제에서는 이전 장에서 다룬 프로그램을 사용해보는데, 더블 앤드 큐에 요소를 집어넣는 것을 보여주겠다. 많이 신경 쓸 필요 없이 단순한 코드이므로, 간단히 결과를 살펴볼 수 있다.

```
import collections

doubleEndedQueue = collections.deque('123456')

print("Deque: {}".format(doubleEndedQueue))

doubleEndedQueue.append('1')
print("Deque: {}".format(doubleEndedQueue))

doubleEndedQueue.appendleft('6')
print("Deque: {}".format(doubleEndedQueue))
```

위 프로그램에서 cProfile을 호출하면, 프로그램을 전체적으로 먼저 실행하고, 코드를 바탕으로 표식화된 통곗값을 보여주기 전에 콘솔 창에 결과를 출력한다.

```
$ python3.6 -m cProfile 04_appendDeque.py
Deque: deque(['1', '2', '3', '4', '5', '6'])
Deque: deque(['1', '2', '3', '4', '5', '6', '1'])
Deque: deque(['6', '1', '2', '3', '4', '5', '6', '1'])
        11 function calls in 0.000 seconds

  Ordered by: standard name

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.000    0.000    0.000    0.000 04_appendDeque.py:1(<module>)
       1    0.000    0.000    0.000    0.000 {built-in method
builtins.exec}
       3    0.000    0.000    0.000    0.000 {built-in method
```

```
builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'append' of
'collections.deque' objects}
        1    0.000    0.000    0.000    0.000 {method 'appendleft' of
'collections.deque' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
        3    0.000    0.000    0.000    0.000 {method 'format' of 'str'
objects}
```

보다시피, 위 코드는 실행에 시간이 소요되지 않았다. append와 appendleft 메소드가 각 총계를 출력했음을 볼 수 있고, 매우 적은 시간이 소요됐다. 마지막 줄에서 format 함수가 세 번 호출됨을 볼 수 있는데, 마찬가지로 매우 적은 시간이 소요됐다.

좀 더 어려운 프로그램을 cProfile을 사용해 다뤄보자.

```
import threading
import random
import time

def myWorker():
for i in range(5):
  print("Starting wait time")
  time.sleep(random.randint(1,5))
  print("Completed Wait")

thread1 = threading.Thread(target=myWorker)
thread2 = threading.Thread(target=myWorker)
thread3 = threading.Thread(target=myWorker)

thread1.start()
thread2.start()
thread3.start()

thread1.join()
thread2.join()
thread3.join()
```

이 프로그램에서 cProfile을 실행하면 이전 프로그램과 비교해 그렇게 복잡하지 않을 텐데, 좀 더 많은 항목이 출력될 것이다.

이 예제에서는 출력이 좀 더 직관적인데, 코드상의 성능 감소가 일어난 부분을 쉽게 알 수 있어 속도를 높이고 최적화하는 데 좀 더 도움이 된다.

```
5157 function calls (5059 primitive calls) in 17.354 seconds

  Ordered by: standard name

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.000    0.000   17.354   17.354 01_timeit.py:1(<module>)
...
```

위 출력에서 함수 호출의 횟수는 5157번으로, 이전의 11번 호출에 비해 몇백 배 늘어났다.

이는 두 프로그램이 얼마나 극명한 성능 차이가 있는지와, 배경지식 없이 처리할 수 있는 순수 작업량을 보여준다. 여타 도구와 마찬가지로 cProfile은 시스템의 내부 작업에 중요한 관점을 제공해주며, 시스템의 전반적인 성능을 높일 수 있도록 다양한 정보를 제공한다.

line_profiler 툴

함수 내에 많은 코드가 있는 오늘날에는 기존의 방법으로 시간을 계산해 모든 줄을 감싸는 것이 거의 불가능하다. 다행히도 이러한 작업을 자동으로 해줄 수 있는데, 코드상에서 부하가 높은 지점을 찾는 등 세세한 분석도 가능하다.

line_profiler 툴의 경우 이러한 작업을 프로그램 실행 시 한 줄씩 따라가며 진행한다. 이는 모든 코드의 기본적인 소요 시간을 계산해 작업량을 줄여준다. 파이썬에 기본적으로 내장돼 있지 않지만 pip 모듈로 쉽게 설치 가능하다.

line_profiler 툴을 pip로 다음과 같이 설치해보자.

```
pip install line_profiler
```

오픈소스인 line_profiler 툴의 작동 방식에 관심이 있다면, https://github.com/rkern/line_profiler를 참조하길 바란다.

kernprof 툴

kerprof 툴은 기본적으로 line_profiler 툴에 포함된 툴이며, line_profiler 툴을 사용해 쉽게 사용할 수 있다. line_profiler를 이용해 프로파일하고자 하는 코드의 함수 상태를 볼 수 있다. 이는 기본적으로 @profile 데코레이터를 이용한다.

이 예제에서는 느린 함수와 빠른 함수, 2개를 갖춘 간단한 프로그램을 만들어보자. slowFunction( )은 프로그램에 안 좋은 영향을 끼치며, 막대한 코드 양으로 인해 시간을 계산하는 모든 코드를 감쌀 수 없다.

모든 코드가 실행되는 데 얼마가 소요되는지 한 줄씩 정확히 분석하려면, 다음과 같이 어떤 것이 잘못됐는지 보고자 @profile 데코레이터를 추가할 수 있다.

```
import random
import time

@profile
def slowFunction( ):
  time.sleep(random.randint(1,5))
  print("Slow Function Executed")

def fastFunction( ):
  print("Fast Function Executed")
```

```
def main():
  slowFunction()
  fastFunction()

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

툴을 실행하려면, .lprof 파일을 생성하고자 kernprof 툴을 먼저 호출해야 한다. 다음은 정확한 출력을 보고자 line_profiler 툴을 거쳐 생성된 파일이니 참고하자.

```
$ python3.6 -m kernprof -l profileTest.py
Slow Function Executed
Fast Function Executed
Wrote profile results to profileTest.py.lprof

$ python3.6 -m line_profiler profileTest.py.lprof
Timer unit: 1e-06 s

Total time: 3.00152 s
File: profileTest.py
Function: slowFunction at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    4                                           @profile
    5                                           def slowFunction():
    6         1      3001412 3001412.0    100.0
time.sleep(random.randint(1,5))
    7         1          106    106.0      0.0    print("Slow Function
Executed")
```

보다시피, kernprof를 처음 프로그램에서 실행하면 성공적으로 profileTest.py.lprof 파일이 생성된다. 그 후 이 파일을 line_profiler 툴에 넣어 실행한 코드의 줄 수와 줄마다 실행한 횟수, 횟수당 걸린 시간이 나타난 표를 출력한다.

6번 줄은 전체 시간의 100.0퍼센트를 차지했음을 볼 수 있다. 약 3초 정도가 걸렸는데, 프로그램의 실행 시간에 큰 영향을 줬다. 이를 바탕으로 불필요한 코드 부분을 파악해 이를 다시 수정하는 과정을 거쳐, 프로그램 실행 시간을 확실히 줄일 수 있다.

메모리 프로파일링

프로그램의 메모리 사용 프로파일은 중급 개발자가 효과적으로 시스템 문제를 디버깅하는 데 꼭 필요한 기술 중 하나다. 나는 한정된 하드웨어를 고려하지 않고 개발하는 개발자를 종종 보곤 한다.

방대한 멀티 JVM 기반 시스템과 세부 작업 및 이에 걸맞은 사항이 포함된 jenkins 인스턴스 서버를 구축해봤는데, 서버에 남은 메모리를 모두 차지하는 특정 프로그램으로 인해 서버가 다운될 수도 있다. 왜 그럴까? 내 경우엔 애플리케이션 내에 적절한 가비지 컬렉션을 구현하지 않아서, 예외가 있을 경우 느리게 처리됐다.

내가 만든 프로그램의 메모리 사용을 분석할 때, 이러한 문제를 심각하게 고민했다. 파이썬 환경에서는 memory_profiler라는 툴이 있어 line_profiler와 마찬가지로 주어진 함수의 각 줄마다 메모리 사용량 정보 테이블을 생성한다.

메모리 프로파일러를 설치하기 위해 다음 명령어를 실행하자.

```
pip install -U memory_profiler
```

마찬가지로, 다음 프로그램을 테스트해본다.

```
import random
import time

@profile
def slowFunction():
```

```
  time.sleep(random.randint(1,5))
  print("Slow Function Executed")

def fastFunction():
  print("Fast Function Executed")

def main():
  slowFunction()
  fastFunction()

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

다행히, 느려진 함수에 메모리 프로파일링이 필요할 때 @profile 애노테이션을 사용할 수 있다.

```
$ python3.6 -m memory_profiler profileTest.py
```

이를 실행하고 나면 콘솔 창에서 line_profiler 출력과 비슷한 형태의 표를 볼 수 있다. 주로 프로파일링한 코드의 줄 수와 해당 메모리 사용량, 그리고 이를 실행할 때 얼마만큼의 메모리 사용 증가가 있었는지를 보여준다.

보다시피, 해당 프로그램의 slowFunction()은 코드 실행 시 시스템에 많은 부하가 없다.

```
Line #    Mem usage    Increment   Line Contents
================================================
    4   33.766 MiB    0.000 MiB   @profile
    5                             def slowFunction():
    6   33.777 MiB    0.012 MiB     time.sleep(random.randint(1,5))
    7   33.785 MiB    0.008 MiB     print("Slow Function Executed")
```

메모리 프로파일 그래프

시간 간격을 설정해 메모리 사용을 보고자 한다면, memory_profiler에 포함된 mprof 툴을 사용하면 된다. mprof 툴은 0.1초 간격으로 메모리 사용 샘플을 제공하며, .data 파일에 사용량을 그래프화한다.

이 .dat 파일은 프로그램의 시간별 메모리 사용을 보고자 matplotlib와 결합해 사용할 수 있다.

```
python -m pip install matplotlib
```

컴퓨터에 여러 파이썬 버전이 설치됐다면, mprof를 작업에 이용하기는 조금 까다로울 수 있다. 나는 맥 OS에서 다음과 같은 명령어로 mprof를 실행해봤다.

```
$ python3.6 /Library/Frameworks/Python.framework/Versions/3.6/bin/mprof run
profileTest.py
```

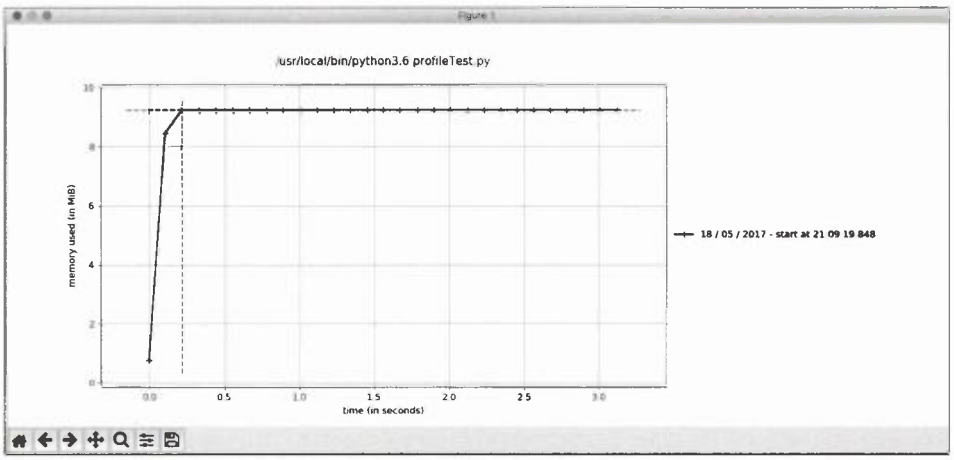
반대로 slowFunction()에 @profile 데코레이터 없이 이전에 사용한 코드를 실행해보자.

```
import random
import time

def slowFunction():
  time.sleep(random.randint(1,5))
  print("Slow Function Executed")

def fastFunction():
  print("Fast Function Executed")

def main():
  slowFunction()
  fastFunction()
```

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

실행 후 시간별로 메모리 사용량을 그래프화하고자 다음 명령어를 사용해보자.

```
$ python3.6 /Library/Frameworks/Python.framework/Versions/3.6/bin/mprof plot
```



그래프에서 볼 수 있듯이 간단하게 기본적인 형태로 출력됐는데, 각 포인트를 확인할 수 있다.

*y*축은 MiB 단위로 전체 메모리 사용량을, *x*축은 초 단위로 시간을 나타낸다. 프로그램이 계속 실행될수록 전체 메모리 사용량 변동은 증가하며, 코드가 더 진행될수록 메모리 문제를 쉽게 확인할 수 있다.

요약

6장에서는 출시 단계에 이르기 전에 버그를 제거하는 등 동시성 파이썬 환경에서 실용적으로 활용할 수 있는 기술을 다양하게 살펴봤다. 코드 로직의 안정성을 보장하는 테스트 방법을 다루고, 버그를 수정할 때 필요한 사항을 다뤘다.

다음으로 파이썬 코드를 디버깅하는 다양한 방법을 내장 Pdb 툴을 사용해 명령행에서 대화형 형태로 사용해보며 알아봤다.

마지막으로, 파이썬 애플리케이션을 벤치마킹하고 프로파일링하는 다양한 기술을 다루고 효율적으로 이를 살펴봤다.

7장에서는 파이썬 Asyncio 라이브러리를 살펴보고, 실행자executor와 풀pool을 활용해 파이썬 애플리케이션의 성능을 향상해본다.

07

실행자와 풀

7장에서는 스레드 풀과 프로세스 풀의 개념, 그리고 프로그램의 실행 속도를 높이고자 이를 파이썬에서 구현하는 방법을 다룬다.

7장에서 다루는 내용은 다음과 같다.

- 동시성 퓨처
- 퓨처 객체
- 프로세스 풀 실행자

5장에서 연습해본 웹 크롤러 예제를 가져와, 결과를 CSV 파일로 저장하고 코드를 리팩토링하는 기능을 더함으로써 더 자세히 사용해본다.

마지막으로, 스레드와 프로세스를 다루는 것과 관련된 작업을 간단히 하는 방법과 실행자 객체를 활용해 파이썬 프로그램의 성능을 향상하는 부분 또한 살펴본다.

동시성 퓨처

동시성 퓨처future는 파이썬 3.2에서 새로 추가된 기능이다. 자바 기반의 배경지식이 있다면 ThreadPoolExecutor에 대해 잘 알 것이다. 동시성 퓨처는 ThreadPoolExecutor를 파이썬에서 구현한 형태다.

멀티스레드 작업을 실행할 때, 가장 많은 연산을 필요로 하는 작업은 스레드를 시작하는 것이다. ThreadPoolExecutor는 스레드 풀이 필요할 동안 이를 생성함으로써 문제를 다룬다. 사용자는 작업을 수행할 때 더 이상 스레드를 생성하거나 실행할 필요가 없으며, 덕분에 한 번만 스레드를 생성해 새로운 작업에 기여한다.

다양한 직원이 있는 사무실을 상상해보자. 직원을 고용하지 않고 교육 후 한 명씩 올바른 부서에 배치한다. 대신 많은 비용이 드는 직원 교육을 한 번으로 끝내고, 다양한 부서로 업무에 맞춰 배치해 효율성을 높이게 된다.

스레드 풀도 이와 같다. 여러 개의 스레드를 한 번에 불러와, 실행되는 동안 다양한 작업이 이뤄지기 전에 스레드를 생성하는 일을 한 번만 한다.

Executor 객체

Executor 클래스는 concurrent.futures 모듈에 있으며, 동시성 형태로 여러 가지 호출을 지원한다. 자체적인 객체 및 컨텍스트 관리자 같은 방법으로 사용되고, 스레드 생성, 시작, 조인 같은 지루한 작업을 가독성이 높은 코드로 바꾸는 작업을 한다.

ThreadPoolExecutor 생성하기

먼저 ThreadPoolExecutor를 정의하는 방법을 살펴보자. 보다시피 매우 간단한 형태다.

```
executor = ThreadPoolExecutor(max_workers=3)
```

이 코드는 ThreadPoolExecutor를 인스턴스화해 최대 작업자 수를 인자로 취한다. 여기서는 3을 취해 스레드 풀이 사용자가 제출하는 작업을 처리하는 3개의 동시성 스레드만 갖는다.

ThreadPoolExecutor에 스레드를 전달하고자 한다면, submit() 함수를 호출해 다음과 같이 첫 번째 파라미터를 취한다.

```
executor.submit(myFunction())
```

예제

이 예제에서는 ThreadPoolExecutor 객체를 생성하고 새로 인스턴스화된 객체에 작업을 제출하는 방법을 모두 배워본다. 0에서 9까지의 숫자를 간단히 더해보는 함수를 구현하고, 결과를 출력한다. 완벽한 소프트웨어는 아니지만, 적절한 예제 역할을 하고 있다.

task 함수를 정의한 후, 기본적인 main 함수를 구현한다. 여기서 executor 객체는 2개의 작업을 새로운 스레드 풀에 제출하기 전에 언급한 동일한 형태로 정의한다.

```
from concurrent.futures import ThreadPoolExecutor
import threading
import random

def task():
  print("Executing our Task")
  result = 0
  i = 0
  for i in range(10):
    result = result + i
  print("I: {}".format(result))
  print("Task Executed {}".format(threading.current_thread()))

def main():
  executor = ThreadPoolExecutor(max_workers=3)
```

```
  task1 = executor.submit(task)
  task2 = executor.submit(task)

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

출력

위 프로그램을 실행하면 비교적 간단한 출력이 나타나고, 계산 결과는 명령행에 출력된다.

어떤 스레드가 해당 작업을 수행하는지 알고자 threading.current_thread() 함수를 활용한다. 출력된 두 값이 각기 구별되는 데몬 스레드임을 볼 수 있다.

```
$ python3.6 05_threadPool.py
Executing our Task
I: 45
Executing our Task
I: 45
Task Executed <Thread(<concurrent.futures.thread.ThreadPoolExecutor object
at 0x102abf358>_1, started daemon 123145333858304)>
Task Executed <Thread(<concurrent.futures.thread.ThreadPoolExecutor object
at 0x102abf358>_0, started daemon 123145328603136)>
```

컨텍스트 관리자

두 번째로 자주 사용되는 방법은 다음과 같이 컨텍스트 관리자를 이용해 ThreadPool Executor를 인스턴스화하는 것이다.

```
with ThreadPoolExecutor(max_workers=3) as executor:
```

위 방법은 이전 절에서 살펴본 방법과 동일한 역할을 하지만, 문법적으로 더 훌륭하며 개발자들에게 더 좋을 수도 있다.

컨텍스트 관리자는 문법적으로 훌륭한 코드를 작성하는 데 많은 도움을 준다.

예제

이번에는 변수 n을 입력으로 사용하는 그 밖의 작업을 정의해본다. task 함수는 'n'을 처리하고 있다는 출력문만 나타낸다.

메인 함수에서는 컨텍스트 관리자가 ThreadPoolExecutor를 활용해 future = executor.submit(task, (n))을 세 번 호출해 스레드 풀에서 스레드를 전달한다.

```
from concurrent.futures import ThreadPoolExecutor

def task(n):
  print("Processing {}".format(n))

def main():
  print("Starting ThreadPoolExecutor")
  with ThreadPoolExecutor(max_workers=3) as executor:
    future = executor.submit(task, (2))
    future = executor.submit(task, (3))
    future = executor.submit(task, (4))
  print("All tasks complete")

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

출력

위 프로그램을 실행하면 제출된 3개의 작업을 실행하기 전에 ThreadPoolExecutor를 시작하고, '모든 작업이 완료됨'이 출력된다.

```
$ python3.6 01_threadPoolExe.py
Starting ThreadPoolExecutor
Processing 2
Processing 3
```

```
Processing 4
All tasks complete
```

맵

파이썬의 맵은 특정 함수를 iterables 내의 모든 요소에 적용해준다.

```
map(func, *iterables, timeout=None, chunksize=1)
```

다행히도 파이썬에서는 반복자의 모든 요소를 함수에 매핑할 수 있으며, ThreadPool Executor에 독립적인 작업으로 요소들을 제공한다.

```
results = executor.map(multiplyByTwo, values)
```

기본적으로 다음 예제와 같이 간단하게 구현된다.

```
for value in values:
  executor.submit(multiplyByTwo, (value))
```

예제

이 예제에서는 multiplyByTwo 함수를 배열의 모든 값에 적용하고자 새로운 맵 함수를 사용한다.

```
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import as_completed

values = [2,3,4,5,6,7,8]

def multiplyByTwo(n):
  return 2 * n
```

```
def main():
  with ThreadPoolExecutor(max_workers=3) as executor:
    results = executor.map(multiplyByTwo, values)
    for result in results:
      print(result)

if __name__ == '__main__':
  main()
  results = list(map(multiplyByTwo, values))
print(results)
```

출력

보다시피 컴퓨터에서 위의 프로그램을 실행하면 원하는 형태로 출력되며, 배열의 모든 곱셈 결과는 map 함수가 연산을 종료하고 출력한다.

```
$ python3.6 03_threadPoolMap.py
4
6
8
10
12
14
16
[4, 6, 8, 10, 12, 14, 16]
```

실행 객체 종료하기

올바른 방법으로 실행자executor 객체를 종료할 수 있다는 것은 다양한 상황에서 중요한 부분이다. 이는 사용자가 필요로 할 때 유연하게 대처한다.

실행자 객체를 종료할 때 가장 중요한 것은 더 이상 작업을 진행할 수 없다는 점이다. 실행자 객체를 바탕으로 이미 진행 중인 작업은 종료 호출 후 즉시 종료될 테지만, 추가적인 작업을 실행자 객체에 전달할 경우 에러가 발생한다.

예제

이 예제에서는 실행 중인 실행자를 종료하는 방법에 대해 설명한다. 먼저 n초간 '작동'하는 함수를 정의한다. 그 후 작업 횟수를 전달하고 실행자에 종료 메소드를 호출한다. 이 때부터 실행자에 더 이상 작업이 전달되지 않는다.

```
import time
import random
from concurrent.futures import ThreadPoolExecutor

def someTask(n):
  print("Executing Task {}".format(n))
  time.sleep(n)
  print("Task {} Finished Executing".format(n))

def main():
  with ThreadPoolExecutor(max_workers=2) as executor:
    task1 = executor.submit(someTask, (1))
    task2 = executor.submit(someTask, (2))
    executor.shutdown(wait=True)
    task3 = executor.submit(someTask, (3))
    task4 = executor.submit(someTask, (4))

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

출력

위의 파이썬 프로그램을 실행하면 작업 1과 작업 2가 ThreadPoolExecutor를 통해 성공적으로 실행된다. executor.shutdown(wait=True)를 호출한 후, 실행자 객체가 더 이상 작업을 받지 않도록 스택 추적이 콘솔에 출력된다.

이 예제에서는 따로 예외를 처리하고자 하지 않았으며, 프로그램이 적절히 종료되지 않았다. 이와 같은 상황을 다루기 위해 종료 요소를 구현하고자 한다면 예외 처리에 관한 부분도 구현해야 함을 명심하자.

```
$ python3.6 11_shutdownExecutor.py
Executing Task 1
Executing Task 2
Task 1 Finished Executing
Task 2 Finished Executing
.. stack trace removed for brevity
RuntimeError: cannot schedule new futures after shutdown
FAIL
```

퓨처 객체

퓨처 객체는 이전의 ThreadPoolExecutor 예제에서 실행자에 작업을 전달할 때 인스턴스화된다. 퓨처 객체는 결국 주어진 값이 이후에 값이 되는 객체다.

퓨처 객체 내의 메소드

퓨처 객체에는 다음과 같이 접근 및 수정이 가능한 메소드가 있다. 이러한 메소드는 이후 샘플 코드를 바탕으로 자세히 다룬다.

result() 메소드

result() 메소드는 퓨처 객체의 모든 반환값을 전달한다. 이는 다음과 같이 호출한다.

```
futureObj.result(timeout=None)
```

timeout 파라미터를 명시해, 기본적으로 퓨처 객체에 시간 제한을 둘 수 있다. 퓨처 객체가 주어진 시간 범위 내에 완료하지 못한다면, concurrent.futures에서 timeout 에러가 발생한다. 이는 스레드가 실행할 수 있는 시간 에러 양을 조절하고, 프로그램 성능 감소를 막을 수 있다는 점에서 꽤 유용하다.

add_done_callback() 메소드

퓨처를 다룰 때, 퓨처의 완료 지점에 실행되는 콜백 함수를 명시할 수 있다. 이는 모든 퓨처 객체의 상태를 파악하고, 코드를 간단히 해준다. add_done_callback 메소드를 활용해 퓨처 객체에 콜백을 추가할 수 있다.

```
futureObj.add_done_callback(fn)
```

running() 메소드

파이썬 프로그램에서 모르는 퓨처 객체가 현재 실행되고 있는지 알고 싶을 때가 있는데, 이때 running() 메소드를 호출해 현재 상태를 확인할 수 있다.

```
futureObj.running()
```

위 메소드는 퓨처 객체의 현재 실행 상태에 따라 참 또는 거짓을 반환한다.

cancel() 메소드

cancel() 메소드는 퓨처 객체를 취소할 때 사용되며, 퓨처 객체가 실행되기 전에만 호출할 수 있다.

```
futureObj.cancel()
```

exception() 메소드

exception() 메소드를 사용해서는 퓨처로부터 전달받은 예외를 검색할 수 있다. timeout 파라미터를 명시해 기본적으로 퓨처 객체가 x초 안에 완료하지 않았을 경우 concurrent.futures.TimeoutError를 던진다.

```
futureObj.exception(timeout=None)
```

done() 메소드

done() 메소드는 퓨처 객체가 성공적으로 종료했는지 실패했는지에 따라 참 또는 거짓을 반환한다.

```
futureObj.done()
```

퓨처 객체의 단위 테스트

동시성 애플리케이션의 단위 테스트는 다소 어려운 부분이지만 다행히도 concurrent.futures에서 지원하고 있다. future 객체는 다음 세 가지 방법으로 단위 테스트를 구현하고 있다.

set_runnining_or_notify_cancel() 메소드

set_running_or_notify_cancel() 메소드는 퓨처 객체가 완료됐는지 취소됐는지 확인할 때 활용된다.

```
futureObj.cancel() # 시작하기 전 퓨처를 취소한다.
futureObj.)set_runnining_or_notify_cancel() # False를 반환한다.
…
futureObj.running() # True를 반환한다.
futureObj.)set_runnining_or_notify_cancel() # True를 반환한다.
```

set_result() 메소드

set_result() 메소드는 단위 테스트상의 퓨처 객체 결과를 설정한다. 실행자의 결과를 확인하는 등 다양한 경우에 활용된다.

이는 퓨처 객체에 숨겨진 논리 구조를 테스트하지 않고자 하는 상황에서 사용되지만, 퓨처 객체가 반환한 후의 상황 논리 구조는 테스트한다. 예컨대, 웹 크롤러를 테스트하고 크롤링의 결과를 수행하는 메소드가 있다고 하자. 사용자는 퓨처 객체의 결과를 기댓값에 설정할 수 있고, 이러한 특정 값의 테스트를 만들 수 있다.

다음과 같이 set_result를 호출해보자.

```
futureObj.set_result(result)
```

set_exception() 메소드

set_result 메소드와 마찬가지로, set_exception() 메소드는 퓨처 객체가 반환할 예외를 설정한다.

```
futureObj.set_exception(exception)
```

호출 가능한 작업 취소하기

시스템에서 실행자 객체로부터 처리되기 전 어떤 작업을 취소하고자 할 때가 있다. 이는 실제로 프린터를 하는 상황에서 프린터 대기열에 있는 원하는 출력 작업을 취소하고자 하는 경우다.

하지만 일반적인 프린터는 출력이 진행되는 중간에 취소할 수 없다. 이는 ThreadPoolExecutor 및 ProcessPoolExecutor에 전달되는 작업에 적용된다.

실행자에 전달된 작업을 취소하려면 다음과 같이 특정 작업에 cancel() 함수를 호출하면 된다.

```
with ThreadPoolExecutor(max_workers=2) as executor:
  myTask = executor.submit(someTask, (1))
  print(myTask.cancel())
```

cancel 함수는 성공적으로 퓨처 객체를 취소했으면 참을, 아니면 거짓을 반환한다. 다음 예제에서는 실행자 객체를 갖고 있는 myTask에 앞서 작업을 전달함에도 불구하고 거짓을 반환한다.

예제

제대로 된 예제를 바탕으로 더 자세히 살펴보자. 먼저 ThreadPoolExecutor에 넣은 임의의 작업을 정의하는데, 이를 someTask라 하며 실행 상태를 출력하고 n초 동안 대기한다.

메인 함수에서는 마찬가지로 with 명령어를 사용해 ThreadPoolExecutor를 컨텍스트 관리자로 인스턴스화하며, 4개의 각 작업을 전달한다. 해당 작업을 모두 전달하면, task3.cancel()의 결과를 출력한다.

ThreadPoolExecutor에서 max_workers=2라고 정의했으니, task3.cancel()을 수행하면 실행자는 작업 1과 2를 미리 수행하고, 작업 3은 시작하지 않는다.

```
import time
import random
from concurrent.futures import ThreadPoolExecutor

def someTask(n):
  print("Executing Task {}".format(n))
  time.sleep(n)
  print("Task {} Finished Executing".format(n))

def main():
  with ThreadPoolExecutor(max_workers=2) as executor:
    task1 = executor.submit(someTask, (1))
    task2 = executor.submit(someTask, (2))
    task3 = executor.submit(someTask, (3))
    task4 = executor.submit(someTask, (4))

    print(task3.cancel())

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

출력

앞의 코드를 실행하면 4개의 작업이 ThreadPoolExecutor에 전달된다. 작업 1과 2는 풀 내의 두 작업자로부터 선택된다. 풀에 2개의 작업이 있는 동안, 사용자는 작업 3을 취소한다. 아직 시작되지 않았기에, 콘솔에 참으로 출력됨을 볼 수 있다. 그 후 작업 1, 2, 4가 실행되고 프로그램은 종료된다.

```
$ python3.6 10_cancellingCallable.py
Executing Task 1
Executing Task 2
True
Task 1 Finished Executing
Executing Task 4
Task 2 Finished Executing
Task 4 Finished Executing
```

결과 얻어내기

일반적으로 몇몇 상황에서는 실행자 객체의 작업을 취소하거나 모를 수도 있으며, 반환값을 생각하지 않을 수도 있다. 하지만 이러한 결과를 검색하고 저장할 수 있는 방법이 있다.

executor.map 함수를 활용해 간단하게 결과를 검색하고, executor.map에 대한 호출에서 반환된 결과를 검색하고자 다음 코드를 활용할 수 있다.

```
for result in results:
  print(result)
```

예제

다음 코드에서는 위의 방법을 바탕으로 결과를 검색할 수 있는 예제를 보여준다. 입력값 배열을 선언하고, executor.map을 호출해 배열의 값을 실행자에서 선택된 작업과 매핑해 처리한다.

```
import time
import random
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import as_completed

values = [2,3,4,5,6,7,8]

def multiplyByTwo(n):
  time.sleep(random.randint(1,2))
  return 2 * n

def done(n):
  print("Done: {}".format(n))

def main():
  with ThreadPoolExecutor(max_workers=3) as executor:
    results = executor.map(multiplyByTwo, values)

    for result in results:
      done(result)

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

출력

위 프로그램을 실행하면 배열에서 전달된 순서에 따라 작업의 결과가 저장되고 출력된다.

```
$ python3.6 13_results.py
Done: 4
Done: 6
Done: 8
Done: 10
Done: 12
Done: 14
Done: 16
```

as_completed 사용하기

이와 같은 map 함수가 필요에 따라 적합하지 않을 수 있는데, 퓨처 객체의 배열 내 ThreadPoolExecutor에 전달하는 모든 퓨처 객체를 저장해야 할 수도 있다.

이러한 방법은 concurrent.futures 모듈의 as_completed 메소드를 활용해 실행자에 전달된 모든 작업 결과를 처리하고 반환할 수 있다.

예제

이 예제에서는 웹사이트가 작동하는지 확인하고자 몇 개의 링크로 구성된 배열을 생성한다. 해당 URL에 요청하고자 urllib.request 모듈을 활용할 것이며, http 상태 코드 반환 결과를 살펴본다.[1]

```
import time
from urllib.request import Request, URLError, urljoin, urlopen
from concurrent.futures import ThreadPoolExecutor, as_completed

URLS = [
  'http://www.google.com',
  'http://www.naver.com',
  'http://www.acornpub.co.kr',
  'http://www.knu.ac.kr'
]

def checkStatus(url):
  print("Attempting to crawl URL: {}".format(url))
  req = Request(url, headers={'User-Agent': 'Mozilla/5.0'})
  response = urlopen(req)
  return response.getcode(), url

def printStatus(statusCode):
  print("URL Crawled with status code: {}".format(statusCode))
```

1 URLS 배열에 본인이 원하는 사이트를 넣어 확인해본다. – 옮긴이

```
def main():
  with ThreadPoolExecutor(max_workers=3) as executor:

    tasks = []
    for url in URLS:
      task = executor.submit(checkStatus, (url))
      tasks.append(task)

    for result in as_completed(tasks):
      printStatus(result)

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

출력

웹 크롤러를 실행하면 크롤링하고자 하는 URL이 출력된다. 각 요청이 완료되면 프로그램은 퓨처 객체와 해당 상태를 출력한다.

여기서는 순서에 상관없으니 참고하자. 실행자에 전달된 순서와 반환되는 순서가 같지 않을 것이다. 몇몇 작업은 시간이 더 오래 걸린다. 순서를 유지하고 싶으면, 정렬 알고리즘을 사용해 결과 반환값 옆에 인덱스를 표시하면 된다.

```
$ python3.6 14_asCompleted.py
Attempting to crawl URL: http://www.google.com
Attempting to crawl URL: http://www.naver.com
Attempting to crawl URL: http://www.acornpub.co.kr
Attempting to crawl URL: http://www.knu.ac.kr
URL Crawled with status code: (200, 'http://www.acornpub.co.kr')
URL Crawled with status code: (200, 'http://www.knu.ac.kr')
URL Crawled with status code: (200, 'http://www.google.com')
URL Crawled with status code: (200, 'http://www.naver.com')
```

콜백 설정하기

콜백에 대해서는 9장과 10장에서 이벤트 기반 프로그래밍과 반응형 프로그래밍을 다룰 때 자세히 살펴볼 것이다. 콜백을 이해하기 위해, 누군가에게 시간이 꽤 걸리는 어떤 작업을 부탁하는 상황을 상상해보자. 보통은 그 사람이 해당 작업을 끝낼 때까지 그냥 앉아서 기다리는 게 아니라, 다른 업무를 보고 있을 것이다.

대신에, 작업을 완료했을 때 다시 호출해주기를 요청할 것이다. 프로그래밍에서는 이러한 호출을 콜백[callback]이라고 하며, ThreadPoolExecutor와의 결합을 이용한 개념이다.

ThreadPoolExecutor에 작업을 전달하기까지 다음과 같이 add_done_callback을 사용한 함수로 콜백을 명시화한다.

```
with ThreadPoolExecutor(max_workers=3) as executor:
  future = executor.submit(task, (2))
  future.add_done_callback(taskDone)
```

여기서 taskDone 함수는 작업이 완료된 후 호출된다. 이러한 간단한 함수 호출 방식으로 작업이 전달되는 순간마다 살펴봐야 하는 번거로움이 사라지고, 결과적으로 한 번에 그 외 함수를 시작할 수 있게 된다. 이러한 콜백은 마지막에 작업을 간단하게 해주며, 매우 유용하다.

예제

콜백 함수를 이용한 간단한 코드를 작성해 살펴보자. 먼저 입력한 것에 따라 처리해 출력하는 task 함수와, 콜백 함수로 구현한 taskDone 함수를 정의해보자.

taskDone 함수에서는 퓨처 객체가 취소됐거나 완료됐는지 여부를 먼저 확인한다. 그러고 콘솔에 적절한 출력을 하게 된다.

메인 함수를 정의해 ThreadPoolExecutor를 생성하고 콜백을 구성할 때 단일 작업을 전달한다.

```
from concurrent.futures import ThreadPoolExecutor

def task(n):
  print("Processing {}".format(n))

def taskDone(fn):
  if fn.cancelled():
    print("Our {} Future has been cancelled".format(fn.arg))
  elif fn.done():
    print("Our Task has completed")

def secondTaskDone(fn):
  print("I didn't think this would work")

def main():
  print("Starting ThreadPoolExecutor")
  with ThreadPoolExecutor(max_workers=3) as executor:
    future = executor.submit(task, (2))
    future.add_done_callback(taskDone)
    future.add_done_callback(secondTaskDone)

  print("All tasks complete")

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

출력

위 프로그램을 실행하면 ThreadPoolExecutor가 시작되고, 작업이 전달되면 주어진 실행자가 이를 선택해 보다시피 콜백 함수가 실행된다.

```
$ python3.6 04_settingCallbacks.py
Starting ThreadPoolExecutor
Processing 2
Our Task has completed
I didn't think this would work
All tasks complete
```

콜백 연결하기

몇몇 상황에서 콜백 함수가 복잡해질 수 있는데, 규모가 큰 콜백 함수를 여러 개의 함수로 나누는 작업이 필요하다. 다행히도 하나의 퓨처 객체에 여러 개의 콜백을 추가해 동일한 결과를 나타내고자 이들을 연결할 수 있다.

```
with ThreadPoolExecutor(max_workers=3) as executor:
  future = executor.submit(task, (2))
  future.add_done_callback(taskDone)
  future.add_done_callback(secondTaskDone)
```

여기서 콜백이 실행되는 순서는 해당 콜백이 추가된 순서임에 유의하자.

예외 클래스

이전 장에서는 자식 스레드에서 그 밖의 스레드로 예외를 전달하고자 큐를 활용해 일반적인 자식 스레드 내의 예외를 다루는 방법을 살펴봤다. 이러한 일이 다소 힘들지만, 그래도 ThreadPoolExecutor에서는 이 모든 것을 다룰 수 있도록 지원한다.

퓨처 객체에서 결과를 검색하는 방법과 동일하게 예외도 반환할 수 있다.

예제

예제에서 isEven 함수를 정의해 값을 넣어보자. 해당 함수에서 먼저 객체의 인자 타입이 정수인지 확인한다. 그렇지 않다면 새로운 예외를 발생시킨다.

확인을 마치면 해당 숫자가 홀수인지 짝수인지 확인한다. 짝수이면 참을, 그렇지 않으면 거짓을 반환한다.

```
from concurrent.futures import ThreadPoolExecutor
import concurrent.futures
import threading
```

```
import random

def isEven(n):
  print("Checking if {} is even".format(n))
  if type(n) != int:
    raise Exception("Value entered is not an integer")
  if n % 2 == 0:
    print("{} is even".format(n))
    return True
  else:
    print("{} is odd".format(n))
    return False

def main():
  with ThreadPoolExecutor(max_workers=4) as executor:
    task1 = executor.submit(isEven, (2))
    task2 = executor.submit(isEven, (3))
    task3 = executor.submit(isEven, ('t'))

  for future in concurrent.futures.as_completed([task1, task2, task3]):
    print("Result of Task: {}".format(future.result()))

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

출력

위 프로그램을 실행하면 2, 3, *t*가 짝수인지 확인하게 된다. 2와 3은 성공적으로 처리되지만, *t*의 경우 실행자가 예외를 전달하고 마친다.

*t*의 경우 유효하지 않은 입력 예외를 전달하며 다음과 같이 출력된다.

```
$ python3.6 12_exceptionThread.py
Checking if 2 is even
2 is even
Checking if 3 is even
```

```
3 is odd
Checking if t is even
Result of Task: True
Result of Task: False
Traceback (most recent call last):
  File "12_exceptionThread.py", line 28, in <module>
  # ...
    raise Exception("Value entered is not an integer")
Exception: Value entered is not an integer
```

ProcessPoolExecutor

ProcessPoolExecutor는 기본적으로 ThreadPoolExecutor와 동일한 기능으로 사용된다. ThreadPoolExecutor와 마찬가지로 Executor 클래스의 하위 클래스로, 많은 메소드가 동일하게 나타난다.

ProcessPoolExecutor 생성하기

ProcessPoolExecutor를 생성하는 과정은 concurrent.futures 모듈에서 클래스를 불러오는 부분과 다음과 같이 생성자 객체를 인스턴스화하는 것 빼고는 ThreadPoolExecutor와 동일하다.

```
executor = ProcessPoolExecutor(max_workers=3)
```

예제

다음 예제를 바탕으로 자체적인 ProcessPoolExecutor를 어떻게 인스턴스화하고 풀에 작업을 전달하는지 알아보자. 이 예제의 task 함수는 간단해서 일반적인 단일 스레드 기반 프로세스보다 느리므로, 멀티프로세스의 장점을 찾기 힘들다.

풀에서 실행하는 각 작업의 현재 PID를 찾고자 os 모듈을 사용해본다.

```
from concurrent.futures import ProcessPoolExecutor
import os

def task():
  print("Executing our Task on Process {}".format(os.getpid()))

def main():
  executor = ProcessPoolExecutor(max_workers=3)
  task1 = executor.submit(task)
  task2 = executor.submit(task)

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

출력

이를 실행하면 전달된 작업 모두 프로세스 ID와 마찬가지로 실행됨을 볼 수 있다. 간단한 예제이지만, 정말 멀티프로세스로 작업을 실행하는지 확인하기에 좋은 예다.

```
$ python3.6 06_processPool.py
Executing our Task on Process 40365
Executing our Task on Process 40366
```

컨텍스트 관리자

ProcessPoolExecutor에서 다룰 다음 메소드는 ThreadPoolExecutor에서와 마찬가지로 컨텍스트 관리자다. 다음과 같이 with 명령어로 실행할 수 있다.

```
with ProcessPoolExecutor(max_workers=3) as executor:
… 기타 작업을 전달함 …
```

컨텍스트 관리자는 특정 자원의 할당과 릴리스를 관리하며, 프로세스 풀을 다루는 더 좋은 방법이다. 이러한 문법은 그 밖의 메소드보다 우수하며 훨씬 가독성 좋은 코드를 작성할 수 있다.

예제

이 예제에서 컨텍스트 관리자를 바탕으로 코드의 가독성을 높여본다. 전달된 함수 호출에서 task 함수의 인자를 넣어보는 작업을 해본다.

```
from concurrent.futures import ProcessPoolExecutor

def task(n):
  print("Processing {}".format(n))

def main():
  print("Starting ThreadPoolExecutor")
  with ProcessPoolExecutor(max_workers=3) as executor:
    future = executor.submit(task, (2))
    future = executor.submit(task, (3))
    future = executor.submit(task, (4))

  print("All tasks complete")

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

출력

위 프로그램을 실행하면 작업이 실행되고, 각 task 함수에 전달되는 값이 출력된다.

```
$ python3.6 02_processPoolExe.py
Starting ThreadPoolExecutor
Processing 2
Processing 3
Processing 4
All tasks complete
```

연습

ProcessPoolExecutor를 잘 사용하기 위해, 수많은 사진의 흑백 버전을 생성하는 프로그램을 작성해보자. 여기에는 파이썬 이미징 라이브러리인 Pillow가 필요하다.

이미지 프로세싱은 이미지의 픽셀이 흑백으로 변환될 때 처리와 재연산이 필요하기에 많은 연산이 필요하다.

주요 요구사항은 다음과 같다.

- 작업 처리에는 ProcessPoolExecutor 클래스를 무조건 이용해야 한다.
- 프로젝트는 병렬적으로 여러 이미지를 처리할 수 있어야 하지만, 동일한 그림을 두 가지 방법으로 처리하지 않아야 한다.

시작하기

시작하기에 앞서, 일반적인 RGB 그림을 흑백 버전으로 어떻게 변환하는지 다음 샘플 코드를 참조해보자.

```
from PIL import Image
image_file = Image.open("convert_image.png") # 컬러 이미지를 연다.
image_file = image_file.convert('1') # 이미지를 흑백으로 변환한다.
image_file.save('result.png')
```

연산 속도 높이기

이제 파이썬 애플리케이션에서 ThreadPoolExecutor와 ProcessPoolExecutor를 어떻게 사용하는지 살펴보며, 언제 사용되는지 또한 이해한다. 둘의 차이점은 이들의 메커니즘을 이해하며 살펴본다.

ThreadPoolExecutor는 현재 실행을 위해 스레드를 활용하기 때문에 이름 지어졌다.

ProcessPoolExecutor는 프로세스를 활용한다.

프로세서와 스레드의 차이점은 이전 장에서 살펴봤지만, 여기서는 코드를 바탕으로 더 학습해본다. 이 예제에서는 배열의 모든 값은 소수이거나 소수가 아님을 확인해본다. 소수인지 확인보기 위해 에라토스테네스의 체sieve of Eratosthenes[2]를 활용해본다.

```
def is_prime(n):
  if n % 2 == 0:
    return False

  sqrt_n = int(math.floor(math.sqrt(n)))
  for i in range(3, sqrt_n + 1, 2):
    if n % i == 0:
      return False
  return True
```

코드

ThreadPoolExecutor와 ProcessPoolExecutor를 사용해 소수 배열을 동작시키는 전체 코드는 다음과 같다. 이 특정한 샘플에서는 이전 장에서 자세히 살펴본 timeit 모듈을 활용해본다.

```
import timeit
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import ProcessPoolExecutor
import math

PRIMES = [
  112272535095293,
  112582705942171,
  112272535095293,
```

2 그리스의 수학자이자 지리학자인 에라토스테네스가 고안한 소수를 찾는 방법으로, 2 이외의 2의 배수, 3 이외의 3의 배수, 5 이외의 5의 배수의 순서로 수를 지워나가면 남는 수가 소수다. – 옮긴이

```
  115280095190773,
  115797848077099,
  1099726899285419
]

def is_prime(n):
  if n % 2 == 0:
    return False

  sqrt_n = int(math.floor(math.sqrt(n)))
  for i in range(3, sqrt_n + 1, 2):
    if n % i == 0:
      return False
  return True

def main():

  t1 = timeit.default_timer()
  with ProcessPoolExecutor(max_workers=4) as executor:
    for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
      print('%d is prime: %s' % (number, prime))

  print("{} Seconds Needed for ProcessPoolExecutor".format(
timeit.default_timer() - t1))
  t2 = timeit.default_timer()
  with ThreadPoolExecutor(max_workers=4) as executor:
    for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
      print('%d is prime: %s' % (number, prime))
  print("{} Seconds Needed for ThreadPoolExecutor".format(
timeit.default_timer() - t2))

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

출력

위 프로그램을 실행하면, 프로세스 풀이 배열의 모든 원소를 돌며 해당 원소가 소수인지 아닌지 여부가 마지막에 전체 소요 시간과 함께 출력됨을 볼 수 있다.

다음으로 스레드 풀 실행자에서 동일하게 연산 작업이 진행되고 전체 실행 시간이 출력된다. 터미널에서 출력된 형태는 동일하다.

```
$ python3.6 08_poolImprovement.py
112272535095293 is prime: True
...
1099726899285419 is prime: False
2.8411907070549205 Seconds Needed for ProcessPoolExecutor
112272535095293 is prime: True
...
1099726899285419 is prime: False
4.426182378898375 Seconds Needed for ThreadPoolExecutor
```

ProcessPoolExecutor에서는 약 2.8초의 시간이 소요됐으며, ThreadPoolExecutor에서는 4.4초 동안 실행됐다. 따라서 프로세서를 활용하면 전체 처리 시간이 60% 감소한다.

단일 스레드 기반 방식에서 시도해보면, 배열 내 값을 처리하는 시간이 ThreadPoolExecutor에서 실행되는 시간보다 더 빠를 것이다. 이를 테스트해보고자 하면, 다음 코드를 이전 예제의 main 함수 아래에 추가해 실행해보자.

```
t3 = timeit.default_timer()
for number in PRIMES:
  isPrime = is_prime(number)
  print("{} is prime: {}".format(number, isPrime))
print("{} Seconds needed for single threaded
execution".format(timeit.default_timer()-t3))
```

웹 크롤러 성능 높이기

이제 ThreadPoolExecutor와 ProcessPoolExecutor에 대해 자세히 알아보면서, 이를 예제에 적용해보자. 앞서 5장에서는 주어진 웹사이트의 연결 가능한 모든 링크를 크롤링하

는 멀티스레드 기반 웹 크롤러를 만들어봤다.

파이썬 웹 크롤러 코드는 https://github.com/elliotforbes/python-crawler에서 확인할 수 있다.

하지만 출력이 여러 형태의 포맷을 지원하지 않기에, 코드는 `ThreadPoolExecutor`를 사용해 성능을 높일 수 있다. 이제 코드의 가독성을 높이고 결과도 보기 좋게 만들어보자.

계획하기

시작하기에 앞서 크롤러의 성능을 어떻게 높일 수 있는지 살펴보자.

새로운 개선점

만들고자 하는 개선점은 다음과 같다.

- `ThreadPoolExecutor`를 사용해 코드를 리팩토링하기
- CSV 파일이나 JSON 파일 포맷으로 크롤링의 결과를 출력하기
- 크롤링하는 모든 페이지 및 각 정보를 저장해, 데이터를 활용한 사이트 성능 높이기

위 세 가지 모두 성능을 향상할 수 있는 시작점이다. 성능 개선과 관련된 더 자세한 사항은 9장의 RESTful API 웹 크롤러 서비스 사용에서 알아보겠다.

코드 리팩토링하기

먼저 애플리케이션의 멀티스레딩으로 구현한 부분을 리팩토링하는 것이다. 모든 스레드는 시작과 종료를 관리할 수 있으며, `ThreadPoolExecutor`에서 다룰 수 있다.

크롤링 코드로 돌아가 여러 스레드를 시작하고자 한다면, 다음과 같이 해보자.

```
import threading
import queue
from crawler import *
from CheckableQueue import *

THREAD_COUNT = 20
linksToCrawl = CheckableQueue()

def createCrawlers():
  for i in range(THREAD_COUNT):
    t = threading.Thread(target=run)
    t.daemon = True
    t.start()

def run():
  while True:
    url = linksToCrawl.get()
    try:
      if url is None:
        break
      Crawler.crawl(threading.current_thread(), url, linksToCrawl)
    except:
      print("Exception thrown with link: {}".format(url))
    linksToCrawl.task_done()

def main():
  url = input("Website > ")
  Crawler(url)
  linksToCrawl.put(url)
  createCrawlers()
  linksToCrawl.join()
  print("Total Links Crawled: {}".format(len(Crawler.crawledLinks)))
  print("Total Errors: {}".format(len(Crawler.errorLinks)))

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

하지만 ThreadPoolExecutor에서 with 명령어를 사용해 몇몇 줄을 줄일 수 있다. 다음과 같이 코드가 훨씬 명료해졌음을 볼 수 있다.

```
import threading
import queue
from concurrent.futures import ThreadPoolExecutor
from crawler import *
from CheckableQueue import *

THREAD_COUNT = 20
linksToCrawl = CheckableQueue()

def run(url):
  try:
    Crawler.crawl(threading.current_thread(), url, linksToCrawl)
  except:
    print("Exception thrown with link: {}".format(url))
  linksToCrawl.task_done()

def main():
  url = input("Website > ")
  Crawler(url)
  linksToCrawl.put(url)
  while not linksToCrawl.empty():
    with ThreadPoolExecutor(max_workers=THREAD_COUNT) as executor:
      url = linksToCrawl.get()
      if url is not None:
        future = executor.submit(run, url)

print("Total Links Crawled: {}".format(len(Crawler.crawledLinks)))
print("Total Errors: {}".format(len(Crawler.errorLinks)))

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

결과를 CSV 파일로 저장하기

CSV 파일은 스크래핑한 결과를 쉽고 빠르게 저장할 수 있는 형태로, 엑셀 같은 소프트웨어로 데이터 분석을 편리하게 할 수 있다.

결과를 CSV로 추가하기 위해서는 파이썬에서 기본적으로 지원하는 csv 모듈을 이용한다.

appendToCSV 함수를 정의해 입력된 결과를 results.csv 파일로 다음과 같이 만들어보자.

```
import csv
...

def appendToCSV(result):
  print("Appending result to CSV File: {}".format(result))
  with open('results.csv', 'a') as csvfile:
    resultwriter = csv.writer(csvfile, delimiter=' ', quotechar='|',
quoting=csv.QUOTE_MINIMAL)
resultwriter.writerow(result)
```

위의 appendToCSV 함수는 메인 스레드에서 작동되며, 결과가 실행자 객체로부터 반환될 때 호출된다. 메인 스레드에 단독으로 존재한다는 건 경합 조건이 일어나지 않는다는 뜻이기에 리소스에 락이 필요 없다.

이제 appendToCSV 파일을 정의해 결과를 얻을 때마다 해당 메소드가 호출된다. 이를 위해, main 함수를 다음과 같이 고쳐 URL은 큐에서 실행자로 전달한다.

```
for future in as_completed(futures):
  try:
    if future.result() != None:
      appendToCSV(future.result())
  except:
    print(future.exception())
```

다음과 같이 main 메소드를 볼 수 있다.

```
def main():
  url = input("Website > ")
  Crawler(url)
  linksToCrawl.put(url)
  while not linksToCrawl.empty():
    with ThreadPoolExecutor(max_workers=THREAD_COUNT) as executor:
      url = linksToCrawl.get()
      futures = []
      if url is not None:
        future = executor.submit(run, url)
        futures.append(future)

    for future in as_completed(futures):
      try:
        if future.result() != None:
          appendToCSV(future.result())
      except:
        print(future.exception())
```

연습: 각 페이지에서 크롤링한 더 많은 정보 얻기

지금까지 크롤링한 사이트의 결과를 CSV 파일로 저장하는 메소드를 다뤄봤는데, 이제 파일화는 생각보다 간단하다. 각 페이지마다 크롤링한 상당한 양의 데이터를 더 개선해 볼 수 있다.

다음과 같이 여러 사항을 추가해보자.

- 다운로드 소요 시간
- 페이지 크기(KB)
- HTML에서 텍스트 비율

이는 파일화에서 해볼 만한 중요한 특징들을 나타낸 것이며, 그 밖에도 중요한 것들이 많으니 찾아보자.

해당 크롤러 코드는 다음의 웹사이트 분석기를 생성하는 데 사용된다.
https://github.com/elliotforbes/python-crawler

파이썬 2.7에서의 concurrent.futures

기존의 시스템 및 관습으로 인해 파이썬 2.7 환경이거나, 구형 코드 기반 및 파이썬 3.* 에서 패키지 부족 같은 여러 이유로 문제가 있어도 걱정하지 말자. futures 모듈은 다음 pip를 사용한다.

```
pip install futures
```

위 명령어는 기본적으로 파이썬 3.2 이상의 버전에서 concurrent.futures 모듈이 코드 마이그레이션 없이 사용할 수 있게 제공된다.

요약

7장에서는 스레드 풀, 프로세스 풀, 퓨처 객체의 모든 부분을 다뤄봤다. 자체적인 스레드 풀과 프로세스 풀을 인스턴스화하는 방법과 더불어 스레드와 프로세스 풀 실행자를 기존의 방식으로 사용할 때의 장점도 살펴봤다.

이제 멀티스레드 기반, 멀티프로세스 기반 애플리케이션을 풀 자원 개념을 활용해 코드의 성능을 향상할 수 있다.

기존의 웹 크롤러 프로그램의 성능을 향상하는 방법도 알아봤고, 여기서 다룬 핵심 개념을 활용해 더 쉬운 코드로 리팩토링도 해봤다. 8장에서는 애플리케이션 내에서 다양한 프로세스를 활용하는 방법을 더 자세히 살펴보자.

08

멀티프로세싱

8장에서는 파이썬에서 멀티프로세싱과 병렬화 실행을 어떻게 수행하는지에 대한 물음에 답해본다.

8장에서 다루는 내용은 다음과 같다.

- 멀티프로세싱이 GIL의 영향을 없애는 방법
- 파이썬에서의 프로세스 라이프(자식 프로세스를 실행하고, 이를 확인하고, 필요 없을 경우 완벽하게 종료하기)
- 멀티프로세싱 풀 API 활용하기
- 여러 프로세스 간의 통신과 동기화

8장을 배우고 나면 멀티프로세스의 진정한 병렬화를 활용해 파이썬 프로그램을 좀 더 쉽게 작성할 수 있을 것이다.

다행히도 이전에 살펴봤듯이 8장에서 다루는 많은 개념은 파이썬 2.7에서도 동작한다. 멀티프로세싱 모듈은 2.6 버전에서 소개됐다.

GIL 작업

전역 인터프리터 락GIL, global interpreter lock은 CPU 기반 작업에서 성능을 저해하는 메커니즘이다. 이 책에서는 비동기적 프로그래밍을 바탕으로 GIL이 파이썬 시스템 성능에 미치는 영향을 최소화해봤다.

하지만 멀티프로세싱을 이용하면 이러한 한계를 모두 극복할 수 있다. 여러 프로세스를 활용함으로써 여러 GIL 인스턴스를 활용해 프로그램에서 한 스레드의 바이트코드를 실행하는 데 있어 한 번에 정제화할 필요가 없다.

파이썬에서의 멀티프로세싱은 CPU의 처리 능력을 최대로 활용할 수 있게 해준다.

하위 프로세스 활용하기

파이썬에서는 CPU의 독립적인 코어에서 실행 가능한 여러 프로세스를 실행할 수 있다.

예제

이 예제에서는 간단한 출력문을 실행하는 자식 스레드를 생성해보자. 멀티프로세싱 모듈을 활용해 자식 스레드가 실행되는지 간단하게 확인할 수 있으며, 이로써 파이썬에서 멀티프로세싱을 조잡하게나마 구현할 수 있다.

```
import multiprocessing

def myProcess():
```

```
  print("Currently Executing Child Process")
  print("This process has it's own instance of the GIL")
  print("Executing Main Process")
  print("Creating Child Process")

myProcess = multiprocessing.Process(target=myProcess())
myProcess.start()
myProcess.join()
print("Child Process has terminated, terminating main process")
```

출력

위 프로그램을 실행하면, 메인 프로세스가 해당 출력문을 실행하고 자식 프로세스가 실행됨을 볼 수 있다.

```
$ python3.6 08_subprocess.py
Currently Executing Child Process
This process has it's own instance of the GIL
Executing Main Process
Creating Child Process
Child Process has terminated, terminating main process
```

프로세스 라이프

멀티프로세싱 모듈에는 파이썬 프로그램 내에서 프로세스를 시작하는 메소드가 세 가지 있다.

- spawn
- fork
- forkserver

파이썬에서 멀티프로세스 프로그램을 작성할 때 이러한 메소드를 사용하지 않더라도 메커니즘이 어떻게 작동하는지는 알아볼 필요가 있으며, 여러 운영체제상에서 어떤 차이가 있는지도 이해해두면 좋다.

공식 파이썬 문서를 확인해보길 바란다.
https://docs.python.org/3.6/library/multiprocessing.html?highlight=multiprocessing#contexts-and-start-methods

fork를 사용해 프로세스 시작하기

포킹forking이란 부모 프로세스에서 자식 프로세스를 생성하기 위해 유닉스 시스템에서 사용되는 메커니즘이다. 이러한 자식 프로세스는 실제 현실처럼 부모 프로세스와 동일하게 부모의 모든 자원을 상속받는다.

fork 명령어는 유닉스 운영 환경에서의 기본 시스템 명령어다.

프로세스 스폰

개별적인 프로세스를 스폰spawn함으로써 그 밖의 파이썬 인터프리터 프로세스를 실행할 수 있다. 여기에는 자체 전역 인터프리터 락이 포함되며, 각 프로세스는 병렬적으로 실행할 수 있어, 더 이상 전역 인터프리터 락의 한계에 대해 걱정할 필요가 없다.

새로 스폰된 프로세스는 해당 실행 메소드에 어떤 인자든 실행하기 위해 필요한 자원만 상속받는다. 이는 윈도우 시스템에서 새로운 프로세스를 실행할 때 일반적으로 사용되는 방법이지만, 유닉스 시스템에서도 마찬가지로 사용된다.

forkserver

forkserver는 개별적인 프로세스를 생성하는 다소 이상한 메커니즘이지만, 유닉스 파이프를 넘어 파일 기술어를 전달하도록 지원하고 유닉스 플랫폼에서만 사용 가능한 메커니즘이다.

프로그램이 프로세스를 시작할 때 이 메커니즘을 선택한다면, 일반적으로 서버가 인스턴스화된다. 그 후 프로세스를 생성하는 모든 요청을 다루며, 파이썬에서 새로운 프로세스를 생성하려면 새로 인스턴스화된 서버에 요청을 전달한다. 그러면 해당 서버는 프로세스를 생성하고 프로그램에서 자유롭게 사용할 수 있다.

데몬 프로세스

데몬 프로세스daemon process는 이 책에서 앞서 언급한 데몬 스레드와 동일한 형태를 따른다. 데몬 플래그를 참으로 설정해 실행 중인 프로세스를 데몬화할 수 있다.

이러한 데몬 프로세스는 메인 스레드가 실행되는 동안 계속되며, 실행이 끝나거나 메인 프로그램을 종료할 경우에만 종료된다.

예제

다음 예제를 통해, 파이썬에서 어떻게 데몬 프로세스를 정의하고 실행하는지 살펴보자.

```
import multiprocessing
import time

def daemonProcess():
  print("Starting my Daemon Process")
  print("Daemon process started: {}".format(multiprocessing.current_process()))
  time.sleep(3)
  print("Daemon process terminating")
  print("Main process: {}".format(multiprocessing.current_process()))
```

```
def main():
  myProcess = multiprocessing.Process(target=daemonProcess)
  myProcess.daemon = True
  myProcess.start()
  print("We can carry on as per usual and our daemon will continue to execute")
  time.sleep(1)

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

코드에 관한 분석

위 프로그램에서는 필요한 멀티프로세싱 및 시간 관련 모듈을 먼저 불러온다. 다음으로 생성하고자 하는 데몬 프로세스의 타깃을 설정하는 함수를 정의한다.

daemonProcess 함수에서 프로세스의 시작을 출력하고, 다음으로 현재 프로세스를 출력한다. 해당 출력을 마치면 3초 동안 유휴 상태를 거치며, 데몬 스레드가 종료됐음을 출력한다.

daemonProcess 함수 정의를 마치고, 현재 프로세스를 출력하고, 프로세스를 인스턴스화한 후, 데몬 플래그를 참으로 설정한다. 마지막으로, 프로세스에서의 작업이 진행되고 프로그램이 종료된다.

출력

이 프로그램을 실행하면 current_process() 호출로부터 메인 프로세스가 먼저 출력된다. 데몬 프로세스를 시작하고 해당 프로세스로부터 동일한 current_process() 함수를 호출한다.

메인 프로세서와 데몬 프로세스 간의 차이를 주의하자. 프로세스 1은 데몬 프로세스가 시작될 때 시작되고 출력된다.

```
$ python3.6 00_daemonProcess.py
Starting my Daemon Process
Daemon process started: <_MainProcess(MainProcess, started)>
Daemon process terminating
Main process: <_MainProcess(MainProcess, started)>
We can carry on as per usual and our daemon will continue to execute
```

데몬 프로세스에서는 자식 프로세스를 생성할 수 없음을 주의하자. 이를 진행하면 process.start()에서 오류가 날 것이다.

PID를 이용해 프로세스 확인하기

운영체제에 있는 모든 프로세스는 프로세스 확인자를 구성하는데, 일반적으로 PID라고 불린다. 파이썬 프로그램 내에서의 멀티프로세스 동작 시, 프로그램에서 오직 1개의 프로세스 확인자가 있을 것이라 생각하지만 그렇지 않다.

파이썬 프로그램상에서 스폰하는 각 하위 프로세스가 운영체제 내에서 개별적으로 확인하고자 자체 PID 수를 받는다. 자체 할당된 PID가 있는 개별적인 프로세스는 로깅 및 디버깅 같은 작업을 수행할 경우 유용하다.

다음과 같이 파이썬 하위 프로세스의 프로세스 확인자를 볼 수 있다.

```
import multiprocessing
print(multiprocessing.current_process().pid)
```

위 코드는 해당 파이썬 프로그램이 실행될 때 현재 프로세스 확인자를 출력한다.

예제

이 예제에서는 단일 자식 프로세스를 실행하고, childTask 함수를 실행 타깃으로 지정해 본다.

```
import multiprocessing
import time

def childTask():
  print("Child Process With PID: {}".format(multiprocessing.current_process().pid))
  time.sleep(3)
  print("Child process terminating")

def main():
  print("Main process PID: {}".format(multiprocessing.current_process().pid))
  myProcess = multiprocessing.Process(target=childTask())
  myProcess.start()
  myProcess.join()

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

출력

위 프로그램은 다음과 같이 메인 프로세서와 생성된 자식 프로세스 모두 해당 PID를 출력한다.

```
$ python3.6 01_identifyProcess.py
Main process PID: 85365
Child Process With PID: 85367
Child process terminating
```

이는 현재 프로세스의 확인자를 가져와 속도를 높인다. 하지만 현재 프로세스로부터 PID 외의 것을 얻을 수 있다.

threading.Thread 클래스에서와 같이 개별적인 프로세스에 이름을 붙이는 작업을 할 수 있다. 어떤 형태의 작업을 수행하는 2개의 프로세스가 있고, 그 외 형태의 작업을 수행하는 2개의 프로세서가 있다고 하자. 이러한 작업의 출력을 기록하려면, '프로세스 1 x 수행', '프로세스 2 y 수행'과 같은 형태는 보고 싶지 않을 것이다.

대신 자식 프로세스에 좀 더 의미 있는 네이밍naming을 해야 하는데, 디버깅 작업 및 잘못된 부분을 찾는 데 있어 많은 도움이 될 것이다.

프로세스가 생성된 후 이름을 붙이고자 다음과 같이 해보자.

```
import multiprocessing

def myProcess():
  print("{} Just performed X".format(multiprocessing.current_process().name))

def main():
  childProcess = multiprocessing.Process(target=myProcess(),
name='My-Awesome-Process')
  childProcess.start()
  childProcess.join()

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

위 코드의 출력은 다음과 같다.

```
$ python3.6 12_nameProcess.py
My-Awesome-Process Just performed X
```

생산에 관련된 애플리케이션을 작성한다면, 애플리케이션의 모든 결과를 추적할 수 있어야 한다. 가능한 한 많은 로깅으로 자체 관리가 중요하므로, 개발 중인 애플리케이션에 지원을 아끼지 않아야 한다.

프로세스 종료하기

실행 중인 자식 프로세스를 종료하는 것에 대해 살펴보자. 로컬상의 애드혹ad hoc을 실행하는 파이썬 코드에서는 그다지 중요하지 않지만, 방대한 서버를 다루는 기업용 파이썬 프로그램에서는 매우 중요하다.

오랜 기간 실행되는 시스템에서 수천, 수만의 프로세스를 실행할 수는 없으며, 일반적으로 시스템 자원에 그대로 남겨둘 수도 없다. 따라서 프로세스를 종료하는 일은 꽤 중요해 보인다.

파이썬에서 프로세스를 종료하려면, 다음과 같이 Process 객체의 .terminate() 함수를 사용한다.

```
myProcess.terminate()
```

예제

이 예제에서는 단일 자식 프로세스를 실행해 20초 동안 실행을 시뮬레이션해보자. 메인 함수에서 프로세스를 실행하면, 새로 생성된 프로세스를 종료하고자 terminate 함수를 즉시 호출한다.

```
import multiprocessing
import time

def myProcess():
  current_process = multiprocessing.current_process()
  print("Child Process PID: {}".format(current_process.pid))
  time.sleep(20)
  current_process = multiprocessing.current_process()
  print("Main process PID: {}".format(current_process.pid))

myProcess = multiprocessing.Process(target=myProcess)
myProcess.start()
```

```
print("My Process has terminated, terminating main thread")
print("Terminating Child Process")
myProcess.terminate()
print("Child Process Successfully terminated")
```

프로그램을 실행하면 자식 프로세스 종료 후 20초 동안 블로킹한다. 이는 자식 스레드가 성공적으로 종료됐음을 보여준다.

현재 프로세스 얻기

개별적인 프로세스를 확인할 수 있는 것은 로깅 및 디버깅의 관점에서 중요하므로, 파이썬 프로그램상의 모든 프로세스 PID를 검색할 수 있어야 한다. 이는 멀티스레딩에서 현재 스레드를 검색하는 방법을 살펴본 부분과 동일하게 해볼 수 있다.

다음 코드를 바탕으로 현재 프로세스 ID[PID, process ID]를 검색할 수 있다.

```
import multiprocessing
print(multiprocessing.current_process().pid)
```

위 코드는 현재 실행되는 프로세스 확인자를 검색한다.

프로세스를 하위 클래스화하기

3장에서 threading.Thread 클래스를 하위 클래스화해 스레드를 생성하는 방법을 살펴봤다. 이를 프로세스를 생성하는 데 동일하게 적용할 수 있다.

예제

이 예제에서는 multiprocessing.Process 클래스를 하위 클래스화하는 자체 클래스를 구현하는 방법을 살펴보자. MyProcess 클래스를 정의해 생성자와 실행 함수를 구현한다.

생성자 내에서 super(MyProcess, self).__init__()를 호출한다. 이러한 호출은 프로세스를 초기화하고, 일반적인 파이썬 객체의 클래스를 프로세스로 바꾼다.

run 메소드에서 현재 PID를 출력해 Process 클래스를 성공적으로 하위 클래스화했음을 보여준다.

```
import multiprocessing

class MyProcess(multiprocessing.Process):
  def __init__(self):
    super(MyProcess, self).__init__()
  def run(self):
    print("Child Process PID: {}".format(multiprocessing.current_process().pid))

def main():
  print("Main Process PID: {}".format(multiprocessing.current_process().pid))
  myProcess = MyProcess()
  myProcess.start()
  myProcess.join()
  myProcess.run()

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

출력

위 프로그램을 실행하면 메인 프로세스가 PID를 출력하고 자식 프로세스가 생성되고 실행된다. 이 자식 프로세스는 다른 PID를 출력한다.

```
$ python3.6 10_subclass.py
Main Process PID: 24484
Child Process PID: 24485
```

multiprocessing.Process 클래스를 하위 클래스화했으면, 다음과 같이 멀티프로세스를 동작하는 다양한 작업을 구현해본다.

```
import os
...
processes = []
for i in range(os.cpu_count()):
  process = MyProcess()
  processes.append(process)
  process.start()

for process in processes:
process.join()
  process.run()
```

위 코드는 x개(여기서 x는 현재 컴퓨터의 CPU 코어 개수다)의 자식 프로세스를 실행한다.

멀티프로세싱 풀

파이썬 애플리케이션에서 멀티프로세스로 동작할 경우, 멀티프로세싱 모듈 내에서 다양한 기능을 가진 Pool 클래스를 활용할 수 있다.

Pool 클래스는 프로그램 내의 여러 자식 프로세스를 쉽게 실행하고 풀에서 작업자를 선택할 수 있다.

concurrent.futures.ProcessPoolExecutor와 Pool의 차이점

7장에서 concurrent.futures.ProcessPoolExecutor를 다뤄봤는데, 프로세스 풀에서 어떤 구현이 더 필요할까?

프로세스 풀의 multiprocessing.Pool 구현은 병렬 처리 능력을 지원하고자 거의 동일한 구현 형태를 지닌다. 하지만 concurrent.futures 모듈은 프로세스 풀 생성을 쉽게 해주는 인터페이스만 지원한다. 이러한 간단한 인터페이스는 프로그래머들로 하여금 스레드와 프로세스 풀 모두 즉각적으로 시작할 수 있게 해준다. 그러나 이러한 작업이 복잡해 특정 상황에서 세밀한 조정이 필요할 때는 오히려 불필요하다.

ThreadPoolExecutor와 ProcessPoolExecutor 모두 동일한 추상 클래스의 하위 클래스이므로, 상속 메소드를 이해하고 작업하기가 좀 더 쉽다.

파이썬 2와 파이썬 3 모두의 관점에서 바라보면, concurrent.futures가 2.6 버전에서도 소개됐으므로 백포트 버전 대신 멀티프로세싱 모듈에서는 이를 사용하면 되겠다.

일반적으로 multiprocess.Pool 모듈보다는 concurrent.futures 모듈이 필요조건에 적합하므로 추천한다. 하지만 concurrent.futures 모듈이 더 많은 조작을 필요로 하는 한계에 부딪혔을 때를 대비해 그 밖의 대안도 알아두자.

컨텍스트 관리자

ThreadPoolExecutor를 활용했듯이 컨텍스트 관리자를 사용해 멀티프로세싱 풀을 다룰 수 있다. 마찬가지로 with 키워드 형태를 활용해보자.

```
with Pool(4) as myPool:
  # 작업을 풀로 전달한다.
```

위와 같은 스타일은 코드를 간단히 하며, 풀에서 필요한 자원을 릴리스하는 작업을 다룰 필요가 없다. 다음으로 비교적 간단하게 풀의 자식 프로세스 실행을 반복하는 맵과 같은 작업을 진행할 수 있다.

예제

컨텍스트 관리자 형태로 Pool을 활용해 예제를 살펴보자. 간단한 작업을 수행하는 함수를 정의하고, 메인 함수에 컨텍스트 관리자를 추가하자. 다음으로 작업 함수에 [2,3,4] 배열의 모든 값을 매핑해 결과를 출력한다.

```
from multiprocessing import Pool

def task(n):
  print(n)

def main():
  with Pool(4) as p:
    print(p.map(task, [2,3,4]))

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

출력

위 프로그램은 프로세스 풀에서 매핑한 3개의 값을 그대로 출력한다.

프로세스 풀에 작업 전달하기

풀을 사용할 일이 자주 일어날 수도 아닐 수도 있지만 위의 예제보다 더 복잡할 수도 있으며, 다양한 방법으로 이러한 풀과 상호작용할 수도 있다.

따라서 multiprocess.Pool 클래스가 필요하며. 이는 앞으로 다룰 API로 구성된다.

멀티프로세스 기반 파이썬 프로그램의 진가를 알고자, 이런 부분을 배우고 풀 객체에 작업을 전달하는 다양한 방법에 익숙해져야 한다.

apply

apply는 ThreadPoolExecutor의 .submit()과 같다. 다시 말해. 개별적인 작업을 풀 객체에 전달하고자 사용된다.

이 예제에서는 인자 1개를 취하는 간단한 myTask 함수를 정의해본다. 앞서 살펴본 myTask 함수와 동일하니 참고하자.

main 함수로 넘어가 컨텍스트 관리자로 Pool을 실행하고 각 작업의 결과를 출력한다.

```
from multiprocessing import Pool
import time

def myTask(n):
  time.sleep(n/2)
  return n*2

def main():
  with Pool(4) as p:
    print(p.apply(myTask, (4,)))
    print(p.apply(myTask, (3,)))
    print(p.apply(myTask, (2,)))
    print(p.apply(myTask, (1,)))

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

.apply() 함수는 결과가 준비될 때까지 정지되므로 병렬 수행 작업에는 적합하지 않다. 병렬적으로 수행하고자 한다면 .apply()와 비슷한 apply_async를 사용하자.

apply_async

병렬 실행 작업이 필요할 때는 apply_async 함수를 바탕으로 풀에 작업을 전달할 수 있다.

이 예제에서는 4개의 작업을 프로세싱 풀에 전달하고자 함수 내에 for 문을 사용한다. tasks 배열에 해당 작업을 추가해 결과를 기다리고자 배열을 반복한다.

```
from multiprocessing import Pool
import time
import os

def myTask(n):
  print("Task processed by Process {}".format(os.getpid()))
  return n*2

def main():
  print("apply_async")
  with Pool(4) as p:
    tasks = []

    for i in range(4):
      task = p.apply_async(func=myTask, args=(i,))
      tasks.append(task)

    for task in tasks:
      task.wait()
      print("Result: {}".format(task.get()))

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

실행 후 4개의 작업이 프로세스 풀에서 선택되어 각 프로세스에서 실행됨을 볼 수 있다. tasks 배열에 전달한 순서 그대로 task.wait()를 호출했으므로, 결과 또한 순서대로 콘솔에 출력된다.[1]

```
$ python3.6 21_applyAsync.py
apply_async
Task processed by Process 99323
Task processed by Process 99324
```

1 일반적인 윈도우 파이썬 IDLE 환경에서 실행할 경우, 엄밀히 말하면 사용자 코드는 다른 프로세스에서 실행하게 된다. IDLE와 사용자 코드는 소켓을 이용해 연결되는데, 멀티프로세싱으로는 소켓 연결이 상속되거나 넘겨지지 않는다. 따라서 IDLE의 콘솔에 myTask 함수의 출력문이 나타나지 않는다. 명령 창에 C:\Python35 >python -m idlelib(파이썬 2.X에서는 idlelib.idle을 사용한다)를 입력해 나타나는 IDLE를 이용하면 CMD 콘솔에서 print 문이 출력되는 모습을 볼 수 있다. - 옮긴이

```
Task processed by Process 99322
Task processed by Process 99325
Result: 0
Result: 2
Result: 4
Result: 6
```

map

ThreadPoolExecutor와 마찬가지로 multiprocessing.Pool의 map 함수는 프로세스 풀에서 작업자에 의해 선택된 원소를 하나씩 매핑한다.

다음 예제에서 list = [4,3,2,1]을 map 함수를 이용해 프로세스 풀에 매핑해본다.

```
from multiprocessing import Pool
import time

def myTask(n):
  time.sleep(n/2)
  return n*2

def main():
  print("mapping array to pool")
  with Pool(4) as p:
    print(p.map(myTask, [4,3,2,1]))

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

map 함수를 이용하면 다음과 같이 출력된다.

```
$ python3.6 17_mapPool.py
mapping array to pool
[8, 6, 4, 2]
```

하지만 apply 함수와 마찬가지로 각 작업은 결과가 준비될 때까지 정지되며, 병렬적인 비동기 성능이 필요할 경우 map_async가 필요할 것이다.

map_async

map_async 함수는 매핑이 필요한 부분에 비동기 실행이 필요할 경우 사용한다. 주어진 함수에 전달되는 작업이 비동기적으로 실행되는 부분을 제외하고는 일반적인 map 함수와 동일하다.

```
from multiprocessing import Pool
import time

def myTask(n):
  time.sleep(n/2)
  return n*2

def main():
  with Pool(4) as p:
    print("mapping array to pool")
    print(p.map_async(myTask, [4,3,2,1]).get())

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

imap

imap() 함수는 이터레이터를 반환하는 부분을 제외하고는 일반적인 map 함수와 비슷한 형태를 띤다. 나는 이러한 이터레이터의 사용을 즐기는데, 파이썬의 이터레이터를 다루는 경험이 굉장히 좋을 때가 있다.

다음 예제에서 풀에 매핑할 myTask와 list가 전달된 iterable = p.imap()을 호출한다. 다음으로 next() 메소드를 사용해 이터레이터 내의 모든 결과를 콘솔에 출력해본다.

```
from multiprocessing import Pool
import time

def myTask(n):
  time.sleep(n+2)
  return n+2

def main():
  with Pool(4) as p:
    for iter in p.imap(myTask, [1,3,2,1]):
      print(iter)

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

프로그램 실행을 마치면 전달된 배열의 모든 원소에 2가 더해져, 풀에 전달된 순서 그대로 출력된다.

```
$ python3.6 17_imapPool.py
3
5
4
3
```

imap_unordered

imap_unordered의 경우 사용할 때 반복 매핑을 반환하며, 전달된 작업을 순서 없이 실행한다.

```
from multiprocessing import Pool
import time

def myTask(n):
  time.sleep(n+2)
```

```
  return n+2

def main():
  with Pool(4) as p:
    for iter in p.imap_unordered(myTask, [1,3,2,1]):
      print(iter)

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

위 코드를 실행하면, 다음과 같이 출력된다.

```
$ python3.6 17_imap_unordered_Pool.py
3
3
4
5
```

기존의 값에서 변경된 이터레이터가 처음의 입력 순서와 다르게 출력된다. imap_unordered는 완료되는 순서대로 요소를 반환한다.

starmap

starmap은 튜플 리스트를 기존의 단일 변수 대신 풀에 전달하게 한다. 이 함수는 다소 유연하며, 특정 상황에서 유용하게 쓸 수 있다.

다음 예제에서는 각 값의 튜플 리스트 [(4,3),(2,1)]을 풀에 전달하고 결과를 출력한다.

```
from multiprocessing import Pool
import time

def myTask(x, y):
  time.sleep(x/2)
  return y*2
```

```
def main():
  with Pool(4) as p:
    print(p.starmap(myTask, [(4,3),(2,1)]))

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

myTask 함수에 각 튜플에 해당하는 값과 동일한 인자를 취한다. 다음으로 x/2초만큼 정지하고, y*2를 반환한다.

출력은 다음과 같다.

```
$ python3.6 18_starmap.py
[6, 2]
```

starmap_async

그 밖의 비동기적 함수를 구현해보자. 주어진 작업을 비동기적으로 실행하는 것을 제외하고는 starmap과 동일한 기능을 제공한다.

```
from multiprocessing import Pool
import time

def myTask(x, y):
  time.sleep(x)
  return y*2

def main():
  with Pool(4) as p:
    print(p.starmap_async(myTask, [(4,3),(2,1)]).get())

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

maxtasksperchild

일반적으로 아파치Apache나 mod_wdgi 같은 프로그램은 코드상에 임시방편이 있어, 가끔은 시스템 자원을 해방하게 해준다. 이는 일정 작업이 완료되면 프로세스를 중지하며, 효과적으로 다시 사용할 수 있게 한다.

자체적인 풀에서 이용하고자 maxtasksperchild 인자에 전달하고, 재사용 전 작업자 프로세스에 필요한 작업 실행을 설정한다.

다음 예제에서 어떻게 동작하는지 살펴보자. 이전의 starmap_async 예제 코드를 가져와 maxtasksperchild 파라미터를 추가해 해당 풀에 그 외 작업을 전달한다.

```
from multiprocessing import Pool
import time
import os

def myTask(x, y):
  print("{} Executed my task".format(os.getpid()))
  return y*2

def main():
  with Pool(processes=1, maxtasksperchild=2) as p:
    print(p.starmap_async(myTask, [(4,3),(2,1), (3,2), (5,1)]).get())
    print(p.starmap_async(myTask, [(4,3),(2,1), (3,2), (2,3)]).get())

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

프로그램을 실행하면 다음과 같이 출력된다.[2]

```
$ python3.6 20_maxTasks.py
92099 Executed my task
92099 Executed my task
```

2 'apply_async' 절의 예제와 동일하게 해결한다. – 옮긴이

```
92100 Executed my task
92100 Executed my task
[6, 2, 4, 2]
92101 Executed my task
92101 Executed my task
92102 Executed my task
92102 Executed my task
[6, 2, 4, 6]
```

출력을 살펴보면 두 가지 형태다. 먼저 풀에 전달된 starmap_async는 4개의 작업으로 쪼개져 1개의 프로세스에 의해 선택되어 풀에서 현재 실행 중임을 볼 수 있다.

다음으로 프로세스는 2개의 작업을 실행하고 재사용한다. 두 번의 작업을 실행하고 OS 상 프로세스 PID에 다음으로 사용 가능한 PID가 추가됨을 볼 수 있다.

프로세스 간 통신

수많은 하위 프로세스 간 동기화에 있어 다양하게 활용 가능한 선택사항이 있다.

- 큐queue : 기본적으로 FIFO 큐가 있으며, 5장에서 자세히 다뤄봤다.
- 파이프pipe : 새로운 개념으로, 이후 간단히 살펴본다.
- 매니저manager : 데이터를 생성하며, 결과적으로 파이썬 애플리케이션 내에서 여러 프로세스 간에 데이터를 공유한다.
- ctype: 자식 프로세스로부터 접근하는 공유 메모리를 활용하는 객체다.

위 4개의 옵션은 멀티프로세스 애플리케이션에 활용 가능한 다양한 통신 메커니즘을 보여준다. 통신에 관련된 선택을 할 때, 한 주제에 몰두해 시간을 보내보는 것도 좋다.

하지만 이 책에서 모든 해결책을 제공할 수는 없다. 이러한 주제에 대해 자세히 알고 싶다면, 미샤 고렐릭Micha Gorelick과 이안 오스발트Ian Ozsvald가 쓴 『고성능 파이썬Performance Python』(한빛미디어, 2016)을 읽어보자.

파이프

파이프는 한 프로세스에서 그 외 프로세스로 정보를 전달하는 방법을 나타낸다. 여기에는 익명 파이프와 이름이 있는 파이프, 두 종류가 있다. 파이프는 운영체제에서 자체 프로세스 간에 정보를 전달하기 위한 가장 일반적인 메커니즘이기에, 멀티프로세스 프로그램의 작업에 동일하게 적용할 수 있다.

익명 파이프

익명 파이프anonymous pipe는 내부 프로세스 통신을 운영체제에서 사용하는 FIFO 통신 형태다. 쉽게 말하면, 한 번에 한 방향으로만 전달한다. 2개의 무전기가 있다고 하자. 한 번에 한 명만 의사를 전달하고자 버튼을 누르고 말할 수 있다. 나머지 사람은 그 사람이 '오버'라는 신호 후 버튼을 놓으면 이제야 말을 전달할 수 있다.

다시 말해, 이중 통신을 위해서는 2개의 익명 파이프로 프로세스 간 통신을 한다. 이러한 익명 파이프는 운영체제에 의해 관리되며, 고성능 프로세스 통신에 이용된다.

이름이 있는 파이프

이름이 있는 파이프named pipe는 운영체제가 끝날 때까지 지속된다는 점을 제외하고는 익명 파이프와 동일하다. 익명 파이프는 프로세스가 계속될 때까지 이어진다.

파이프로 작업하기

지금까지 파이프가 무엇이며 운영체제에서 어떻게 작동하는지를 살펴봤으니, 이제 파이썬 프로그램에서 어떻게 활용하는지 살펴볼 차례다.

파이썬 프로그램에서 파이프를 생성하고자 `os` 모듈을 불러오고 `os.pipe()`를 호출한다. `os.pipe()` 메소드는 운영체제 내에 파이프를 생성하고 읽기 및 쓰기에 사용되는 튜플을 반환한다. 다음 예제에서 읽기에는 `pipeout`을, 쓰기에는 `pipein`을 이용한다.

```
import os
pipeout, pipein = os.pipe()
```

예제

다음 예제에서는 자식 프로세스를 생성하고, multiprocessing.Process 클래스를 하위 클래스화한다.

```
import os, sys
import multiprocessing

class ChildProcess(multiprocessing.Process):

def __init__(self, pipein):
  super(ChildProcess, self).__init__()
  self.pipein = pipein

def run(self):
  print("Attempting to pipein to pipe")
  self.pipein = os.fdopen(self.pipein, 'w')
  self.pipein.write("My Name is Elliot")
  self.pipein.close()

def main():
  pipeout, pipein = os.pipe()

  child = ChildProcess(pipein)
  child.start()
  child.join()

  os.close(pipein)
  pipeout = os.fdopen(pipeout)

  pipeContent = pipeout.read()
  print("Pipe: {}".format(pipeContent))

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

프로그램을 실행하면 다음과 같이 출력된다.

```
$ python3.6 11_pipes.py
Attempting to pipein to pipe
Pipe: My Name is Elliot
```

예외 처리하기

자식 프로세스에서 예외 발생을 처리하는 것은 애플리케이션 실행에 있어 중요하다. 예외를 생각하지 않으며 자식 프로세스를 복구할 수단이 없다는 것은 애플리케이션에 치명적이다.

예외 처리의 경우 그 외 개발자들이 볼 수 있게 확실히 해야 하는데, 그렇지 않으면 코드상의 문제를 개발자들이 보면서 매우 싫어할 것이다.

이러한 예외를 다루는 데는 다양한 방법이 존재한다. 발생한 예외가 어느 쪽이든 다루기로 한 부모 프로세스로 돌아가고자, 부모 프로세스와 자식 프로세스 간의 통신 방법에 대해 다뤘다. 하지만 그 밖의 방법이 있는데, 바로 프로세스 간 예외를 전달하고자 파이프를 이용하는 것이다.

파이프 이용하기

파이썬 코드에서 파이프를 이용하려면 두 가지 방법이 있다. 파이프를 생성하기 위해 os 모듈을 사용하는 방법과, `multiprocessing.Pipe()`를 활용하는 방법이다.

이 두 가지의 차이점은 `multiprocessing.Pipe`의 경우 일반적으로 `os.pipe`보다 높은 레벨에 있다는 것이다. 이때 양방향 통신을 위해서는 `multiprocessing.Pipe`를 사용해야 한다는 점이 중요하다.

다음 예제를 통해, 부모 프로세스에서 백업한 자식 프로세스로부터 던져진 예외의 정보를 어떻게 전달하는지 살펴보자.

```
import multiprocessing
import os, sys
import traceback

class MyProcess(multiprocessing.Process):
  def __init__(self, pipein):
    super(MyProcess, self).__init__()
    self.pipein = pipein

def run(self):
  try:
    raise Exception("This broke stuff")
  except:
    except_type, except_class, tb = sys.exc_info()

    self.pipein = os.fdopen(self.pipein, 'w')
    self.pipein.write(str(except_type))
    self.pipein.close()

def main():
  pipeout, pipein = os.pipe()

  childProcess = MyProcess(pipein)
  childProcess.start()
  childProcess.join()

  os.close(pipein)
  pipeout = os.fdopen(pipeout)

  pipeContent = pipeout.read()
  print("Exception: {}".format(pipeContent))

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

multiprocessing.Manager

multiprocessing 모듈에는 Manager 클래스가 있는데, 이 클래스는 파이썬 객체를 조절하고 프로그램에서 안전한 스레드와 프로세스를 제공한다.

Manager 클래스를 다음과 같이 초기화한다.

```
import multiprocessing
myManager = multiprocessing.Manager( )
```

manager 객체로 여러 프로세스 간에 리스트나 딕셔너리 같은 데이터를 공유할 수 있으며, 일반적으로 자체 통신을 구현하는 첫 번째 관문이다.

네임스페이스

Manager에는 네임스페이스namespace라는 개념이 있는데, 호출할 수 있는 퍼블릭 메소드는 없고 쓰기가 가능한 애트리뷰트에서는 유용하게 사용한다.

멀티프로세스 간에 몇 개의 애트리뷰트를 공유하는 빠른 방법이 필요할 때는 이 방법이 최고라 할 수 있다.

예제

다음 예제에서는 메인 프로세스와 자식 프로세스 간에 데이터를 공유하고자 네임스페이스를 활용하는 부분을 살펴보자.

```
import multiprocessing as mp
import queue

def myProcess(ns):
  # 네임스페이스의 값을 갱신한다.
  print(ns.x)
```

```
  ns.x = 2

def main():
  manager = mp.Manager()
  ns = manager.Namespace()
  ns.x = 1

  print(ns)

  myProcess(ns)
  process = mp.Process(target=myProcess, args=(ns,))
  process.start()
  process.join()
  print(ns)

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

처음으로 돌아가 프로그램을 실행하면, 세 가지 일이 일어난 것을 볼 수 있다. 먼저, Namespace 객체가 프로그램 실행 처음에 출력된다. 이때 x=1인 상태다. 다음으로 하위 프로세스의 출력 후 새로운 값 2로 변경된다.

이제 부모 프로세스로부터 마찬가지로 네임스페이스를 출력하고, 이번에는 x 값이 2로 바뀐 것을 볼 수 있다.

```
$ python3.6 07_manager.py
Namespace(x=1)
1
Namespace(x=2)
```

큐

5장에서 큐 동기화와 관련된 부분과 멀티스레드 기반 상황에서 사용하는 방법을 살펴봤다. 멀티프로세스 프로그램과 결합 시 사용될 수 있다는 부분에 주목하자.

예제

다음 예제에서 큐 동기화를 이용해 각 자식 프로세스 간 통신 메커니즘을 어떻게 구현하는지 확실히 살펴보자.

myTask 함수를 정의해 공유된 큐를 인자로 가지며 큐에서 단순히 값을 뽑아내 보자. main 함수에서는 manager 객체와 다음을 사용해 공유된 큐를 정의한다.

```
m = multiprocessing.Manager( )
sharedQueue = m.Queue( )
```

myTask 함수를 실행 타깃으로, sharedQueue를 한 인자로 취해 세 번의 프로세스를 정의한다. 각 프로세스를 정의한 후에는 이를 시작하고 조인해본다.

```
from multiprocessing import Pool
import multiprocessing
import queue
import time

def myTask(queue):
  value = queue.get( )
  print("Process {} Popped {} from the shared Queue".format(
multiprocessing.current_process( ).pid, value))
  queue.task_done( )

def main( ):
  m = multiprocessing.Manager( )
  sharedQueue = m.Queue( )
  sharedQueue.put(2)p
  sharedQueue.put(3)
  sharedQueue.put(4)

  process1 = multiprocessing.Process(target=myTask, args=(sharedQueue,))
  process1.start( )
  process1.join( )
```

```
  process2 = multiprocessing.Process(target=myTask, args=(sharedQueue,))
  process2.start()
  process2.join()

  process3 = multiprocessing.Process(target=myTask, args=(sharedQueue,))
  process3.start()
  process3.join()

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

출력

프로그램을 실행하면 세 번의 프로세스가 공유된 큐 객체로부터 2, 3, 4를 그 외 PID로 끄집어내는 모습을 볼 수 있다.[3]

```
$ python3.6 06_mpQueue.py
Process 89152 Popped 2 from the shared Queue
Process 89153 Popped 3 from the shared Queue
Process 89154 Popped 4 from the shared Queue
```

Listener와 Client 클래스

여러 프로세서와 하위 프로세스 간의 대다수 통신이 큐나 파이프로 구현된다 할지라도, 그 외 메커니즘을 적용해볼지 고민해야 한다. 이렇게 고민해볼 대안으로는 multiprocessing.connection을 활용하는 방법이 있다.

multiprocessing.connection 모듈은 일반적으로 소켓과 윈도우상의 이름 있는 파이프를 다루기 위한 하이 레벨의 메시지 기반 API이다.

3 'apply_async' 절의 예제와 동일하게 해결한다. – 옮긴이

예제

multiprocessing.connection 모듈에 관한 공식 문서에는 클라이언트와 리스너가 서로 상호작용하고 메시지를 보내는 방법을 소개하는 훌륭한 예제가 있다.

다음 예제에서는 리스너와 클라이언트 모두를 사용한 통신을 이해하는 기본적인 방법을 다뤄본다.

Listener 클래스

먼저 시스템 시작에 있어 가장 중요한 리스너를 정의한다. multiprocessing.connection 모듈로부터 Listener 클래스를 불러와 'localhost'와 포트번호 6000을 주소 튜플로 취해 정의한다.

다음으로 Listener를 컨텍스트 관리자로 사용해 해당 포트에 연결이 이뤄지도록 기다린다. authkey=b'secret password'는 이상한 프로세스의 정보는 차단하고, authkey와 일치하는 포트와 주소 연결만 허용하는 중요한 부분이다.

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)  # 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
  with listener.accept() as conn:
    print('connection accepted from', listener.last_accepted)

    conn.send([2.25, None, 'junk', float])

    conn.send_bytes(b'hello')

    conn.send_bytes(array('i', [42, 1729]))
```

Client 클래스

다음으로 Client 클래스를 정의해 리스너와 처음 연결한 방식 그대로 하며, Listener 프로세스로부터 전송된 모든 것을 출력한다.

먼저 multiprocessing.connection 모듈로부터 Client 클래스를 불러온다. Listener 클래스에서 정의했듯이 authkey도 마찬가지로 주소에 전달하는 컨텍스트 관리자를 Client 클래스에 활용한다.

성공적으로 연결되면 Listener 클래스로부터 전달받은 모든 사항을 출력한다. connection 모듈의 유연성을 설명하고자 이런 부분을 다양하게 해볼 수 있다.

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
  print(conn.recv())                  # => [2.25, None, 'junk', float]

  print(conn.recv_bytes())            # => 'hello'

  arr = array('i', [0, 0, 0, 0, 0])
  print(conn.recv_bytes_into(arr))    # => 8
  print(arr)                          # => array('i', [42, 1729, 0, 0,0])
```

출력

리스너 코드를 실행하면, 하나의 연결이 있을 때까지 무한정 대기하는 모습을 볼 수 있다. 연결이 되면, 승인한 연결과 해당 IP 주소 및 포트를 출력한다.

```
$ python3.6 23_listener.py
connection accepted from ('127.0.0.1', 56197)
```

리스너 프로그램을 시작하면 클라이언트 프로그램도 시작된다. 실행 중인 리스너와 성공적으로 연결되면, 전송된 메시지 배열을 출력한다.

```
$ python3.6 22_client.py
[2.25, None, 'junk', <class 'float'>]
b'hello'
8
array('i', [42, 1729, 0, 0, 0])
```

여기서 multiprocessing.connection 모듈의 진가가 나타나는데, 동일한 하드웨어에서 실행 중인 멀티프로세스 간의 정보를 효과적으로 어떻게 전달하는지 살펴봤다.

로깅

파이썬을 이용한 고성능 컴퓨팅 세계에 대해 알려면 멀티스레드 및 멀티프로세스 기반의 파이썬 애플리케이션에서 로깅을 어떻게 구현하는지 다시 한번 살펴봐야 한다.

이 책의 수많은 예제에서는 간단한 출력문 형태가 프로그램이 어떻게 진행돼가는지 정확히 로깅해주고 있다. 하지만 어려운 문제에 직면했을 때, 프로그램 출력을 로깅하는 문제에 있어 더 나은 방법이 필요하다.

애플리케이션에서 잘 구성된 로깅 시스템을 구현하는 것은 차후에 프로그램의 특성을 이해하는 좋은 방법이다.

개인적으로 파이썬 로깅 쿡북에 관련된 자료를 추가로 확인해보길 바란다. https://docs.python.org/3/howto/logging-cookbook.html

예제

다행히도 로깅은 파이썬에 기본적으로 설치된 모듈이니 다음과 같이 불러올 수 있다.

```
import logging
```

모듈을 불러온 후 다음과 같이 프로그램 디렉토리 내의 파일에 좀 더 의미 있는 로그 메시지를 로깅하는 시스템을 구성한다.

```
logging.basicConfig(filename='myapp.log', level=logging.INFO,
format='%(processName)-10s %(asctime)%s:%(levelname)s:%(message)s')
```

log를 진행하는 파일 이름과 로깅 수준으로 INFO를 명시했다. 다음으로 log 파일에 추가할 로깅 메시지의 포맷을 명시한다.

포맷 문자열에 추가해야 할 첫 번째 인자는 %(processName)-10s이므로 주의하자. 이는 프로그램의 모든 하위 프로세스의 결과를 확인할 수 있다.

```
import logging
from multiprocessing import Pool

logging.basicConfig(filename='myapp.log', level=logging.INFO,
format='%(processName)-10s %(asctime)%s:%(levelname)s:%(message)s')

def myTask(n):
  logging.info("{} being processed".format(n))
  logging.info("Final Result: {}".format(n*2))
  return n*2

def main():
  with Pool(4) as p:
    p.map(myTask, [2,3,4,5,6,])

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

이 코드를 실행하면 다음 출력이 myapp.logs 파일에 입력됨을 볼 수 있다.

각 결과에는 작업을 진행한 프로세스 이름과 수준, 출력으로 주어진 로그 정보가 나타난다.

```
ForkPoolWorker-1 %s:INFO:2 being processed
ForkPoolWorker-1 %s:INFO:Final Result: 4
ForkPoolWorker-2 %s:INFO:3 being processed
ForkPoolWorker-2 %s:INFO:Final Result: 6
ForkPoolWorker-3 %s:INFO:4 being processed
ForkPoolWorker-3 %s:INFO:Final Result: 8
ForkPoolWorker-4 %s:INFO:5 being processed
ForkPoolWorker-4 %s:INFO:Final Result: 10
ForkPoolWorker-1 %s:INFO:6 being processed
ForkPoolWorker-1 %s:INFO:Final Result: 12
```

순차적인 프로세스 통신하기

순차적인 프로세스 통신, 즉 CSP[Communicating Sequential Processes]는 여러 개의 동시성 모델이 각자 어떻게 상호작용하는지 보여준다. 일반적으로 여러 개의 동시성 프로세스 간에 메시지를 전달하는 채널을 주로 사용하며, 그 내부는 클로저[clojure] 및 Go 언어로 이뤄져 있다.

현재 활발히 사용되고 있는 개념이며, 다양한 도서 및 자료가 있으니 참고하기 바란다.

「Communicating Sequential Processes」(C.A.R. Hoare, 2015)를 참고하자.
http://www.usingcsp.com/cspbook.pdf

이러한 주제를 살펴보면, 기존의 객체지향 구성과 비교해 위 스타일을 이용하는 편이 문제를 구성하기에 더 쉬울 것이라 본다.

PyCSP

PyCSP란 CSP 기반 핵심 프리미티브로 구현된 파이썬 라이브러리다. 2006년부터 시작되어 릴리스가 드물게 되는 부분을 안정화한다. 이 라이브러리는 기본적인 파이썬 모듈로 동작되는 쉬운 분산 통신 모델을 제공한다.

전반적으로 파이썬 내장 데코레이터를 사용한 멀티프로세스 기반 파이썬 애플리케이션을 어떻게 구현하는지 보여준다.

PyCSP를 설치하기 위해 다음 pip를 사용하자.

```
python3.6 -m pip install pycsp
```

PyCSP에서 처리하기

다음 예제에서 PyCSP 모듈을 사용해 동시성 파이썬 애플리케이션을 어떻게 구성하는지 살펴보자. 보다시피, Process1과 Process2 함수를 정의한다. 다음으로 @process 애노테이션을 사용해 데코레이팅하며, 프로세스상 메소드의 전체적인 전환을 다룬다. 마지막으로, PyCSP 기반 프로그램에 Process1과 Process2를 병렬적으로 실행하도록 전달한다.

```
from pycsp.parallel import *
import time

@process
def Process1():
  time.sleep(1)
  print('process1 exiting')

@process
def Process2():
  time.sleep(2)
  print('process2 exiting')
```

```
Parallel(Process1(), Process2()) # 블로킹
print('program terminating')
```

출력

위 프로그램을 실행하면 Process1과 Process2가 성공적으로 실행되고 종료됨을 볼 수 있다.

```
$ python3.6 24_pycsp.py
process1 exiting
process2 exiting
program exiting
```

CSP의 개념은 흥미로운 부분 중 하나인데, 단순히 생각해볼 일임에도 불구하고 이 장의 마지막에 넣었다는 데 의의가 있다고 본다.

요약

8장에서는 멀티프로세싱에 대한 이해와 시스템에서 이를 어떻게 다루는지 살펴봤다. 적절한 종료를 바탕으로 모든 프로세스 라이프의 생성을 알아봤다.

크로스 프로세스 통신과 동기화 같은 다양하고 복잡한 사항을 비롯해, 7장에서 살펴본 ProcessPoolExecutor와 멀티프로세싱 풀의 차이점도 알아봤다.

다음으로 큰 성능 감소 없이 여러 프로세스 간의 통신과 동기화 등을 어떻게 구현하는지 간단히 살펴보고, 경합 조건 없는 시스템 관리에 한 발 더 다가섰다.

9장에서는 이벤트 기반 프로그래밍과 관련해 asyncio 모듈에 대해 자세히 배우며, 자체 이벤트 기반 파이썬 프로그램을 개발하고자 해당 모듈을 어떻게 이용하는지 살펴본다.

09

이벤트 기반 프로그래밍

9장에서는 이벤트 기반 프로그래밍의 세계를 소개하겠다. 이벤트 기반 프로그래밍은 이벤트에 초점을 맞춘 형태다. 다소 뻔한 부분이지만, 프로그래밍 관점에서 '이벤트'의 정의를 정확히 이해하고 요약해 그 내용을 알아보자.

우선, 이벤트 기반 프로그래밍의 일반적인 특징과 대부분의 이벤트 기반 파이썬 프로그램에 사용되는 asyncio 모듈을 소개한다.

asyncio 프로그래밍 세계도 소개한다. 이 내용만 이론적으로 다룬 책이 있을 정도로 유명한 라이브러리다. 중요한 부분은 간단하게 설명하고, 예제 프로그램을 활용하는 방법 또한 알아본다.

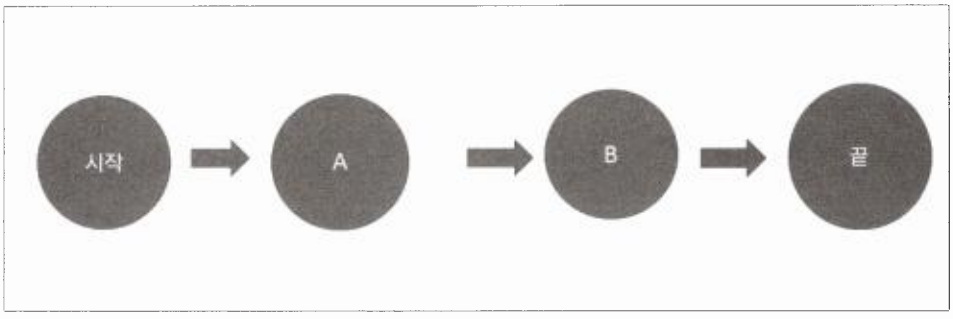
9장에서 다루는 내용은 다음과 같다.

- 이벤트 기반 프로그래밍이란?
- asyncio 모듈의 이해
- asyncio 기반 프로그램 디버깅
- 트위스티드 이벤트 기반 네트워크 엔진
- gevent

이제 파이썬에서 자신만의 이벤트 기반 시스템을 생성하고 동작시켜보자.

이벤트 기반 프로그래밍

이벤트 기반 프로그래밍은 이전에 다룬 한정된 파이썬 애플리케이션과는 다르다. 일반적으로 이러한 우선 프로그램은 플로우를 설정하고 메인 메모리의 실행 관점에서 프로그램의 어떤 부분이 비결정적일지라도 독립적인 접근법을 결정론적으로 따른다.

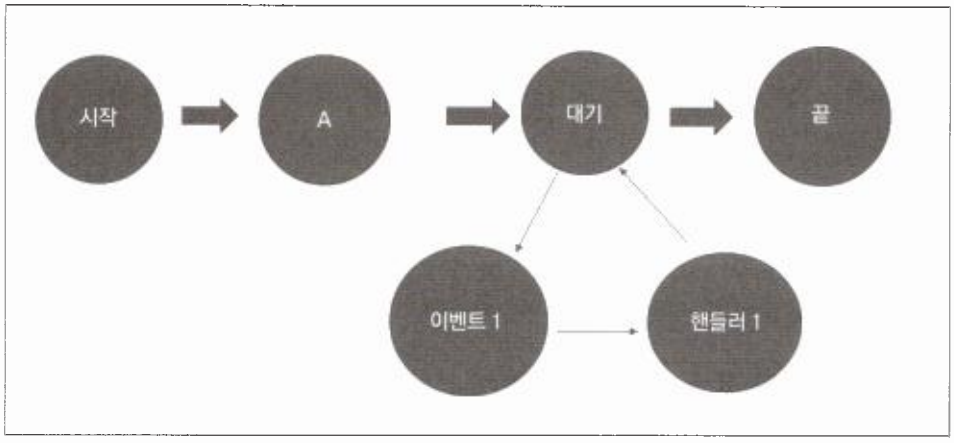


▲ 출처: https://mywebgranth.wordpress.com

이벤트 기반 프로그램에서는 일반적으로 이벤트에 반응하는 일정한 이벤트 루프event loop가 있다. 이벤트 루프가 시작되면, 어떤 것이 실행되고, 그 순서가 어떻게 되는지 결정하는 시스템상에 필요한 이벤트가 내려진다. 쉬운 예로 키보드 입력을 들 수 있다. 다

음에 어떤 키가 눌릴지 결정할 수는 없지만, 프로그램이 key-press 이벤트에 반응했을 때 주어지는 키에 매핑되는 특정 함수를 갖는다.

다음 그림에서 이벤트 기반 프로그램이 일반적으로 따르는 흐름 구성을 볼 수 있다. 프로그램이 시작되고 '대기' 상태에 진입하는 것을 볼 수 있다. 이는 특정 이벤트나 시스템 충돌 발생으로 인한 프로그램 종료 때까지 무한정 계속된다.



▲ 출처: https://mywebgranth.wordpress.com

'대기' 상태에서는 계속 이벤트를 받아들이고, 이를 이벤트 핸들러에게 전달한다. 여기서는 하나의 특정 이벤트가 하나의 핸들러와 매핑된 것을 보여주며, 특정 이벤트 기반 프로그램에서는 이벤트 및 핸들러의 개수가 이론적으로 무한대일 수도 있다.

이는 운영체제 같은 시스템에 이상적이다. 절차형 방식으로 OS를 작성해야 한다면, 디자인이 잘된 시스템과 달리 복잡하게 얽힌 코드를 읽기가 매우 어렵다는 사실을 알게 될 것이다.

시스템에서 이벤트 기반 개념을 활용해 전반적인 프로그램의 구조를 단순화하고 기능성을 높이고 디버깅 과정도 줄일 수 있다.

이벤트 루프

모든 이벤트 기반 파이썬 프로그램의 컴포넌트는 이벤트 루프를 갖고 있다. 예제에서 asyncio 이벤트 루프를 살펴보자. 이벤트 루프 안에서 할 수 있는 일은 다음과 같다.

- 호출 등록, 실행, 취소하기
- 하위 프로세스 및 외부 프로그램에서 관련 통신 실행하기
- 스레드 풀에 가장 많이 호출되는 함수 지정하기

모든 이벤트 루프는 필수적으로, 주어진 이벤트 타입과 매치된 함수가 연결되기 전에 발생하는 이벤트를 기다려야 한다.

이를 잘 설명하는 예시는 바로 웹 서버다. 수많은 웹 페이지로 구성된 웹사이트를 제공하는 서버가 있다고 하자. 이벤트 루프는 계속 대기 중일 테고, 요청이 일어나면 관련된 웹 페이지와 연결하게 된다.

웹 서버로부터 생성된 각 요청은 이벤트라고 보면 된다. 이러한 이벤트는 이벤트가 발생할 때 일어나도록 미리 구성해둔 함수와 매치된다.

asyncio

asyncio는 파이썬 3.4 버전에서 소개됐으며, 훌륭한 함수를 제공해 파이썬 커뮤니티에서 많은 인기를 얻고 있다.

asyncio는 이후에 다뤄볼 코루틴coroutine을 활용한 단일 스레드 기반 및 동시성 프로그램을 쉽게 작성해주는 모듈이다. 소켓과 그 외 자원들의 멀티플렉싱 I/O 접근 작업을 해주며, 비교적 쉽게 스레드 세이프 프로그램을 작성할 수 있도록 동기화 작업도 제공해주고 있다.

이 모듈에는 다음과 같이 다양한 개념들이 있다.

- 이벤트 루프
- 퓨처
- 코루틴
- 태스크
- 트랜스포트
- 프로토콜

이러한 각 개념들은 가독성이 높은 고성능 파이썬 프로그램을 작성하는 데 도움을 준다. 이후에 이러한 개념들에 대해 좀 더 자세히 살펴본다.

asyncio 모듈은 이벤트 기반 방식으로 프로그램을 작성하도록 다양한 툴과 클래스를 제공하므로, 다음 문서를 참고해보길 추천한다.

https://docs.python.org/3/library/asyncio.html

시작하기

다음 예제에서는 asyncio 이벤트 루프를 반환하는 get_event_loop() 메소드를 활용해 본다. 그런 다음, run_until_complete() 메소드를 활용해 실행하면서 코루틴을 취한다.

코루틴에 대해서는 이후에 살펴볼 것이며, 여기서는 이러한 코루틴이 asyncio에서 동시성 실행을 디자인할 경우 필수적인 함수임을 알아두자.

```
import asyncio
async def myCoroutine():
  print("Simple Event Loop Example")

def main():
  loop = asyncio.get_event_loop()
  loop.run_until_complete(myCoroutine())
  loop.close()
```

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

위 프로그램을 실행하는 콘솔에 Simple Event Loop Example이 출력됨을 볼 수 있다. 비교적 간단하면서도 이벤트 기반 프로그램을 이해하는 데 있어 이벤트 루프의 특징을 잘 보여준다.

이벤트 루프

지금까지 간단한 이벤트 루프 기반 프로그램을 만들어봤는데, 이벤트 루프에서 활용 가능한 여러 메소드를 바탕으로 효과적으로 이를 활용하는 방법을 살펴보자.

run_forever() 메소드

run_forever() 메소드는 이름 그대로 이벤트 루프를 시작해 계속 블로킹한다.

다음 예제에서 2개의 코루틴에서 계속 동작하는 이벤트 루프를 생성해보자.[1]

```
import asyncio

@asyncio.coroutine
def hello_world():
  yield from asyncio.sleep(1)
  print('Hello World')
  asyncio.async(hello_world())

@asyncio.coroutine
def good_evening():
  yield from asyncio.sleep(1)
  print('Good Evening')
  asyncio.async(good_evening())
```

1 Ctrl + C를 눌러 키보드 인터럽트를 발생시킨다. – 옮긴이

```
print('step: asyncio.get_event_loop()')
loop = asyncio.get_event_loop()
try:
  print('step: loop.run_until_complete()')
  asyncio.async(hello_world())
  asyncio.async(good_evening())
  loop.run_forever()
except KeyboardInterrupt:
  pass
finally:
  print('step: loop.close()')
  loop.close()
```

run_until_complete() 메소드

run_until_complete() 메소드는 이전에 살펴봤듯이, 자체적으로 종료하기 전에 이벤트 루프가 주어진다.

```
import asyncio
import time

async def myWork():
  print("Starting Work")
  time.sleep(5)
  print("Ending Work")

def main():
  loop = asyncio.get_event_loop()
  loop.run_until_complete(myWork())
  loop.close()

  print(loop.is_closed())

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

이 코드에서는 이벤트 루프가 시작되고 주어진 함수가 완료될 때까지 myWork async 함수를 실행한다.

stop() 메소드

stop() 메소드 또한 이름 그대로 run_forever() 메소드를 바탕으로 동작되는 이벤트 루프로 하여금 이후 적절한 지점에 멈추게 한다.

```
import asyncio
import time

async def myWork():
  print("Starting Work")
  time.sleep(5)
  print("Ending Work")

def main():
  loop = asyncio.get_event_loop()
  loop.run_until_complete(myWork())
  loop.stop()
  print("Loop Stopped")
  loop.close()

  print(loop.is_closed())

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

is_closed() 메소드

is_closed() 메소드는 이벤트 루프가 close() 메소드로부터 호출되어 종료됐다면 True를 반환한다.

```
print(loop.is_closed()) # True 또는 False를 출력한다.
```

close() 함수

close() 함수는 실행 중이 아닌 이벤트 루프를 종료하고 남은 콜백을 정리한다. 실행자의 경우 종료될 때까지 기다리지 않으며 이벤트 루프를 강제로 종료시킨다. 데이터베이스와 브로커처럼 끊임없이 동작하는 자원과 연결돼 있어 복잡하므로, 이러한 상황에서 종료가 필요하다면 주의해야 한다.

```
loop.close()
```

태스크

asyncio의 태스크는 이벤트 내 코루틴의 실행을 담당한다. 이러한 태스크는 한 번에 1개의 이벤트 루프에서만 실행되고, 병렬적으로 실행하고자 멀티스레드를 바탕으로 멀티 이벤트 루프를 실행하게 된다.

이전 장에서 실행자 및 풀 결합을 사용할 때 태스크를 고려해야 하는 상황처럼 asyncio에서도 동일하게 고려한다.

이 절에서는 asyncio 기반 프로그램에서 태스크로 작업하기 위해 사용할 핵심 함수를 살펴본다.

예제

예제 코드에서는 스케줄링되고 실행될 이벤트 루프에 필요한 5개의 코루틴을 생성할 함수를 정의한다. 코루틴의 스케줄링을 위해 이후 자세히 살펴볼 ensure_future() 메소드를 사용한다.

```
import asyncio
import time
```

```
@asyncio.coroutine
def myTask(n):
  time.sleep(1)
  print("Processing {}".format(n))

@asyncio.coroutine
def myGenerator():
  for i in range(5):
    asyncio.ensure_future(myTask(i))
  print("Completed Tasks")
  yield from asyncio.sleep(2)

def main():
  loop = asyncio.get_event_loop()
  loop.run_until_complete(myGenerator())
  loop.close()

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

코드를 실행하면 5개의 태스크가 성공적으로 생성, 스케줄링, 실행됨을 볼 수 있으며, 콘솔에 다음과 같이 출력된다.

```
$ python3.6 04_generator.py
Completed Tasks
Processing 0
Processing 1
Processing 2
Processing 3
Processing 4
```

all_tasks(loop=None) 메소드

all_tasks 메소드는 주어진 이벤트 루프 세트를 반환한다. 아무런 이벤트 루프도 전달되지 않았다면, 현재 이벤트 루프의 모든 태스크를 기본적으로 출력한다.

```
import asyncio

async def myCoroutine():
  print("My Coroutine")

async def main():
  await asyncio.sleep(1)

loop = asyncio.get_event_loop()
try:
  loop.create_task(myCoroutine())
  loop.create_task(myCoroutine())
  loop.create_task(myCoroutine())

  pending = asyncio.Task.all_tasks()
  print(pending)
  loop.run_until_complete(main())
finally:
  loop.close()
```

위 코드를 실행하면 3개의 코루틴이 콘솔에 출력된다. 이는 현재 이벤트 루프를 실행하고자 아직 스케줄링되지 않은 부분을 알려준다.

```
$ python3.6 16_asyncioTasks.py
{<Task pending coro=<myCoroutine() running at 16_asyncioTasks.py:3>>, <Task
pending coro=<myCoroutine() running at 16_asyncioTasks.py:3>>, <Task
pending coro=<myCoroutine() running at 16_asyncioTasks.py:3>>}
My Coroutine
My Coroutine
My Coroutine
```

current_tasks() 함수

현재 어떤 태스크가 실행 중인지 알아야 할 때가 있다. 필요하다면 이벤트 루프에서 현재 실행 중인 태스크 리스트를 모두 돌면서, 원한다면 종료할 수도 있다.

다음 예제에서는 create_task 함수를 사용해 3개의 태스크를 스케줄링하고 이 함수에 myCoroutine 함수를 입력 인자로 전달한다.

```
import asyncio
async def myCoroutine():
  print("My Coroutine")

async def main():
  current = asyncio.Task.current_task()
  print(current)

loop = asyncio.get_event_loop()
try:
  loop.create_task(myCoroutine())
  loop.create_task(myCoroutine())
  loop.create_task(myCoroutine())
  loop.run_until_complete(main())
finally:
  loop.close()
```

코드를 실행하면 모든 코루틴이 정상적으로 실행되고, 지금 실행 중인 코루틴은 메인 코루틴이 실행될 때까지 보류된다.

```
$ python3.6 16_asyncioTasks.py
My Coroutine
My Coroutine
My Coroutine
<Task pending coro=<main() running at 16_asyncioTasks.py:8>
cb=[_run_until_complete_cb() at
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/asyncio/bas
e_events.py:176]>
```

cancel() 함수

cancel 함수는 퓨처나 코루틴을 취소하는 요청을 한다.

```
import asyncio

async def myCoroutine():
  print("My Coroutine")

async def main():
  current = asyncio.Task.current_task()
  print(current)

loop = asyncio.get_event_loop()
try:
  task1 = loop.create_task(myCoroutine())
  task2 = loop.create_task(myCoroutine())
  task3 = loop.create_task(myCoroutine())
  task3.cancel()
  loop.run_until_complete(main())
finally:
  loop.close()
```

코드를 실행하면 task1과 task2가 정상적으로 실행됨을 볼 수 있다. 세 번째로 스케줄링된 태스크는 취소 호출로 인해 실행되지 않는다.

지금까지 태스크를 취소하는 예제를 살펴봤는데, 세 번째 태스크를 취소하는 방법을 알아봤다. 하지만 특정 환경에서는 cancel 함수가 보류 중인 태스크를 확실히 취소함을 보장할 수 없다.

```
$ python3.6 16_asyncioTasks.py
My Coroutine
My Coroutine
<Task pending coro=<main() running at 16_asyncioTasks.py:8>
cb=[_run_until_complete_cb() at
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/asyncio/bas
e_events.py:176]>
```

태스크 함수

이전 절에서 각 태스크 객체에서 사용되는 함수를 살펴봤는데, 각 태스크 조합에서는 어떻게 작동할까? asyncio 모듈에서 비교적 쉽게 멀티 태스크를 활용하도록 태스크 함수를 제공한다.

as_completed(fs, *, loop=None, timeout=None) 함수

as_completed 함수는 실행된 모든 태스크에서 반환된 결과를 사용할 수 있다. 이러한 결과는 다른 코루틴에 전달되어 필요에 따라 간단한 로깅이나 기타 작업에 이용된다.

as_completed( ) 함수는 다음과 같이 반복문으로 반환될 때마다 결과를 받는다.

```
for f in as_completed(fs):
  result = yield from f
  # result를 사용한다.
```

ensure_future(coro_or_future, *, loop=None) 함수

ensure_future 함수는 coroutine 객체의 실행을 스케줄링하며, 퓨처에 감싸져 task 객체를 반환한다.

```
import asyncio
...
async def myCoro( ):
  print("myCoro")
...
task = asyncio.ensure_future(myCoro( ))
...
```

wrap_future(future, *, loop=None) 함수

이 함수는 concurrent.futures.Future 객체를 asyncio.Future 객체로 변환해주는 간단한 어댑터adapter다.

다음과 같이 concurrent.futures.Future 객체를 변환할 수 있다.

```
with ThreadPoolExecutor(max_workers=4) as executor:
  future = executor.submit(task, (4))
  myWrappedFuture = asyncio.wrap_future(future)
```

gather(*coroes_or_futures, loop=None, return_exceptions=False) 함수

gather 함수는 다소 복잡하다. 코루틴 세트 및 퓨처를 인자로 취해 입력된 세트로부터 종합된 결과를 future 객체로 반환한다.

```
import asyncio

async def myCoroutine(i):
  print("My Coroutine {}".format(i))

loop = asyncio.get_event_loop()
try:
  loop.run_until_complete(asyncio.gather(myCoroutine(1), myCoroutine(2)))
finally:
  loop.close()
```

코드를 실행하면 asyncio.gather 함수에 전달된 모든 코루틴이 이벤트 루프가 종료되고 정상적으로 완료됨을 볼 수 있다.

```
$ python3.6 16_asyncioTasks.py
My Coroutine 1
My Coroutine 2
```

wait() 함수

wait 함수는 모든 퓨처 및 코루틴에 전달된 첫 번째 파라미터가 완료될 때까지 프로그램을 블로킹한다.

```
import asyncio

async def myCoroutine(i):
  print("My Coroutine {}".format(i))

loop = asyncio.get_event_loop()
try:
  tasks = []
  for i in range(4):
    tasks.append(myCoroutine(i))
  loop.run_until_complete(asyncio.wait(tasks))
finally:
  loop.close()
```

코드를 실행하면 4개의 코루틴이 정상적으로 생성되고 프로그램이 종료됨을 볼 수 있다.

```
$ python3.6 16_asyncioTasks.py
My Coroutine 0
My Coroutine 1
My Coroutine 2
My Coroutine 3
```

퓨처

7장에서 실행자와 풀을 자세히 다룰 때 퓨처 객체에 대해 다뤄봤다. 하지만 asyncio 모듈은 파이썬 프로그램에서 다소 다른 구현 형태를 제공한다.

퓨처의 asyncio.futures.Future 객체는 내부 개념상 concurrent.futures.Future 객체

와 거의 동일하다. 퓨처 객체는 이후에 결과를 받고자 하는 의도로 생성된다. 기본 퓨처 객체가 가진 메소드 또한 동일하게 나타난다.

하지만 공식 문서를 살펴보면 특징적인 변화 세 가지가 있으니 https://docs.python.org/3/library/asyncio-task.html#asyncio.Future를 참고하자. 이러한 부분은 개발자들에게 유용하다.

- result()와 exception() 함수는 timeout 파라미터를 갖지 않으며 퓨처가 아직 일어나지 않았을 경우 예외를 발생시킨다.
- add_done_callback() 포맷으로 등록된 콜백은 call_soon_threadsafe() 이벤트 루프를 바탕으로만 호출된다.
- future 클래스는 concurrent.futures 패키지의 wait 및 as_completed 함수와 호환되지 않는다.

예제

다음 예제에서는 future 객체에서 myCoroutine으로 정의된 코루틴을 어떻게 감싸는지 살펴보고, 이후 정의할 메인 코루틴으로 다뤄본다.

이 예제에서는 asyncio.ensure_future를 호출하기 전의 await 사용이 중요하다. ensure_future() 메소드는 이후 코루틴을 감쌀 뿐만 아니라 실행 스케줄링 또한 하므로, result()에 접근하기 전에 완료해야 한다.

```
import asyncio

async def myFuture(future):
  await asyncio.sleep(1)
  future.set_result("My Future Has Completed")

async def main():
  future = asyncio.Future()
```

```
  await asyncio.ensure_future(myFuture(future))
  print(future.result())

loop = asyncio.get_event_loop()
try:
  loop.run_until_complete(main())
finally:
  loop.close()
```

출력

프로그램을 실행하면 코루틴이 정상적으로 퓨처 객체로 변환됨을 볼 수 있고, .result() 메소드와 같이 접근할 수 있다.

```
$ python3.6 12_future.py
My Future Has Completed
```

코루틴

코루틴은 threading 모듈에서의 Thread 객체와 유사하다. asyncio 기반 애플리케이션의 코루틴을 활용해 단일 스레드 기반 컨텍스트에서 동작하는 예외를 가진 비동기 프로그램을 작성할 수 있다.

일반적으로 이벤트 기반 프로그램에서 매직이 발생한 부분은 asyncio 모듈에서 중요한 부분이다. asyncio 기반 프로그램을 살펴보면 이러한 코루틴 객체를 자주 활용했음을 볼 수 있다.

코루틴을 구현하는 데는 다양한 방법이 있는데, 가장 먼저 async def 함수(파이썬 3.5부터 지원)를 구현해야 한다. 다음에서 이 함수를 바탕으로 코루틴을 구현하는 것을 볼 수 있다.

```
import asyncio

async def myCoroutine():
  print("My Coroutine")

def main():
  loop = asyncio.get_event_loop()
  loop.run_until_complete(myCoroutine())
  loop.close()

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

두 번째는 @asyncio.coroutine 데코레이터를 사용한 결합에서 제너레이터를 활용하는 것이다.

```
import asyncio

@asyncio.coroutine
def myCoroutine():
  print("My Coroutine")

def main():
  loop = asyncio.get_event_loop()
  loop.run_until_complete(myCoroutine())
  loop.close()

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

코루틴 변경하기

파이썬 시스템에서 높은 성능이 필요할 때 코루틴의 호출을 연결할 수 있다.

공식 문서를 보면 이러한 연결 개념을 잘 설명해주는 샘플 코드가 있다. 이 코드에는

async def로 표시된 2개의 코루틴이 있다. compute 코루틴을 바탕으로 1초 동안의 유휴 상태 후 x와 y의 합을 반환한다.

하지만 두 번째 코루틴은 다음과 같이 첫 번째 코루틴과 연결돼 있다고 해보자.

```
import asyncio

async def compute(x, y):
  print("Compute %s + %s ..." % (x, y))
  await asyncio.sleep(1.0)
  return x + y

async def print_sum(x, y):
  result = compute(x, y)
  print("%s + %s = %s" % (x, y, result))

loop = asyncio.get_event_loop()
loop.run_until_complete(print_sum(1, 2))
loop.close()
```

위 코드를 실행하면 다음과 같이 출력된다.

```
$ python3.6 06_chainCoroutine.py
1 + 2 = <coroutine object compute at 0x1031fc0f8>
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/asyncio/eve
nts.py:126: RuntimeWarning: coroutine 'compute' was never awaited
self._callback(*self._args)
```

기본적으로 print_sum() 메소드 내의 compute 함수가 콜백을 기다리지 않았으며, 프로그램은 결과를 받은 것처럼 계속하려 했다.

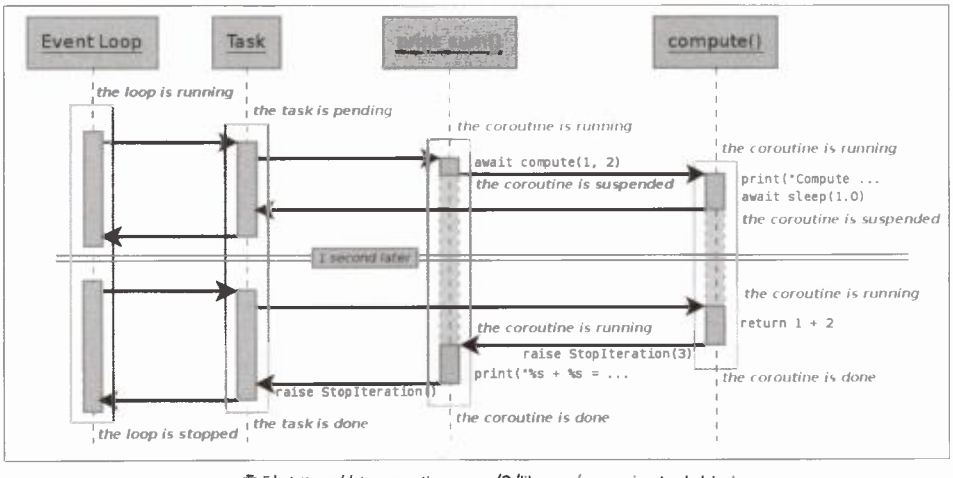
이러한 문제를 해결하고자 await 키워드를 활용해야 한다. await 키워드는 호출된 코루틴이 결과를 반환하기까지 이벤트 루프의 진행을 막는다. 하지만 이러한 코드의 단점은 비동기성의 우수성을 잃어 기본적인 동기적 실행으로 돌아간다는 것이다. 빠르고 쉬운

결정론적 실행을 제공하지만 성능에 영향을 미칠 수 있기에 await의 사용 필요성을 판단해야 한다.

print_sum 코루틴은 compute 코루틴이 반환한 변수 결과를 인스턴스화한다.

```
import asyncio

async def compute(x, y):
  print("Compute %s + %s ..." % (x, y))
  await asyncio.sleep(1.0)
  return x + y

async def print_sum(x, y):
  result = await compute(x, y)
  print("%s + %s = %s" % (x, y, result))

loop = asyncio.get_event_loop()
loop.run_until_complete(print_sum(1, 2))
loop.close()
```

이 프로그램의 공식 문서에서 볼 수 있는 다음 그림은 코루틴 연결이 어떻게 실제로 작동되는지 명확히 보여준다.



▲ 출처: https://docs.python.org/3/library/asyncio-task.html

출력

코드를 실행하면 코루틴이 정상적으로 서로 연결되고, print_result 코루틴이 정상적으로 compute 함수의 결과를 대기한다.

```
$ python3.6 06_chainCoroutine.py
Compute 1 + 2 ...
1 + 2 = 3
```

트랜스포트

트랜스포트는 다양한 통신 타입 구현을 지원하는 asyncio 모듈에 포함된 클래스다. 전체적으로 4개의 트랜스포트 종류가 있는데, 모두 BaseTransport 클래스로부터 상속받았다.

- ReadTransport
- WriteTransport
- DatagramTransport
- BaseSubprocesTransport

BaseTransport 클래스에는 앞에서 나열된 4개의 트랜스포트에 걸쳐 5개의 메소드가 있다.

- close(): 트랜스포트를 닫는다.
- is_closing(): 트랜스포트가 닫혔는지 참, 거짓을 반환한다.
- get_extra_info(name, default=None): 추가적인 트랜스포트 정보를 반환한다.
- set_protocol(protocol): 함수명 그대로 프로토콜을 설정한다.
- get_protocol(): 현재 프로토콜을 반환한다.

프로토콜

프로토콜은 asyncio 모듈에서 다소 이상한 모듈 중 하나로, 이전에 정의된 인터페이스를 따르는 클래스다. 참고로, 인터페이스는 필요조건에 부합하는 방식으로 작성된 클래스와 같은 형태로 동작한다.

다시 말해 프로토콜의 개념은 이 장에서 다루지 않겠으나, 파이썬에서 트랜스포트와 함께 살펴봤으면 한다. 트랜스포트와 프로토콜에 관한 공식 문서는 https://docs.python.org/3/library/asyncio-protocol.html에서 찾아볼 수 있다.

코루틴 간의 동기화

asyncio의 코루틴은 비결정적인 방식으로 동작하며, 단일 스레드로 동작되는 부분과 관계없이 코드가 경합 조건이 될 가능성이 여전히 있다.

경합 조건에 대한 두려움으로 인해 asyncio 모듈의 결합을 바탕으로 이전에 다룬 동기화 원리를 어떻게 사용할지 꼭 알아야 한다.

이 절에서는 락, 이벤트, 조건, 큐를 바탕으로 이들이 이벤트 기반 파이썬 프로그램에서 어떻게 작동되는지 살펴본다. 내용의 한계상 asyncio 코루틴 간의 동기화와 관련해 가장 중요한 개념인 락과 큐에 대해서만 전체 코드 예제를 살펴본다.

Lock

asyncio 모듈의 Lock 메소드는 threading 모듈 내의 기본적인 락 구현 형태와 비교해 실용적인 부분에 있어 거의 동일하다.

asyncio 모듈 구현은 어떤 코루틴도 동일한 중요한 부분에 동시에 실행할 수 없기에 asyncio 기반 프로그램의 중요 부분에 락이 가능토록 한다.

asyncio 모듈에서 락을 인스턴스화하기 위해 다음과 같이 해보자.

```
import asyncio
myLock = asyncio.Lock( )
```

전체 코드를 살펴보자. myWorker를 호출해 async 코루틴을 정의하며 시작하는데, 락을 인자로 한다. myWorker 코루틴에서 with 키워드를 사용해 락을 받는다. 락을 획득하면 간단히 print 문을 출력하는 등 중요한 상태를 실행한다.

메인 코루틴에서는 coroutine 함수에 인자로 전달되는 락을 인스턴스화한다.

```
import asyncio
import time

async def myWorker(lock):
  with await lock:
    print(lock)
    print("myWorker has attained lock, modifying variable")
    time.sleep(2)
  print(lock)
  print("myWorker has release the lock")

async def main(loop):
  lock = asyncio.Lock( )
  await asyncio.wait([myWorker(lock),myWorker(lock)])

loop = asyncio.get_event_loop( )
try:
  loop.run_until_complete(main(loop))
finally:
  loop.close( )
```

위 코드를 실행하면 각 작업자의 차례에 락을 얻고 이전의 락을 릴리스하기 전에 실행해야 할 부분을 진행하고, 두 번째 코루틴이 효과적으로 계속되며 코드상의 중요한 부분을 실행한다.

```
$ python3.6 asyncioLock.py
<asyncio.locks.Lock object at 0x103121dd8 [locked]>
myWorker has attained lock, modifying variable
<asyncio.locks.Lock object at 0x103121dd8 [unlocked]>
myWorker has release the lock
<asyncio.locks.Lock object at 0x103121dd8 [locked]>
myWorker has attained lock, modifying variable
<asyncio.locks.Lock object at 0x103121dd8 [unlocked]>
myWorker has release the lock
```

큐

asyncio.Queue는 파이썬에서 주어진 기본 큐 모듈과 구성이 거의 동일하다.

이 예제에서는 프로듀서와 컨슈머 모두를 생성한다. 프로듀서는 1에서 5 사이의 articleIds를 생성하며, 컨슈머가 이러한 모든 아티클을 읽는다. 프로듀서와 컨슈머 모두 네임드 코루틴을 정의한다. newProducer 코루틴에서 asyncio.Queue에 전달되는 ID를 집어넣는 put 명령어를 1초마다 실행한다.

newConsumer 함수에서는 공유된 큐로부터 아티클 ID를 검색하고 get 요청을 거듭 실행한다. 주어진 아티클 ID에서 컨슈밍된 사항을 출력하고, 다음 아티클을 반복하며 진행한다.

```
import asyncio
import random
import time

@asyncio.coroutine
def newsProducer(myQueue):
  while True:
    yield from myQueue.put(random.randint(1,5))
    yield from asyncio.sleep(1)

@asyncio.coroutine
def newsConsumer(myQueue):
  while True:
```

```
    articleId = yield from myQueue.get()
    print("News Reader Consumed News Article {}", articleId)

myQueue = asyncio.Queue()

loop = asyncio.get_event_loop()

loop.create_task(newsProducer(myQueue))
loop.create_task(newsConsumer(myQueue))

try:
  loop.run_forever()
finally:
  loop.close()
```

코드를 실행하면 newConsumer 코루틴으로부터 일정한 형태의 출력문이 나타난다. 이는 정상적으로 asyncio.Queue 객체에서 프로듀서와 컨슈머를 이용했고 통신이 진행됐음을 보여준다.

```
$ python3.6 13_asyncioQueue.py
News Reader Consumed News Article {} 4
News Reader Consumed News Article {} 1
News Reader Consumed News Article {} 4
News Reader Consumed News Article {} 2
...
```

이벤트와 조건

4장에서 살펴봤듯이, 이벤트는 플래그가 지정된 부분까지 실행하는 여러 컨슈머를 블로킹한다. 다음과 같이 인스턴스화할 수 있다.

```
myEvent = asyncio.Event()
```

컨디션도 마찬가지로 그 외 코루틴에 의해 계속됨을 알려주는 지점까지 작업을 블로킹한다. 다음과 같이 인스턴스화할 수 있다.

```
myCondition = asyncio.Condition( )
```

세마포어와 한정된 세마포어

asyncio 모듈은 세마포어와 threading 모듈에서 찾아볼 수 있는 BoundedSemaphore 프리미티브 모두 자체 구현으로 제공한다.

4장에서 보여준 구현과 정확히 동일한 함수를 제공한다. 코루틴을 얻으면 카운터가 감소하고 릴리스하면 증가한다. 다시 말해, 주어진 자원에 접근하면서 코루틴의 수를 조절할 수 있으며 자원 부족이 프로그램상에 발생하지 않게 한다.

다음과 같이 인스턴스화한다.

```
import asyncio
...
mySemaphore = asyncio.Semaphore(value=4, *, loop=None)
...
boundedSemaphore = asyncio.BoundedSemaphore(value=4, *, loop=None)
```

하위 프로세스

단일 프로세스 프로그램은 소프트웨어에서 적절한 함수 필요조건을 충족하지 못할 수 있다. 이전 장에서 멀티프로세스를 바탕으로 성능을 높이는 메커니즘을 살펴봤는데, asyncio는 다행히 이벤트 기반 프로그램에서 하위 프로세스의 성능을 충분히 활용한다.

이는 프로그램의 복잡성을 갑자기 높일 수도 있기에 적극적으로 추천하지 않는다. 하지만 유용한 부분도 있는데, 다음 공식 문서를 참고하자.

https://docs.python.org/3/library/asyncio-subprocess.html

asyncio 프로그램 디버깅

asyncio 기반 애플리케이션을 디버깅할 때 다행히도 다양하게 고려해볼 부분이 있다. asyncio 모듈 개발자는 친절하게 디버깅 모드를 제공하는데, 시스템 코드를 많이 고칠 필요 없이 디버깅 작업이 가능토록 도와준다.

디버깅 모드

asyncio 기반 프로그램에서 디버깅 모드는 비교적 쉬우며 다음 함수만 호출하면 된다.

```
loop.set_debug(True)
```

다음 전체 코드를 살펴보면서 기본적인 로깅과 어떻게 다른지 알아보자. 예제에서는 간단한 이벤트 루프를 생성하고 여기에 태스크를 전달한다.

```
import asyncio
import logging
import time

logging.basicConfig(level=logging.DEBUG)

async def myWorker():
  logging.info("My Worker Coroutine Hit")
  time.sleep(1)
```

```
async def main():
  logging.debug("My Main Function Hit")
  await asyncio.wait([myWorker()])

loop = asyncio.get_event_loop()
loop.set_debug(True)
try:
  loop.run_until_complete(main())
finally:
  loop.close()
```

코드를 실행하면 다음과 같이 출력된 것을 볼 수 있다.

```
$ python3.6 11_debugAsyncio.py -Wdefault
DEBUG:asyncio:Using selector: KqueueSelector
DEBUG:root:My Main Function Hit
INFO:root:My Worker Coroutine Hit
WARNING:asyncio:Executing <Task finished coro=<myWorker() done, defined at
11_debugAsyncio.py:7> result=None created at
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/asyncio/tas
ks.py:305> took 1.004 seconds
DEBUG:asyncio:Close <_UnixSelectorEventLoop running=False closed=False
debug=True>
```

추가된 로그 출력이 있음을 주목하자. 이러한 추가된 로그는 이벤트 루프와 관련해 앞으로 어떻게 할지 고민해보게 한다.

예제의 디버깅 모드는 코루틴 중 하나가 이벤트 루프가 종료됐을 때 100밀리초 이상 실행된 부분을 알 수 있다.

이러한 asyncio의 디버깅 모드는 코루틴이 어떤 것을 전달하지 않을지 결정하거나 프로그램의 성능을 줄이는 부분에 유용하다. 좀 더 알아봐야 할 사항들을 살펴보자.

- call_soon()과 call_at() 메소드는 잘못된 스레드로부터 호출될 경우 예외를 발생시킨다.

- 선택자의 실행 시간 로깅하기
- ResourceWarning 경고는 트랜스포트와 이벤트 루프가 명시적으로 닫히지 않았을 경우 발생한다.

이는 초보자를 위한 것이다. 전반적으로 비교적 유용한 툴이며, 프로그램을 살펴보는 수단으로 일반적인 Pdb보다 다양한 옵션을 제공한다. 필요하다면 좀 더 자세한 예제가 있으니 asyncio 디버깅 모드에 관한 공식 문서를 살펴보기 바란다.

https://docs.python.org/3/library/asyncio-dev.html#asyncio-debug-mode

트위스티드

파이썬의 트위스티드Twisted는 웹 서버, 메일 클라이언트, 하위 시스템 등 대규모 프로젝트에 사용되는 강력한 이벤트 기반 네트워킹 엔진이다.

트위스티드는 복잡한 코드 없이 강력한 프로그램을 생성할 수 있고, 로우 레벨 API를 모두 제공하고 있다. 비동기 및 이벤트 기반 형태로 디자인돼 asyncio 모듈과 견줄 만하다.

트위스티드 프레임워크에 대해 더 알아보고 싶다면, 아베 페티크(Abe Fettig)와 제시카 맥켈러(Jessica McKellar)가 쓴 『Twisted Network Programming Essentials』(2014)를 참고하자.

간단한 웹 서버 예제

트위스티드는 웹사이트 구축에 있어 매우 좋은 프레임워크다. 브라우저에서 상대 파일 위치로부터 파일을 제공하는 특정 지점에 반응하는 간단한 TCP 서버를 구축한다.

예제에서는 프로그램과 동일한 디렉토리 내의 tmp 내 로컬 파일을 제공하는 간단한 웹 서버를 구축한다.

먼저 twisted.web과 twisted.internet 모듈에서 필요한 부분을 불러온다. 그런 다음, 구축할 서버의 디렉토리 파일 리소스 인스턴스를 정의한다. 다음으로 새로 정의한 리소스를 사용해 사이트 팩토리의 인스턴스를 생성한다.

마지막으로, Site 팩토리를 TCP 포트와 매핑하고 리스닝을 시작하고 reactor.run()을 호출하며, 유입되는 모든 TCP 연결과 모든 경로의 바이트를 전달하는 등의 작업을 다루는 전체 프로그램을 가동한다.[2]

```
from twisted.web.server import Site
from twisted.web.static import File
from twisted.internet import reactor, endpoints

resource = File('tmp')
factory = Site(resource)
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8888)
endpoint.listen(factory)
reactor.run()
```

이는 .html 파일을 바탕으로 제공하는데, <h1> 태그로 바디[body]를 나타낸다. 따라서 간단하게 웹사이트를 구축할 수 있다.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Twisted Home Page</title>
</head>
<body>
  <h1>Twisted Homepage</h1>
</body>
</html>
```

2 모듈을 불러오는 데 오류가 있다면 파이썬 2를 사용해본다. – 옮긴이

이 코드를 실행하고 브라우저에 http://localhost:8888/을 입력해 들어가면 방금 구축한 서버가 생성된 것을 볼 수 있다.

여기까지 트위스티드 프레임워크의 한 부분만 예제로 구현했는데, 내부 작동 방식은 분량상 다루지 못했다. 전반적인 프레임워크의 성능이 우수하기에 과거에 많이 사용했다.

gevent

gevent 네트워크 라이브러리는 코루틴의 상단을 구성한다. 트위스티드와 비교해 비슷한 면이 있으며, 이벤트 기반 파이썬 애플리케이션을 구성할 수 있는 함수를 제공한다.

특징은 다음과 같다.

- 빠른 이벤트 루프
- greenlet 패키지를 기반으로 한 가벼운 실행 단위
- 파이썬 기본 라이브러리에서 개념을 재사용할 수 있는 API
- 협동적 소켓과 SSL 모듈
- TCP/UDP/HTTP 서버
- 스레드 풀
- 하위 프로세스 지원

그 외에도 다양한 특징들이 있는데, 개발자들이 강력하고 유연한 시스템을 구성하도록 클래스와 내부 메소드를 제공한다. 여기서 제공하는 모든 부분을 알고 싶다면, 다음 공식 문서를 참조하자.

http://www.gevent.org/contents.html

이벤트 루프

asyncio 모듈과 마찬가지로 gevent도 이벤트 루프 개념을 활용한다. 여기서의 이벤트 루프는 내부에 등록된 상태로 이벤트를 다루기에 효율적인 디자인이 가능하다. 이벤트로 인한 자원 낭비를 줄이고 이벤트의 실제 진행에 집중하며 OS가 이벤트 알림의 전달을 처리하게 한다.

greenlet

gevent 프레임워크의 핵심은 greenlet이다. greenlet은 협업 스케줄링된 C 언어로 작성한 매우 가벼운 코루틴이다. 이는 매우 간단한 스레드와 비슷한 객체를 제공해 멀티스레드 동작 없이 파이썬 프로그램에서 동시성 프로그램이 가능하다.

이러한 의사pseudo 스레드는 greenlet 인스턴스 생성을 바탕으로 만들고, 시작 메소드를 호출한다. 다음으로 코드를 실행하고, 유효 상태를 거쳐 그 밖의 스레드가 차지한다. 이 같은 반복 사이클은 프로그램의 타깃이 완료되고 종료될 때까지 진행한다.

다음과 같은 spawn 함수를 사용해 greenlet을 정의한다.

```
import gevent

def myGreenlet():
  print("My Greenlet is executing")

gevent.spawn(myGreenlet)
gevent.sleep(1)
```

마찬가지로, 다음과 같이 하위 클래싱을 정의해본다.

```
import gevent
from gevent import Greenlet
```

```
class MyNoopGreenlet(Greenlet):
  def __init__(self, seconds):
    Greenlet.__init__(self)
    self.seconds = seconds

  def _run(self):
    print("My Greenlet executing")
    gevent.sleep(self.seconds)

  def __str__(self):
    return 'MyNoopGreenlet(%s)' % self.seconds

g = MyNoopGreenlet(4)
g.start()
g.join()
print(g.dead)
```

예제: 호스트이름

다음 예제에서는 파이썬 애플리케이션에서 gevent를 사용해본다. 3개의 URL을 취하는 배열을 생성하고, socket.gethostbyname() 함수를 사용해 URL의 IP 주소를 검색하고자 gevent 배열을 스폰한다.

다음으로 joinall()을 사용해 2초의 시간 제한을 두어 IP 주소 반환을 출력한다.

```
import gevent
from gevent import socket

def main():
  urls = ['www.google.com', 'www.example.com', 'www.python.org']
  jobs = [gevent.spawn(socket.gethostbyname, url) for url in urls]
  gevent.joinall(jobs, timeout=2)
  print([job.value for job in jobs])

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

출력

코드를 실행하면 다음과 같은 형태로 출력된다.

```
$ python3.6 19_geventSimple.py
['216.58.211.100', '93.184.216.34', '151.101.60.223']
```

monkey 패키지를 이용한 패치

gevent 라이브러리를 활용한 그 밖의 방법 중 하나는 monkey 패키지를 활용한 패치다. 실제 monkey 패키지로는 아무런 작업을 안 하며, 런타임에서 클래스 및 모듈의 동적 변경에 집중한다.

이를 이용한 작업은 데이터베이스, REST API 같은 외부 의존성을 바탕으로 값을 검색하는 메소드 기반 파이썬 프로그램을 생각해보면 된다. monkey 패키지의 패치는 단위 테스트 도중 외부 의존성에 영향을 주지 않게 메소드를 종료하는 수단으로 사용된다.

그렇다면 gevent 라이브러리에 왜 이것이 필요할까? gevent에서는 monkey 패치를 함수와 클래스를 협업적인 구성으로 바꾸는 데 활용할 수 있다. 가장 우수한 예제는 socket 모듈이다. DNS 요청은 기본적으로 직렬 형태이기에 양이 많으면 속도가 느리다.

이러한 DNS 요청을 동시에 진행하고자 하면 monkey 패치를 활용한다. gevent.monkey를 이용해 socket 모듈의 함수와 클래스를 협업 구성체로 바꿀 수 있고 성능 문제로 해결할 수 있다.

```
from gevent import monkey; monkey.patch_socket()
import urllib2 # 현재의 greenlet 코루틴에서는 유용하다.
```

성능을 높일 수 있는 기술에 대해 더 알고 싶다면, 다음 공식 문서를 참조하자.

http://www.gevent.org/intro.html#monkey-patching

요약

9장에서는 asyncio 모듈의 작동 방식과, 이벤트 기반 파이썬 시스템에서 어떻게 활용 가능한지 이벤트 기반 프로그램의 체계를 살펴봤다.

asyncio 모듈에 대해 자세히 다루고, 이벤트 루프의 구성 및 루프 내 코루틴의 연결, 이벤트 핸들러의 구성 등에 대해서도 이야기했다.

10장에서는 RxPY 모듈을 활용해 리액트 프로그램을 작성하는 방법과 기존의 이벤트 기반 프로그램과의 차이점을 살펴본다.

10

리액트 프로그래밍

이벤트 기반 프로그래밍은 이벤트와 관련된 반면, 리액트 프로그래밍reactive programming은 순수하게 데이터 중심이다. 다시 말해, 새로운 데이터를 받을 때마다 이벤트로 간주한다. 이로 인해 기술적으로 정의하면 이벤트 기반 프로그래밍의 한 갈래라고 볼 수 있다. 하지만 이러한 방식이 기존과 다르며 자주 이용되기에 10장에서 소개한다.

10장에서는 리액트 프로그래밍과 관련된 파이썬 라이브러리인 RxPY에 대해 자세히 다뤄본다. 이 라이브러리의 특징과 이를 활용해 비동기 프로그램을 작성하는 방법도 배운다.

시작하기에 앞서 RxPY의 기본적인 사항에 대해 배워보자.

- 관찰자 및 관찰 가능 속성 다루기
- 람다 함수 사용하기

- 다양한 오퍼레이터 연결해 활용하기
- 핫 및 콜드 관찰 가능 속성의 차이점
- 멀티캐스팅

PyFunctional 라이브러리에 대해서도 알아보는데, RxPY와의 차이점과 특정 상황에서 어떻게 활용하는지 살펴본다. 공식 문서에 나오는 예제는 토마스 닐드Thomas Nield의 'Reactive Python for Data Science'라는 동영상 강의에서 다뤘으니 참고하자. 여기에는 이 책에서 다루지 못한 다양한 부분이 나오니 꼭 들어보길 추천한다. http://shop.oreilly.com/product/0636920064237.do를 참고하자.

리액트 프로그래밍의 기본

리액트 프로그래밍은 기존의 일반적인 프로그램 스타일과는 완전히 다르다. 리액트 프로그램의 장단점을 이해함으로써 훌륭한 소프트웨어에 도달할 수 있다.

리액트 프로그래밍은 기존의 관습적인 스타일에서 벗어나 이벤트 흐름으로서의 데이터 표현에 집중할 수 있게 한다. 프로그램은 이러한 연속적 데이터 흐름을 받고 이벤트를 취한다. 이는 시스템 흐름을 간단하고 빠르게 해 기존의 아키텍처 및 스타일에 비해 다루기 쉽고 유지 관리에도 좋다.

리액트 라이브러리는 시스템상의 다양한 함수를 이벤트에 넣는 복잡한 부분을 제거하고, 데이터 검색 및 실시간 처리 작업을 효과적으로 가능하게 한다. 이런 방식은 기본적으로 주식 시세, 소셜 미디어 처리 같은 무한정으로 이뤄지는 이벤트 흐름에 사용된다.

이러한 리액트 프로그래밍은 통계 데이터 및 센서 데이터를 사용하는 데이터 사이언티스트에게 각광받고 있으며, 분석과 활용에 많은 도움을 준다.

진정한 리액트 프로그래밍

리액트 개념에서 비보존형 트랜잭션이 중요한데, 이는 프로그램 실행에 영향을 주는 내부 문제점을 줄여준다.

이러한 프로그래밍 스타일은 함수형 프로그래밍에서 사용되는데, 좀 더 탄력적인 분산 시스템을 구성하는 데 많은 역할을 한다. 따라서 이런 시스템을 개발할 경우 꼭 참고하고 따라야 할 부분이다.

ReactiveX(RX)

먼저 리액트RX, Reactive Extensions는 2010년 넷플릭스Netflix라는 회사가 RxJava를 사용하면서 커지고 널리 알려졌다.

현재 다양한 프로그래밍 언어에 맞춘 리액트 버전이 있으니, 다음을 살펴보자.

- 자바: RxJava
- 자바스크립트: RxJS
- 파이썬: RxPY
- 스위프트: RxSwift

각 언어의 RX 형태는 https://github.com/ReactiveX에서 참고하자.

파이썬의 리액트 형태인 RxPYReactive Extensions for Python는 관찰 가능 컬렉션과 LINQ 스타일 쿼리 오퍼레이터를 사용하는 비동기 및 이벤트 기반 프로그래밍 구성 라이브러리다. 내 경우 앵귤러Angular 프레임워크에서 작업할 때 ReactiveX와 비슷한 버전을 사용했다. 웹 소켓 스트림은 관찰 가능 속성으로 보고, 앵귤러 애플리케이션 및 브라우저에 나타냈다.

마찬가지로 RxPY도 유용한데, 관찰자 및 관찰 가능 속성을 다루는 애플리케이션을 작성할 경우 활용된다.

내가 보기에 이러한 라이브러리를 가장 잘 활용할 수 있는 부분은 주식 거래다. 이론적으로 볼 때 주기적으로 주식 가격을 측정하는 API와 특정 조건일 때 주식을 거래하는 RxPY 기반 형태다. 예컨대, 가격이 20퍼센트 떨어진 주식이 있다고 하자. 이러한 형태의 이벤트를 받고, 주식을 팔거나 다시 사는 형태의 상황으로 리액트한다.

RxPY 설치하기

다음과 같이 pip를 사용해 RxPY를 설치한다.

```
pip install rx
```

RxPY는 PyPy 및 IronPython뿐만 아니라 파이썬 2.7 및 3.4 이상의 버전에서도 동작한다.

관찰 가능 속성

관찰 가능 속성은 RxPY 애플리케이션에서 가장 중요한 부분이다. 이는 현재 등록된 모든 관찰자에게 이벤트를 전달하게 된다. 여기서 중요한 건, 관찰자가 풀 메커니즘pull mechanism과 반대로 새로운 이벤트 구독자를 알고자 푸시 메커니즘push mechanism을 활용한다는 점이다.

관찰자 생성하기

RxPY에서는 거의 대부분이 관찰 가능 속성이 될 수 있는데, 수많은 라이브러리에서 관찰 가능 속성을 소비하는 관점에 있어 굉장히 많은 경우의 수가 있다. 필요시 람다 함수를 간단히 사용할 수 있거나 이를 다루는 클래스를 정의한다.

예제

다음 예제에서는 rx.Observer의 하위 클래스인 PrintObserver를 구현한다. 여기서 on_next(), on_completed(), on_error() 함수를 정의한다. 이 세 함수는 관찰자에서 중요한 역할을 한다.

- on_next(self, value): 관찰자가 새로운 이벤트를 받았을 경우 호출된다.
- on_completed(self): 관찰 가능 속성이 작업을 완료했을 경우 호출된다.
- on_error(self, error): 에러 상황을 다룰 경우 호출된다. 다행히도 간단한 예제에서는 따로 고려할 필요가 없다.

on_next() 함수로부터 호출되어 값을 받아 단순히 출력하는 부분은 비교적 간단하나, 자체 관찰자를 어떻게 정의할지 생각하게 해준다.

주의: 다음 코드는 공식 RxPY 문서에서 참고했다.

간단한 RxPY 관찰자와 관찰 가능 속성을 나타낸 예제 코드를 살펴보자.

```
from rx import Observable, Observer

def push_five_strings(observer):
  observer.on_next("Alpha")
  observer.on_next("Beta")
  observer.on_next("Gamma")
  observer.on_next("Delta")
  observer.on_next("Epsilon")
  observer.on_completed()

class PrintObserver(Observer):

  def on_next(self, value):
```

```
    print("Received {0}".format(value))

  def on_completed(self):
    print("Done!")

  def on_error(self, error):
    print("Error Occurred: {0}".format(error))

source = Observable.create(push_five_strings)

source.subscribe(PrintObserver())
```

위 코드를 실행하면 5개의 문자열이 출력되고 on_completed() 함수가 호출된다. 이는 더 이상 이벤트가 없음을 관찰자에게 알려주며, 프로그램은 종료된다.

```
$ python3.6 07_createObservable.py
Received Alpha
Received Beta
Received Gamma
Received Delta
Received Epsilon
Done!
```

예제 2

다음으로 주식을 사고파는 간단한 주식 거래 시스템을 만들어본다. 주식 가격이 기준과 일치하면 주문 요청을 관찰자에게 전달해 처리하게 한다. 물론 이것으로 백만장자가 되지는 못하지만, 그래도 많은 부분을 배울 수 있을 것이다.

```
from rx import Observable, Observer

stocks = [
{ 'TCKR' : 'APPL', 'PRICE': 200},
```

```
{ 'TCKR' : 'GOOG', 'PRICE': 90},
{ 'TCKR' : 'TSLA', 'PRICE': 120},
{ 'TCKR' : 'MSFT', 'PRICE': 150},
{ 'TCKR' : 'INTL', 'PRICE': 70},
]

def buy_stock_events(observer):
  for stock in stocks:
    if(stock['PRICE'] > 100):
      observer.on_next(stock['TCKR'])
  observer.on_completed()

class StockObserver(Observer):

  def on_next(self, value):
    print("Received Instruction to buy {0}".format(value))

  def on_completed(self):
    print("All Buy Instructions have been received")

  def on_error(self, error):
    print("Error Occurred: {0}".format(error))

source = Observable.create(buy_stock_events)
source.subscribe(StockObserver())
```

코드에 관한 분석

코드를 살펴보면 TCKR로 나타낸 주식 시세 표시와 PRICE로 나타낸 주식 가격을 저장한 딕셔너리 배열을 먼저 정의했다. 물론 이런 부분은 모두 가짜이며 실제 의미는 없다.

다음으로 buy_stock_events 관찰 가능 속성을 정의해 주식 및 가격 배열을 지나면서 주식을 살 수 있는지 확인한다. 주식이 기준과 일치하면 주식의 TCKR을 이벤트의 키로서 전달한다. 모든 주식 확인이 끝나면 on_completed()를 정의해 주식 관련 작업과 더 이상 구매가 없음을 관찰자에 알린다.

마지막으로, StockObserver 클래스 정의에서 Observable 객체를 생성해 buy_stock_events 함수를 인자로 취한다. 그 후 StockObserver 클래스의 새로운 인스턴스를 통해 관찰 가능 속성을 받는다.

출력

컴퓨터에서 이를 실행하면 다섯 번 주식을 살펴보고 100보다 큰지 가격을 비교한다. APPL, TSLA, MSFT 모두 기준에 만족해 구매 지시를 내린다. StockObserver 객체는 해당 전달을 받아 적절히 수행해 순서대로 출력한다.

5개의 주식 모두를 살펴보고 나면 관찰 가능 객체가 폐기되며 StockObserver 객체가 종료된다.

```
$ python3.6 00_observables.py
Received Instruction to buy APPL
Received Instruction to buy TSLA
Received Instruction to buy MSFT
All Buy Instructions have been received
```

람다 함수

이전의 방법은 모든 클래스가 불필요한 상황에서는 다소 장황한 부분이다. 이럴 때는 람다 함수lambda function를 가져다 활용할 수 있다.

람다 함수란 비동기 함수를 생성하게 해주는 함수다. 이런 상황에서 꽤 강력하게 작용하는데, 이전 예제를 다음과 같이 수정할 수 있다.

예제

다음 예제에서는 람다 함수를 활용해 주어진 주식 시장 거래를 어떻게 실행하는지 보여준다.

```
from rx import Observable

stocks = [
{ 'TCKR' : 'APPL', 'PRICE': 200},
{ 'TCKR' : 'GOOG', 'PRICE': 90},
{ 'TCKR' : 'TSLA', 'PRICE': 120},
{ 'TCKR' : 'MSFT', 'PRICE': 150},
{ 'TCKR' : 'INTL', 'PRICE': 70},
]

def buy_stock_events(observer):
  for stock in stocks:
    if(stock['PRICE'] > 100):
      observer.on_next(stock['TCKR'])

observer.on_completed()

source = Observable.create(buy_stock_events)

source.subscribe(lambda value: print("Received Instruction to buy {0}".format(
value)))
```

코드에 관한 분석

코드를 살펴보면 관찰 가능 속성 객체는 이전 예제와 동일한데, 한 가지 주목해야 할 점이 있다. 더 이상 StockObserver 클래스 정의를 복잡하게 할 필요 없이 한 줄로 대체해 모든 함수를 나타낼 수 있다는 것이다.

```
source.subscribe(lambda value: print("Received Instruction to buy {0}".format(
value)))
```

이 특정한 예제에서는 코드를 줄이고 어떤 부분이 잘못됐는지 클래스 정의를 모두 살필 필요가 없다. 하지만 여기서는 이점이긴 하나, 모든 코드를 하나의 람다식에 넣으면 코드 가독성이 떨어지며 유지 관리 및 버그 수정에 애를 먹을 것이다.

람다식에서의 on_next, on_completed, on_error 함수

이전 예제에서 중요한 부분은 코드가 간결해졌지만 on_next, on_completed, on_error 함수가 이전에 다뤄진 형태에 비해 정의하기 쉽지 않아졌다는 점이다. 그렇지만 위의 세 함수를 다음과 같이 람다 함수 형태로 구현할 수 있다.

```
from rx import Observable

stocks = [
{ 'TCKR' : 'APPL', 'PRICE': 200},
{ 'TCKR' : 'GOOG', 'PRICE': 90},
{ 'TCKR' : 'TSLA', 'PRICE': 120},
{ 'TCKR' : 'MSFT', 'PRICE': 150},
{ 'TCKR' : 'INTL', 'PRICE': 70},
]

def buy_stock_events(observer):
  for stock in stocks:
    if(stock['PRICE'] > 100):
       observer.on_next(stock['TCKR'])
  observer.on_completed()

source = Observable.create(buy_stock_events)

source.subscribe(on_next=lambda value:
print("Received Instruction to buy {0}".format(value)), on_completed=lambda:
print("Completed trades"), on_error=lambda e: print(e))
```

출력

위 코드의 source.subscribe 호출에서 구독자가 필요한 3개의 람다 함수를 정의했는데, 적절한 호출에 매핑했다. 추가된 Completed trades 출력 외에는 동일하며, on_completed 함수가 정상적으로 매핑됐음을 알 수 있다.

```
$ python3.6 10_allLambda.py
Received Instruction to buy APPL
Received Instruction to buy TSLA
Received Instruction to buy MSFT
Completed trades
```

오퍼레이터와 연결

오퍼레이터는 이미션emission과 구독자 사이에서 동작한다. 이는 구독자로부터 흡수된 포맷으로 출력된 원 데이터를 변환한다.

전체적으로 RxPY는 130개의 오퍼레이터를 바탕으로 새로운 관찰 가능 속성을 얻으며, 이러한 오퍼레이터를 연결해 강력하게 만들 수 있다. Observable에서 연산하는 대부분의 오퍼레이터는 구독되는 Observable을 반환하게 된다. 이는 바람직한 상황을 구성하고자 여러 오퍼레이터를 연결하기 때문이다.

필터 예제

먼저 길이가 다섯 글자 이상인 문자열의 길이를 출력하도록 간단한 오퍼레이터를 어떻게 함께 연결하는지 살펴보자.

5개의 문자열 배열을 취해, .map() 오퍼레이터를 사용해 길이를 매핑하고 .filter() 오퍼레이터를 사용해 다섯 글자 이하인 문자열을 걸러낸다. 매핑된 구독자를 가지며 길이를 출력하는 간단한 람다 함수를 이용해 필터링한다.

```
from rx import Observable

source = Observable.from_(["Alpha", "Beta", "Gamma", "Delta", "Epsilon"])

lengths = source.map(lambda s: len(s))
```

```
filtered = lengths.filter(lambda i: i >= 5)

filtered.subscribe(lambda value: print("Received {0}".format(value)))
```

코드에 관한 분석

위의 코드 예제에서는 5개의 그리스 알파벳 배열을 전달하는 Observable.from_ 함수를 정의한다.

다음으로 source.map을 활용해 해당 문자열의 길이를 길이 변수로 매핑했고, 문자열의 길이를 변환하는 람다 함수를 제공했다.

그 후 5 이상인 값을 반환하는 lambda 함수를 사용해 길이를 필터링한다.

마지막으로, 각 문자열의 길이를 출력하는 최종 lambda 함수를 사용해 필터링된 이벤트 스트림을 구독한다.

연결된 오퍼레이터

이전에 언급한 연결 메커니즘을 활용해 이 예제를 나타내면, 다음과 같다.

```
from rx import Observable

Observable.from_(["Alpha", "Beta", "Gamma", "Delta", "Epsilon"]) \
  .map(lambda s: len(s)) \
  .filter(lambda i: i >= 5) \
  .subscribe(lambda value: print("Received {0}".format(value)))
```

이 코드는 이전 코드와 함수성에서는 아무런 차이가 없지만, 각 오퍼레이터가 저장되는 출력을 따라갈 필요가 없으므로 가독성에 더 좋다.

다양한 오퍼레이터

이 절에서는 RxPY에서 사용할 수 있는 다양한 오퍼레이터를 살펴본다. 안타깝게도 130개가 넘는 오퍼레이터를 모두 이 절에서 소개할 수는 없기에 몇 개만 알아본다. 좀 더 자세히 알고 싶다면 오퍼레이터 공식 문서를 참고하자(http://reactivex.io/documentation/operators.html). 이 리스트는 다양한 운영체제에 사용 가능한 언어이며, ReactiveX로 작업할 경우 북마크된 자원을 살펴본다.

관찰 가능 속성 생성하기

RxPY에서 관찰 가능 속성 생성과 관련해 다양한 경우의 수가 존재한다. 오퍼레이터를 생성하고자 `Create`, `From`, `Interval` 같은 오퍼레이터를 활용한다.

프로그래밍적으로 관찰자 메소드를 호출하고자 스크래치로부터 `Observable`을 생성하는 `Create` 오퍼레이터가 있다. 반대로 `Defer`는 관찰자가 구독하는 지점에서만 `Observable`을 생성하고 실제로는 각 구독 호출에 독특한 `Observable`을 생성한다.

관찰 가능 속성 변환하기

관찰 가능 속성 변환과 관련해 이를 생성하는 부분보다 더 많은 경우의 수가 있다. 파이썬 프로그램에서 사용 가능한 모든 관찰 가능 속성 변환을 활용하는 중요한 오퍼레이터를 아래에서 살펴보자.

몇 가지 중요한 오퍼레이터를 정의해보자.

- `Buffer`: 이미 생성된 `Observable`로부터 주기적으로 아이템을 모으는 버퍼 역할을 하며 하나로 출력되게 묶는다. 이는 효과적으로 수백만의 출력으로 시스템 흐름을 막을 수 있다.
- `FlatMap`: 이미 `merge_all` 오퍼레이터를 사용해 관찰 가능 속성을 결합하나, `FlatMap` 오퍼레이터는 이러한 관찰 가능 속성을 하나로 병합하는 대안을 제공한다.

- Map: 주어진 관찰 가능 속성에서 출력된 모든 형태를 변환하고 주어진 각 항목에 함수를 적용한다.

그 외에 스캔 및 윈도 오퍼레이터도 존재하지만, 이 정도가 시작하는 데 적절해 보인다.

관찰 가능 속성 필터링하기

RxPY는 관찰 가능 속성을 필터링하는 API를 제공한다. 이전 예제에서 사실상 Filter 오퍼레이터를 사용해봤는데, 내가 추천하는 Distinct, ElementAt, Sample, Take 오퍼레이터도 사용해보길 바란다.

필터링과 관련된 오퍼레이터의 한 부분인 다음 리스트를 살펴보자.

- Distinct: Observable에서 중복된 항목을 출력하지 않게 한다. 중복된 리스트를 필터링해 복잡성을 관리하는 오퍼레이터다.
- Take: Observable로부터 출력된 *n*개의 하위 항목을 취한다.
- ElementAt: Observable로부터 출력된 *n* 위치의 항목을 반환한다.

관찰 가능 속성 에러 다루기

에러를 다루는 것은 모든 프로그래머가 소프트웨어에서 겪어야 할 부분이다. RxPY 기반 프로그램은 예외를 다룰 수 없는데, 다음과 같은 에러 처리 오퍼레이터 2개를 제공한다.

- Catch
- Retry

핫 및 콜드 관찰 가능 속성

ReactiveX에는 다양한 구독자로 관찰 가능 객체를 생성하는 것을 인지하는 핫 및 콜드 관찰 가능 객체 개념이 존재한다.

RxPY 프로그램에 다음과 같은 관찰 가능 속성 중 하나가 있다.

- 핫(hot) 관찰 가능 속성: 구독자와 관계없이 알림을 발행한다.
- 콜드(cold) 관찰 가능 속성: 구독자가 내부에서 응답할 경우 내보낸다.

예컨대, 주식의 새로운 부분을 관찰하는 관찰 가능 속성이 있다고 하자. 현재 해당 관찰 가능 속성에 응답하고 내보내는 구독자가 1개가 있다. 이제 이 관찰 가능 객체에 여러 구독자를 추가해야 한다.

이런 상황에서 관찰 가능 객체는 이미 그 외 구독자의 이벤트가 작동된 부분에 응답하지 않는다. 대신에 이후 내보내게 한다. 이는 핫 관찰 가능 속성의 예시 중 하나다.[1]

관찰 가능 속성이 이미 내보낸 부분에 응답할 수 있다면, 이는 콜드 관찰 가능 객체다.

이벤트 내보내기

RxPY에서는 이벤트 기반 프로그래밍과 유사한 일반적인 이벤트 및 데이터 모두와 작업하는 복합 관찰 가능 객체를 생성할 수 있다.

예제

다음 예제에서 키보드 입력을 바탕으로 데이터를 내보내는 이벤트와 결합해 원하는 이벤트 스트림을 종료해보자.

```
from rx import Observable

Observable.interval(1000) \
  .map(lambda i: "{0} Mississippi".format(i)) \
  .subscribe(lambda s: print(s))

input("Press any key to quit\n")
```

1 즉, 구독하는 시점 이후로 이벤트를 받는다. – 옮긴이

코드에 관한 설명

코드에서 보다시피 1000밀리초마다 Mississippi와 이어져 i의 결과가 출력되는 관찰 가능 객체를 생성한다. 다음으로 어떤 것을 내보내는지 간단히 출력하는 람다 함수 이벤트를 구독한다.

input() 호출로 프로그램 종료를 조절한다. 이는 각 스레드의 Observable.interval() 오퍼레이터 때문인데, 프로그램은 호출 없이 비정상적으로 종료될 수 있다.

출력

프로그램을 실행하면 다음과 같이 출력하고 종료된다.

```
0 Mississippi
1 Mississippi
2 Mississippi
3 Mississippi
4 Mississippi
5 Mississippi
6 Mississippi
...
```

멀티캐스팅

지금까지 살펴본 예제에서는 1개의 관찰 가능 속성 및 관찰자가 있는 경우를 다뤘다. 하지만 RxPY 라이브러리의 강력한 부분은 다양한 구독자로 내보낼 수 있는 멀티캐스팅 multicasting이 가능하다는 것이다.

멀티캐스팅은 여러 구독자가 하나의 관찰 가능 객체를 구독하는 경우에 필요한데, 이는 모든 구독자가 받는 이벤트와 관련해 차이가 생길 수 있다. 하지만 멀티캐스팅을 사용하면 효과적으로 이러한 차이를 없애며, 구독자의 개수와 상관없이 정상적으로 이벤트를 받을 수 있다.

예시로 다음 코드를 살펴보자.

```
from rx import Observable
from random import randint

three_emissions = Observable.range(1, 3)

three_random_ints = three_emissions.map(lambda i: randint(1, 100000))

three_random_ints.subscribe(lambda i: print("Subscriber 1 Received:
{0}".format(i)))
three_random_ints.subscribe(lambda i: print("Subscriber 2 Received:
{0}".format(i)))
```

이 코드는 3개의 무작위 이벤트를 내보내는 간단한 관찰 가능 속성인데, 각 속성은 1부터 100,000까지 범위의 무작위 수를 생성한다. 코드를 실행하면 구독자 1과 구독자 2 모두 동일한 이벤트를 받는다. 하지만 주어진 프로그램의 출력을 살펴보면 예상과는 다르다.

```
$ python3.6 03_multicast.py
Subscriber 1 Received: 21097
Subscriber 1 Received: 19863
Subscriber 1 Received: 68053
Subscriber 2 Received: 69670
Subscriber 2 Received: 11348
Subscriber 2 Received: 8860
```

예제

다음 예제에서 이전 예제를 확장해 살을 붙여보자. Observer 클래스의 하위 클래스인 Subscriber 클래스를 정의한다. 이 클래스에서 on_next( ), on_completed( ), on_error( ) 함수를 구현해 전달된 값을 출력한다.

다음으로, 마찬가지로 이전 예제의 three_random_inits 관찰 가능 객체를 정의해 .publish( ) 함수를 사용해 발행한다.

아래로 내려가 3개의 구독자를 관찰 가능 속성으로 구독하고 .connect() 함수를 호출해 모든 구독자가 동일하게 전달된 스트림을 받도록 준비한다.

```
from rx import Observable, Observer
from random import randint

class Subscriber(Observer):

  def __init__(self, ident):
    self.id = ident

  def on_next(self, value):
    print("Subscriber: {} Received: {}".format(self.id, value))

  def on_completed(self):
    print("Subscriber: {} Received Events".format(self.id))

  def on_error(self, error):
    print("Error Occurred: {}".format(error))

three_emissions = Observable.range(1,3)
three_random_ints = three_emissions.map(lambda i:
randint(1, 10000)).publish()

three_random_ints.subscribe(Subscriber("Grant"))
three_random_ints.subscribe(Subscriber("Barry"))
three_random_ints.subscribe(Subscriber("Sophie"))
three_random_ints.connect()
```

출력

3개의 구독자가 3개의 새로운 이벤트를 받고 마지막에 완료 상태가 됐음을 알린다.

```
$ python3.6 14_multicasting.py
Subscriber: Grant Received: 211
```

```
Subscriber: Barry Received: 211
Subscriber: Sophie Received: 211
Subscriber: Grant Received: 7120
Subscriber: Barry Received: 7120
Subscriber: Sophie Received: 7120
Subscriber: Grant Received: 2802
Subscriber: Barry Received: 2802
Subscriber: Sophie Received: 2802
Subscriber: Grant Received Events
Subscriber: Barry Received Events
Subscriber: Sophie Received Events
```

관찰 가능 속성 연결하기

다른 시간에 데이터를 전송하는 2개의 관찰 가능 속성이 필요할 수 있다. 이런 상황에서 해당 관찰 가능 속성을 연결해 효과적으로 하나의 관찰 가능 속성으로 하여 구독자가 순차적으로 구독할 수 있게 한다.

실시간 뉴스를 기반으로 하는 주식 거래 프로그램을 생각해보면, 새로운 뉴스 기사를 살펴보고 시스템상의 그 밖의 컴포넌트로 이벤트를 넘겨주는 다양한 관찰 가능 속성이 있다. 이런 상황에서 효과적으로 모든 관찰 가능 속성을 통합해 하나의 핵심 뉴스로 연결하며 주식 프로그램을 효율적으로 사용할 수 있다.

실제 관찰 가능 속성을 연결하는 컴포넌트를 구현하는 문제의 경우 RxPY에서 다양한 오퍼레이터를 활용할 수 있다. 다음을 참조하자.

- `and() / then() / when()`
- `combineLatest()`
- `join`
- `merge()`
- `startWith()`

- switch()
- zip()

프로그램 내에서 각 오퍼레이터를 다르게 활용할 수 있으므로 오퍼레이터를 활용하고 이해하는 것이 중요하다.

zip() 예제

다음 예제에서는 .zip() 오퍼레이터를 사용해 리스트 관찰 가능 속성과 intervals 속성을 하나의 관찰 가능 속성으로 연결한다. 다음으로 .subscribe() 오퍼레이터를 사용해 새로 연결된 객체로 관찰자를 구독한다.

```
from rx import Observable

list1 = [23, 38, 43, 23]

letters = Observable.from_(list1).to_blocking()
intervals = Observable.interval(1000)

def main():
  Observable \
    .zip(letters, intervals, lambda x, y: (x*y, y)) \
    .subscribe(lambda t: print(t))

if __name__ == '__main__':
  main()
  input("Press any key to quit\n")
```

출력

코드를 실행하면 다음과 같이 출력된다. 먼저 비정상적으로 종료되는 것을 막는 input 함수 결과를 볼 수 있고, list1 값과 현재 인터벌 값의 곱 그리고 인터벌 값이 차례대로 튜플로 출력된다.

```
$ python3.6 06_combining.py
Press any key to quit
(0, 0)
(38, 1)
(86, 2)
(69, 3)
```

merge_all() 오퍼레이터

관찰 가능 속성을 연결하기 위해서는 merge_all() 오퍼레이터 또한 2개 이상의 속성을 구독할 수 있는 하나의 속성으로 병합할 수 있다.

다음 예제에서 2개의 배열로부터 간단한 관찰 가능 속성 2개를 정의한다. 다음으로 sources라는 관찰 가능 속성 배열을 생성하고, 이어서 merge_all() 오퍼레이터를 바탕으로 새로 생성된 배열을 기존의 두 속성 모두에 병합해 연결한다.

```
from rx import Observable

list1 = [23, 38, 43, 23]
list2 = [1,2,3,4]
source1 = Observable.from_(list1)
source2 = Observable.from_(list2)
sources = [source1, source2]

def main():
  Observable.from_(sources) \
    .merge_all() \
    .subscribe(lambda x: print(x))

if __name__ == '__main__':
  main()
  input("Press any key to quit\n")
```

출력

출력을 살펴보면 기존의 관찰 가능 객체 모두 다방면으로 결합된 것을 볼 수 있다.

```
$ python3.6 11_mergeAll.py
23
38
1
43
2
23
3
4
Press any key to quit
```

동시성

이 장에서 다루는 대다수의 코드는 단일 프로세스에 초점이 맞춰져 있는데, 동시성 프로그래밍이 RxPY에서 다음 두 오퍼레이터를 바탕으로 지원하고 있다.

- `subscribe_on()`
- `observe_on()`

두 오퍼레이터 모두 재사용 가능한 작업자 스레드 풀을 생성하고자 `ThreadPoolScheduler` 스케줄러를 제공한다.

예제

마찬가지로 공식 문서에서 제공하는 라이브러리 코드를 활용해 RxPY에서 동시성을 구현해보자.

다음 코드에서 필수 크로스 스레드 스케줄러로 re.concurrency에서 ThreadPoolScheduler 클래스를 사용한다. 그 후 여러 인터벌을 가진 이벤트를 전달하는 3개의 관찰 가능 속성을 생성한다.

```
import multiprocessing
import random
import time
from threading import current_thread

from rx import Observable
from rx.concurrency import ThreadPoolScheduler

def processHeavyCalc(value):
  time.sleep(random.randint(5,20) * .1)
  return value

# CPU의 개수를 계산하고 스레드 개수만큼 ThreadPoolScheduler를 생성한다.
optimal_thread_count = multiprocessing.cpu_count()
pool_scheduler = ThreadPoolScheduler(optimal_thread_count)

# Process 1 생성
Observable.from_(["Alpha", "Beta", "Gamma", "Delta", "Epsilon"]) \
  .map(lambda s: processHeavyCalc(s)) \
  .subscribe_on(pool_scheduler) \
  .subscribe(on_next=lambda s: print("PROCESS 1:
{0} {1}".format(current_thread().name, s)),
  on_error=lambda e: print(e),
  on_completed=lambda: print("PROCESS 1 done!"))

# Process 2 생성
Observable.range(1, 10) \
  .map(lambda s: processHeavyCalc(s)) \
  .subscribe_on(pool_scheduler) \
  .subscribe(on_next=lambda i: print("PROCESS 2:
{0} {1}".format(current_thread().name, i)),
  on_error=lambda e: print(e),
  on_completed=lambda: print("PROCESS 2 done!"))
```

```
# Process 3 생성(무한)
Observable.interval(1000) \
  .map(lambda i: i * 100) \
  .observe_on(pool_scheduler) \
  .map(lambda s: processHeavyCalc(s)) \
  .subscribe(on_next=lambda i: print("PROCESS 3:
{0} {1}".format(current_thread().name, i)),
  on_error=lambda e: print(e))

input("Press any key to exit\n")
```

출력

코드를 실행하면 정상적으로 동시성 스레드가 실행된 것을 볼 수 있다. 맨 앞에 종료를 막기 위한 "Press any key to exit"를 볼 수 있다.

그 후 3개의 프로세스가 동시에 실행되고 일정한 출력문이 완전히 얽혀 동기적으로 실행되지 않는다.

```
$ python3.6 09_concurrency.py
Press any key to exit
PROCESS 1:<concurrent.futures.thread.ThreadPoolExecutor object at
0x102abda20>_0 Alpha
PROCESS 2: <concurrent.futures.thread.ThreadPoolExecutor object at
0x102abda20>_1 1
PROCESS 1: <concurrent.futures.thread.ThreadPoolExecutor object at
0x102abda20>_0 Beta
PROCESS 3: <concurrent.futures.thread.ThreadPoolExecutor object at
0x102abda20>_2 0
PROCESS 2: <concurrent.futures.thread.ThreadPoolExecutor object at
0x102abda20>_1 2
PROCESS 3: <concurrent.futures.thread.ThreadPoolExecutor object at
0x102abda20>_3 100
PROCESS 1: <concurrent.futures.thread.ThreadPoolExecutor object at
0x102abda20>_0 Gamma
PROCESS 2: <concurrent.futures.thread.ThreadPoolExecutor object at
```

```
0x102abda20>_1 3
PROCESS 3: <concurrent.futures.thread.ThreadPoolExecutor object at
0x102abda20>_3 200
PROCESS 2: <concurrent.futures.thread.ThreadPoolExecutor object at
0x102abda20>_1 4
PROCESS 1: <concurrent.futures.thread.ThreadPoolExecutor object at
0x102abda20>_0 Delta
...
```

PyFunctional

PyFunctional은 리액트 프로그램과 비슷한 개념을 따르므로 지금 이 시점에서는 그다지 중요하지 않다. 이는 파이썬 언어를 사용해 함수형 프로그램을 만들게 해준다.

함수형 및 리액트 프로그래밍 모두 에러나 추가적인 상황을 저장할 필요 없는 순수한 함수를 활용하는 경향이 있다. 프로그램에서 일어나는 부분과 상관없이 동일한 결과를 반환하는 함수를 정의하게 된다. PyFunctional과 RxPY의 가장 큰 차이점은 데이터 흐름은 동일한 형태이나, 다뤄지는 데이터가 상이하다는 것이다. RxPY는 시스템에서 다뤄지는 데이터 및 이후의 이벤트에 초점이 있다. PyFunctional은 함수형 프로그래밍 개념을 사용해 데이터의 변환을 주로 사용한다.

PyFunctional은 함수형 오퍼레이터가 연결된 데이터 파이프라인을 쉽게 생성할 수 있게 한다. 이 라이브러리는 Ph.D 학생인 페드로 로드리게즈Pedro Rodriguez가 스파크Spark[2]와 스칼라Scala[3]에 착안해 만들었으며, 데이터 스트림을 쉽게 다루고자 오퍼레이터가 연결된 LINQ 스타일을 사용해 코드를 작성한다.

2 빅데이터에 사용되는 클러스터 컴퓨팅 시스템 – 옮긴이

3 자바를 기반으로 한 객체지향 및 함수형 프로그래밍 언어 – 옮긴이

설치와 공식 문서

다음과 같이 pip로 PyFunctional을 설치한다.

```
pip install pyfunctional
```

다음 예제에서 functional 모듈을 불러와 사용해본다.

PyFunctional 모듈에 대해 알고 싶다면, 깃허브 저장소(https://github.com/EntilZha/PyFunctional)를 참조하거나 공식 사이트(http://www.pyfunctional.org/)를 살펴보자.

간단한 예제

이제 PyFunctional 모듈을 사용해 함수형 프로그램을 어떻게 작성하는지 예제를 바탕으로 살펴보자. 이는 PyFunctional 모듈을 얼마나 잘 사용하는지 기본적인 사항을 확인해주는 예제다.

다음 예제에서는 스트림 객체의 반복 및 조작을 수행하는 seq를 활용한다. 먼저 각 값을 두 배로 하는 람다 함수를 사용해 연속된 수를 매핑한다. 다음으로 x가 4보다 큰 값을 필터링하고, 나머지 값의 합으로 연속된 수를 합쳐본다.

```
from functional import seq

result = seq(1, 2, 3, 4)\
  .map(lambda x: x * 2)\
  .filter(lambda x: x > 4)\
  .reduce(lambda x, y: x + y)
print("Results: {}".format(result))
```

출력

필터링 및 연속적으로 매핑된 수를 합쳐서 14라는 값이 출력된 것을 볼 수 있다.

```
$ python3.6 12_pyFunctional.py
Results: 14
```

스트림, 변환, 액션

PyFunctional에는 API를 구분할 수 있는 세 가지 형태가 있다.

- 스트림
- 변환
- 액션

해당 라이브러리에서 각자의 역할을 하게 된다. 스트림은 조작 및 활용하고자 하는 데이터를 읽어들인다. 변환 항목은 스트림된 데이터를 변환할 수 있으며, 액션은 이렇게 변환된 시리즈를 구체적인 값으로 계산한다.

seq( )를 호출해 인자에 데이터를 전달하면 데이터는 스트림으로 변환되며, 변환 및 조작을 할 수 있게 된다.

필터링 리스트

PyFunctional에서는 필터링과 관련된 다양한 오퍼레이터를 제공하는데, 스트림은 원하는 결과 형태로 변환하는 데 사용할 수 있다.

이 예제에서는 주식 거래 배열을 필터링하는 작업을 살펴보고, 해당 배열의 결과를 매핑하고 주어진 거래의 전체 가격을 알기 위해 배열을 더해본다.

Stock 튜플을 정의하기 위해서는 collections 패키지의 namedtuple 모듈을 활용한다. 해당 튜플에는 주식 시세 표시기와 거래 가격으로 구성된 배열을 정의한다.

```
from functional import seq
from collections import namedtuple

Stock = namedtuple('Stock', 'tckr price')
stocks = [
  Stock('AMZN', 100),
  Stock('FACE', 200),
  Stock('JPM', 80),
  Stock('TSLA', 500),
  Stock('TSLA', 450)
]

costs = seq(stocks)\
  .filter(lambda x: x.tckr == 'TSLA')\
  .map(lambda x: x.price)\
  .sum()

print("Total cost of TSLA transactions: {}".format(costs))
```

출력

코드를 실행하면 모든 주식 중 TSLA로 정상적으로 필터링된 것을 볼 수 있다. 다음으로 필터링된 리스트를 매핑해 합을 구하고 거래의 전체 가격을 콘솔에 출력한다. 다음과 같이 출력된다.

```
$ python3.6 13_pyfilter.py
Total cost of TSLA transactions: 950
```

SQLite3 읽기 및 쓰기

PyFunctional 라이브러리는 SQLite3 작업을 제공하며, 다양한 오퍼레이터를 바탕으로 DBS 쿼리 및 복잡한 작업을 다소 쉽게 해준다.

예컨대, sqlite3 데이터베이스를 쿼리할 때 seq.sqlite3(db, query).to_list()를 활용해 데이터베이스의 수를 쿼리하고 리스트로 변환할 수 있다. 코드 작성을 쉽게 할 뿐만 아니라 간결하고 가독성 높은 코드를 만들 수 있다.

공식 문서에서 제공하는 예제는 데이터베이스의 모든 사용자를 쿼리한다. 데이터베이스 경로는 sqlite3 데이터베이스로 전달하고 모든 사용자를 반환하고자 select * from user를 호출한다. to_list() 오퍼레이터는 파이썬 기본 리스트로 전달된 SQL 쿼리로부터 반환된 행을 변환한다.[4]

```
db_path = 'examples/users.db'
users = seq.sqlite3(db_path, 'select * from user').to_list()
# [(1, 'Tom'), (2, 'Jack'), (3, 'Jane'), (4, 'Stephan')]]

sorted_users = seq.sqlite3(db_path, 'select * from user order by name').to_list()
# [(2, 'Jack'), (3, 'Jane'), (4, 'Stephan'), (1, 'Tom')]
```

sqlite3 내의 데이터베이스로 코드를 작성하는 부분에는 다양한 방법이 있다.

```
import sqlite3
from collections import namedtuple

with sqlite3.connect(':memory:') as conn:
  conn.execute('CREATE TABLE user (id INT, name TEXT)')
  conn.commit()
  User = namedtuple('User', 'id name')

  # 쿼리를 통해 작성
```

4 https://github.com/EntilZha/PyFunctional 참조 – 옮긴이

```
  seq([(1, 'pedro'), (2, 'fritz')]).to_sqlite3(conn, 'INSERT INTO user (id, name)
VALUES (?, ?)')

  # 튜플/리스트로부터 값을 네임드 테이블로 삽입해 작성한다.
  seq([(3, 'sam'), (4, 'stan')]).to_sqlite3(conn, 'user')

  # 네임드 튜플로부터 스키마를 추론해 작성한다.
  seq([User(name='tom', id=5), User(name='keiga', id=6)]).to_sqlite3(conn, 'user')

  # 딕셔너리로부터 스키마를 추론해 작성한다.
  seq([dict(name='david', id=7), dict(name='jordan', id=8)]).to_sqlite3(conn, 'user')

  # 올바르게 작성됐는지 처음부터 확인한다.
  print(list(conn.execute('SELECT * FROM user')))

  # [(1, 'pedro'), (2, 'fritz'), (3, 'sam'), (4, 'stan'), (5, 'tom'), (6, 'keiga'),
(7, 'david'), (8, 'jordan')]
```

지금까지 사용자 테이블로부터 리스트를 작성하는 여러 가지 방법을 알아봤다.

압축된 파일

PyFunctional은 gzip, lzma/xz, bz2 같은 압축 파일을 쉽게 다룰 수 있게 제공한다. 여기서 자동으로 파일이 압축됐는지 확인하고 일반적인 파일과 동일하게 작업할 수 있다.

압축 데이터의 작업에는 연산이 필요하므로 프로그램 속도에 영향을 미친다. 하지만 내 경험상 그렇게 차이가 있지 않으니 걱정 말자.

이러한 파일을 재작성할 때는 아래 함수에 압축 인자를 지정해야 한다.

- gzip 압축에는 gzip 또는 gz
- lzma 압축에는 lzma 또는 xz
- bz2 압축에는 bz2

이 파라미터는 PyFunctional의 to_ 함수에 있으며, 이를 조합해 자동으로 확인하고 압축 파일을 쉽게 다룰 수 있다.

병렬 실행

파이썬 프로그램을 기반으로 한 PyFunctional에서 병렬 실행을 위해서는 코드에서 딱 한 가지, 즉 seq를 pseq로 변경해야 한다.

그 외 현재 병렬 실행을 지원하는 세 가지 오퍼레이터가 있는지 참고하자.

- map/select
- filter/filter_not/where
- flat_map

프로그램을 수정해 코드를 단위 프로세스 형태로 바꾼 후 multiprocessing 모듈을 활용하면 복잡한 연산 작업의 성능 향상을 기대할 수 있을 것이다.

요약

10장에서는 리액트 프로그래밍의 핵심 원리를 다뤘다. 리액트 프로그래밍과 일반적인 이벤트 기반 프로그램의 차이점을 살펴보고, RxPY 파이썬 라이브러리에 대해 자세히 살펴봤다. 그리고 관찰자, 관찰 가능 속성, 이벤트 전달하기, 다양한 구독자로 멀티캐스팅하기 등을 알아봤다.

지금까지 리액트 프로그래밍의 전반적인 사항을 살펴봤는데, 이제 본인만의 리액트 시스템을 만들어볼 차례다. 파이썬을 사용해 함수형 프로그램을 어떻게 구성하는지 PyFunctional의 다양한 오퍼레이터를 활용해 다뤄봤다.

11장에서는 데이터 분석과 빅데이터 연구 작업을 수행하기 위해 그래픽 카드의 연산 능력을 활용해 성능을 향상하는 방법을 알아본다. 병렬적으로 작동하는 수천 개의 코어를 활용하는 방법과 Numba와 PyOpenCL 같은 라이브러리를 활용해 성능 향상에 도달하는 부분도 배워본다.

11

GPU 사용하기

11장에서는 파이썬 프로그램의 성능 향상을 위해 GPU를 활용하는 방법을 살펴본다. GPU가 무엇이며, 파이썬 프로그램에서 어떻게 활용해 이점을 얻는지 알아본다.

다음으로 세분화된 부분을 자세히 살펴볼 필요 없이 일반적인 프로그램에서 GPU를 활용할 수 있는 파이썬 래퍼wrapper를 살펴본다.

PyCUDA 라이브러리는 C, C++ 같은 복잡한 언어를 알지 못해도 고성능 애플리케이션을 생성할 수 있게 해준다.

11장에서는 GPU 프로그래밍에서 다양하게 사용되는 라이브러리를 소개한다. 이러한 라이브러리를 시작하고 GPU 및 APU 하드웨어에서 어떻게 라이브러리의 개념이 사용되는지 배워본다. 지금부터 다룰 라이브러리는 다음과 같다.

- PyCUDA
- PyOpenCL
- Numba
- Theano

그 전에 앞서 GPU가 기존의 CPU에 비해 어떤 부분이 더 좋은지 살펴보자.

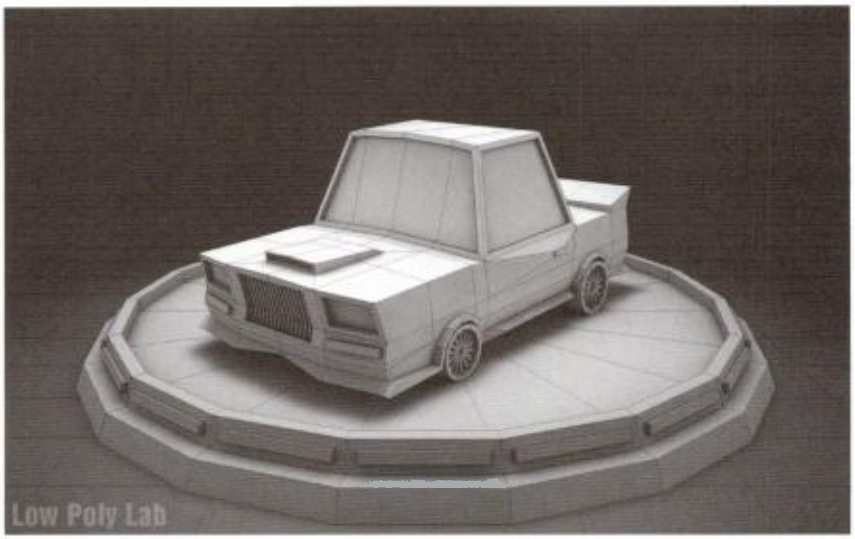
GPU 소개

그래픽 처리 장치GPU, Graphics Processing Unit는 일반적으로 고성능 게이밍 시장에 초점이 맞춰져 있다. 보통 고성능 3D 비디오 게임을 원활하게 하고자 좋은 그래픽 카드를 구비하는 경우가 많다.

비디오 게임은 3D 객체를 랜더링할 때 분당 수백만 번의 연산이 필요하다. 게임상의 장면들은 간단한 3D 객체부터 수천 개의 복잡한 모델까지 다양한 형태로 이뤄져 있다. 각 프레임에서는 비교적 정확한 위치, 크기, 회전 및 정상적으로 렌더링되기 위한 다양한 요소들이 결정된다.

비교적 간단한 모델일지라도 수백 또는 수천 개의 정점vertex이 있다. 다음 페이지의 자동차 모델링 사진을 보면 전체적으로 간단한 다각형으로 이뤄졌지만, 휠의 경우 200~300개 이상의 정점으로 구성돼 있다.

이런 객체를 업데이트하면서 나타내려면 행렬을 변환하고 회전함으로써 각 정점을 곱해야 한다. 한 장면에 건물, 캐릭터 같은 수많은 객체가 있다고 하면 변환해야 할 정점의 개수는 훨씬 많다. 초당 최소 24~30프레임을 곱한다는 것은 웬만한 컴퓨터에서 쉽지 않은 작업이다.



▲ 출처: https://www.lowpolylab.net/

왜 GPU를 사용하는가?

초당 수많은 행렬 연산 같은 작업은 일반적인 CPU에게는 너무 과도하다. 원래 이러한 방식은 병렬 처리 및 고성능 계산을 다루도록 디자인되지 않았다. 그렇기에 수백만의 연산을 담당할 독립적인 많은 코어가 있는 하드웨어가 나오게 됐다.

2장에서 그래픽 카드가 따르는 SIMD 아키텍처 스타일에 대해 다뤘다. 이러한 작업에 SIMD 스타일이 얼마나 유용한지 살펴봤는데, 데이터 사이언스 및 머신 러닝에 사용되는 부분은 다루지 않았다.

GPU는 이러한 고성능 그래픽 연산에 특화됐는데, 통계 분석, 데이터 마이닝, 암호 해독 같은 작업에도 사용된다. 이 장에서는 그래픽 카드를 이용한 다양한 라이브러리를 소개하고, 이를 활용하는 여러 방법도 배워본다.

데이터 사이언스

앞으로는 많은 데이터 사이언티스트가 필요해질 텐데, 패턴 분석 및 주식 거래의 핵심적인 특징을 찾아내는 금융 분야에서 특히 그럴 것이다.

전 세계의 많은 회사가 축적해온 수많은 데이터를 처리하고 모델에 적용하고자 거대한 연산 능력을 갖춘 조직과 학문이 필요해진 것이다.

파이썬은 비교적 간단한 표현으로 인해 이런 분야에서 일찍이 자리 잡아온 언어 중 하나다.

데이터 사이언스의 종류

데이터 사이언스 분야는 다양하게 나뉘는데, 각기 다른 영역에서 각광받으며 활발히 연구 중이다. 주요 분야는 다음과 같다.

- 머신 러닝machine learning
- 분류classification
- 클러스터 분석cluster analysis
- 데이터 마이닝data mining

이어지는 절들에서 하나씩 살펴보자.

머신 러닝

일반적으로 지도 및 비지도 학습으로 분류되는데, 머신 러닝 알고리즘은 최적의 결과를 내고자 가능한 한 많은 학습 데이터를 살펴본다. 데이터를 중심으로 알고리즘이 실행되면서 이후의 데이터 경향을 예측하고자 학습 후 남은 데이터를 사용한다.

분류

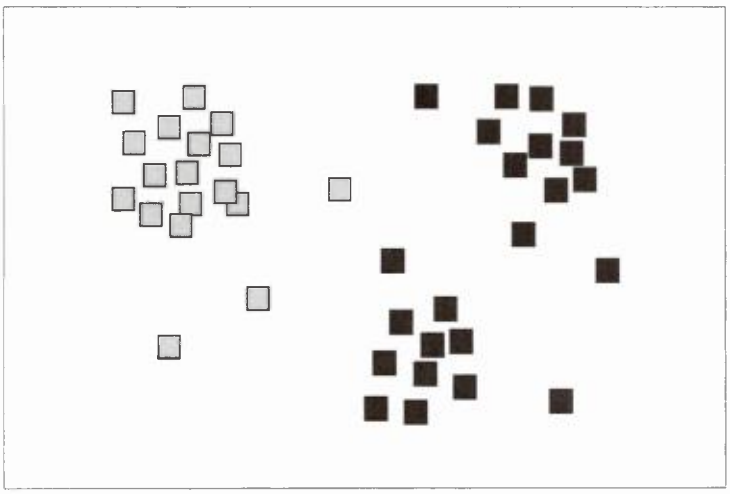
분류는 머신 러닝의 일종인데, 카테고리별로 요소를 개별적으로 분류하는 것을 말한다. 머신 러닝도 이미 분류된 샘플 데이터 세트 자체를 학습하는 동일한 과정을 거친다. 그

후 샘플 데이터로부터 학습된 사항을 기반으로 하여 해당하는 카테고리로 요소를 분류하게 된다.

클러스터 분석

클러스터 분석이란 동일한 특성을 지닌 클러스터로 각 데이터 세트가 그룹화하는 것을 말한다. 주로 검색 엔진의 정보 검색에 사용되는데, 특정 단어를 검색할 때 그룹화된 웹 페이지를 나타내고자 응집 클러스터링 같은 알고리즘이 사용된다.

다음 그림은 다양한 데이터 형태를 보여준다. 응집 클러스터링 같은 기술을 이용해 세 부분의 카테고리로 데이터를 그룹화할 수 있다. 각 데이터 점들이 검색된 웹 페이지라고 생각해보자. 그리고 노란색은 고양이, 파란색은 개, 빨간색은 거북이에 관한 페이지라고 해보자. 응집 클러스터링은 이러한 데이터를 효과적으로 묶어 나타낸다.

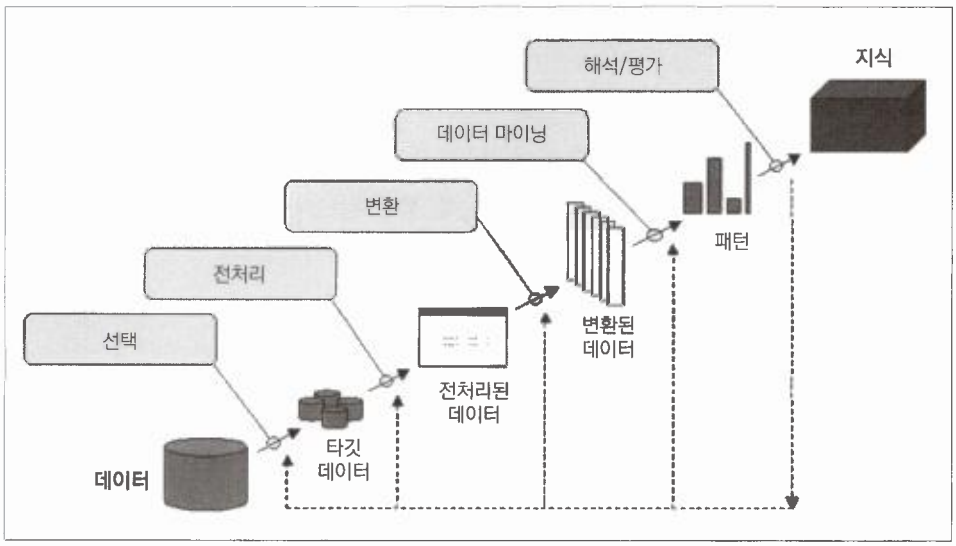


▲ 출처: https://en.wikipedia.org/wiki/Cluster_analysis#agglomerative_clustering

데이터 마이닝

데이터 마이닝은 수많은 데이터 세트에서 필요한 정보를 추출하는 방법을 말한다. 일반적으로 다음 다섯 과정을 거친다.

1. 시험하고자 하는 데이터를 확인한다.
2. 데이터를 전처리한다.
3. 데이터를 변환한다.
4. 데이터를 마이닝한다.
5. 결과를 해석 및 평가한다.



▲ 출처: http://www2.cs.uregina.ca/~dbd/cs831/notes/kdd/1_kdd.html

이 주제와 관련해서는 버나드 마Bernard Marr가 쓴 『Big Data』(교학사, 2016)를 참조하자. 이 책은 파이썬 라이브러리를 사용해 이러한 과정을 빠짐없이 다루고 있다.

CUDA

지금까지 CUDA를 사용하는 데 필요한 라이브러리를 살펴봤는데, CUDA가 정확히 무슨 뜻이며 실제 사용과는 어떤 관련이 있는지 알아보자.

CUDA는 엔비디아NVIDIA사에서 개발된 병렬 컴퓨팅 플랫폼 및 API이다. 이는 GPU에서

제공하는 강력한 병렬화를 일반적인 프로그래밍 형태로 간단하게 활용할 수 있도록 디자인됐다.

GPU를 활용하고자 CUDA에서 제공하는 다양한 API를 사용해 쉽게 파이썬 프로그램을 제작할 수 있다. 이런 API는 코드를 적절히 분리해 많은 연산이 요구되는 부분은 따로 다루면서 성능을 향상할 수 있다.

엔비디아 그래픽 카드 없이 CUDA로 작업하기

궁극적으로 CUDA로 작업하고 PyCUDA 같은 라이브러리를 사용하기 위해서는 적절한 하드웨어가 있어야 한다. 다행히도 해당 라이브러리를 활용하기 위한 문제를 해결할 방법이 있는데, 안타깝게도 무료는 아니다.

그래도 가장 빠른 해결책은 현재 서비스 중인 클라우드를 이용하는 것이다. 예컨대, AWS에서는 엔비디아 하드웨어로 구성된 서버로 작업할 수 있으며 이용한 시간만큼 사용료를 지불하면 된다. 시작하기에 비교적 적절한 방법인데, AWS 인프라를 구축하고 학습하기 위해서는 어느 정도 시간 투자가 필요하다.

두 번째로 가장 확실한 방법은 엔비디아 기반의 그래픽 카드를 컴퓨터에 장착하는 것이다. 고성능인 데다가 최신의 그래픽 카드 구매 시 많은 지출이 일어나므로 적절한 구성이 필요하다.

단기적으로는 전자의 구성이 비용이 적게 들겠지만, 내 경험상 GPU 없이 많은 시간을 클라우드로 작업하면 그래픽 카드 값보다 오히려 더 많은 비용이 지출됐다.

PyCUDA

PyCUDA는 파이썬 개발자에게 엔비디아의 CUDA 병렬 컴퓨팅을 활용하도록 한 파이썬 라이브러리다.

PyCUDA의 공식 문서를 참조하자. 여기에는 공식 소스 코드와 메일링 리스트가 있다.
https://mathema.tician.de/software/pycuda/

이 절에서는 엔비디아 GPU가 필요한 복잡한 연산 프로그램에서 PyCUDA를 어떻게 활용하는지 간단한 예제를 통해 살펴본다.

특징

PyCUDA는 다양한 특징이 있으므로 살펴보자.

- 객체 정리는 객체가 올려져 있는 시간을 조정해 문제없이 코드가 작성되고 이후의 실행에 충돌되지 않게 한다.
- 이전에 언급했듯이, PyCUDA의 가장 큰 장점은 `pycuda.compiler.SourceModule`과 `pycuda.gpuarray.GPUArray`로 대부분의 API를 사용할 수 있다는 것이다.
- PyCUDA는 자동으로 파이썬 예외 처리된 모든 에러를 변환하므로 에러 확인 코드를 작성할 필요가 없다.
- 성능을 높이기 위한 C++ 라이브러리다.

이 책의 한계상 라이브러리의 모든 개념에 관한 예제를 다룰 수는 없다. 독자 스스로 샘플 코드를 참조해 연습해보길 바란다.

간단한 예제

PyCUDA의 공식 홈페이지에 나온 예제 코드를 살펴보자. 이 코드를 통해 PyCUDA 생태계에 대한 간단한 소개와 PyCUDA에 관한 기본적인 사항을 배울 수 있다.

코드 상단에서 PyCUDA 및 NumPy 모듈에 필요한 부분을 불러온다.

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
  drv.Out(dest), drv.In(a), drv.In(b),
  block=(400,1,1), grid=(1,1))

print dest-a*b
```

커널

커널은 그래픽 프로그래밍 및 언어에 사용되는 개념이다. 커널 함수는 CPU 코드에서 호출되는 GPU 함수의 일종이다. 따라서 이론적으로 파이썬 프로그램은 CPU보다 위에서 실행되며, GPU 작업을 커널 함수의 형태로 본다.

이런 커널 함수는 파이썬 프로그램에서 다음과 같다.

```
mod = SourceModule("""
  __global__ void doublify(float *a)
```

```
{
  int idx = threadIdx.x + threadIdx.y*4;
  a[idx] *= 2;
}
""")
```

OpenCL 커널 프로그래밍을 자세히 다룬 『Open CL 프로그래밍The OpenCL Programming Book』(Fixstars 저, 한빛미디어, 2012)을 참고하자. 영문판은 https://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/first-opencl-program/에서 온라인으로 볼 수 있다.

PyCUDA 및 커널을 이용해 작업하고자 한다면, 커널 프로그래밍을 자세히 배워보길 추천한다.

GPU 배열

GPU 배열은 PyCUDA 라이브러리에서 중요한 부분이다. GPU로 작업하는 복잡한 부분을 줄여주고 대신에 numpy.ndarray와 비슷하게 작업할 수 있게 해준다. 클래스 정의는 다음과 같다.

```
class pycuda.gpuarray.GPUArray(shape, dtype, *, allocator=None, order="C")
```

다음 코드에서 gpuarray 인스턴스를 어떻게 초기화하는지 살펴본다. gpuarray.to_gpu()를 호출해 gpuarray 인스턴스를 초기화한다. 다음으로 gpuarray에 무작위로 채워진 값을 세 배한 a.tripled를 정의한다. 마지막으로, 콘솔에 결과를 출력한다.

```
import numpy
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
```

```
a.gpu = gpuarray.to_gpu(numpy.random.randn(2,2).astype(numpy.float64))
a.tripled = (3*a_gpu).get()
print(a.tripled)
```

이 책의 한계상 공식 문서의 내용을 모두 담지 못했다. 다음을 꼭 참조하자.
https://documen.tician.de/pycuda/array.html#

Numba

연속적 분석의 한 형태인 Numba 파이썬 컴파일러는 높을 병렬성을 바탕으로 한 강력한 성능의 인터프리터 언어다.

공식 pydata 웹사이트 문서는 Numba에 대한 소개와 파이썬 프로그램에서 활용할 수 있는 방법을 제공한다. http://numba.pydata.org/#을 참고하자.

이 절에서는 아나콘다Anaconda 형태의 Numba 환경을 살펴본다. 빅데이터를 효과적이고 효율적으로 분석하기 위해 주변의 다양한 Numba 패키지를 활용하는 방법 또한 알아본다. Numba의 기초적인 부분과 GPU 및 APU를 프로그램에서 활용하는 복잡한 측면도 직접 해본다.

개관

Numba는 LLVM 컴파일러 인프라를 사용해 파이썬 코드를 최적화된 기계어 코드로 생성한다. 현재 코드에서 조금만 수정하면 프로그램 성능에 큰 차이를 가져다줄 수 있다.

Numba의 특징

Numba는 개발자들이 좋아할 만한 큰 특징 세 가지가 있다.

- 다양한 코드 생성
- CPU(기본) 및 GPU에 네이티브 코드 생성
- 파이썬 과학 소프트웨어와의 통합

파이썬 2 및 파이썬 3를 다양한 운영체제에 크로스 지원하기에, Numba 사용은 파이썬 버전과 운영체제 간을 넘나드는 시간을 줄인다.

LLVM[1]

LLVM은 개발자들이 모듈형, 재사용 컴파일러, 툴체인 기술로 구성한 프로젝트 모니커moniker다. 연구 프로젝트로부터 시작됐는데, 컴파일 기술에 관심이 많은 사람들의 이목을 끌게 되자 기존 셸은 더 이상 사용할 수 없게 됐다.

이는 최적화 로우 레벨 및 중간 코드, 바이너리를 생성하는 데 초점을 맞췄다. C++로 주로 작성됐지만, 다음과 같이 다양한 언어와 프로젝트를 바탕으로 한다.

- 에이다Ada
- 포트란Fortran
- 파이썬Python
- 루비Ruby

컴파일러 구성에 관심이 많다면 다음 기사를 참고하자.

https://gnuu.org/2009/09/18/writing-your-own-toy-compiler/

1 현재 LLVM은 하나의 프로젝트로 그 의미가 더 넓어져 이용되고 있다. 현존하는 최고 인기의 오픈소스 기반 컴파일러이므로 그 역할과 역사에 대해 배워볼 필요가 있다. 『LLVM Cookbook』(메이유르 판디 저, 에이콘출판, 2017)을 참고하자. – 옮긴이

하드웨어 간 호환성

이 장에서는 GPU에서 일반적인 프로그래밍을 다루지만, Numba는 CPU 및 GPU 하드웨어 모두에서 동작하는 파이썬 컴파일러를 지원한다. 이 절의 예제는 엔비디아 외에도 다양한 GPU에서 실행할 수 있다.

파이썬 컴파일러

Numba 라이브러리를 자세히 살펴보기에 앞서, CPython 프로그램과 Numba 파이썬 프로그램이 어떻게 컴파일되고 실행되는지 차이점을 살펴봐야 한다. 다음 비교표를 살펴보자.

	AOT(Ahead Of Time)	JIT(Just In Time)
CPython/libpython 기반	Cython Shedskin Nuitka(최근 버전) Pythran **Numba**	**Numba** HOPE Theano Pyjion
CPython/libpython 대체	Nuitka(이전 버전)	Pyston PyPy

▲ 파이썬 컴파일러

JIT와 AOT 컴파일러

이러한 두 종류의 컴파일러는 많은 사람에게 익숙하지 않은데, 이번 기회에 두 종류의 차이점과 언제 어떻게 사용되지 확실히 배워보자.

JIT Just-In-Time 컴파일러는 작성된 파이썬 코드의 인터프리터 오버헤드를 제거해 프로그램의 속도를 높여준다. 인터프리팅된 코드의 오버헤드를 제거하면 C, C++ 같은 컴파일 언어의 성능과 거의 동일해진다. Numba는 이러한 JIT 컴파일러를 채택하는데, 인터프리터 오버헤드를 제거해주지만 C 언어의 성능 관점에서 코드 성능을 완전히 높여주지는 않고 단지 도와줄 뿐이다.

AOT 컴파일러는 함수가 온디스크on-disk 바이너리 객체로 컴파일되어 독립적으로 분산 실행된다. 일반적으로 C, C++, 포트란이 채택하는 기술이다. 여기서는 인터프리팅된 코드가 필요 없으며, 컴퓨터는 바이너리 코드를 컴파일하고 동시에 실행하는 것이 아닌, 이미 구성된 바이너리 코드를 실행하는 작업만 수행한다.

Numba 프로세스

Numba 문서를 먼저 살펴보면, 파이썬 코드의 성능을 높이는 가장 쉬운 방법은 코드의 모든 함수에 `@jit` 데코레이터를 추가하는 것이다. 하지만 이런 경우는 실제로 거의 없으며, Numba 라이브러리의 이점을 알기 힘들다.

연속적 분석으로 과학적 소프트웨어를 개발하는 스탠리 세이버트Stanley Seibert는 Numba를 효과적으로 사용하기 위한 다섯 가지 지침을 발표했다. 다음을 살펴보자.

1. 실질적인 벤치마크 테스트를 시도해본다. 시스템 함수가 얼마만큼 작동되는지 측정하는데, 기본 단위 테스트 라이브러리는 아니다.
2. 프로파일러를 활용하고 벤치마크에서 실행한다. 가장 좋은 경우는 6장에서 다룬 cProfile 툴이다.
3. 코드상에서 실행 시간이 좀 더 요구되는 핫스팟을 확인한다.
4. 중요한 함수에 `@numba.jit` 및 `@number.vectorize` 데코레이터를 활용한다.
5. 벤치마크를 재실행하고 프로그램의 성능을 실제로 향상하고자 어떤 부분을 결정할지 결과를 분석한다.

실제 강연에 관한 부분은 https://www.youtube.com/watch?v=eYIPEDnp5C4를 꼭 참고하길 바란다.

아나콘다

Numba를 시작하기 앞서 연속적 분석 웹사이트에서 아나콘다Anaconda 패키지를 설치해야 한다.

https://www.anaconda.com/download/

아나콘다는 파이썬 데이터 사이언스 부분에 있어 중요하다. 오픈소스이며, 파이썬 및 R 언어 모두 고성능의 분산 형태를 띤다. R 언어에 대한 부분은 개인적으로 알아보길 바란다. R 언어는 데이터 사이언티스트들이 많이 채택하는 언어이고, 많은 데이터 세트를 분석할 때 사용된다.

아니콘다는 자체 패키지 및 의존성 매니저인 Conda를 제공한다. 1000개의 데이터 사이언스 특화 패키지가 있으니 차근차근 알아보자.[2]

기본적인 Numba 파이썬 프로그램 작성하기

지금까지 컴파일러 부분에 있어 Numba를 어떻게 다룰지 기본적인 개념을 살펴봤는데, 파이썬 프로그램에 이러한 부분을 적용해보자.

직접 파이썬 프로그램을 작성해본다. 먼저 전달된 두 값을 단순히 더해 반환하는 함수를 작성하자.

```
def f(x, y):
  return x + y

f(2,3)
```

여기에 numba의 jit 데코레이터를 추가하고, 다음과 같이 데코레이터에 새로 정의된 덧셈 함수를 데코레이팅한다.

```
from numba import jit

@jit
def f(x, y):
```

2 지금부터는 아나콘다 프롬프트를 사용해 파이썬을 실행해본다. 기본적인 명령어는 모두 동일하며, 파이썬 버전이 다를 경우 https://conda.io/docs/user-guide/tasks/manage-python.html#installing-a-different-version-of-python을 참고하자. – 옮긴이

```
  return x + y

f(2,3)
```

데코레이터는 해당 함수가 컴파일될 것임을 의미한다.

컴파일링 옵션

@jit 데코레이터는 암묵적으로 컴파일러에게 어떻게 함수를 컴파일할지 다양한 키워드를 제공한다. 이러한 옵션을 바탕으로 GIL로부터 벗어나고 특정 컴파일러 모드를 사용할 수 있다.

nopython

Numba는 2개의 컴파일링 모드가 있는데 nopython과 object 모드다. Numba 프로그램에는 암묵적으로 코드를 컴파일하고자 하는 모드를 선택해 해당 모드에서는 불가능한 에러를 발생시킬 수 있다.

nopython 모드는 파이썬 C API에 직접적으로 접근 못 하는 코드를 생성한다. 즉, 이렇게 생성된 코드는 매우 빠르다는 뜻이다.

object 모드는 기본적으로 성능 측면에서 일반적인 인터프리팅된 프로그램과 비슷한 코드를 생성한다.

```
@jit(nopython=True)
def func(x, y):
  return x + y
```

nogil

Numba에서 함수를 nopython 모드식으로 컴파일할 수 있다면 nogil 플래그를 활용한다. 해당 함수에 진입하면 GIL을 피할 수 있다.

이 책의 여러 장에서 특정 상황의 GIL 문제를 다루면서 세세하게 이를 살펴봤다. GIL이 발생하면 해당 함수는 그 외 파이썬에서 실행 중인 스레드로 동시에 실행하고, 다음 코드로 상당한 성능 향상을 볼 수 있다.

```
@jit(nogil=True)
def func(x, y):
  return x + y
```

cache 옵션

cache 옵션을 설정하면 Numba가 온디스크on-disk 캐시 파일로 특정 함수에 컴파일된 코드를 저장하게 한다. 즉, 함수가 한 번 컴파일되면 실행할 때마다 컴파일할 필요가 없다.

```
@jit(cache=True)
def func(x, y):
  return x + y
```

parallel 옵션

parallel 옵션은 병렬 의미인 특정 함수의 작업을 자동적으로 병렬화하기 위한 함수다.

```
@jit(nopython=True, parallel=True)
def func(x, y):
  return x + y
```


이러한 부분은 nopython 옵션과 함께 사용해야 한다.

Numba의 문제점

Numba를 사용해 소프트웨어 시스템을 구축할 때는 Numba의 문제점을 알고 있어야 한다. 어떤 형태인지 추론하는 것이 항상 가능하진 않은데, 어디서 문제가 생겼는지 궁금할 것이다.

문제 해결과 팁에서 제공하는 예제를 살펴보자. `@jit(nopython=True)` 데코레이터로 나타낸 함수를 보자. 이 함수는 `x`, `y` 인자를 취하고 두 인자의 합을 반환한다.

```
@jit(nopython=True)
def f(x, y):
  return x + y
```

`f(2, 3)` 형태로 2와 3을 넣으면, Numba는 5가 반환됨을 볼 수 있다. 즉, 형태 추론이 가능한데, `f(1, (2,))`라고 하면 어떤 일이 발생할까? 두 값은 형태가 서로 다르며 Numba에서 추론이 안 되고 다음과 같은 에러가 나타난다.

```
>>> f(1, (2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    [...]
  File "/home/antoine/numba/numba/typeinfer.py", line 242, in resolve
    raise TypingError(msg, loc=self.loc)
numba.typeinfer.TypingError: Failed at nopython frontend
Undeclared +(int64, (int32 x 1))
File "<stdin>", line 2
```

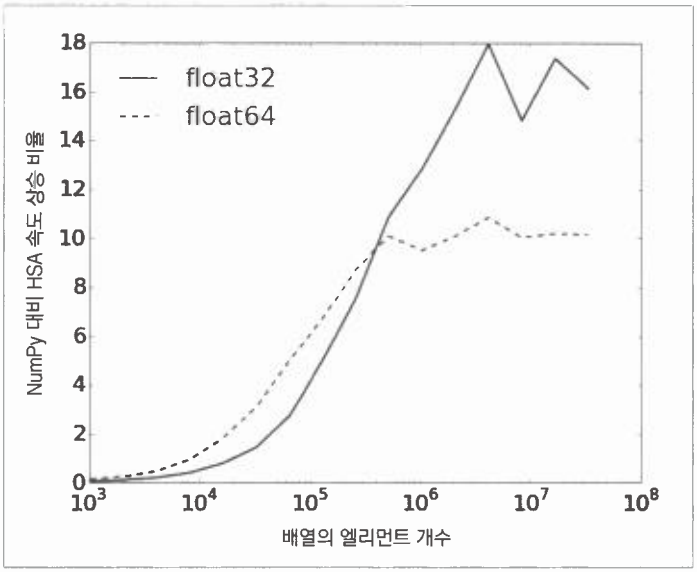
CUDA 기반 GPU에서의 Numba

지금까지 Numba의 기초적인 부분을 살펴봤는데, GPU를 사용해 어떻게 번역되는지 살펴보자.

AMD APU에서의 Numba

Numba 버전 0.21에서는 이기종 시스템 아키텍처HSA, Heterogeneous System Architecture를 지원한다. HSA는 주어진 공유 메모리 공간을 바탕으로 CPU와 GPU를 하나의 연산장치로 활용하는 기술이다.

해당 기술에 대해 더 알고 싶다면 https://www.anaconda.com/developer-blog/programming-amd-apus-numba/를 참조하자. 성능과 관련해 APU의 연산 개념을 알아보기 위해 다음 그래프를 살펴보자. 여기서 각 프로세스의 엘리먼트 개수가 증가할수록, NumPy에서의 성능 대비 APU에서 실행되는 프로그램의 속도 상승 정도가 증가함을 볼 수 있다.



▲ 출처: https://www.continuum.io/blog/developer/programming-amd-apus-numba

배열의 엘리먼트 개수가 선형적으로 증가할수록 HSA를 바탕으로 성능이 11x 마크까지 기하급수적으로 증가함을 볼 수 있는데, 이후 선형적으로 계속되고 15~18x 증가에서 마무리된다.

Accelerate

아나콘다의 Accelerate 패키지는 엔비디아의 GPU와 인텔 CPU의 성능을 활용하게 해주는 패키지다. 문서에서 볼 수 있듯이 아나콘다의 애드온add-on 패키지다.

- cuBLAS, cuFFT, cuSPARSE, cuRAND, CUDA Sorting 같은 CUDA 라이브러리를 통합한다.
- 인텔의 Math Kernel Library를 사용해 NumPy, SciPy, scikit-learn, NumExpr에서 속도가 향상된 선형대수 작업을 가능케 한다.
- NumPy의 빌트인 UFuncs의 속도 향상 변형 형태다.
- NumPy에서 고속 푸리에 변환FFT, Fast Fourier Transformations 속도를 높인다.

여타 라이브러리와 마찬가지로, 수많은 데이터 세트를 이용하고자 디자인됐다.

아나콘다의 Accelerate 공식 문서는 https://docs.anaconda.com/accelerate/#에서 볼 수 있다.

Theano

딥 러닝 및 머신 러닝에 대해 관심 있다면, Theano를 사용해보길 바란다. Theano는 NumPy의 `ndarray` 같은 다차원 배열을 작업하는 데 적합하고 좋은 성능을 낸다. 일반적인 CPU상의 C 언어를 몇 배 능가하는 성능을 제공하고자 GPU를 사용한다.

필요사항

Theano는 파이썬 2.7 및 파이썬 3.3부터 3.6까지 지원한다. NumPy와 SciPy 또한 필요하다. 전체적인 필요사항은 다음 공식 문서를 확인해보자.

http://deeplearning.net/software/theano/requirements.html

시작하기

Theano는 매우 쉽게 시작할 수 있어 이 장에서 재밌게 배울 수 있다. 라이브러리의 공식 문서는 좋은 예제를 제공해 파이썬 애플리케이션을 작성할 수 있게 한다.

예제

다음 예제에서는 두 수를 취하고 이를 더하는 함수를 정의해본다. 간단하지만 Theano 라이브러리를 소개하는 예제를 살펴보자.

```
from theano import *
import numpy
import theano.tensor as T

x = T.dscalar('x')
y = T.dscalar('y')
z = x + y
f = function([x,y],z)
print(f(2,3))
```

코드를 실행하면, 보다시피 5.0이 콘솔에 출력된다.

```
$ python3.6 02_theano.py
5.0
```

두 행렬 더하기

이제 여기에 두 행렬을 더하는 함수를 정의한다. 이전 예제와 마찬가지로 동일한 형태를 따르며, dscalar 대신 dmatrix를 사용한다.

```
from theano import *
import numpy
```

```
import theano.tensor as T

x = T.dmatrix('x')
y = T.dmatrix('y')
z = x + y
f = function([x,y],z)

print(f(\
[[2,3],[4,5]],\
[[2,3],[4,5]]\
))
```

코드를 실행하면 Theano가 정상적으로 두 행렬을 더한 것을 볼 수 있다. 그 후 콘솔에 다음과 같이 출력된다.

```
$ python3.6 03_matrices.py
[[ 4. 6.]
[ 8. 10.]]
```

다양한 생성자

Theano 라이브러리에는 광범위한 배열인 TensorType이 있다. 이전에 dmatrix와 dscalar 메소드를 다뤘는데, 그 밖의 다양한 옵션이 있다. 이렇게 효과적으로 타입 시스템을 활용하게 된다. 타입별 생성자의 전체 리스트는 다음 링크를 참조하자.

http://deeplearning.net/software/theano/library/tensor/basic.html#all-fully-typed-constructors

GPU에서 Theano 사용하기

Theano는 계산 모델링을 개발자들에게 알리고자 디자인됐다. 전체 코드를 변경하지 않고 하드웨어의 성능을 이용해 간단한 코드를 작성하게 한다.

현재 GPU를 사용해 프로그램을 작성하도록 2개의 메소드를 제공한다. 엔비디아 그래픽 카드와 OpenCL 디바이스를 활용하게 하고, 나머지는 엔비디아 그래픽 카드만 지원한다.

GPU를 사용하기 위해서는 다음과 같이 CUDA를 디바이스 플래그로 설정한다.

```
$ THEANO_FLAGS='device=cuda' python3.6 04_gpu.py
```

대신에 더 넓은 의미의 device=gpu로 설정 가능하다.

```
$ THEANO_FLAGS='device=gpu' python3.6 04_gpu.py
```

예제

Theano 문서에서 제공하는 예제를 활용해 GPU를 정상적으로 활용 가능한지 살펴보자.

```
from theano import function, config, shared, tensor
import numpy
import time

vlen = 10 * 30 * 768  # 10 x 코어 x 코어당 스레드
iters = 1000

rng = numpy.random.RandomState(22)

x = shared(numpy.asarray(rng.rand(vlen), config.floatX))

f = function([], tensor.exp(x))
print(f.maker.fgraph.toposort())
t0 = time.time()
for i in range(iters):
  r = f()
t1 = time.time()
print("Looping %d times took %f seconds" % (iters, t1 - t0))
```

```
print("Result is %s" % (r,))

if numpy.any([isinstance(x.op, tensor.Elemwise) and
              ('Gpu' not in type(x.op).__name__)
               for x in f.maker.fgraph.toposort()]):
  print('Used the cpu')
else:
  print('Used the gpu')
```

코드를 실행하면 다음과 같이 콘솔에 출력된다.

```
$ python3.6 04_gpu.py
[Elemwise{exp,no_inplace}(<TensorType(float64, vector)>)]
Looping 1000 times took 2.136130 seconds
Result is [ 1.23178032 1.61879341 1.52278065 ..., 2.20771815 2.29967753
1.62323285]
Used the cpu
```

GPU가 아닌 CPU를 사용해 출력한다. THEANO_FLAGS=device=cuda0로 설정해 GPU를 활용하면 다음과 같이 출력된다.

```
THEANO_FLAGS=device=cuda0 python3.6 04_gpu.py
Mapped name None to device cuda0: GeForce GTX 680 (cuDNN version 5004)
[GpuElemwise{exp,no_inplace}(<GpuArrayType<None>(float64, (False,))>),
HostFromGpu(gpuarray)(GpuElemwise{exp,no_inplace}.0)]
Looping 1000 times took 1.202734 seconds
Result is [ 1.23178032 1.61879341 1.52278065 ..., 2.20771815 2.29967753
1.62323285]
Used the gpu
```

CPU와 GPU 모두 정상적으로 출력됨을 볼 수 있다.

멀티 GPU 활용하기

시스템에 2개의 고성능 그래픽 카드가 구성돼 있다면, 병렬적으로 해당 GPU를 모두 활용한 theano 라이브러리를 사용해볼 수 있다. 해당 라이브러리에서 실험적인 부분이니 프로그램 결과가 다를 수 있음을 주의하자.

공식 문서를 참고하자.
http://deeplearning.net/software/theano/tutorial/using_multi_gpu.html

컨텍스트 맵 정의하기

멀티 GPU 활용과 관련해 해당 디바이스를 매핑해 작업을 선정한다. 이러한 매핑은 다양한 디바이스로 구성됐으며, 컨텍스트 이름은 '->'와 cuda0 같은 디바이스 이름으로 구성된다.

```
$ THEANO_FLAGS="contexts=dev0->cuda0;dev1->cuda1" python -c 'import theano'
Mapped name dev0 to device cuda0: GeForce GTX TITAN X
Mapped name dev1 to device cuda1: GeForce GTX TITAN X
```

간단한 그래프 예제

공식 문서의 간단한 예제에서 병렬적으로 그래픽 카드를 활용하는 방법을 살펴보자.

```
import numpy
import theano

v01 = theano.shared(numpy.random.random((1024, 1024)).astype('float32'),
                    target='dev0')
v02 = theano.shared(numpy.random.random((1024, 1024)).astype('float32'),
                    target='dev0')
```

```
v11 = theano.shared(numpy.random.random((1024, 1024)).astype('float32'),
                    target='dev1')
v12 = theano.shared(numpy.random.random((1024, 1024)).astype('float32'),
                    target='dev1')

f = theano.function([], [theano.tensor.dot(v01, v02),
                         theano.tensor.dot(v11, v12)])

f()
```

2개의 디바이스에서 실행하면 선형적으로 속도가 증가함을 볼 수 있다. 여러 개의 그래픽 카드가 없다면 멀티 컨텍스트를 매핑해 하나의 디바이스에서 실행해보자.

다음 코드를 사용하면 된다.

```
THEANO_FLAGS="contexts=dev0->cuda0;dev1->cuda0" python myApp.py
```

PyOpenCL

개방형 컴퓨팅 언어OpenCL, Open Computing Language는 CUDA 기반의 GPU에서 동작하는 이기종 컴퓨팅heterogeneous computing 로우 레벨 API이다. PyOpenCL은 CPU, GPU, DSP, FPGA 같은 다양한 플랫폼을 활용해 파이썬 애플리케이션을 작성하게 하는 API 구현 형태다.

PyOpenCL 라이브러리 공식 문서는 다음을 참고하자.[3]
https://documen.tician.de/pyopencl/

3 동일하게 아나콘다에서 설치해서 사용해본다. - 옮긴이

예제

공식 문서의 예제를 자세히 살펴보자. 이후에 더 나아갈 수 있는 좋은 예제이니 참고하자.

먼저 NumPy, PyOpenCL을 np, cl로 불러와 맨 위에 위치시킨다. 다음으로 numpy.float32 타입의 두 무작위 수를 생성한다. 다음으로 파라미터 호출 없이 PyOpenCL 프로그램 내의 컨텍스트를 생성한다. 이 함수는 프로그램이 실행되는 플랫폼 및 디바이스를 선택해 빠르게 실행하게 한다.

컨텍스트를 생성하면 새로 생성된 컨텍스트를 취하는 명령어 큐를 생성한다.

다음으로 컨텍스트에 빈 디바이스 문자열과 OpenCL 커널 형태의 바이너리를 취하는 Program 객체를 정의한다.

```
from __future__ import absolute_import, print_function
import numpy as np
import pyopencl as cl

a_np = np.random.rand(50000).astype(np.float32)
b_np = np.random.rand(50000).astype(np.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a_np)
b_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b_np)

prg = cl.Program(ctx, """
__kernel void sum(
  __global const float *a_g, __global const float *b_g, __global float
*res_g)
{
  int gid = get_global_id(0);
  res_g[gid] = a_g[gid] + b_g[gid];
}
""").build()
```

```
res_g = cl.Buffer(ctx, mf.WRITE_ONLY, a_np.nbytes)
prg.sum(queue, a_np.shape, None, a_g, b_g, res_g)

res_np = np.empty_like(a_np)
cl.enqueue_copy(queue, res_np, res_g)

# Numpy로 CPU 확인하기
print(res_np - (a_np + b_np))
print(np.linalg.norm(res_np - (a_np + b_np)))
```

출력

내 맥북 프로의 통합 그래픽에서 실행했다. 먼저 어떤 플랫폼에서 코드를 실행할지 묻고, 반대로 해당 플랫폼의 어떤 디바이스에서 실행할지도 묻는다.

```
$ python3.6 06_example.py
Choose platform:
[0] <pyopencl.Platform 'Apple' at 0x7fff0000>
Choice [0]:0
Choose device(s):
[0] <pyopencl.Device 'Intel(R) Core(TM) i5-4288U CPU @ 2.60GHz' on 'Apple'
at 0xffffffff>
[1] <pyopencl.Device 'Iris' on 'Apple' at 0x1024500>
Choice, comma-separated [0]:1
Set the environment variable PYOPENCL_CTX='0:1' to avoid being asked again.
[ 0. 0. 0. ..., 0. 0. 0.]
0.0
```

요약

지금까지 살펴본 내용을 떠올려보자. GPU가 무엇인지, 그리고 어떻게 일반적인 작업에 활용할 수 있는지 살펴봤다. 데이터 사이언티스트가 일반적으로 직면하는 문제와 GPU 래퍼 라이브러리를 활용해 해결하는 부분을 알아봤다.

다음으로 오늘날 그래픽 처리 장치의 성능을 활용할 수 있는 라이브러리를 소개했다. 데이터 사이언티스트나 그 밖의 사람들을 위해 GPU와 APU 기반 애플리케이션을 어떻게 작성하는지 생각해봤다.

이 책의 마지막 장에서는 지금까지 다룬 여러 기술을 되짚어보고 중요한 부분을 요약해 알아본다.

12

솔루션 선택하기

마지막 장에서는 이 책에서 다루지 못한 라이브러리를 살펴본다. 지금 소개하는 라이브러리를 다룬 책들이 많이 있을 테지만, 이 책의 독자들을 위해 12장을 할애해봤다.

효과적으로 라이브러리와 프로그래밍 개념을 파이썬에 활용하고자 따라야 할 과정을 간단하게 살펴본다. 학습에 도움이 될 만한 책이나 자료도 소개한다.

책에서 다루지 못한 라이브러리

이 책을 집필할 당시 다양한 라이브러리를 찾아보고 연구했다. 하지만 한계상 각 장에 꼭 필요한 라이브러리만 소개했다.

현재 사용 가능한 많은 라이브러리를 좀 더 자세히 소개하지 못한 점이 안타깝다. 파이썬을 활용한 소프트웨어 공학 지식을 확장할 수 있는 여러 자료를 익히는 일도 중요하다고 본다.

GPU

GPU는 이 책에서 자세히 다루지 못한 주제 중 하나다. 애플리케이션 성능을 높이기 위해 GPU를 활용하는 방법은 다양한데, 다음 내용을 알아보자.

PyGPU

PyGPU를 살펴보자. PyGPU는 이미지 프로세싱에 특화된 라이브러리이며, 기타 라이브러리와 마찬가지로 높은 속도를 자랑하고 로우 레벨 GPU API를 다룰 수 있게 한다.

고차원 함수, 이터레이터, 리스트 및 GPU 알고리즘 구성에 특화된 형태와 같이 파이썬 코어 특징을 나타내고 있는 임베디드 언어다.

PyGPU에 관한 공식 문서는 다음을 참조하자.
http://fileadmin.cs.lth.se/cs/Personal/Calle_Lejdfors/pygpu/

이벤트 기반과 리액트 라이브러리

이벤트 기반과 리액트 프로그래밍 개념을 이전에 재밌게 배웠다. 어려운 문제를 푸는 직관적인 방법을 제공하고, asyncio와 RxPY를 사용해 샘플 프로젝트를 구성해봤다. 그 외에도 여기에 활용할 수 있는 라이브러리가 있으니 확인해보자.

Tornado

Tornado는 파이썬 웹 프레임워크이며 비동기 네트워킹 라이브러리다. 이는 논블로킹 네트워크 I/O를 활용하고 수천 개의 네트워크 연결을 관리할 수 있다.

Tornado에 관한 공식 문서는 다음을 참조하자.
http://www.tornadoweb.org/en/stable/

RESTful API를 바탕으로 간단하게 'Hello World'를 출력하는 코드를 작성해보자.

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
  def get(self):
    self.write("Hello, world")

def make_app():
  return tornado.web.Application([
    (r"/", MainHandler), ])

if __name__ == "__main__":
  app = make_app()
  app.listen(8888)
  tornado.ioloop.IOLoop.current().start()
```

Flask

Flask도 Tornado와 같은 형태다. 하지만 더 작은 프레임워크다. 프레임워크를 구성하고 실행하기가 더 쉽고, 다섯 줄만으로도 Tornado와 동일한 결과를 낼 수 있다.

다음 예제 코드를 통해 Flask 애플리케이션을 작성해보자.

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
  return "Hello World!"
```

Flask에 관한 공식 문서는 다음을 참조하자.
http://flask.pocoo.org/

Celery

Celery는 분산 메시지 패싱을 기반으로 한 비동기 작업 및 큐/잡 형태의 큐이며, Tornado 및 Flask 같은 웹 프레임워크와 밀접한 관련이 있다. 하지만 때론 필수적인 소프트웨어 시스템에 복잡한 레이어를 추가하는 브로커 기술 구성이 필요하다.

이러한 메시지 브로커는 마이크로 서비스 간 통신을 가능케 하는 중간 중단기다. 기업 시스템 같은 대규모 작업에 사용되므로 필요하다면 참고 자료를 살펴보자.

Celery는 현재 4개의 브로커와 호환된다.

- RabbitMQ
- Redis
- Amazon SQS
- Zookeeper(호환 진행 중)

Celey 프로젝트에 관한 공식 문서는 다음을 참조하자.
http://www.celeryproject.org/

데이터 사이언스

나는 대학에 들어간 후 데이터 사이언스에 관심을 가졌는데, 특히 데이터 검색과 구글 같은 시스템에서 어떻게 최적의 결과를 도출하는지 궁금했다. 이런 부분을 공부해본 적이 없다면, 꼭 한번 알아보자.

Pandas

Pandas는 대규모 데이터 분석 프로젝터에 사용되는 고성능 오픈소스 라이브러리다. 데이터 분석과 모델링보다는 준비에 초점이 맞춰져 있다. 일반적으로 R 프로그래밍 언어가 풀 수 있는 부분을 다룬다.

데이터 사이언티스트에게 있어 Pandas는 데이터 사이언스 작업에 도움을 주는 툴이다. Pandas의 공식 문서는 다음을 참조하자.

http://pandas.pydata.org/

Matplotlib

Matplotlib는 분석하고자 하는 데이터를 좋은 수준의 2D 시각화로 제공한다. 특히나 그래프로 작업하는 데 어려움을 없애주어 실제 문제에 집중하게 한다.

Matplotlib에 관한 공식 문서는 다음을 참조하자.

https://matplotlib.org/

TensorFlow

TensorFlow는 데이터 플로우 그래프를 사용해 수치 계산을 수행하는 오픈소스 라이브러리다. 데이터 플로우 그래프DFG, Data-Flow Graph는 여러 연산 간의 데이터 의존성을 나타낸다(캘리포니아 대학교 브루어Brewer 교수).

전반적으로 TensorFlow는 딥러닝과 신경망을 이용한 복잡한 AI를 구성하는 프레임워크다. 원래 2015년 11월 구글에서 오픈소스로 개발했으며, 이후 성공해 구글 본사가 위치한 캘리포니아주 마운틴 뷰에서 컨퍼런스를 개최했다.

TensorFlow를 더 자세히 알아보고 싶다면, 다음 링크의 '시작하기' 가이드를 살펴보길 바란다. https://www.tensorflow.org/get_started/get_started

시스템 디자인하기

지금까지 이 책에서는 다양한 문제에 적합한 개념들을 살펴봤다. 이 절에서는 언제 어떻게 각 개념을 제대로 사용할지 알아본다.

하지만 권장사항일 뿐, 결국 묘책은 없다. 힌 개인에게 작용하는 사항이 다른 모두에게 적합할 수는 없다. 기업 시스템을 디자인한다면, 코드를 작성하기 전에 가능한 한 많은 생각과 연구를 해보자.

어떤 문제는 다방면의 여러 솔루션이 필요하다. 소프트웨어 아키텍처는 많은 시간과 노력이 필요한 분야이므로, 능수능란해지면 본인의 연봉이 바뀔 것이다.

요구사항

작업하기에 앞서 상당한 시간을 프로젝트의 핵심 부분에 필요한 사항을 모으는 데 쏟아붓는다. 시간과 노력의 비율은 핵심 부분의 양과 비례한다. 프로젝트가 개인용이라면 쉽게 즐길 수 있으나, 많은 사람을 대상으로 한다면 여러 요구사항을 모아야 한다.

요구사항을 모으는 데는 두 가지 종류가 있다. 함수형 요구사항 및 비함수형 요구사항인데, 둘 다 요구사항을 충분히 이해하고 상당한 시간을 소비해야 한다.

함수형 요구사항

함수형 요구사항은 시스템이 해야 할 부분을 정의한다.

예컨대, 주식 시장 거래와 돈을 벌어들이는 프로젝트를 생각해보자. 이 시스템에는 다음과 같은 함수형 요구사항이 필요하다.

- 5초마다 주식 가격을 확인하고 주식 거래가 있으면 실행한다.
- 주식 거래자와 주식을 사고팔 수 있어야 한다.
- 준수사항을 따르며 모든 거래를 살펴볼 수 있어야 한다.

비함수형 요구사항

비함수형 요구사항은 시스템이 어떻게 결정할지에 대한 요구사항이다.

이전에 이어서 시스템의 비함수형 요구사항을 살펴보자.

- 성능: 무거운 시스템 연산을 유지하기 위한 성능이 필요하다.
- 시스템 에러를 다룰 수 있어야 한다. 에러로 인해 회사 시스템에 차질이 있으면 안 된다. https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/을 참조하자.
- 확장성: 정상적인 거래 알고리즘으로부터 수백 개로 확장할 수 있어야 한다.
- 문서화: 기업용 시스템의 경우 전부는 아니지만 어느 정도는 문서화해야 한다.

디자인

함수형 및 비함수형을 비롯한 최종 요구사항이 끝났으면, 소프트웨어 구조를 디자인한다.

최종 솔루션 디자인과 관련해 어떤 것이 있고 가능한지 연구해보자. 내 경우에는 프로젝트를 시작하기 전에 여러 라이브러리 개념을 개념증명PoC하고 이를 적용했다.

한 가지 살펴볼 점은 라이선스 문제다. 목적에 맞는 완성된 제품을 팔아야 한다는 사실을 염두에 두자. 상용 라이선스 라이브러리를 사용하고 있다면, 사용료를 지불하면서 해결

해야 한다. 하지만 이 책에서 살펴본 대부분의 라이브러리는 라이선스와 관련해서는 아무 문제가 없으니 걱정 말자.

복잡한 연산

시스템에서 스레드와 프로세스를 사용할 때의 차이점과 I/O 한계 및 CPU 한계를 살펴봤다. 하지만 다양한 상황에서 어떤 라이브러리를 적용할지가 중요하다.

복잡한 연산이 필요한 애플리케이션에서는 다음과 같은 라이브러리나 모듈을 사용한다.

- 멀티프로세싱 모듈: 비교적 작은 데이터 세트의 빠른 연산에 필요하다. 사용하기 전에 멀티프로세스 사용에 있어 계산이 보증되는지 확인한다.
- PyCUDA 및 Theano 라이브러리: 대규모 데이터 세트의 빠른 연산에 필요하다. 주로 빅데이터에서 사용된다.

다량 이벤트 애플리케이션

해당 애플리케이션이 사용자 입력 및 프로그래밍적으로 생성된 이벤트를 처리한다면, 다음 라이브러리를 사용해보자.

- Asyncio: 일반적인 이벤트 기반 프로그램에 적합하다.
- Twisted: 네트워킹 이벤트 기반 시스템(웹 서버)에 적합하다.
- RxPY: 데이터 기반 이벤트를 다루는 데 적합하다.

다량 I/O 애플리케이션

해당 애플리케이션이 다량의 I/O 기반 태스크를 처리한다면, 일반적인 스레딩 모듈, 즉 `concurrent.futures` 모듈을 활용해보자.

- `threading` 모듈(3장부터 살펴봄)
- `concurrent.futures` 모듈(7장의 `ThreadPoolExecutor`에서 살펴봄)

concurrent.futures 모듈을 사용하길 권장하는데, 해당 API가 멀티스레드를 사용하는 데 있어 복잡함을 줄여주면서 이러한 프로젝트에 적합하기 때문이다.

디자인과 관련된 책

이 책에서 다양한 파이썬 라이브러리를 다뤘지만, 특정 라이브러리를 언제, 어디서 사용할지 자세히 배우진 못했다. 소프트웨어 디자인 분야를 깊이 있게 다룬 좋은 책들이 현재 시중에 많이 나와 있다. 여기서 내가 가장 좋다고 보는 책 두 권을 소개한다.

Software Architecture with Python

「Software Architecture with Python」(Anand Balachandran 저, Packt, 2017)은 백과사전 형태로 다양하고 중요한 주제를 다룬 책이다. 이 책에서 다루는 내용은 다음과 같다.

- 수정이 쉽고 가독성 좋은 코드 작성하기
- 테스트 용이성
- 확장 가능한 코드 작성하기
- 파이썬 디자인 패턴

그 외에도 소프트웨어 공학과 관련해 생각해볼 다양한 부분을 다룬다.

Python: Master the Art of Design Patterns

「Python: Master the Art of Design Patterns」(Phillips 저, Packt, 2017)는 객체지향 파이썬 프로그래밍과 소프트웨어 시스템에 적용되는 다양한 디자인 패턴 등의 주제를 다루고 있다.

소프트웨어 디자인의 전반적인 부분을 아우르는 좋은 책이며, 이전 책에 비해 디자인 패턴을 더 자세히 다루면서 보완했다.

연구

마지막 장에서 강조하는 가장 중요한 것은 바로 디자인의 필요성이며, 이로 인해 시간과 고통을 줄일 수 있다.

거대한 소프트웨어 프로젝트를 디자인하고 연구하는 데 있어, 르네상스 시대의 과학자 레오나르도 다 빈치Leonardo Da Vinci(1452~1519)의 명언을 새겨보자.

> "단순함이란 궁극의 정교함이다."

특정 문제를 해결하는 소프트웨어 시스템을 작성하고자 단순한 방법으로 주어진 문제를 해결할 수 있어야 한다. 이는 다른 개발자들이 속한 팀보다 프로젝트를 더 쉽고 훌륭하게 이끌고 나가게 한다.

요약

안타깝게도 이 책과 함께한 긴 여정은 여기까지다.

이 책에서는 스레드와 프로세스, 그리고 이를 응용한 이벤트 기반 프로그래밍, 리액트 프로그래밍 같은 개념을 다뤄봤다.

성능적인 측면에서 부하를 다루는 비동기 시스템을 구성하고 디자인하는 방법도 배워봤다. 문제에 다양하게 접근해보고, 이 방법이 왜 다른 방법보다 훌륭한지 깨달았을 것이다.

궁금한 사항이 있으면 편하게 트위터(@elliot_f)로 연락하거나 이메일(elliot@elliotforbes.co.uk)을 보내길 바란다. 도움을 주는 것은 언제나 환영이며, 이 책에 대한 피드백이나 궁금한 사항이 있으면 연락하길 바란다.

찾아보기

T

U

V

에이콘출판의 기틀을 마련하신 故 정완재 선생님 (1935-2004)

파이썬 동시성 프로그래밍

명료하고 훌륭한 동시성 파이썬 코드 작성하고 개념 이해하기

발 행 | 2018년 9월 17일

지은이 | 엘리엇 포브스
옮긴이 | 이 창 화

펴낸이 | 권 성 준
편집장 | 황 영 주
편 집 | 이 지 은
디자인 | 박 주 란

에이콘출판주식회사
서울특별시 양천구 국회대로 287 (목동)
전화 02-2653-7600, 팩스 02-2653-0433
www.acornpub.co.kr / editor@acornpub.co.kr

ISBN 979-11-6175-204-4
ISBN 978-89-6077-210-6 (세트)
http://www.acornpub.co.kr/book/concurrency-python

이 도서의 국립중앙도서관 출판시도서목록(CIP)은 서지정보유통지원시스템 홈페이지(http://seoji.nl.go.kr)와 국가자료공동목록시스템(http://www.nl.go.kr/kolisnet)에서 이용하실 수 있습니다.(CIP제어번호: CIP2018029212)

책값은 뒤표지에 있습니다.