병렬 프로그래밍

Hadoop

Byeongjoon Noh

powernoh@sch.ac.kr

SCH SOON CHUN HYANG UNIVERSITY

Most of the slides are available on Senseable AI Lab homepage: https://sailab.space/courses/

Contents

1. 병렬처리의 기본 개념

2. 병렬처리의 구현 방법

3. 병렬처리와 분산처리

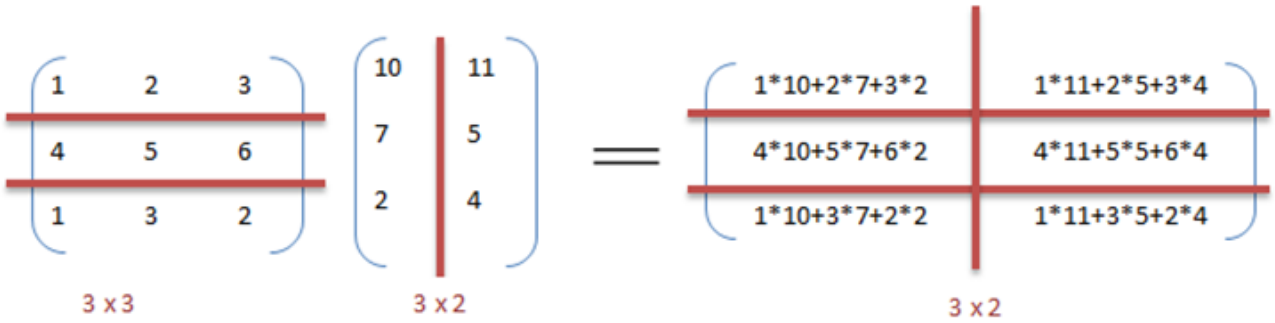
1. 병렬처리의 기본 개념

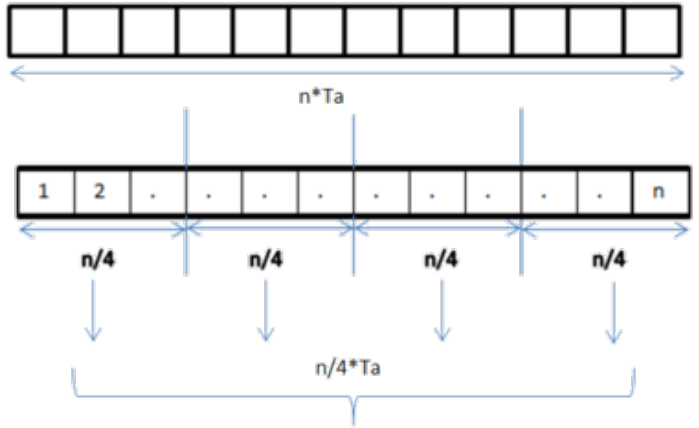
병렬처리 개요

- 병렬처리 (Parallel processing)
  - 다수의 프로세서들이 여러 개의 프로그램 또는 한 프로그램의 분할된 부분들을 분담하여 동시에 처리하는 기술
  - 컴퓨터시스템의 성능 향상을 위하여 가장 널리 사용되고 있는 방법

병렬처리 개요

- 순차처리 (Sequential processing)과 병렬처리 (Parallel processing) 예시






```
function matrix_multiply_sequential(A, B):
  let C = new matrix[3][2]

  for i from 1 to 3 do // A의 행
    for j from 1 to 2 do // B의 열
      C[i][j] = 0

      for k from 1 to 3 do // A의 열과 B의 행
        C[i][j] += A[i][k] * B[k][j]

  return C
```

```
if CPU = "a" then
  lower_limit = 1
  upper_limit = round(d.length / 2)

else if CPU = "b" then
  lower_limit = round(d.length / 2) + 1
  upper_limit = d.length

for i from lower_limit to upper_limit do
  foo(d[i])
```

병렬처리 개요

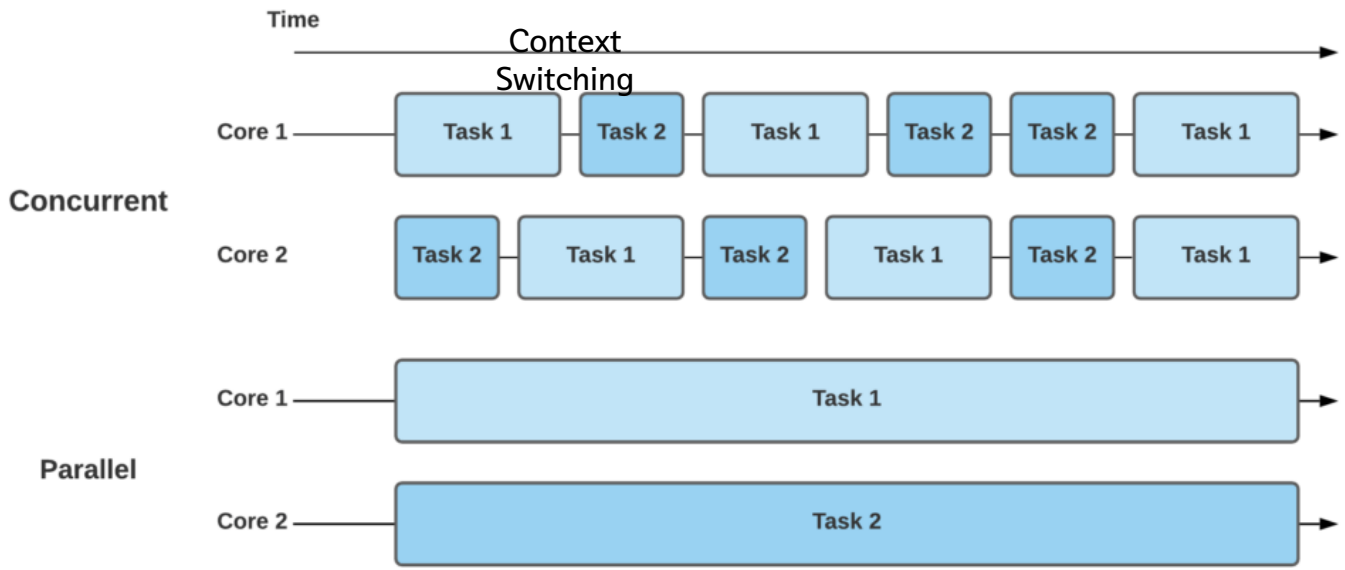
- 병렬처리의 조건
  - 다수의 프로세서들로 하나의 시스템을 구성할 수 있도록 작고 저렴하며 고속인 프로세서들의 사용이 가능해야 함
    - ➔ 반도체 기술 발전에 따라 어느정도 해결
  - 한 프로그램을 여러 개의 작은 부분들로 분할 가능해야 하며, 분할된 부분들에 대한 병렬처리 결과가 순차적 처리의 결과와 동일해야 함
  - 새로이 야기되는 과제들
    - 분할성 (decomposability): 한 프로그램을 여러 개로 분할 필요
    - 복잡성 (complexity): 병렬 알고리즘의 개발 필요
    - 프로세서간 통신 (inter-processor communication): 프로세서들 간 데이터 교환을 위한 통신 필요

동시성과 병렬성

- 동시성 (concurrency)
  - 단일 프로세서(CPU)가 여러 작업을 번갈아가면서 수행되는 방식
  - 여러 작업이 시간적으로 겹쳐서 수행됨 (단일 프로세서에서는 동시에 진행되는 것 처럼 보임)
  - 동시성의 목적: 주어진 자원의 활용을 극대화 하는 것

- 병렬성 (parallelism)
  - 여러 작업이 동시에 실행되는 것
  - 여러 개의 코어를 가진 CPU를 이용하는 방식

- 순차성 (sequential)
  - 일반적인 프로세스의 수행 방법



병렬처리의 한계와 가능성

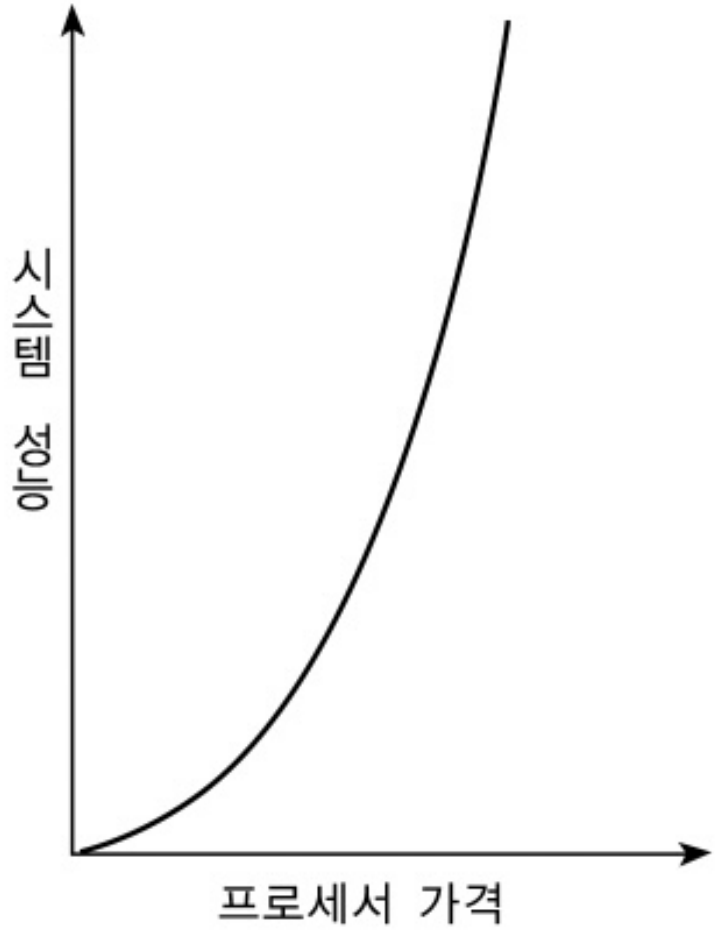
- 단일 프로세서의 속도 한계
  - Chip 상의 전자 흐름 속도 = $3x10^7$ m/sec (광속도의 1/10)
  - 지름이 3cm인 칩 내 최대 전송시간 = 10^{-9} sec

    ➔ 최대 계산 능력 : 10^9 FLOPS = 1 GFLOPS

  - 속도의 한계를 극복할 수 있는 방법 ➔ 병렬처리

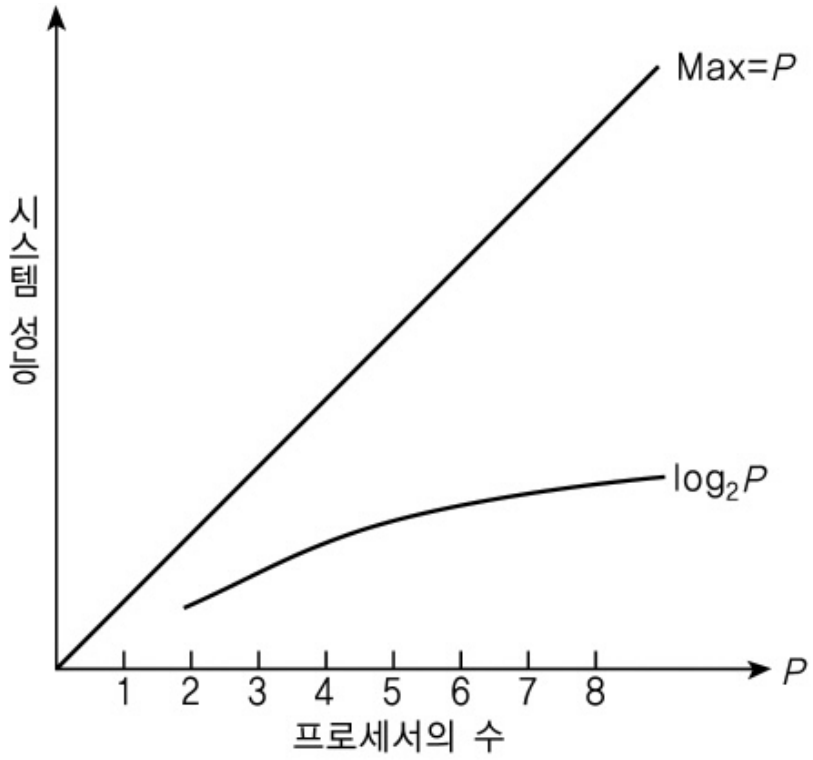
병렬처리의 한계와 가능성

- 병렬처리에 대한 비판적 이론
  - Glosch's law
    - 프로세서의 성능이 가격의 제곱에 비례하여 높아질 것으로 예측
    - 즉, $n^{1/2}$ 배의 비용만 투자해도 n배 빠른 프로세서 개발 가능
      - n개의 프로세서들의 병렬 사용 ➔ 속도: n배, 비용: n배
    - 여건의 변화
      - VLSI 기술의 발전으로 저렴한 가격의 소형 프로세서 개발 가능
        - VLSI: Very Large Scale Integration (초 고밀도 집적 회로)



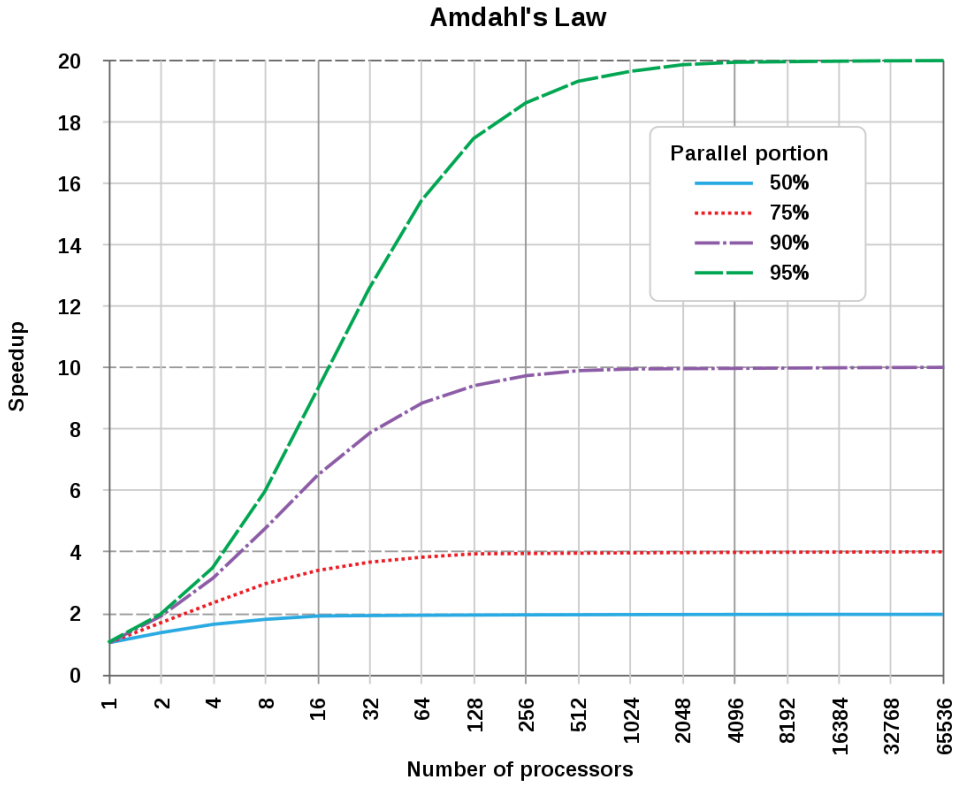
병렬처리의 한계와 가능성

- 병렬처리에 대한 비판적 이론
  - Minsky's conjecture
    - P개의 프로세서들을 사용한 병렬 프로세서에서 프로세서들 간의 정보 교환을 위한 통신 overhead 때문에 시스템 성능은 P배가 아닌 최대 $\log_2 P$배 까지만 개선될 수 있다고 추측
    - 여건의 변화
      - 효율적 병렬 알고리즘들의 개발
      - 고속 상호연결망 출현
      - 프로세서 스케줄링 기술



병렬처리의 한계와 가능성

- 병렬처리에 대한 비판적 이론
  - Amdahl's law (암달의 법칙)
    - 병렬처리를 이용하여 얻을 수 있는 속도 향상은 병렬화가 불가능한 비율 (Amdahl's fraction, a)에 의해 제한
    - $\text{Speedup} \leq \frac{1}{(1-P)+\frac{P}{N}}$
      - P: 프로그램 중 병렬처리가 가능한 부분의 비율
      - $1-P$: 순차적으로 처리되어야 하는 부분의 비율
      - N: 병렬처리에 사용되는 프로세서의 수



2. 병렬처리의 구현 방법

병렬처리의 구현 방법

- 멀티 태스킹
  - 단일 프로세서(CPU)를 이용하여 여러 프로세스를 번갈아가며 조금씩 실행시키는 방식
  - OS의 스케줄링 방식(Context Switching)을 이용 - Concurrency 방식을 사용한 병렬처리시스템

- 멀티 스레딩
  - 하나의 프로세스 내부에 다수의 스레드를 생성하여 작업을 처리하는 방식
  - 동일한 프로세스 내부의 스레드 간의 데이터 공유가 가능 (데이터 공유로 인한 장단점 존재)

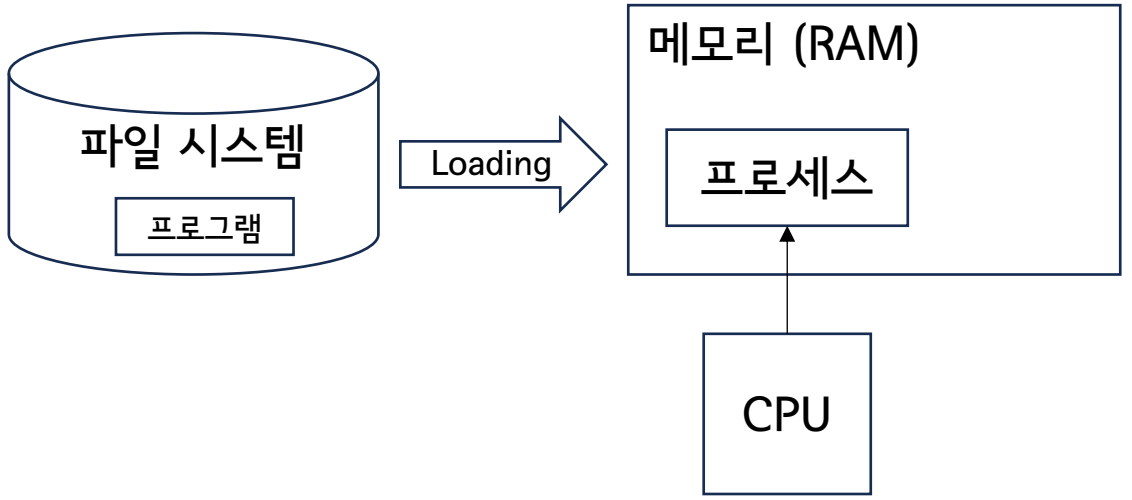
- 멀티 프로세싱
  - 다수의 프로세서가 협력하여 여러가지 프로그램을 동시 다발적으로 처리하는 것
    - 듀얼코어, 쿼드코어, ..(멀티 코어 시스템)

프로그램

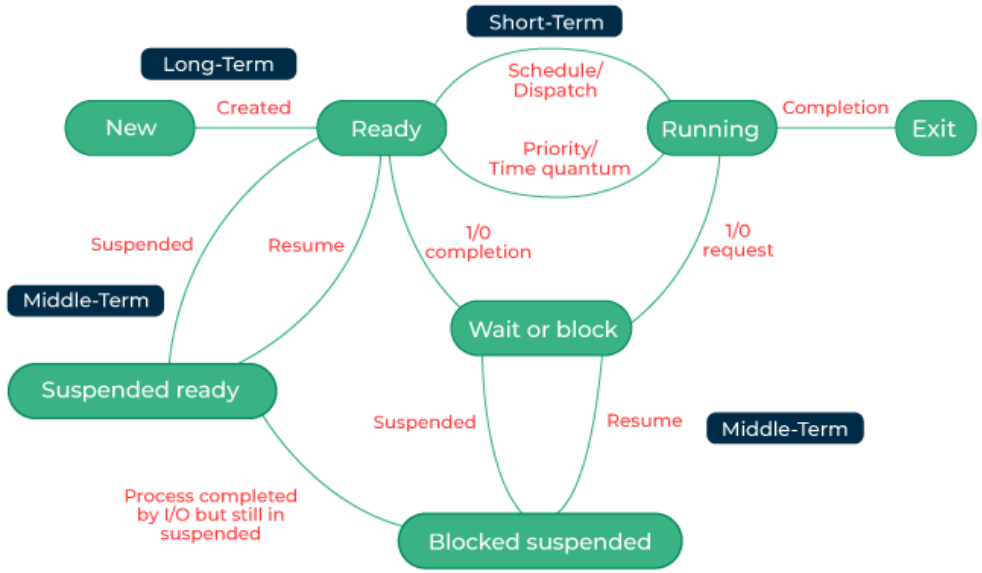
- 프로그램의 사전적 의미
  - 어떤 작업을 하기 위해 해야 할 명령(일)들을 순서대로 나열한 것
  - ➔ 컴퓨터에서 어떤 작업을 하기 위해 실행할 수 있는 정적인 상태의 파일
    - .exe, .bat, 등

프로세스 (Process)

- 프로세스의 사전적 의미
  - 프로그램이 실행되어 돌아가고 있는 상태
  - 컴퓨터에서 연속적으로 실행되고 있는 동적인 상태

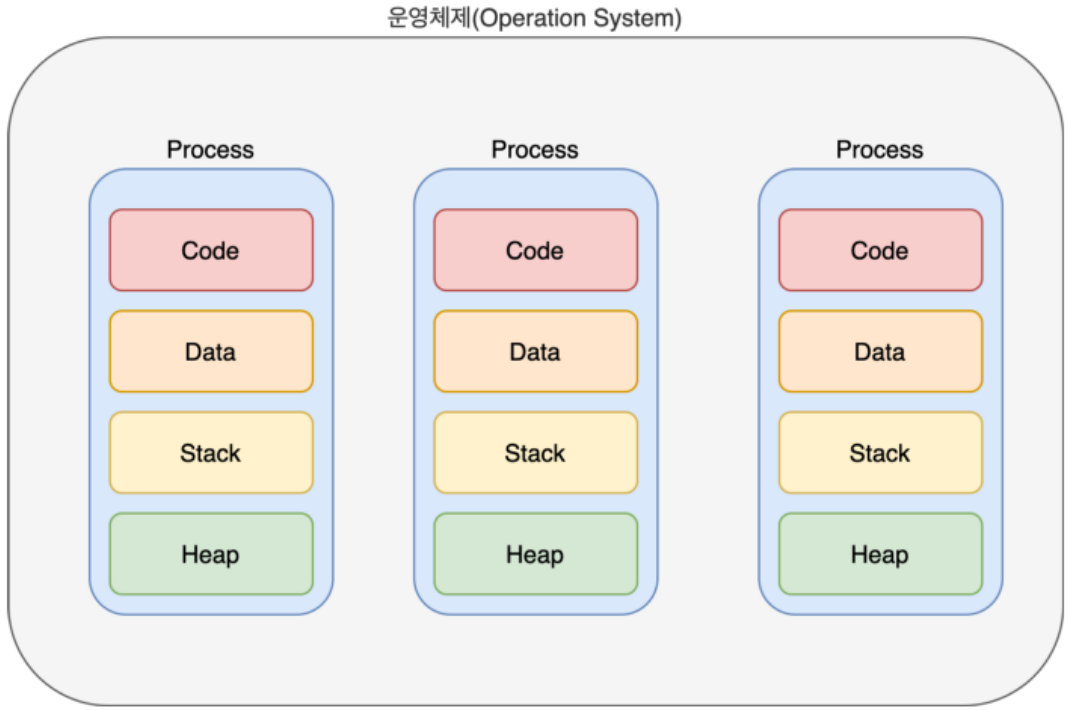


- 프로세스의 개념
  - OS가 메모리/CPU 등 필요한 자원을 “할당” 해준 실행중인 프로그램
    - (시스템)자원: CPU 시간, 메모리 주소, **메모리 영역**
    - **(참고) 메모리영역: Code, Data, Stack, Heap**



(참고) 메모리 영역

- Code 영역
  - 실행할 프로그램의 소스코드가 machine code로 저장되는 영역
    - CPU가 한 줄 씩 처리
- Data 영역
  - 소스코드 내 전역 변수와 정적 변수가 저장되는 영역
  - 해당 변수는 프로그램 실행 시 생성, 프로그램 종료 시 소멸
- Stack 영역
  - 함수 내에서 선언된 지역변수, 매개변수, 반환값 등을 저장
  - 함수 호출 시 생성, 함수 호출 시 소멸
- Heap 영역
  - 관리가 가능한 데이터 외 다른 형태의 데이터 (메타 데이터 등) 저장하기 위한 여유 공간



(참고) 메모리 영역: Java code 예시

- Code 영역: 실행할 프로그램의 바이너리 코드 저장
- Data 영역: 정적 변수 저장
- Stack 영역: 메소드 호출 관련 변수 및 지역 변수 저장
- Heap 영역: 객체와 배열 저장

```
public class JavaMemoryExample {
    static int staticNumber = 100; // Data 영역
    int instanceNumber; // Heap 영역
    public JavaMemoryExample() {
        this.instanceNumber = 50;
    }
    public void performCalculation(int a) {
        int result = a * 10; // Stack 영역
        System.out.println(result);
    }
    public static void main(String[] args) {
        JavaMemoryExample example = new JavaMemoryExample(); // Heap 영역
    }
}
```

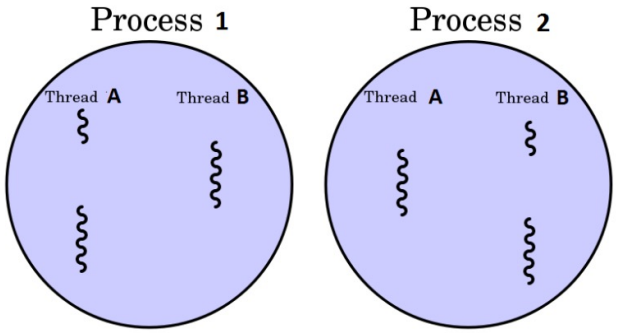
(참고) 메모리 영역: Python code 예시

- Code 영역: 실행할 프로그램의 코드 저장
- Data 영역: CPython 구현의 내부 상태와 같은 내부 데이터 저장
- Stack 영역: 함수 호출 관련 변수 및 지역 변수 저장
- Heap 영역: 객체와 데이터 구조(list, tuple, ...)를 저장

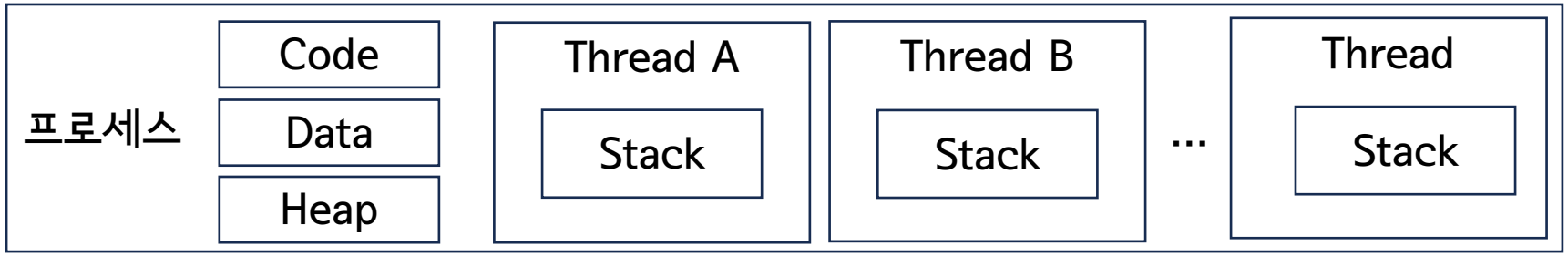
```
class PythonMemoryExample:
    class_variable = 100  # 클래스 변수, Heap 영역

    def __init__(self):
        self.instance_variable = 50  # 인스턴스 변수, Heap 영역

    def perform_calculation(self, a):
        result = a * 10  # Stack 영역(참조는 Stack에, 실제 객체는 Heap에 저장)
        print(result)

def main():
    example = PythonMemoryExample()  # 객체 생성, Heap 영역

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

스레드 (Thread)

- 스레드의 사전적 의미
  - 프로세스 내에서 동작하는 **여러 실행의 흐름** 단위
    - 프로세스가 할당받은 자원의 실행 단위 (프로세스의 특정 수행 경로)



- 스레드의 특징
  - 한 프로세스 내에서 각각 Stack만 따로 할당받고 Code, Data, Heap 영역은 공유
  - 프로세스 내의 주소 공간이나 자원들을 같은 프로세스 내 스레드 끼리 공유하며 동작 (실행)
  - 다른 프로세스에서는 접근 불가능
  - 한 스레드가 (프로세스 내) 자원을 변경하면, 같은 프로세스 내 다른 스레드 (sibling thread)도 변경 결과를 공유



스레드의 활용

- 응용 프로그램 (Application)에서의 스레드 활용 예시
  - 웹 서버: 동시에 사용자의 요청을 처리해야함
    - ➔ 임영웅 콘서트 티켓 예매
  - 채팅 프로그램: 사용자간 대화의 처리 방식을 (멀티)스레드를 활용하여 처리해야 함
    - ➔ 카카오톡 채팅
  - 게임 프로그램: 여러 사용자의 입력(마우스 클릭, qwer 콤보 등)을 동시에 처리해야 함
    - ➔ 리그오브레전드

스레드의 생성과 관리

- ‘theading’ 모듈
  - Python에서 스레드를 생성하고 관리하기 위한 모듈

  - 단일 스레드 생성 및 시작 예제
    - `Thread()`: 활용하여 스레드 객체 생성
    - `start()`: 메소드를 활용하여 스레드 시작

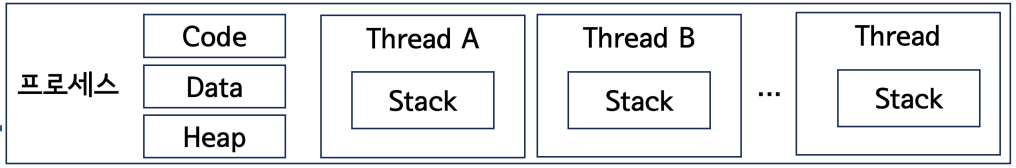
```
import threading

def print_numbers():
  for i in range(10):
    print(i)

if __name__ == '__main__':
  thread = threading.Thread(target=print_numbers)
  thread.start()
```

스레드의 생성과 관리

- 스레드 상태 확인
  - `is_alive()`: 스레드가 실행 중인 경우 True, 아니면 False를 반환

```
import threading
import time

def print_numbers():
  for i in range(5):
    print(i)
    time.sleep(1)

if __name__ == '__main__':
  thread = threading.Thread(target=print_numbers)
  thread.start()

  while thread.is_alive():
    print("Thread is still running")
    time.sleep(0.5)

  print("Thread has finished")
```

멀티 스레딩 (Multithreading)



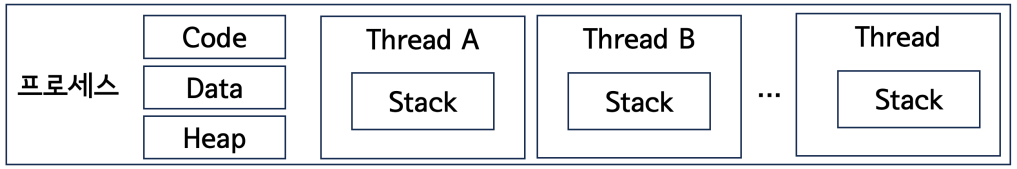
- 프로그램이 동일한 프로세스 내에서 여러 스레드를 동시에 실행할 수 있도록 하는 프로그래밍 기법
- 각각의 스레드는 독립적으로 실행되며 자신의 작업을 수행할 수 있으므로 병렬 처리를 하는 것 “처럼” 보임

```
if __name__ == '__main__':
  thread1 = threading.Thread(target=print_numbers, args=("Thread-1",))
  thread2 = threading.Thread(target=print_numbers, args=("Thread-2",))

  thread1.start()
  thread2.start()

  thread1.join()
  thread2.join()

  print("Finished all threads")
```

← thread가 종료되기를 기다림 (동기화)

멀티 스레딩 (Multithreading)

- 프로그램이 동일한 프로세스 내에서 여러 스레드를 동시에 실행할 수 있도록 하는 프로그래밍 기법
- 각각의 스레드는 독립적으로 실행되며 자신의 작업을 수행할 수 있으므로 병렬 처리를 하는 것 “처럼” 보임




```
if __name__ == '__main__':

  num_threads = 5
  count = 5

  threads = []

  for i in range(num_threads):
    thread = threading.Thread(target=print_numbers, args=("Thread-"+str(i+1),))
    threads.append(thread)
    thread.start()

  for thread in threads:
    thread.join()

print("Finished all threads")
```

멀티 스레딩 예제

- 두 개의 스레드로 두 함수 (print_numbers(), print_letters())를 각각 실행

```
import threading
import time

def print_numbers(thread_name):
  for i in range(5):
    print(f"{thread_name}: {i}")
    time.sleep(1)

def print_letters(thread_name):
  for i in range(5):
    print(f"{thread_name}: {chr(i+65)}")
    time.sleep(2)

if __name__ == '__main__':
  thread1 = threading.Thread(target=print_numbers, args=("Thread-1",))
  thread2 = threading.Thread(target=print_letters, args=("Thread-2",))

  thread1.start()
  thread2.start()

  thread1.join()
  thread2.join()

  print("Finished all threads")
```

멀티 스레딩 예제

- 전역변수 “data”를 3개의 스레드에서 각 스레드 마다 값을 1씩 증가시키며 실행

```
data = 0

def adds(thread_name):
  global data

  for i in range(5):
    new = data + 1
    data = new
    print(f"{thread_name}: updated number {data}")
    time.sleep(i)

if __name__ == '__main__':
  num_threads = 3
  threads = [threading.Thread(target=adds, args=("Thread-"+str(i+1),)) for i in range(num_threads)]

  for thread in threads:
    thread.start()

  for thread in threads:
    thread.join()

  print("Finished all threads")
```

멀티 스레딩 동기화

- 동기화 (Synchronization)
  - 다수의 스레드나 프로세스가 자원을 공유할 때 발생할 수 있는 충돌과 일관성 문제를 방지하기 위해 사용
    - 멀티스레딩 환경에서는 매우 중요한 개념
  - 데이터의 일관성과 정확성을 유지하면서 여러 스레드가 동시에 작업을 수행할 수 있도록 조정하는 과정

- 크리티컬 섹션 (Critical section)
  - 한 번에 한 스레드만이 실행할 수 있어야 하는 영역
  - 크리티컬 섹션을 보호하기 위해 다양한 동기화 메커니즘을 활용
    - Mutex
    - Semaphore
    - Spinlock
    - ...

멀티 스레딩 동기화

- 경쟁 상태 (Race Condition)
  - 멀티 스레딩 환경에서 다수의 스레드가 동시에 같은 메모리 영역이나 자원에 접근할 때 각 스레드의 실행 순서에 따라 결과가 달라질 수 있음
    - ex) 두 스레드가 동시에 같은 변수를 수정하려고 할 때, 하나의 스레드가 변수를 읽고 수정하는 동안 다른 스레드가 해당 변수의 값을 수정하는 경우 ➔ 최종 결과가 의도하지 않는 방식으로 변경됨

- 데드락 (Deadlock)
  - 두 개 이상의 스레드가 서로의 스레드가 소유한 자원을 기다리는 상황 (서로가 서로를 기다림)
  - 무한 대기 상황에 진입하며 프로그램이 정지됨

멀티 스레딩 동기화

- 멀티스레딩 동기화 방법
  - Lock
    - 가장 기본적인 동기화 도구
    - 특정 시점에 단 하나의 스레드만이 특정 섹션(크리티컬 섹션)의 코드를 실행하도록 함
    - Mutex (Lock) 매커니즘
      - 1) 스레드는 공유 자원을 사용하기 전에 락을 획득(잠금)
      - 2) 자원 사용 후에는 락을 해제(잠금 해제)하여 다른 스레드가 사용할 수 있도록 함

멀티 스레딩 동기화

- 멀티스레딩 동기화 방법
  - Semaphore
    - Lock의 일반화된 형태
    - 동시에 여러 스레드가 특정 자원에 접근할 수 있도록 허용
    - 내부적으로 카운터를 유지하여 동시에 자원에 접근할 수 있는 스레드의 최대 허용 수를 계산
      - 스레드가 Semaphore를 획득할 때마다 카운터가 감소
      - 스레드가 Semaphore를 해제할 때마다 카운터가 증가
      - 카운터가 0이 되면, 다른 스레드가 자원에 접근하려 할 때 대기 상태로 전환

멀티 스레딩 동기화 예제

- Mutex

```
data = 0

def adds(thread_name):
  global data

  with threading.Lock():
    for i in range(5):
      new = data + 1
      data = new
      print(f"{thread_name}: updated number {data}")
      time.sleep(i)

if __name__ == '__main__':
  num_threads = 3
  threads = [threading.Thread(target=adds, args=("Thread-"+str(i+1),)) for i in range(num_threads)]

  for thread in threads:
    thread.start()

  for thread in threads:
    thread.join()

  print("Finished all threads")
```

← 해당 thread가 종료될 때 까지 다른 thread를 실행하지 않음

멀티 스레딩 동기화 예제

- Semaphore

```
semaphore = threading.Semaphore(2)    ← 2개의 thread까지만 접근 허용함

def access_resource(thread_id):
  print(f"Thread {thread_id} is attempting to access the resource...")
  semaphore.acquire()

  print(f"Thread {thread_id} has accessed the resource!")
  time.sleep(2) # working processes

  print(f"Thread {thread_id} is releasing the resource...")
  semaphore.release()

if __name__ == '__main__':
  threads = [threading.Thread(target=access_resource, args=(i,)) for i in range(0, 10)]

  for thread in threads:
    thread.start()

  for thread in threads:
    thread.join()

  print("All threads have finished")
```

멀티 스레딩 동기화 예제

- Deadlock 상황의 구현

```
lock1, lock2 = threading.Lock(), threading.Lock()

def thread1_routine():
  while True:
    with lock1:
      print("Thread 1: lock1 acquired")
      time.sleep(1)

      with lock2:
        print("Thread 1: lock2 acquired")

      print("Thread 1: lock1 and lock2 released")
      time.sleep(1)

def thread2_routine():
  while True:
    with lock2:
      print("Thread 2: lock2 acquired")
      time.sleep(1)

      with lock1:
        print("Thread 2: lock1 acquired")

      print("Thread 2: lock2 and lock1 released")
      time.sleep(1)
```

```
t1 = threading.Thread(target=thread1_routine)
t2 = threading.Thread(target=thread2_routine)
t1.start()
t2.start()

t1.join(timeout=2)
t2.join(timeout=2)
print("Deadlock might have occurred.")
```

Multithreading을 활용한 병렬처리

- Python에서 multithreading을 활용한 병렬처리
  - 문제: 1부터 10,000,000까지의 합을 구하는 프로그램 (답: 49999995000000)
    - 순차 처리: 소요시간 약 0.18 sec

```
def heavy_work():
  result = 0

  for i in range(10000000):
    result += i
    print(result)

if __name__ == '__main__':
  start = time.time()

  heavy_work()

  end = time.time()

  print("Proc time: %f sec" % (end - start))
```

Multithreading을 활용한 병렬처리

- Python에서 multithreading을 활용한 병렬처리
  - 문제: 1부터 10,000,000까지의 합을 구하는 프로그램 (답: 49999995000000)
    - 병렬 처리 : 소요시간 약 0.48 sec (vs. 순차처리 시 약 0.18 sec) ← why?!

```
import queue

result_queue = queue.Queue()

def heavy_work(name, start, end):
  global result_queue

  result = 0
  for i in range(start, end):
    result += i

  result_queue.put(result)

  print('%s done' % name)
```

```
if __name__ == '__main__':
  start = time.time()

  threads = []
  for i in range(10):
    t = threading.Thread(target=heavy_work, args=(i, 1000000*i, 1000000*(i+1)))
    t.start()
    threads.append(t)

  for t in threads:
    t.join()

  end = time.time()
  print("Proc time: %f sec" % (end - start))

  total_sum = 0

  while not result_queue.empty():
    total_sum += result_queue.get()

  print(total_sum)
```

Python GIL

- Global Interpreter Lock (GIL)
  - 하나의 스레드만 Python 인터프리터의 제어를 유지할 수 있도록 하는 lock (뮤텍스)
    - Python의 메모리 관리가 스레드에 안전하지 않기 때문에 도입됨
  - CPU 바운드 작업에서 멀티스레딩의 이점을 제한
    - 즉, 여러 스레드가 CPU를 사용하는 작업을 수행할 경우, 실제로는 한 번에 하나의 스레드만이 실행될 수 있습니다. 이로 인해 멀티코어 프로세서의 이점을 활용하지 못하는 경우가 발생
    - * CPU 바운드 작업: 프로그램의 성능이 CPU의 성능에 의해 주로 결정되는 작업
      - ex) 팩토리얼 계산 등
    - * I/O 바운드 작업: 프로그램의 성능이 입출력 작업에 의해 결정되는 작업
      - ex) txt 파일 읽어오기 등

Python GIL

- Global Interpreter Lock (GIL)
  - https://www.itworld.co.kr/news/302737

개발자

파이썬에서 GIL 삭제된다…“병렬 처리의 혁신적 진전”

Serdar Yegulalp | InfoWorld 2023.08.09

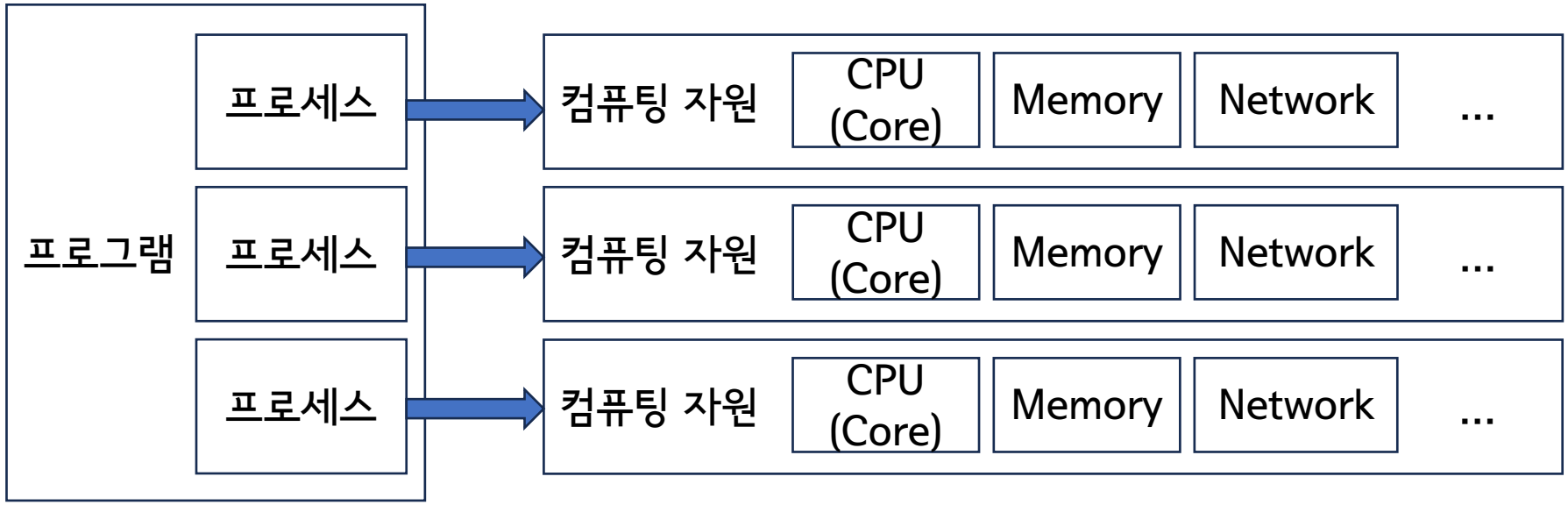
Summary - 멀티스레딩

- Concurrency
- I/O-bound tasks
  - particularly useful for I/O bound tasks
    - because threads can perform non-CPU-intensive operations
      - e.g., reading/writing files, making network requests, or waiting for user input

    threads can yield control when waiting for I/O, allowing other threads to execute
- Resource sharing
  - threads can share data and resources within the same process, making it easier to coordinate work among different parts of your program

멀티프로세싱 (Multiprocessing)

- 여러 프로세스를 사용하여 작업을 동시에 수행하는 프로그래밍 기술
- 각 프로세스는 독립적으로 실행되며 CPU 코어를 포함한 자체 메모리 공간, 프로그램 카운터 및 리소스 소유
  - vs 멀티스레딩: 여러 스레드가 단일 프로세스 내에서 동일한 메모리 공간을 공유



멀티 프로세싱

- ‘multiprocessing’ 모듈
  - Python에서 멀티 프로세싱을 사용하여 병렬처리를 수행하기 위한 모듈
  - 다수의 프로세스를 생성하고, 각 프로세스는 독립적인 자원을 할당 받음 (GIL 적용 X)

- 병렬화 방법

멀티 프로세싱

- ‘multiprocessing’ 모듈의 활용
  - 문제: 두 개의 리스트를 각각 처리하는 프로세스 2개를 생성

```python
import random
import time
import multiprocessing

def square(numbers):
  time.sleep(10)
  print([num**2 for num in numbers])

if __name__ == '__main__':
  num_list1 = [2, 4, 6, 8, 10]
  num_list2 = [1, 3, 5, 7, 9]

  p1 = multiprocessing.Process(target=square, args=(num_list1, ))
  p2 = multiprocessing.Process(target=square, args=(num_list2, ))

  p1.start()
  p2.start()

  p1.join()
  p2.join()

  print("Completed")
```

```
(bj) (base) user@workstation:~/Lecture/lecture_hadoop/parallel$ ps -ef | grep python
root         726       1  0  4월 19 ?      00:00:00 /usr/bin/python3 /usr/bin/networkd-dispatc
her --run-startup-triggers
root         905       1  0  4월 19 ?      00:00:00 /usr/bin/python3 /usr/share/unattended-upg
rades/unattended-upgrade-shutdown --wait-for-signal
user      495990  495663  7 11:48 ?        00:00:34 /home/user/.vscode-server/bin/0ee08df0cf4
527e40edc9aa28f4b5bd38bbff2b2/node /home/user/.vscode-server/extensions/ms-python.vscode-pyla
nce-2024.3.100/dist/server.bundle.js --cancellationReceive=file:5d2dfad6aa579c1860055ddd13ef5
264084e298cd9 --node-ipc --clientProcessId=495663
user      496696  495717  0 11:55 pts/3    00:00:00 python ex12-multiproc-multiproc-intro.py
user      496697  496696  0 11:55 pts/3    00:00:00 python ex12-multiproc-multiproc-intro.py
user      496698  496696  0 11:55 pts/3    00:00:00 python ex12-multiproc-multiproc-intro.py
user      496784  496337  0 11:55 pts/6    00:00:00 grep --color=auto python
```

멀티 프로세싱 예제

- 순차 처리 vs 병렬 처리 (멀티 프로세싱)
  - 문제: 1천만개의 랜덤값을 가지는 리스트의 각 요소를 제곱하는 작업

```python
import random
import time

def func(numbers):
  return [num**2 for num in numbers]

if __name__ == '__main__':
  numbers = [random.randint(1, 100) for _ in range(10000000)]

  start_time = time.time()

  sequential_result = func(numbers)

  sequential_time = time.time() - start_time

  print(f"Proc time: {sequential_time:.2f} sec")
```

멀티 프로세싱 예제

- 순차 처리 vs 병렬 처리 (멀티 프로세싱)
  - 문제: 1천만개의 랜덤값을 가지는 리스트의 각 요소를 제곱하는 작업
    - CPU core 만큼의 프로세스를 생성하고 각 CPU core에 분할된 작업을 할당
      - 작업 분할: 1천만개 / CPU core 수

```python
import multiprocessing
import random
import time

def square_numbers(data):
  return [x**2 for x in data]

def worker(input_data):
  square_numbers(input_data)
```

- Output 획득하려면?

```python
if __name__ == '__main__':
  data = [random.randint(1, 100) for _ in range(10000000)]

  processes = []

  cpu_count = multiprocessing.cpu_count()
  chunk_size = len(data) // cpu_count
  chunks = [data[i * chunk_size:(i + 1) * chunk_size] for i in range(cpu_count)]

  start_time = time.time()

for chunk in chunks:
  process = multiprocessing.Process(target=worker, args=(chunk,))
  processes.append(process)
  process.start()

for process in processes:
  process.join()

print("Proc time: ", time.time() - start_time)
```

멀티 프로세싱 예제

- 순차 처리 vs 병렬 처리 (멀티 프로세싱)
  - 문제: 1천만개의 랜덤값을 가지는 리스트의 각 요소를 제곱하는 작업
    - CPU core 만큼의 프로세스를 생성하고 각 CPU core에 분할된 작업을 할당
      - 작업 분할: 1천만개 / CPU core 수

```python
import multiprocessing
import random
import time

def square_numbers(data):
  return [x**2 for x in data]

def worker(input_data, output_queue):
  result = square_numbers(input_data)
  output_queue.put(result)
```

- Output의 순서가 보장되는가?

```python
if __name__ == '__main__':
  data = [random.randint(1, 100) for _ in range(10000000)]

  processes = []
  output_queue = multiprocessing.Queue()

  '''data split by # of CPU cores'''

  for chunk in chunks:
    process = multiprocessing.Process(target=worker, args=(chunk, output_queue))
    processes.append(process)
    process.start()

  results = []
  for _ in range(cpu_count):
    results.extend(output_queue.get())

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

멀티 프로세싱 예제

- 순차 처리 vs 병렬 처리 (멀티 프로세싱)
  - 문제: 1천만개의 랜덤값을 가지는 리스트의 각 요소를 제곱하는 작업
    - CPU core 만큼의 프로세스를 생성하고 각 CPU core에 분할된 작업을 할당
      - 작업 분할: 1천만개 / CPU core 수

```python
import multiprocessing
import random
import time

def square_numbers(data):
  return [x**2 for x in data]

def worker(input_data, output_queue):
  result = square_numbers(input_data)
  output_queue.put(result)
```

- 병렬처리 시간이 보장되는가?

```python
if __name__ == '__main__':
  data = [random.randint(1, 100) for _ in range(10000000)]

  processes = []
  output_queue = multiprocessing.Queue()

  '''split data by the number of CPU cores'''
  '''generate multiprocesses'''

  results = [None] * 4

  for _ in range(4):
    index, result = output_queue.get()
    results[index] = result

  for process in processes:
    process.join()

  final_results = []
  for result in results:
    final_results.extend(result)
```

멀티 프로세싱 예제

- Pool.map()의 활용
  - 각 프로세스에 작업을 자동으로 할당하고, 결과를 순서대로 수집 가능

```
if __name__ == '__main__':
  data = [random.randint(1, 100) for _ in range(10000000)]

  with multiprocessing.Pool(processes=multiprocessing.cpu_count()) as pool:
    chunk_size = len(data) // multiprocessing.cpu_count()
    chunks = [data[i * chunk_size:(i + 1) * chunk_size] for i in range(multiprocessing.cpu_count())]

    results = pool.map(square_numbers, chunks)

  final_results = []

  for result in results:
    final_results.extend(result)
```

멀티 프로세싱 예제

- N개의 csv 파일 읽어서 pandas dataframe 형태로 저장하는 프로그램

```
import os
import pandas as pd
import multiprocessing

def read_csv_file(file_path):
  return pd.read_csv(file_path)

if __name__ == '__main__’:
  csv_files = [f for f in os.listdir('.') if f.endswith('.csv’)]

  with multiprocessing.Pool(processes=multiprocessing.cpu_count()) as pool:
    dataframes = pool.map(read_csv_file, csv_files)

  dataframe_dict = {file: df for file, df in zip(csv_files, dataframes)}
```

Multithreading vs. multiprocessing

- Complexity
  - multiprocessing can be more complex to set up compared to multithreading because of dealing with separate processes
  - multithreading is often simpler to implement than multiprocessing
- Memory isolation
  - in multiprocessing, since each process has its own memory space, there is no shared memory be default
  - in multithreading, since threads share the same memory space, they can easily share data by accessing the same variables
    - however, there are some issues about data races and synchronization, which must be managed carefully

Multithreading vs. multiprocessing

- Parallelism
  - multiprocessing provides true parallelism, meaning that each process runs independently on a separate CPU core
    - this is suitable for CPU-bound tasks such as complex calculation
  - in multithreading, threads can run concurrently, Python's GIL restricts the execution of Python bytecode to one thread at a time

3. 병렬처리와 분산처리

병렬처리 vs. 분산처리

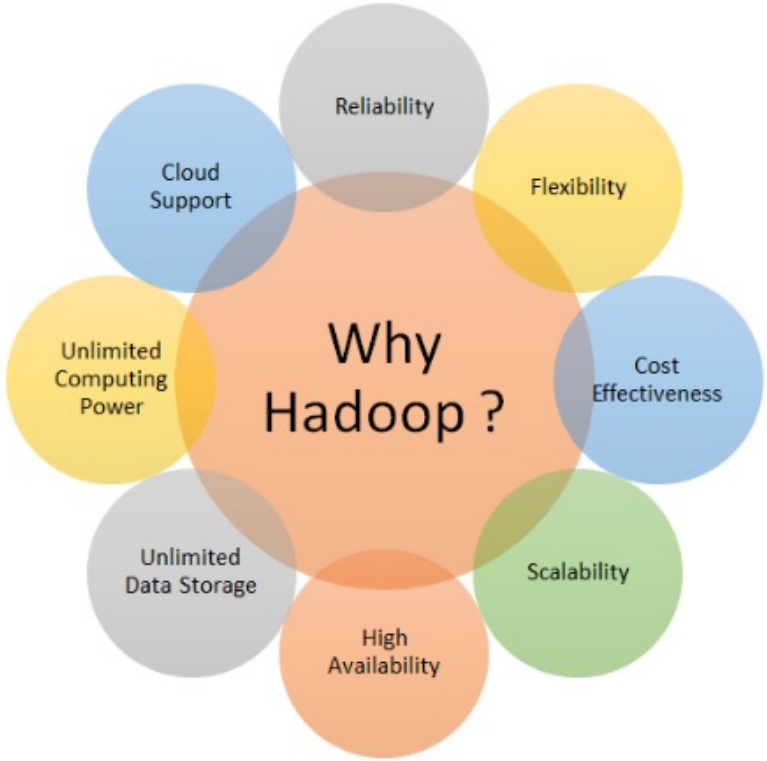
- 병렬처리 (Parallel processing)
  - 여러 계산을 동시에 수행하여 처리 속도를 높이는 기술
  - 단일 머신 내의 멀티코어 프로세서나 GPU 등의 하드웨어 자원을 활용
  - 하나의 큰 문제를 여러 작은 부분으로 나누어 이를 동시에 처리하는 방법

- 분산처리 (Distributed processing)
  - 네트워크를 통해 연결된 여러 컴퓨터(노드)가 작업을 공유하고 처리하는 방식
  - 하나의 큰 문제를 여러 작은 부분으로 나누어 이를 각 노드에 전송하여 처리하는 방법
  - 대표적인 분산처리 플랫폼: Hadoop

분산처리 플랫폼

- Apache Hadoop
  - 대용량의 데이터를 적은 비용으로 더 빠르게 분석할 수 있는 플랫폼
  - 빅데이터 처리와 분석을 위한 플랫폼 중 사실상 표준

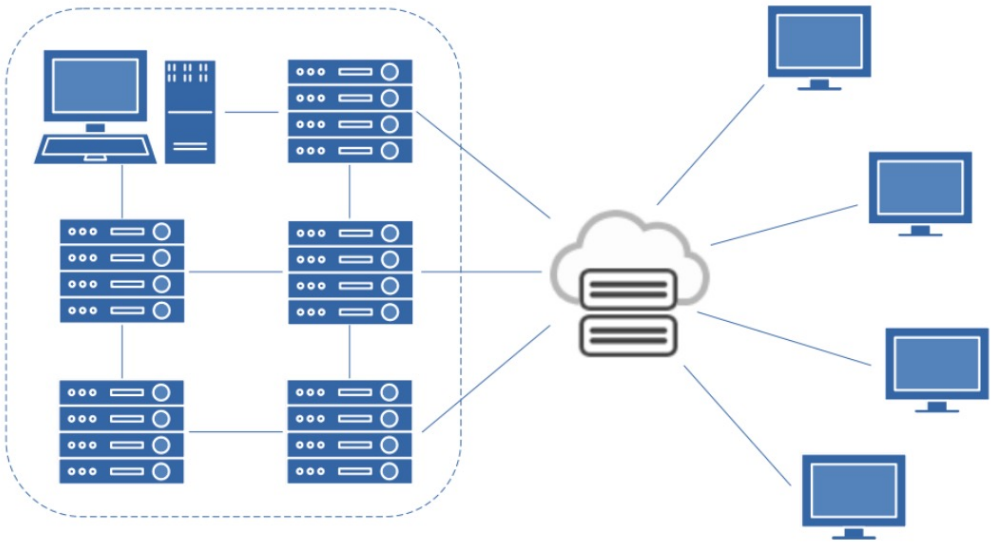
  - 하나의 고성능 컴퓨터 ➔ 범용 컴퓨터 여러 대를 클러스터화
  - 큰 데이터 크기의 데이터를 클러스터에서 병렬로 처리 (분산 처리)
  - 처리 속도의 비약적 향상



- 오픈 소스 기반 플랫폼
- Java 기반 분산 컴퓨팅 플랫폼

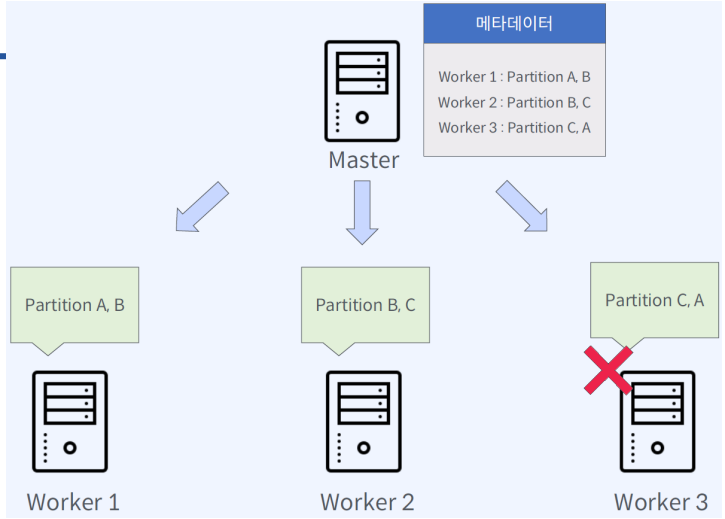
Hadoop의 특징

- Scalability
  - “Cluster”에 machine을 추가함으로써 수평적인 확장이 가능함
    - 수천~수만 대의 서버 또는 컴퓨터를 사용하여 대량의 데이터 분산 처리
    - 시스템의 처리 능력 확장
    - 비용 절감의 효과



Hadoop의 특징

- Fault tolerance
  - 동작 중 일부가 실패 (Fault) 해도 시스템은 계속 작동하도록 설계 및 구현
  - Master-Worker 구조
    - Hadoop의 분산 시스템 관리 방법 중 하나
    - Master 노드: 모든 노드의 동기화/조정
    - Worker 노드: 데이터 처리 작업, Master 노드 고장 시 그 역할을 수행
  - 여러 Worker 노드에 데이터를 복제하여 저장
    - 데이터의 손상 및 노드의 장애 시에도 다른 노드에서 데이터 복구
    - 노드 상태를 주기적으로 확인하여 자체적인 복구 가능

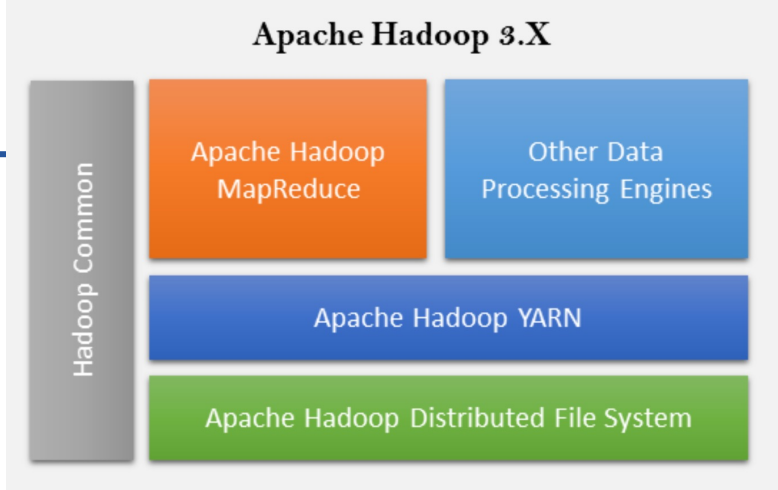


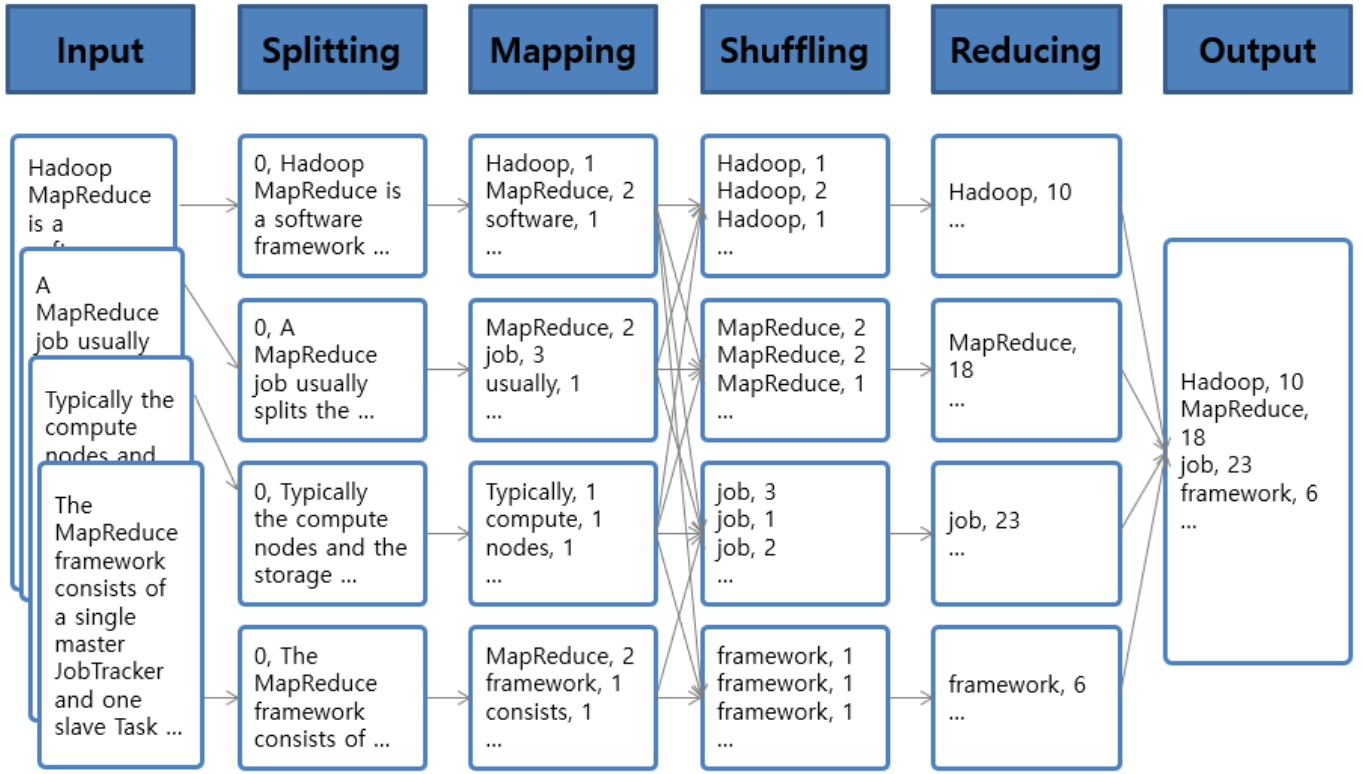
Hadoop의 특징

- Flexibility
  - 다양한 데이터 유형 지원
    - Structured, semi-structure, unstructured data 지원
    - Hadoop ecosystem 내 다른 데이터 처리 도구들과 연동 가능
  - 다양한 데이터 처리 방식 지원
    - MapReduce 등 다양한 프로그래밍 모델 제공
    - HDFS 등 다양한 저장소 및 데이터 처리 도구를 제공
  - 확장 가능한 아키텍처
    - 새로운 데이터 처리 방식이나 유형에 대한 지원을 비교적 쉽게 추가 가능
    - 다양한 오픈소스와 연동 가능
  - 비용 효율적인 하드웨어 지원
    - 클러스터 기반의 시스템 ➔ 범용 기기의 확장을 통한 시스템 구축

Hadoop 4 core principles

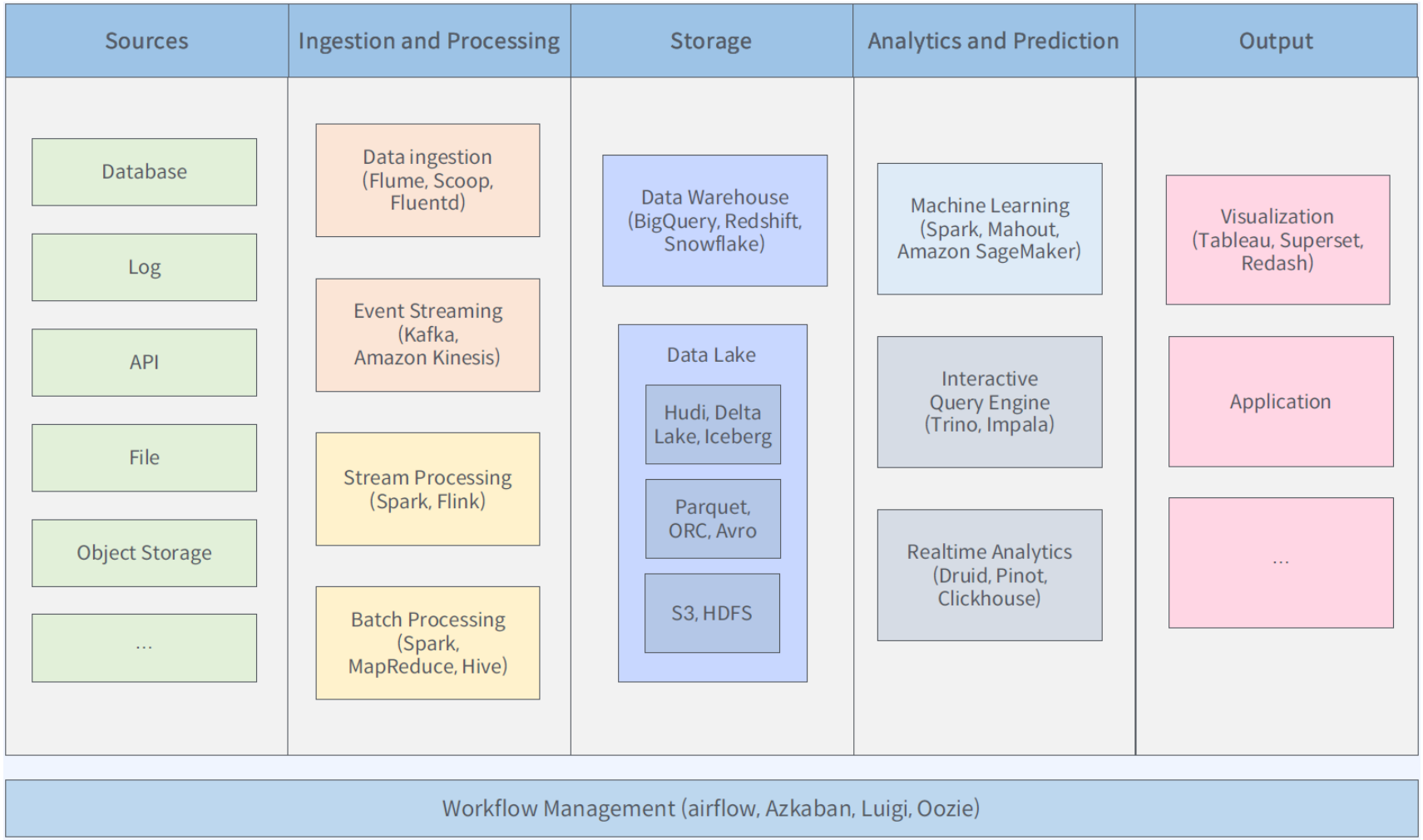
- Hadoop common library
  - Hadoop FS Shell
- Hadoop Distributed File System (HDFS)
- MapReduce (MR)
- Yet Another Resource Negotiator (YARN)





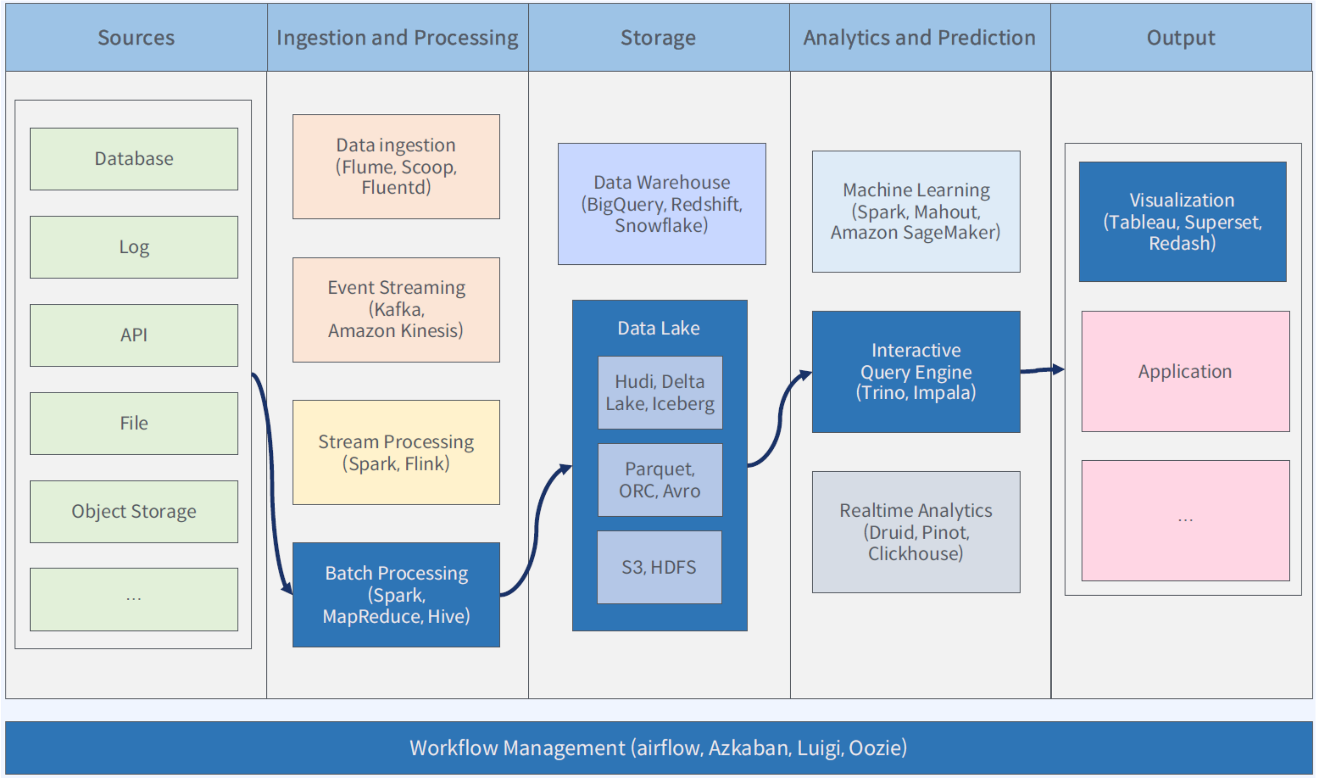
Hadoop ecosystem

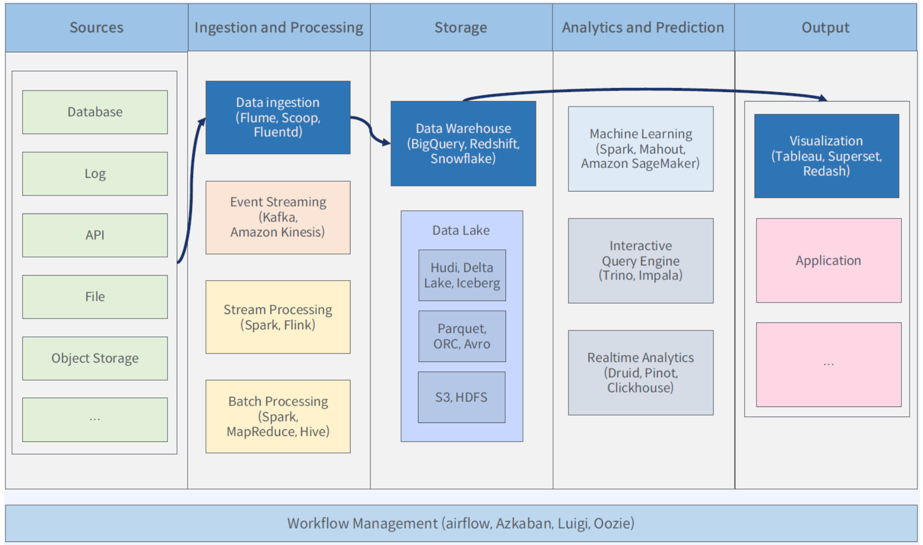
- 빅데이터 처리/분석 아키텍쳐

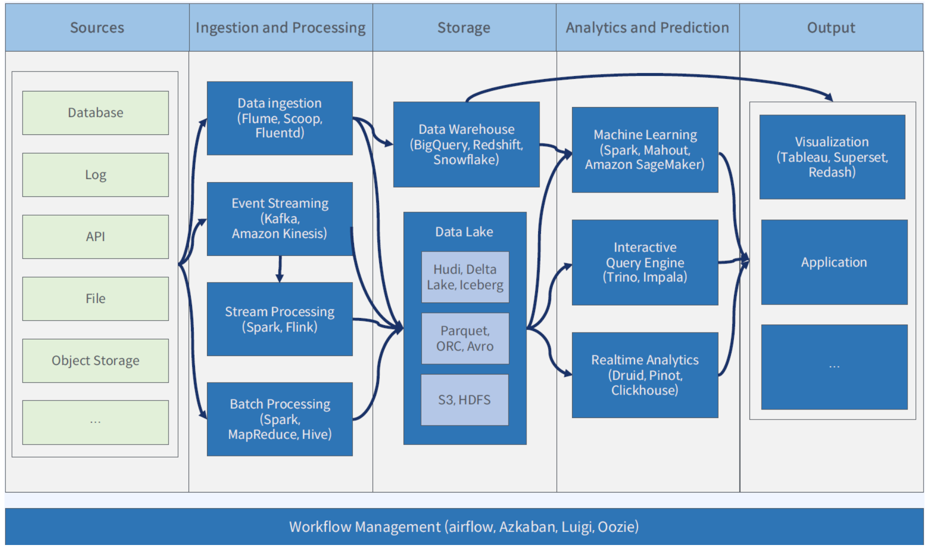


Hadoop ecosystem

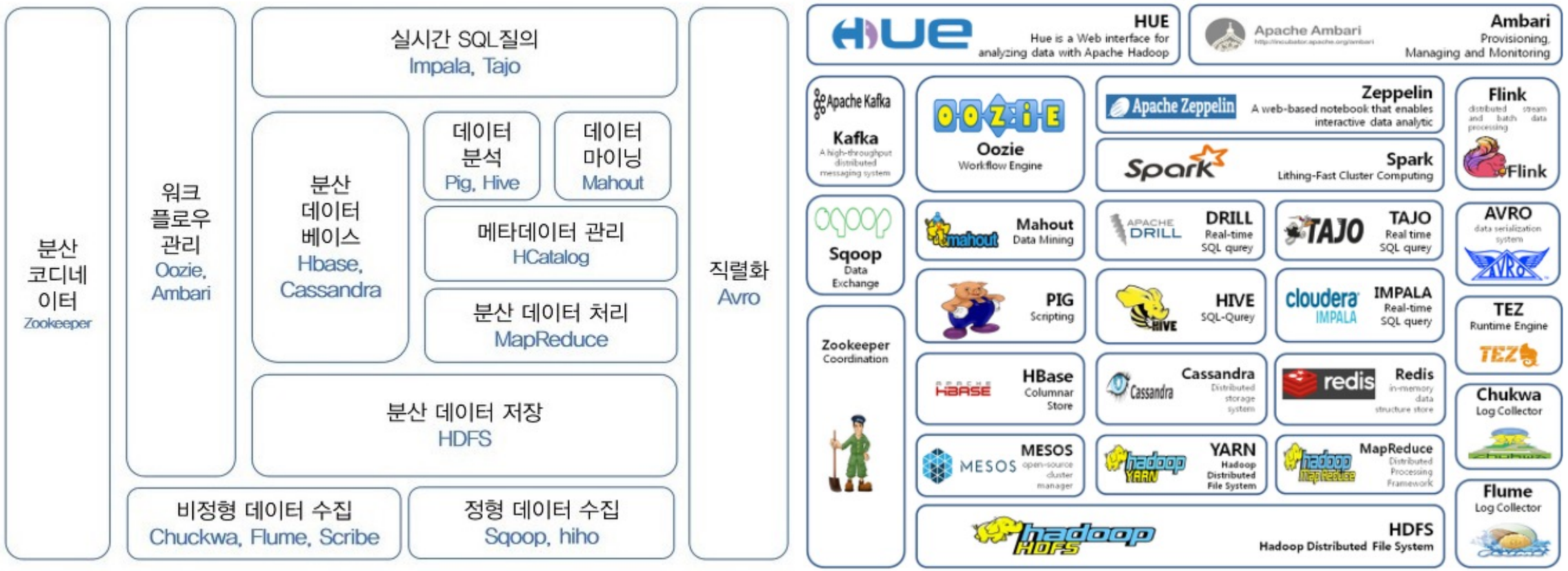
- 필요/요구에 따라 적절한 처리/저장/분석 파이프라인을 구성해야 함







Hadoop ecosystem



Hadoop ecosystem

- Data storage
  - Flume
    - 분산된 각 서버에 설치된 agent로부터 데이터를 전달받는 기능
    - 전체 데이터의 흐름을 관리하기 위해 master 서버가 존재
  - Kafka
    - 실시간 데이터, 스트림 데이터 관리를 위한 분산 메시징 시스템
    - Publish – subscribe 모델로 구성됨
  - HBase
    - NoSQL용 HDFS의 칼럼 기반 DB
    - 실시간 랜덤 조회 및 업데이트 기능 제공
    - MapReduce방식으로 처리가능

Hadoop ecosystem

- Data processing
  - Hive
    - Hadoop 기반의 data warehousing solution
    - SQL과 매우 유사한 HiveQL 쿼리 기능 제공 ➔ MapReuce job으로 변환 가능
  - Spark
    - In-memory 기반의 범용 데이터 처리 플랫폼
    - MR과 유사한 task처리 ➔ 메모리 기반 ➔ 빠른 속도
  - Mahout
    - Hadoop 기반 데이터 마이닝 알고리즘을 구현한 오픈소스
    - Classification, Clustering, Regression 등 주요 algorithm 지원

Hadoop ecosystem

- Visualization
  - Hue
    - Hadoop User Experience
    - Hadoop과 Hadoop ecosystem에 최적화된 웹 인터페이스 및 시각화 기능 제공
    - HiveQL 기반의 데이터 조회 및 시각화, job 스케줄링 가능
  - Zeppelin
    - Python, hive, spark 등 다양한 솔루션의 API를 제공
    - API기반의 웹 시각화 가능

Hadoop ecosystem

- Distribution management
  - Zookeeper
    - 분산 환경에서 서버들간의 상호 조정이 필요한 다양한 서비스를 제공하는 시스템
    - Load balancing
    - 데이터 동기화 ➔ 안정성 보장
    - Load scheduling

End of slide