

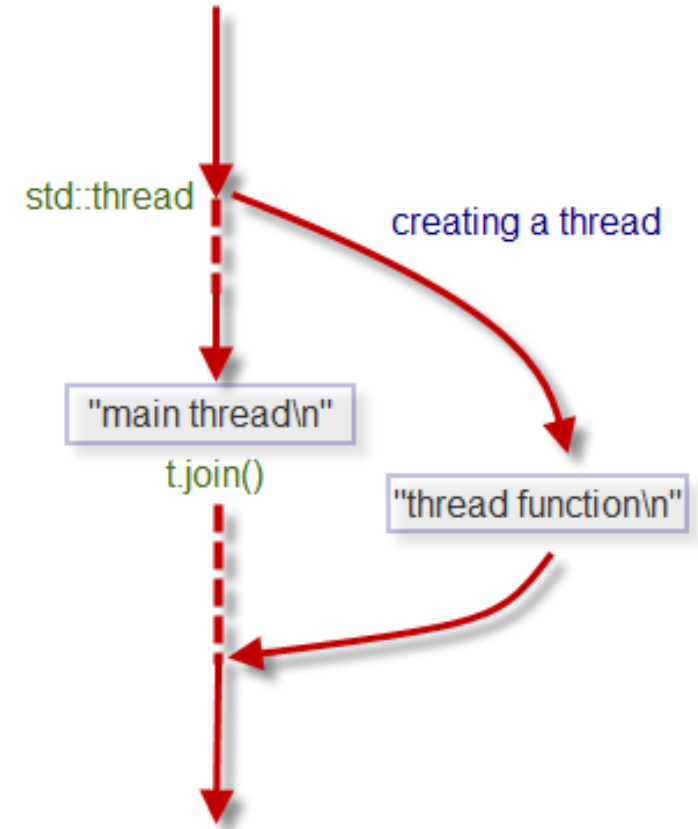
**CODINGO** x **posco**

**K-Digital Training** 스 마 트 팩 토 리

# Thread

# Thread

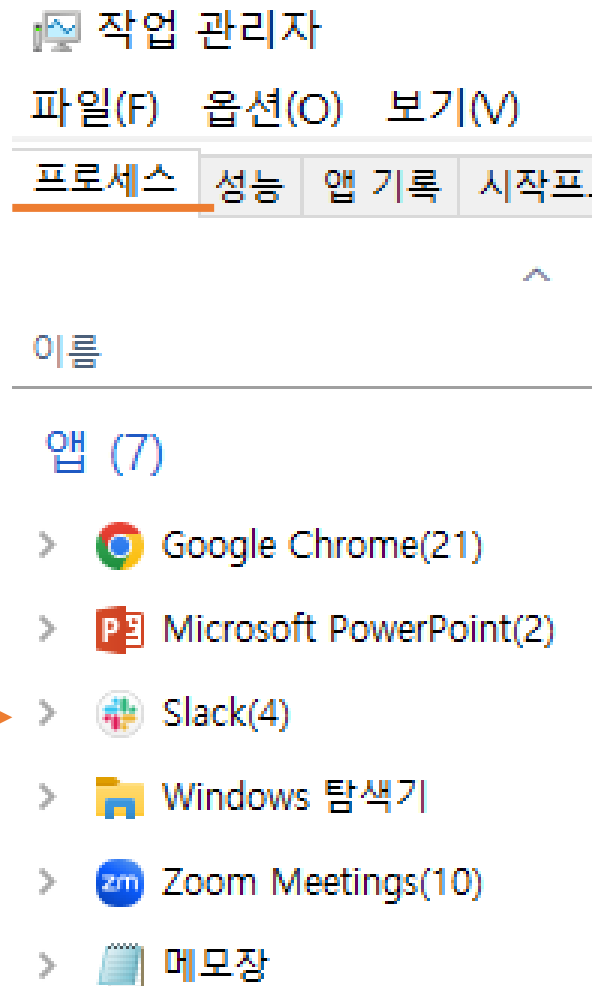
- 어떠한 프로그램 내에서, 특히 프로세스 내에서 실행되는 흐름의 단위
- 일반적으로 한 프로그램은 하나의 스레드를 가지고 있지만, 프로그램 환경에 따라 둘 이상의 스레드 동시에 실행가능
- 이러한 실행 방식을 **멀티스레드(multi-thread)** 라고 한다.



# Thread / Process

- Process
  - 운영체제에서 실행되고 있는 컴퓨터 프로그램
  - 운영체제로부터 시스템 자원을 할당 받는 단위
- Thread
  - Process 내부에서 실행되는 여러 흐름의 단위
  - Stack 영역만 할당받고, Code, Data, Heap 영역은 공유

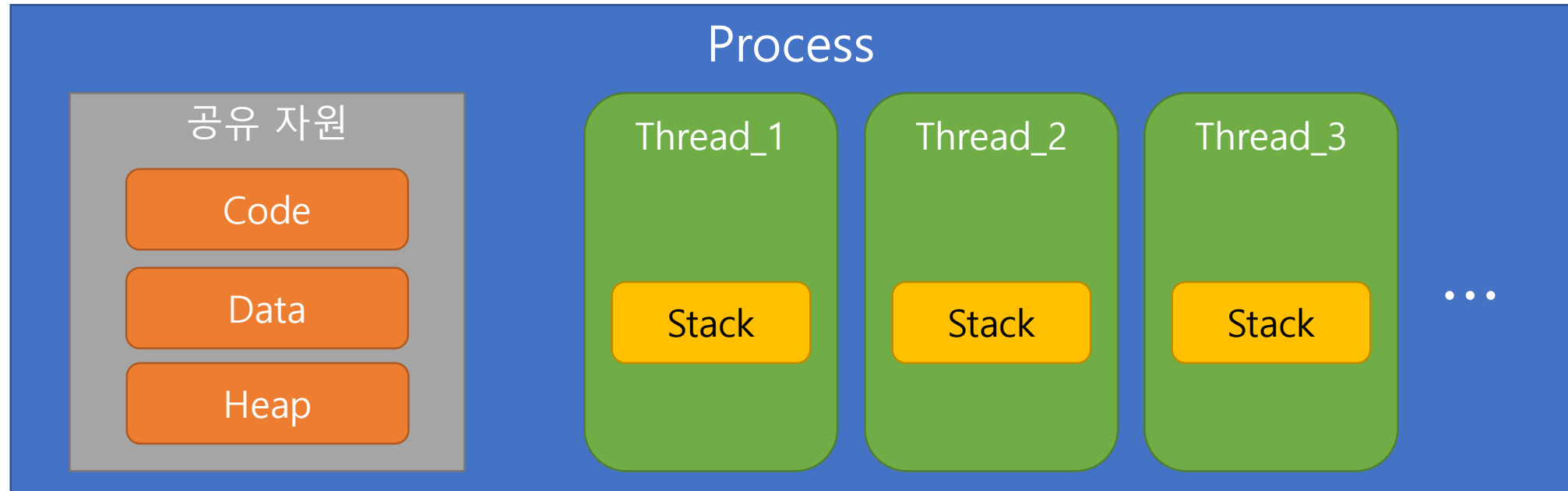
각각 모두 하나의 프로세스 →



# Thread

- **Thread**

- Process 내부에서 실행되는 여러 흐름의 단위
- Stack 영역만 할당받고, Code, Data, Heap 영역은 공유



# Multi-thread 사용 목적

## 1. 성능 향상

- 사실상 병렬로 계산이 이루어지므로 처리 속도가 빨라짐

## 2. 응답성 개선

- GUI 프로그래밍의 경우, UI 스레드와 기능 구동 스레드를 구분하여 응답성 개선  
리소스 공유

## 3. 리소스 공유

- 동일 프로세스 내 스레드들은 메모리와 파일 같은 자원을 공유하므로 프로세스 두 개를 작동시키는 것에 비해 효율적

## 4. 비동기 처리

- I/O 작업 등을 따로 처리하여 전체적인 효율성을 향상

# C++ Thread

- 기본적인 사용법 `std::thread`

```
#include <iostream>
#include <thread>

void printMessage() {
    std::cout << "Hello from thread!" << std::endl;
}

int main() {
    std::thread t(printMessage); // 새로운 스레드 생성
    t.join(); // 메인 스레드가 t 스레드가 종료될 때까지 대기
    return 0;
}
```

# C++ Thread

- 스레드에 입력 값을 전달하기

```
int main() {  
    std::thread t1(printMessage, 5); // 새로운 스레드 생성 및 입력 값 전달  
    std::thread t2(printMessage, 10); // 새로운 스레드 생성 및 입력 값 전달  
  
    t1.join();  
    t2.join(); // 메인 스레드가 t1, t2 스레드가 모두 종료될 때까지 대기  
    return 0;  
}
```



# Thread Scheduling

- 다중 스레드 환경에서 어떤 스레드가 CPU 를 할당 받고 실행될지를 관리하는 프로세스
- 스레드 스케줄링은 운영 체제 및 프로세서(CPU)에 따라 다를 수 있음
  - 운영체제 및 CPU 상황에 따라 각 스레드의 “실행 순서”가 매번 달라짐

# Thread Scheduling

- 주요 개념과 원칙
  1. 스레드 우선순위 (Thread Priority)
  2. 스케줄링 알고리즘
  3. 문맥 교환 (Context Switching)
  4. 우선순위 역전 (Priority Inversion)

# Thread Scheduling

## 1. 스레드 우선순위 (Thread Priority)

- 스레드가 CPU를 얼마나 자주 할당 받을지 결정하는데 사용
- 각 스레드는 **우선순위 값**을 가지며, 우선순위 값이 높은 스레드는 낮은 스레드보다 자주 실행될 확률이 높음

```
// 스레드 우선순위 설정
if (!SetThreadPriority(hThread, THREAD_PRIORITY_HIGHEST)) {
    std::cerr << "스레드 우선순위 설정 실패" << std::endl;
    return 1;
}
```

Thread\_1

Priority: 9

Stack

Thread\_2

Priority: 3

Stack

Thread\_3

Priority: 1

Stack

# Thread Scheduling

## 2. 스케줄링 알고리즘

- 라운드 로빈(순환 처리)
  - 각 프로세스에 동일한 시간 할당량(Time Quantum)을 주어 순차적으로 CPU를 사용할 수 있게 함
- 우선 순위 기반 스케줄링
  - 우선순위가 높은 프로세스가 먼저 CPU를 할당 받음
  - 단, 우선순위가 낮은 프로세스는 무한 대기 상태에 빠질 위험이 있음
- FCFS(First-Come, First-Served)
  - 가장 일반적인 스케줄링 알고리즘
  - 가장 먼저 요청한 프로세스가 실행됨
  - 작동 시간이 긴 프로세스가 먼저 도착하면 나머지 프로세스들의 실행이 늦어 짐

# Thread Scheduling

## 3. 문맥 교환 (Context Switching)

- CPU에서 실행 중인 스레드를 바꿀 때 문맥 교환을 수행
- 실행 중인 스레드의 상태를 저장하고 다음 실행할 스레드의 상태를 복원
- Ex.
  - 1번 스레드의 소스코드를 **20번째 줄**까지 실행
  - 갑자기 2번 스레드의 소스코드를 **1~10번째 줄**까지 실행
  - 1번 스레드로 돌아와서 **21번째 줄**부터 마지막 줄까지 실행
  - 2번 스레드로 돌아와서 **11번째 줄**부터 나머지 부분 실행

```
Thread 5: 10
: 3
Thread 1: 4
Thread 1: 5
Thread 1: 6
Thread Thread 1: 72: 3
Thread 2: 4
Thread
2: 5Thread 3: 2

Thread 1: 8
Thread 3: 3
Thread 2: 6
Thread 2: Thread 3: 4
Thread 2: 5
```

# Thread Scheduling

## 4. 우선순위 역전 (Priority Inversion)

- 공유 자원에 대한 접근이 잘못 관리되어, 낮은 우선순위의 스레드가 높은 우선순위의 스레드보다 먼저 실행되는 것

- Ex.

High, Mid, Low 세 개의 스레드가 각각 이름대로 우선 순위를 가질 때,  
Low 스레드가 공유 자원을 갖고 작업 중

High 스레드는 공유 자원을 사용하려고 대기 중

Mid 스레드가 실행되며 High 스레드보다 먼저 공유 자원을 차지함

(어쨌든 Low 스레드보다는 우선순위가 높으므로)

# Race condition

- 다중 스레드 또는 다중 프로세스 환경에서 여러 스레드 또는 프로세스가 공유된 자원에 동시에 접근하려고 할 때 발생하는 프로그래밍 문제
  - 의도치 않은 Context Switch
  - 의도치 않은 Priority Inversion
  - 등등..

# Race condition

- 주요 특징과 원인
  1. 공유 자원
  2. 동시 접근
  3. 올바른 순서 없음
- 그로 인한 문제
  1. 데이터 손실 또는 오염
  2. 데드락(deadlock)
  3. 비정상적인 동작



# Race condition

- 해결방법

- **std::mutex**

- 여러 스레드가 동시에 공유 데이터에 접근하는 것을 막기 위해 사용
    - lock, unlock

- **std::atomic**

- 원자적 연산을 수행하기 위한 도구, 공유 변수를 안전하게 업데이트
    - 여러 스레드 간에 동시에 변수 값을 수정하더라도 데이터 무결성을 보장

- **std::condition\_variable**

- 주로 스레드가 특정 조건을 만족할 때까지 대기하고, 조건이 충족되면 다른 스레드에게 신호를 보내고 깨우는데 사용
    - 일반적으로 mutex와 함께 사용


# C++ std::mutex

```
[-] #include <iostream>
    #include <thread>
    #include <mutex>
```

```
std::mutex mtx; // 뮉텍스 객체 생성
int counter = 0;
```

```
[-] void increment() {
[-]     for (int i = 0; i < 1000; ++i) {
        mtx.lock(); // 임계 구역에 들어가기 전에 잠금
        ++counter;  // 공유 자원에 접근
        mtx.unlock(); // 임계 구역을 벗어나면 잠금 해제
    }
}
```

# C++ std::mutex

```
 int main() {  
    std::thread t1(increment);  
    std::thread t2(increment);  
  
    t1.join();  
    t2.join();  
  
    std::cout << "Final counter value: " << counter << std::endl;  
  
    return 0;  
}
```

# C++ std::lock\_guard( )

- lock\_guard 객체 생성 시 Lock이 걸리고, 소스코드 생명주기가 끝나서 (중괄호를 벗어나거나, delete 되거나 등) 객체가 소멸하면 자동으로 unlock이 됨

```
int counter = 0;  
std::mutex mtx;
```

```
void incrementCounter() {  
    for (int i = 0; i < 1000; ++i) {  
        std::lock_guard<std::mutex> lock(mtx);  
        ++counter;  
    }  
}
```

# C++ std::atomic

- include <atomic> 필요
- std::atomic 객체를 생성하여 변수처럼 사용
- atomic으로 만들어진 객체가 사용 중일 때는 다른 스레드가 간섭할 수 없음
- 즉, 일반 변수는 counter++; 이 수행 될 때 내부적으로 counter = counter + 1; 이 수행되는데, 이 시점에 context switching이 발생하여 오류가 생길 수 있음

```
// 일반적인 변수 선언
int counter = 0;
// atomic으로 선언
std::atomic<int> counter(0);
```

# C++ std::atomic

```
int main() {
    const int num_threads = 10;
    const int num_iterations = 1000;
    std::vector<std::thread> threads;

    for (int i = 0; i < num_threads; ++i) {
        threads.push_back(std::thread(increment, num_iterations));
    }

    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Final counter value: " << counter << std::endl;

    return 0;
}
```

```
std::atomic<int> counter(0);

void increment(int iterations) {
    for (int i = 0; i < iterations; ++i) {
        ++counter;
    }
}
```

# C++ std::atomic

// 안전하게 값 삽입

```
counter.store(999);
```

// 안전하게 값 복사

```
int num = counter.load();
```

// 안전하게 값 비교

// counter의 값이 999면 true를 반환하고 30으로 값을 변경

```
int expected = 999;
```

```
bool result = counter.compare_exchange_strong(expected, 30);
```

// 반복문 안에서 돌거나, 성능이 중요하다면 weak를 사용

```
bool result = counter.compare_exchange_weak(expected, 30);
```

// counter에 10을 더하고 이전에 있던 값을 반환

```
int before = counter.fetch_add(10);
```

# C++ `std::unique_lock`

1. `std::lock_guard` 와 동일하게 사용 가능
2. `std::unique_lock` 객체 생성시 `std::defer_lock` 옵션을 추가하여 바로 Lock을 하지 않고 원하는 순간에 Lock 및 Unlock을 수행
3. 멤버 함수 `try_lock()` 을 사용하여 Lock을 시도하고 성공 여부 확인 가능
4. 멤버 함수 `release()` 를 사용하여 할당된 mutex를 반납 가능
  - mutex를 반납하고 다른 `lock_guard` 또는 `unique_lock` 객체에서 사용 가능



# C++ std::unique\_lock

```
std::mutex mtx;
```

- include <chrono> 필요

```
void try_to_lock(int id) {  
    std::unique_lock<std::mutex> lock(mtx, std::defer_lock); // lock을 바로 획득하지 않음  
  
    if (lock.try_lock()) { // lock을 시도  
        std::cout << "Thread " << id << " acquired the lock." << std::endl;  
        std::this_thread::sleep_for(std::chrono::seconds(1)); // 자원 사용 시뮬레이션  
  
        lock.unlock(); // lock 해제  
        std::cout << "Thread " << id << " released the lock." << std::endl;  
    }  
    else {  
        std::cout << "Thread " << id << " could not acquire the lock." << std::endl;  
    }  
}
```

# C++ std::unique\_lock

```
void use_release(int id) {  
    std::unique_lock<std::mutex> lock(mtx); // lock을 바로 획득  
    std::cout << "Thread " << id << " acquired the lock and will release it soon." << std::endl;  
    std::this_thread::sleep_for(std::chrono::seconds(1)); // 자원 사용 시뮬레이션  
  
    std::mutex* mtx_ptr = lock.release(); // lock을 해제하지만 뮤텍스는 유지  
    std::cout << "Thread " << id << " released the lock but retains the mutex." << std::endl;  
  
    // mtx_ptr을 사용하여 수동으로 뮤텍스를 잠글 수 있음  
    mtx_ptr->unlock();  
}
```

# C++ std::unique\_lock

```
int main() {  
    std::thread t1(try_to_lock, 1);  
    std::thread t2(try_to_lock, 2);  
    std::thread t3(use_release, 3);  
  
    t1.join();  
    t2.join();  
    t3.join();  
  
    return 0;  
}
```

# C++ `std::condition_variable`

- include `<condition_variable>` 필요
- 보통 `std::unique_lock` 과 함께 사용
- 여러 스레드 간 커뮤니케이션을 위해 사용
- **조건**을 걸어서 다른 스레드가 조건을 만족 할 때까지 대기하는 것이 가능

# C++ `std::condition_variable`

- **`notify_one()`**
  - 다른 스레드 하나를 작동시키고 조건을 검사하게 함
  - 두 개의 스레드만 이용할 때 자원 절약을 위해 사용
- **`notify_all()`**
  - 다른 모든 스레드를 작동시키고 조건을 검사하게 함
- **`wait( std::unique_lock<std::mutex>& lock, 조건식 )`**
  - `notify_one()` 또는 `notify_all()` 이 다른 스레드에서 호출되면 작동
  - 조건식을 확인 후 조건이 맞으면 밑에 코드를 마저 실행

# C++ std::condition\_variable

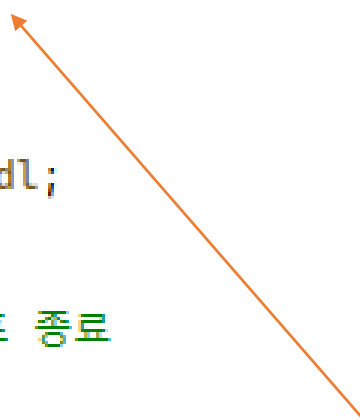
```
std::mutex mtx;
std::condition_variable cv;
std::queue<int> data_queue;
bool done = false;

void producer() {
    for (int i = 0; i < 10; ++i) {
        std::this_thread::sleep_for(std::chrono::milliseconds(500)); // 작업 지연 시간
        std::unique_lock<std::mutex> lock(mtx);
        data_queue.push(i);
        std::cout << "Produced: " << i << std::endl;
        cv.notify_one(); // 데이터가 준비되었음을 알림
    }
    // 작업 완료를 알림
    // unique_lock 객체의 생명 주기를 조절하기 위해 종괄호(코드 블록)를 사용
    {
        std::unique_lock<std::mutex> lock(mtx);
        done = true;
        cv.notify_all(); // 다른 모든 스레드를 깨우고 조건을 확인하게 함
        // condition_variable 객체를 사용하는 또 다른 스레드가 하나만 있을 경우
        // cv.notify_one(); 을 사용
    }
}
```

설정한 시간 만큼  
이 스레드만 쉬기

# C++ std::condition\_variable

```
void consumer() {  
    while (true) {  
        std::unique_lock<std::mutex> lock(mtx);  
        cv.wait(lock, [] { return !data_queue.empty() || done; }]); // 데이터가 준비될 때까지 대기  
        if (!data_queue.empty()) {  
            int value = data_queue.front();  
            data_queue.pop();  
            std::cout << "Consumed: " << value << std::endl;  
        }  
        else if (done) {  
            break; // 생산자가 작업을 마쳤음을 알림 받으면 루프 종료  
        }  
    }  
}
```



중괄호를 사용하여 소스코드를 전달하기 위한 **람다 함수 문법**  
return 된 값(!data\_queue.empty())이 true 또는 false 인지 체크

# C++ std::condition\_variable

```
int main() {  
    std::thread producer_thread(producer);  
    std::thread consumer_thread(consumer);  
  
    producer_thread.join();  
    consumer_thread.join();  
  
    return 0;  
}
```