

# 멀티코어프로그래밍

2024 가을학기

한국공학대학교 게임공학과

정내훈



# 1. 소개

멀티쓰레드 프로그래밍

정내훈

# 내용

---

- 강좌 소개
- “멀티쓰레드 프로그래밍”에 대한 소개
- 간단한 멀티쓰레드 프로그램 작성법
- Case Study

# 강사 소개

- KAIST 전산과 박사
  - 전공 : 멀티프로세서 CPU용 일관성 유지 HW
- NCSoft 근무
  - Alterlife 프로그램 팀장
  - Project M(현 Blade & Soul) 프로그램 팀장
  - CTO 직속 게임기술연구팀
- 현 : 한국공학대학교 게임공학과 부교수
  - 게임서버프로그래밍, 멀티코어프로그래밍
- 최근 연구
  - 넷마블 리니지2레볼루션 서버 멀티스레드 안정화
  - 서울 지하철 승무원 배치 알고리즘 멀티쓰레드 최적화
- 멀티코어 프로그래밍 강의
  - 삼성전자
  - 넷마블

# 강의 목적

- 현업의 요구
  - 고 사양 게임은 100% 멀티코어 활용
    - 4-Core 컴퓨터에서 4배의 FPS 또는 동접
- 면접 시 기출문제
  - 게임 제작회사에서 필요성 상승

# 강의 목적

## ● 현업의 요구(넥슨 신입, 2023-09-01)

The screenshot shows the Nexon recruitment page with the following job openings and their requirements:

Job Title	Requirements
[마케팅] 클라이언트 프로그래머	신입 게임프로그래밍
[메이플스토리M] 클라이언트 프로그래머	신입/경력 게임프로그래밍
[메이플스토리 월드] 게임서버 프로그래머	신입/경력 게임프로그래밍
[메이플스토리 월드] 유니티 테크니컬 프로그래머	신입/경력 게임프로그래밍
[메이플스토리 월드] 프론트엔드 프로그래머	신입/경력 게임프로그래밍
[메이플스토리 월드] 게임로직 프로그래머	신입/경력 게임프로그래밍
[메이플스토리 월드] 코어엔진 프로그래머	신입/경력 게임프로그래밍
[메이플스토리 유니버스] 메이플스토리N 게임 서버/클라이언트 프로그래머 (C/C++)	신입/경력 게임프로그래밍
[인텔리전스랩스] 매치메이킹 서버 개발자	신입/경력 게임프로그래밍
[국내메이플] 게임 프로그래머	신입/경력 게임프로그래밍

At the bottom, there is a banner for "인재 Pool 등록" (Talent Pool Registration) with the text: "포지션 발생 시 적합한 후보자에게 제안 드립니다." (We will propose to suitable candidates when a position arises.)

### 지원자격

- Unity, C#, VC++을 이용하여 개발 가능하신 분
- Unity의 NGUI 개발 가능하신 분
- 2D/3D 프로그래밍에 대한 이해가 있으신 분
- 네트워크 프로그래밍과 멀티 스레드 환경에 대한 이해가 있으신 분

### 지원자격

- C# 언어 사용에 능숙하신 분
- 네트워크 프로그래밍, 멀티 스레드 프로그래밍 경험이 있으신 분

### 지원자격

- C# 언어 사용에 능숙하신 분
- 유니티 사용에 능숙하신 분
- 자료구조 및 알고리즘의 이해가 있으신 분

### 지원자격

- C# 언어 사용에 능숙하신 분
- 멀티 스레드 프로그래밍 경험이 있으신 분
- 유니티를 사용한 개발 경험이 있으신 분
- CPU 및 메모리 성능 최적화된 로직과 구조에 관심이 많으신 분
- 디버깅 및 프로파일링을 통해 문제 원인 파악 및 해결 경험이 있으신 분
- 대규모 코드 기반으로 효율적으로 작업하며 기존 시스템의 리팩토링 및 최적화 경험이 있으신 분

# 교재

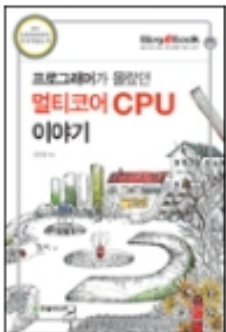
- 교재 : 용어 정의 및 예제
  - “The Art of Multiprocessor Programming”  
Second Edition, Morgan Kaufmann, 2021
    - 번역:모리스 헐리히, 니르 샤비트 “멀티프로세서 프로그래밍” 한빛미디어, 2009
    - 장점 : 멀티프로세서 프로그래밍의 기초를 전부 엄밀하게 다루고 있다.
    - 단점
      - 알고리즘 위주로 실제 구현과는 차이가 있다.
      - 한글판의 경우 치명적인 번역오류
      - 게임과는 맞지 않는 **Java**를 사용한 구현

# 교재

## ● 좀더 쉬운 책

- 강의와는 상관 없음.
- 앞의 책이 너무 어려울 경우 읽으면 많은 도움이 됨

2



[도서] **프로그래머가 몰랐던 멀티코어 CPU 이야기** -Blog2Book 009

김민장 저 | 한빛미디어 | 2010년 05월

22,000원 → **19,800원**(10% 할인) | YES포인트 1,100원(5%지급)

**도착 예상일** : 지금 주문하면 **오늘** 도착예정

판매지수 1,671 | 회원리뷰 (6개) | 내용 ★★★★★ 편집/구성 ★★★★★

관련상품 중고상품 **9개** / eBook **13,200원** 구매

[미리보기](#)

1

[카트](#)

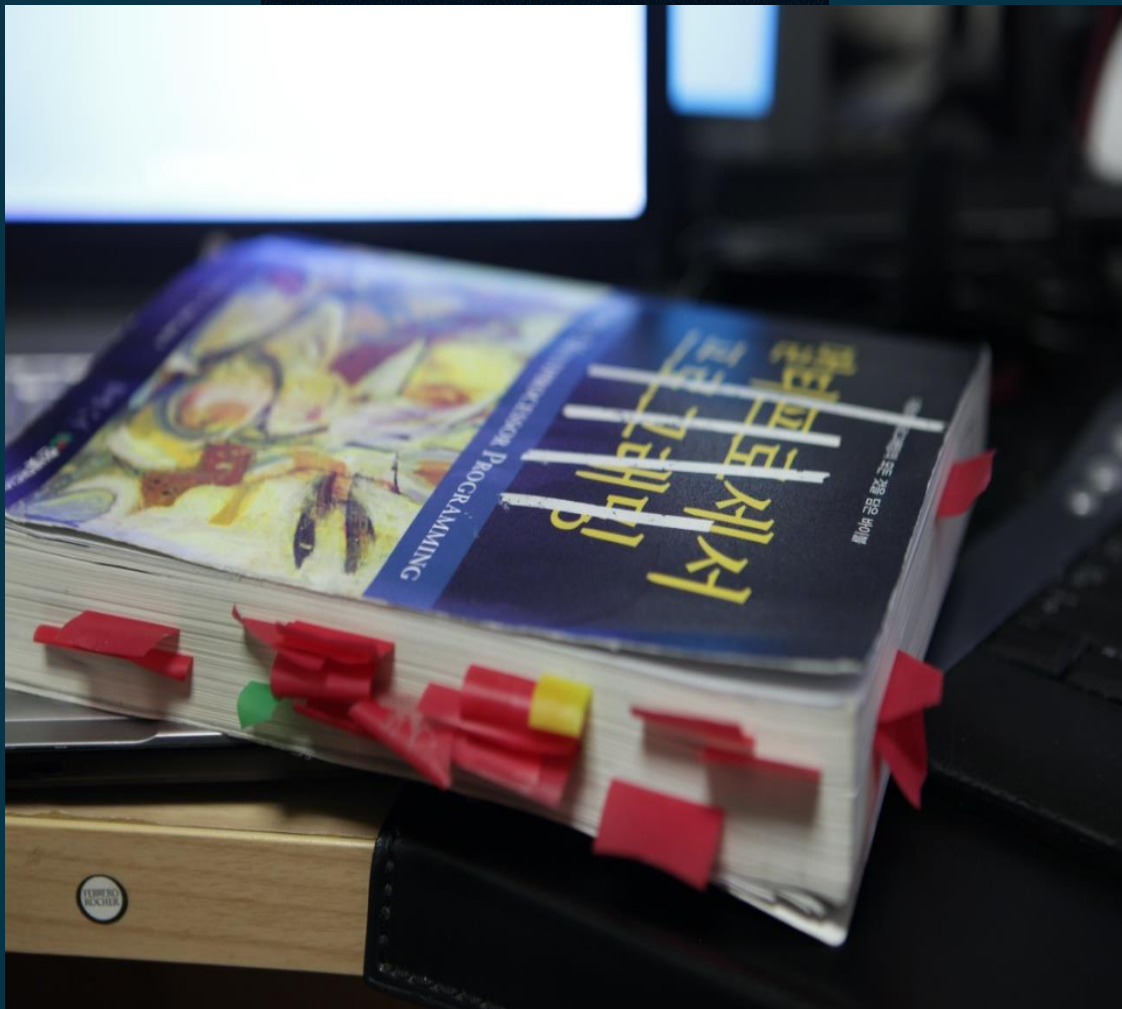
[리스트](#)

[바로구매](#)



# 현실

- 좋은 교재??



# 현실

- 좋은 교재??

- 게임프로그래머가 번역 (넥슨, NCSoft 재직)
- 좋은 평판
- 많은 사람들이 Study를 시도
- 1달 이상 지속되는 스터디를 본적이 없음
- 문제는 난이도....

각오가 필요

- 더 중요한 건 아는 사람의 지도

# 강의 계획

		내용
1	1-2주	강의 소개, 환경 설정, 개요
	3-4주	멀티쓰레드 프로그램 실습, 컴파일러, Data Race
2	5-6주	메모리 일관성, Atomic, Non-Blocking
	7-8주	동기화 연산, CAS
3	9 주	Non-Blocking 자료구조 작성 실습 - LIST
	10 주	Non-Blocking 자료구조 작성 실습 - LIST
4	11 주	Non-Blocking 자료구조 작성 실습 - QUEUE
	12 주	Non-Blocking 자료구조 작성 실습 - STACK
5	13 주	Non-Blocking 자료구조 작성 실습 - SKIP-LIST
	14-15 주	상용 라이브러리 리뷰 및 총정리, 평가

# 강의 평가

- 과제 30% (대면 수업 시 실습으로 대체)
  - 1차 표절 -100% 점수
  - 2차 표절 F학점
    - 오류까지 그대로 복사하는 것은 너무 명확한 증거
    - “같이 협력해서 작성 했는데요.” -표절
    - “보여주기만 했는데요” -표절
- 중간고사 30%
- 기말고사 30%
- 출석 10%

# 강의 계획

- 강의의 흐름

1. 멀티쓰레드 프로그래밍의 필요성
2. 멀티쓰레드 프로그램 작성시 문제점
3. Non-Blocking 알고리즘의 필요성
4. Non-Blocking 알고리즘 작성 테크닉 및 실습
5. 정리 및 상용라이브러리 소개

# 선수 과목

- C 프로그래밍 언어
- C++11 프로그래밍 언어
  - STL 필요
- 자료구조
- 알고리즘
- 컴퓨터구조
  - CPU 명령어 구조, Pipeline
  - Cache, Out-of-order execution, multiprocessor
- 운영체제
  - 프로세스, 스레드

# 개발 환경

---

- 운영 체제
  - Windows 10/11
- 컴파일러
  - Visual Studio 2022 community
- 컴퓨터
  - 일반 PC (x86 멀티코어 아키텍처, 4 core 이상)

# 개발 환경

- Linux?

- 똑같다, C++11 덕분

- 모바일???? (IOS, Android)

- CPU가 다르다. (x86 vs ARM)

- 하지만 똑같다. C++11 덕분
    - 내부 기계어 구현에 차이점이 있을 뿐, 7주에 다름.

- OSX?

- C++11 덕분에 같음.

- 멀티쓰레드 프로그래밍에 운영체제의 역할은 거의 없음 => 모든 것은 CPU에 달려 있음.



# 개발 환경

- 실습

- 실습환경 테스트
- Windows 기동
- Quad Core 이상인가 확인
- Visual Studio 2022 실행
- HelloWorld.cpp 작성 / 컴파일 / 실행

# 개발 환경

## ● 실습 #1

- HelloWorld.cpp 프로그램을 작성하고 컴파일 후 실행하시오
- HelloWorld.cpp
  - “HelloWorld!” 라는 문장을 출력하는 프로그램
- 목적
  - 에디터 사용
  - 컴파일러 사용
  - 프로그램 실행

# 내용

---

- 강좌 소개
- “멀티쓰레드 프로그래밍”에 대한 소개
- 간단한 멀티쓰레드 프로그래밍 작성법
- Case Study

# 컴퓨터

- (직렬) 컴퓨터 (Serial computer, Single Core 컴)
  - 하나의 CPU(또는 core)만을 갖는 컴퓨터
  - 듀얼코어가 대중화 되기 이전의 대부분의 컴퓨터
  - 여러 분이 배운 자료구조/알고리즘은 전부 직렬 컴퓨터를 가정하고 있음.
  - **현재 멸종 : 싱글코어 컴퓨터를 본 적이 있는 사람???**
- 병렬 컴퓨터 (Parallel Computer)
  - 여러 개의 CPU(또는 core)가 명령들을 실행하는 컴퓨터
  - 직렬 컴퓨터의 속도 제한을 극복하기 위해 제작
  - **여러분이 작성하는 프로그램이 실행되는 컴퓨터**

# 병렬 컴퓨터

- 병렬 컴퓨터의 사용 목적

- (X) 여러 개의 작업을 빨리 실행하기 위해서
  - 여러 대의 컴퓨터를 사용해도 됨 => 더 싸다
- 하나의 작업을 보다 빨리 실행하기 위해서.

- 병렬 프로그램

- 병렬 컴퓨터에서 실행했을 때 속도가 증가하는 프로그램
  - 기존의 프로그램을 병렬 컴퓨터에서 실행했을 경우 속도 증가는 **0%**
  - 병렬 알고리즘을 사용해서 새로 프로그래밍 해야 함.

# 병렬 프로그램

- 병렬 프로그램의 특징

- 실행된 프로그램(프로세스)의 내부 여러 곳이 동시에 실행됨
- 병렬로 실행되는 객체(Context로 불림)사이의 협업(동기화 또는 **synchronization**)이 필수
- 크게 공유메모리 (Shared Memory) 모델과 메시지 패싱(Message Passing) 모델이 있음

우리가 다루는 것은 공유 메모리 모델

# 병렬 프로그램

- 병렬 프로그램 요구사항 : 정확성과 성능
  - 여러 흐름(Context)에서 **동시 다발적으로** 호출해도 문제 없이 실행되는 알고리즘이 필요.
    - 오류가 발생하면 모든 것이 의미 없음.
  - Context의 증가에 따른 **성능 향상**이 높아야 한다.
    - 잘하지 못하면 기존의 직렬 프로그램보다 느려질 수 있다.
    - 코어의 개수 이상의 성능 향상은 불가능 하다.

# 멀티쓰레드 프로그래밍

- 멀티쓰레드 프로그래밍

- 현재 운영체제에서 지원하는 병렬 프로그래밍의 (유일한) 구현 수단
- 하나의 프로세스 안에서 여러 개의 쓰레드를 실행 시켜 병렬성을 얻는 프로그래밍 방법
- Windows, Linux, Android, iOS에서 기본으로 제공하는 유일한 병렬 프로그래밍 API
  - HW와 운영체제가 직접 지원하는 것은 이것 뿐
  - 다른 API로는 GPGPU가 있음.



# 멀티 쓰레드 프로그래밍

## ● 복습 – 프로세스 (1/2)

- 운영체제는 사용자의 프로그램을 프로세스 단위로 관리한다.
- 실행파일의 실행 => 운영체제가 파일내용을 메모리에 복사 후 시작 주소로 점프하는 것.
  - 읽어 들일 때 여러 가지 초기화가 필요하다.
- 시분할 운영체제는 여러 프로세스를 고속으로 번갈아 가면서 실행한다.
  - 실행중인 프로세스의 상태를 강제로 준비(ready)로 변경가능

# 멀티 쓰레드 프로그래밍

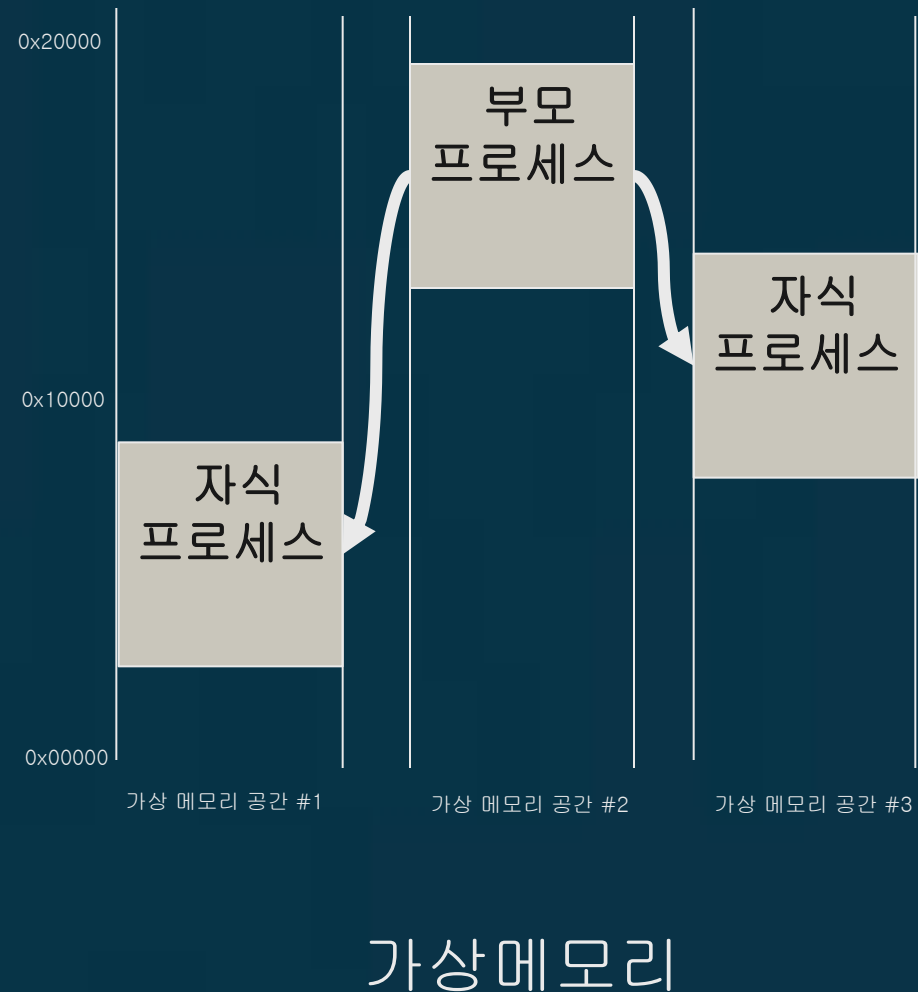
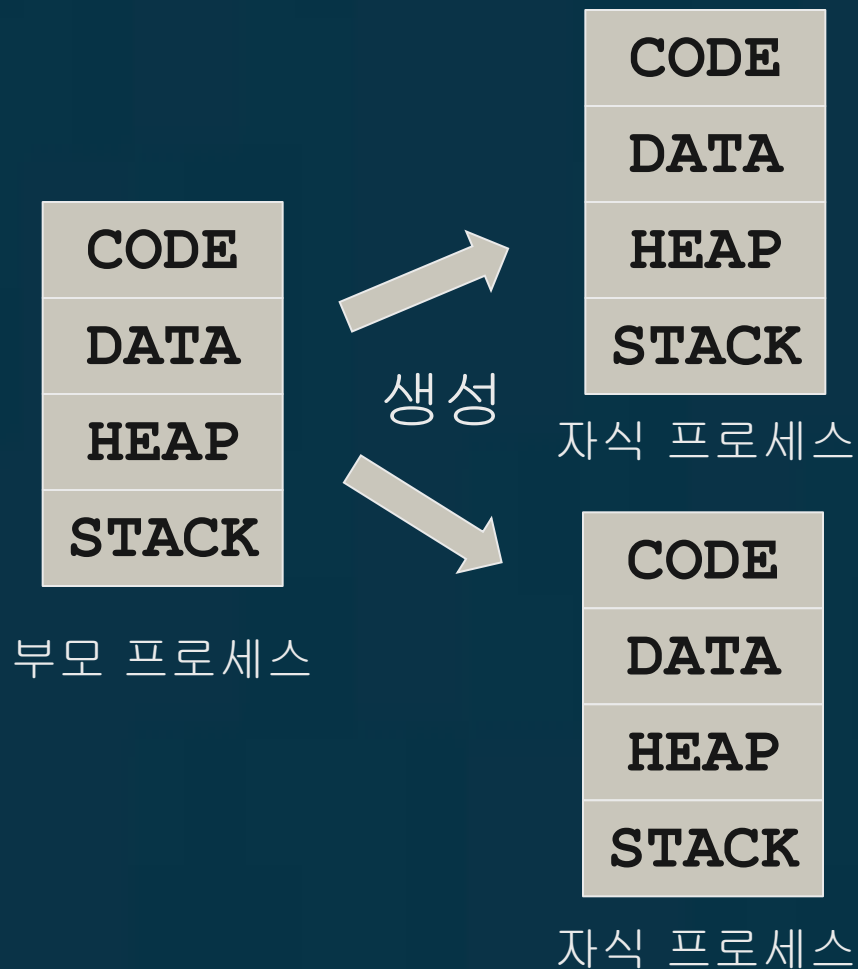
## ● 복습 – 프로세스 (2/2)

– 프로세스의 메모리구조는 다음과 같다.

- Code : 실행될 명령어가 들어가는 구역
- Data : 전역변수가 들어가는 구역
- Stack : 지역변수와 함수 리턴 주소가 들어가는 구역
- Heap : malloc이나 new로 할당 받은 메모리가 들어가는 구역
- PCB : Process Control Block

# 멀티 쓰레드 프로그래밍

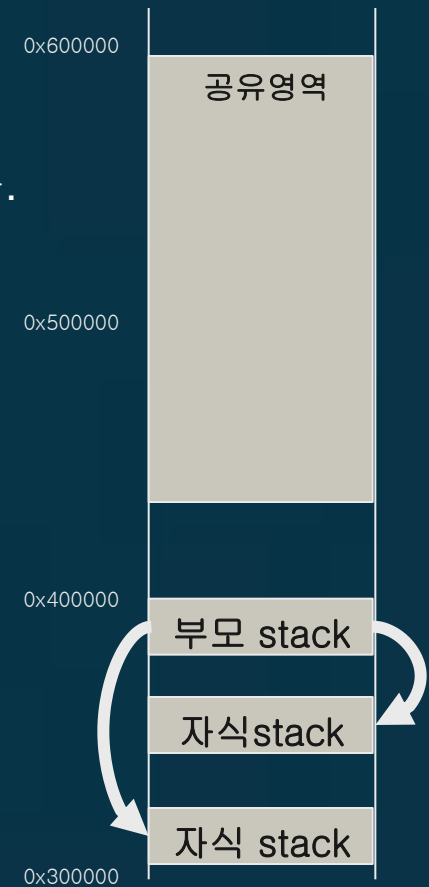
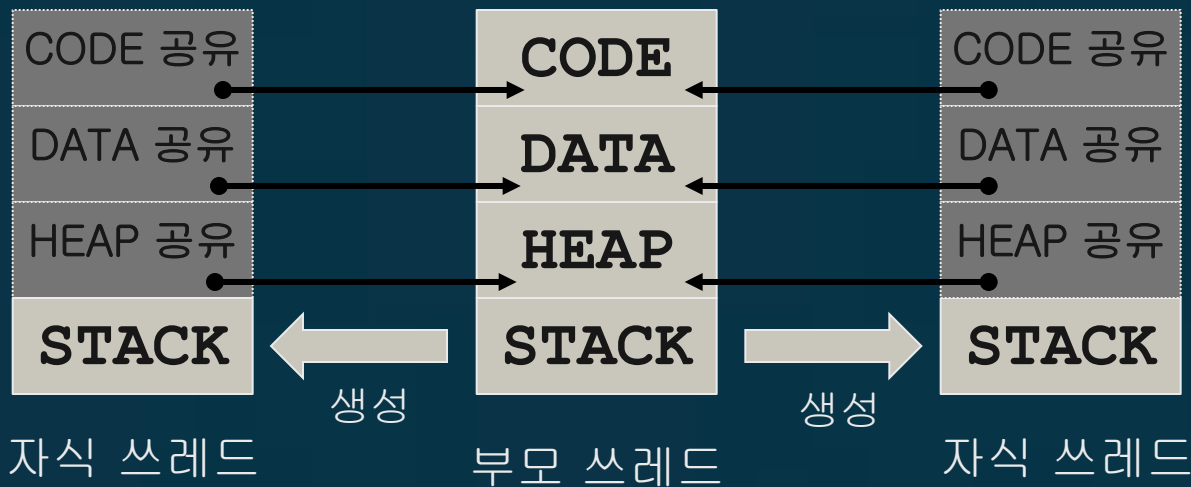
## ● 프로세스의 생성



# 멀티 쓰레드 프로그래밍

## ● 쓰레드

- 프로세스는 초기에 하나의 시작 쓰레드를 가짐
- 쓰레드는 다른 쓰레드를 만들 수 있다.
- 쓰레드 생성은 프로그래머가 프로그램에 적은 대로 이루어진다.
- 모든 쓰레드는 자신 고유의 스택을 가지고 있고, 나머지를 공유한다.
- (옆 쓰레드의 **stack**은 접근 불가인가??)



가상메모리

# 멀티 쓰레드 프로그래밍

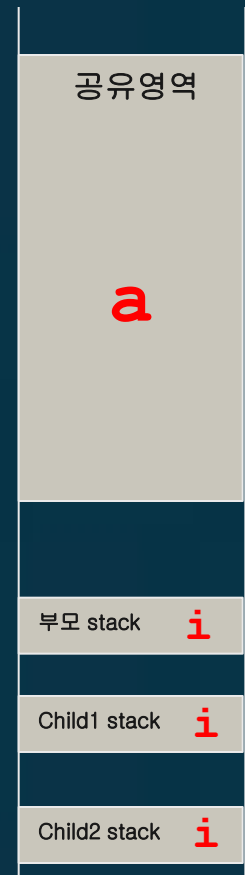
## ● 멀티 쓰레드의 메모리 공유 <실습필요>

```
int a;

void func()
{
    for (int i = 0; i < 10; ++i) a = a + i;
}

int main()
{
    thread child1(func);
    thread child2(func);
    func();
}
```

- **a**의 값은 모든 쓰레드가 공유
  - **a**의 주소 값은 모든 쓰레드에서 같은 값
- **i**의 값은 모든 쓰레드가 독자적으로 보유
  - **i**의 주소 값은 동시에 실행되는 모든 쓰레드에서 다른 값



가상메모리

# 멀티 쓰레드 프로그래밍

- 프로세스의 대한 쓰레드의 장점
  - 생성 Overhead가 적다.
  - Context Switch Overhead가 적다.
    - Virtual memory (TLB 교체 오버헤드)
  - Thread간의 통신이 간단하다.
- 쓰레드의 단점
  - 하나의 쓰레드에서 발생한 문제가 전체 프로세스를 멈추게 한다.
  - 디버깅이 어렵다.

# 두번째 시간

---

- 멀티 코어 HW소개
- 멀티쓰레드 프로그래밍 설명

# 멀티 쓰레드 프로그래밍

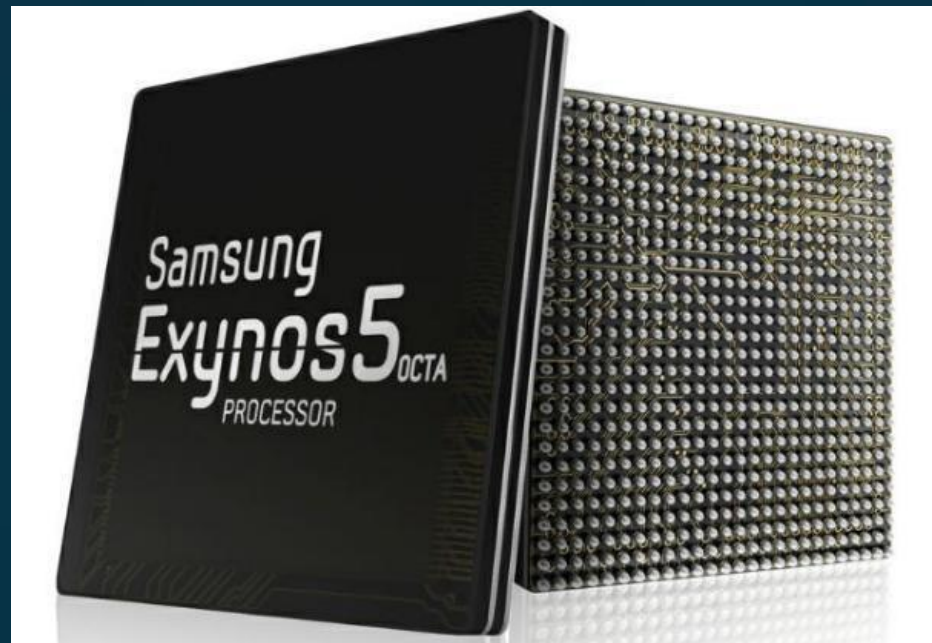
- 멀티 쓰레드의 사용 목적
  - 멀티 코어 CPU에서의 프로그램 성능향상 (O)
  - 멀티 CPU 컴퓨터에서의 프로그램 성능향상 (O)
  - 분산 컴퓨터에서의 프로그램 성능향상 (X)
  - 싱글 코어(CPU)컴퓨터 환경에서의 프로그램 성능향상 (X)
  - 프로그램을 모듈화 해서 알아보기 쉽게 (X)

병렬컴퓨터에서의 프로그램 성능향상



# 멀티코어 CPU?

- 한 개 이상의 코어로 구성된 CPU를 뜻함.
  - 예) 여러분이 쓰고 있는 컴퓨터 (i3, i5, i7, Ryzen)
  - 예) 갤럭시S24, iPhone15, Playstation 5
- 현재 멀티코어가 아닌 CPU가 존재하는가?
  - 여러분의 컴퓨터는?



# 멀티프로세서

## — 현재

### Steam Hardware & Software Survey: July 2024


PC Physical CPU details

Sort By:  Sort

PHYSICAL CPUS (WINDOWS)	MAR	MAY	JUL	
1 cpu	0.05%	0.06%	<b>0.06%</b>	0.00%
2 cpus	5.16%	5.30%	<b>5.00%</b>	0.00%
3 cpus	0.29%	0.27%	<b>0.27%</b>	0.00%
4 cpus	19.30%	19.85%	<b>18.47%</b>	0.00%
5 cpus	0.03%	0.03%	<b>0.03%</b>	0.00%
6 cpus	35.69%	33.94%	<b>33.67%</b>	0.00%
7 cpus	0.00%	0.00%	<b>0.00%</b>	0.00%
8 cpus	19.49%	20.87%	<b>19.95%</b>	0.00%
9 cpus	0.00%	0.01%	<b>0.00%</b>	0.00%
10 cpus	5.60%	5.37%	<b>6.44%</b>	0.00%
11 cpus	0.00%	0.00%	<b>0.00%</b>	0.00%
12 cpus	5.49%	5.47%	<b>5.62%</b>	0.00%
13 cpus	0.00%	0.00%	<b>0.00%</b>	0.00%
14 cpus	3.61%	3.89%	<b>4.00%</b>	0.00%
15 cpus	0.00%	0.00%	<b>0.00%</b>	0.00%
16 cpus	3.37%	3.61%	<b>3.82%</b>	0.00%
17 cpus	-	0.00%	<b>0.00%</b>	0.00%
18 cpus	0.05%	0.05%	<b>0.05%</b>	0.00%
19 cpus	-	-	<b>0.00%</b>	0.00%
20 cpus	0.45%	0.60%	<b>0.86%</b>	0.00%
21 cpus	0.00%	-	<b>0.00%</b>	0.00%
22 cpus	0.00%	0.01%	<b>0.00%</b>	0.00%
23 cpus	0.00%	0.00%	<b>0.00%</b>	0.00%
24 cpus	1.36%	1.60%	<b>1.70%</b>	0.00%
26 cpus	-	0.00%	<b>0.00%</b>	0.00%
28 cpus	0.02%	0.01%	<b>0.01%</b>	0.00%
32 cpus	0.01%	0.01%	<b>0.01%</b>	0.00%

# 멀티코어프로세서

- Intel과 ARM에서 멀티 코어 프로세서를 만드는 이유
  - CPU의 성능을 올려야 한다.
    - 안 그러면 아무도 안 산다. => 파산!
  - 클럭 속도를 높일 수 없다.
    - 발열 문제 <물리법칙>
  - 클럭 속도 말고 CPU의 속도를 올리는 법
    - 아키텍처 개선 : 캐시, 파이프라인, 예측 분기, 동적수행.....
      - 한계에 부딪힘
  - 남은 방법
    - 멀티 코어 (또는 Ctrl-C & Ctrl-V)



14th Gen Intel Core (**NDA, do not share!**)

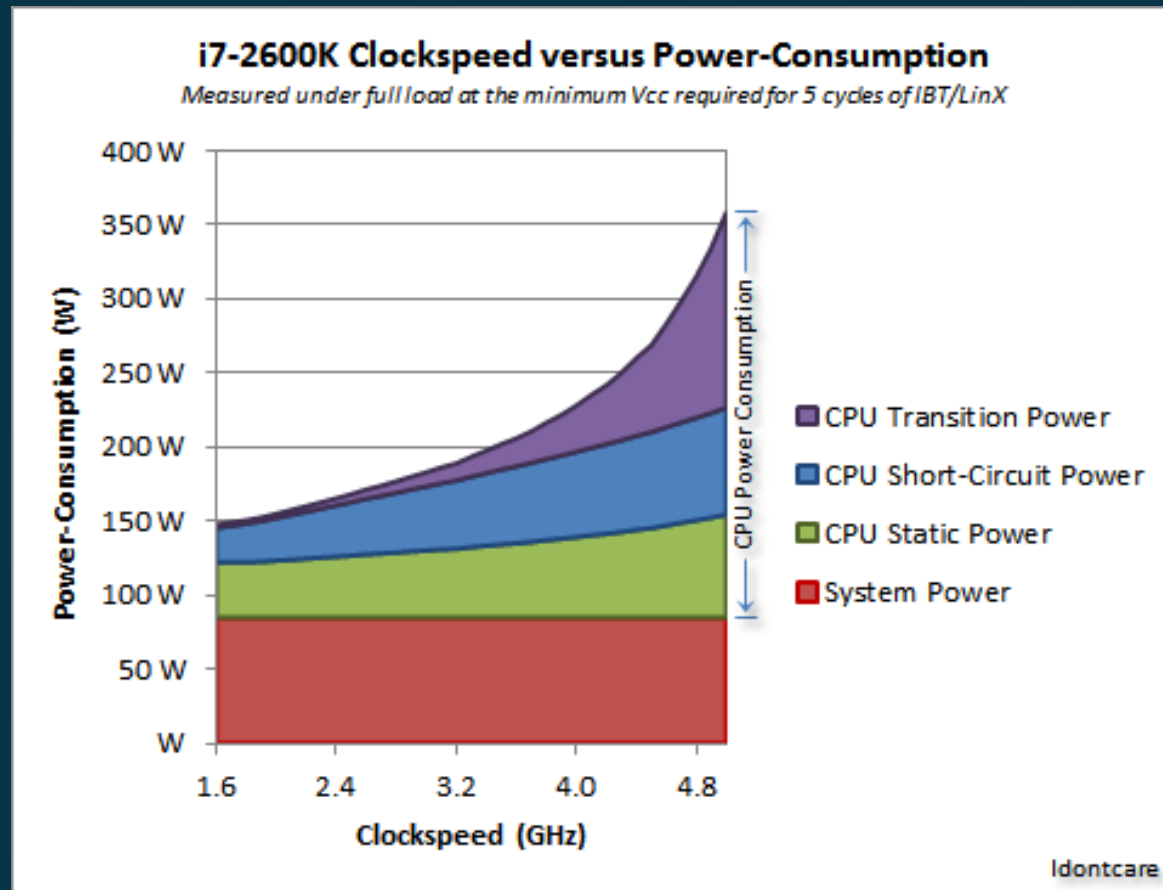
- 13th Gen = Raptor Lake-S, 14th Gen = **Raptor Lake-S Refresh**
- **No architectural changes**, same 'Intel 7' (10nm) process
- On average **~3% faster** than 13th Gen same segment SKU
- **i7-14700K** on average **~17% faster** (MT) because of **extra E-cores**

msi

13th Gen		14th Gen	
SKU	Core count	SKU	Core count
i5-13600K	6P + 8E	i5-14600K	6P + 8E
<b>i7-13700K</b>	<b>8P + 8E</b>	<b>i7-14700K</b>	<b>8P + 12E</b>
i9-13900K	8P + 16E	i9-14900K	8P + 16E

# 멀티코어프로세서

- Intel과 ARM에서 멀티 코어 프로세서를 만드는 이유



# 멀티코어프로세서

## ● 5GHz 달성????

### FX PROCESSORS SKU INFORMATION



AMD

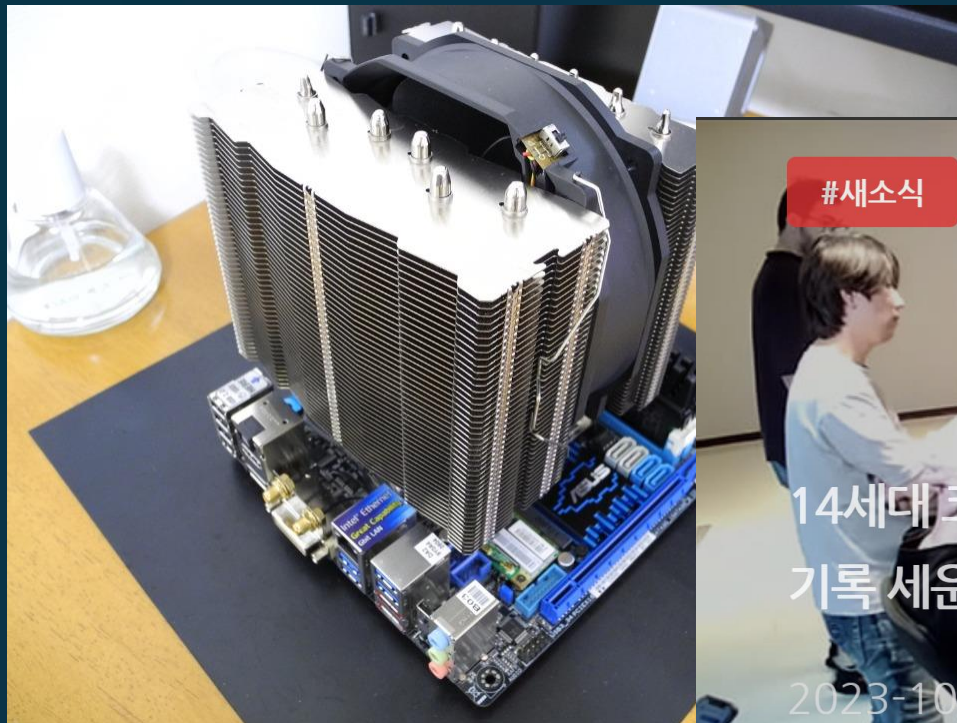
Model	FX-9590	FX-9370	FX 8350
"Piledriver" CPU Cores	8	8	8
Max Turbo	5.0 GHz	4.7 GHz	4.2 GHz
CPU Base	4.7 GHz	4.x GHz	4.0 GHz
TDP	220 W	220 W	125 W
L2 Cache	8 MB	8 MB	8 MB
L3 Cache	8 MB	8 MB	8 MB
Max DDR3	1866	1866	1866
AMD Turbo CORE 3.0	Yes	Yes	Yes
Unlocked	Yes	Yes	Yes





# 멀티코어프로세서

## ● 5GHz 달성????



9GHz 달성

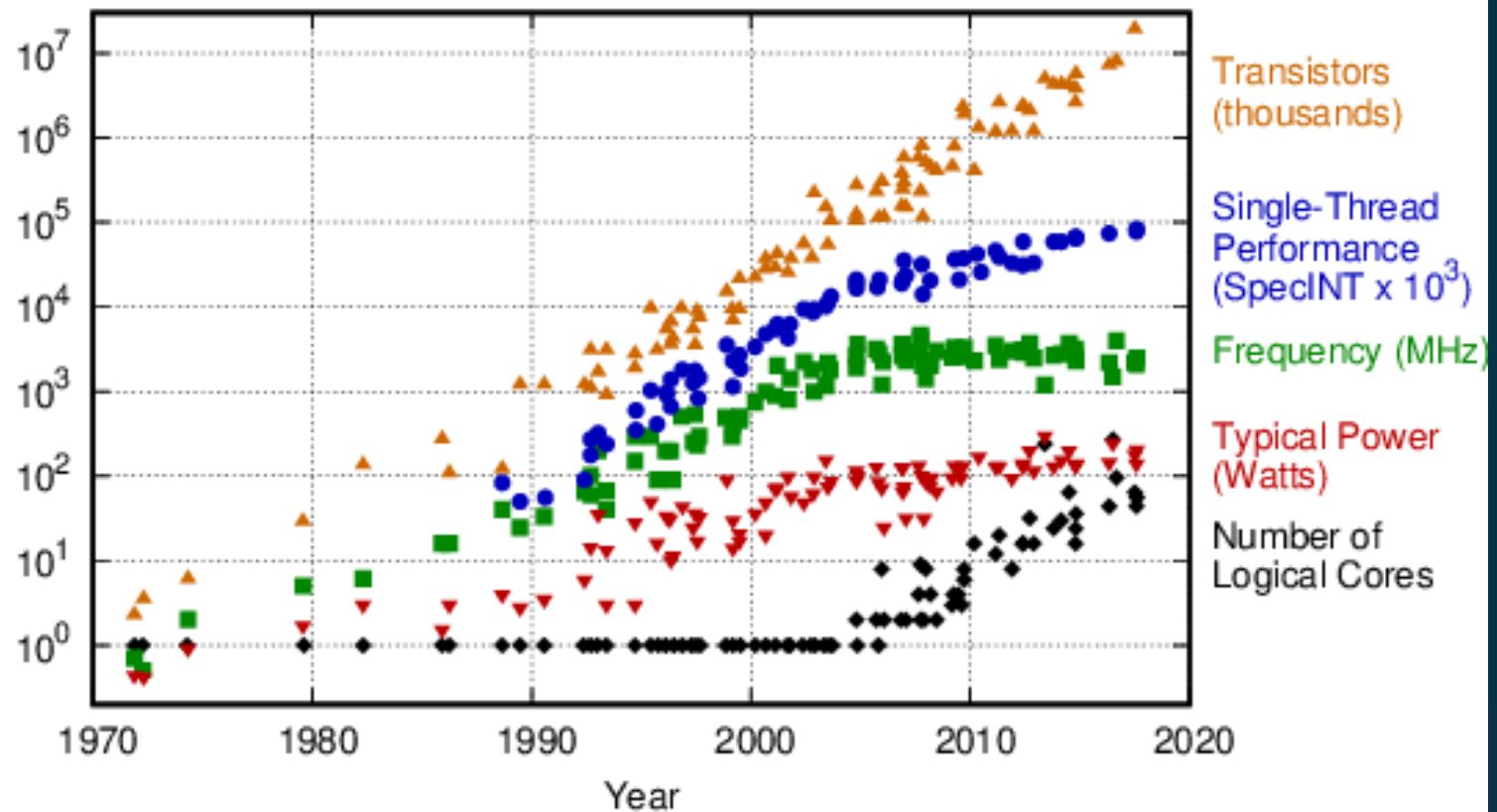


#새소식

14세대 코어 i9-14900KF에서 작동 클럭 9.044GHz  
기록 세운 에이수스

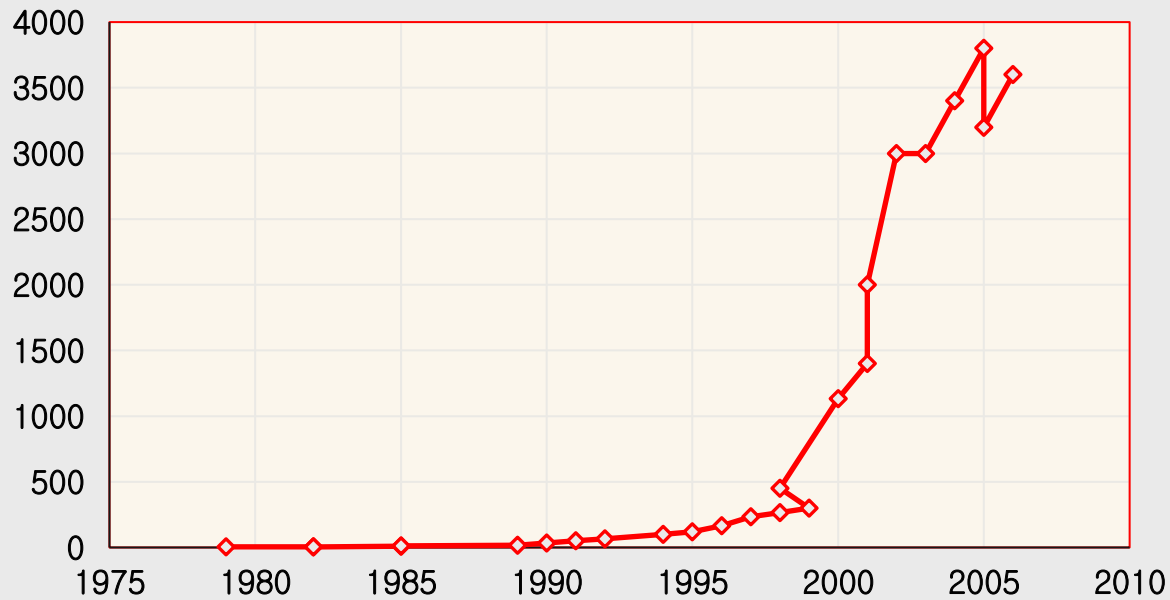
2023-10-18

# 멀티코어프로세서



# 멀티코어프로세서

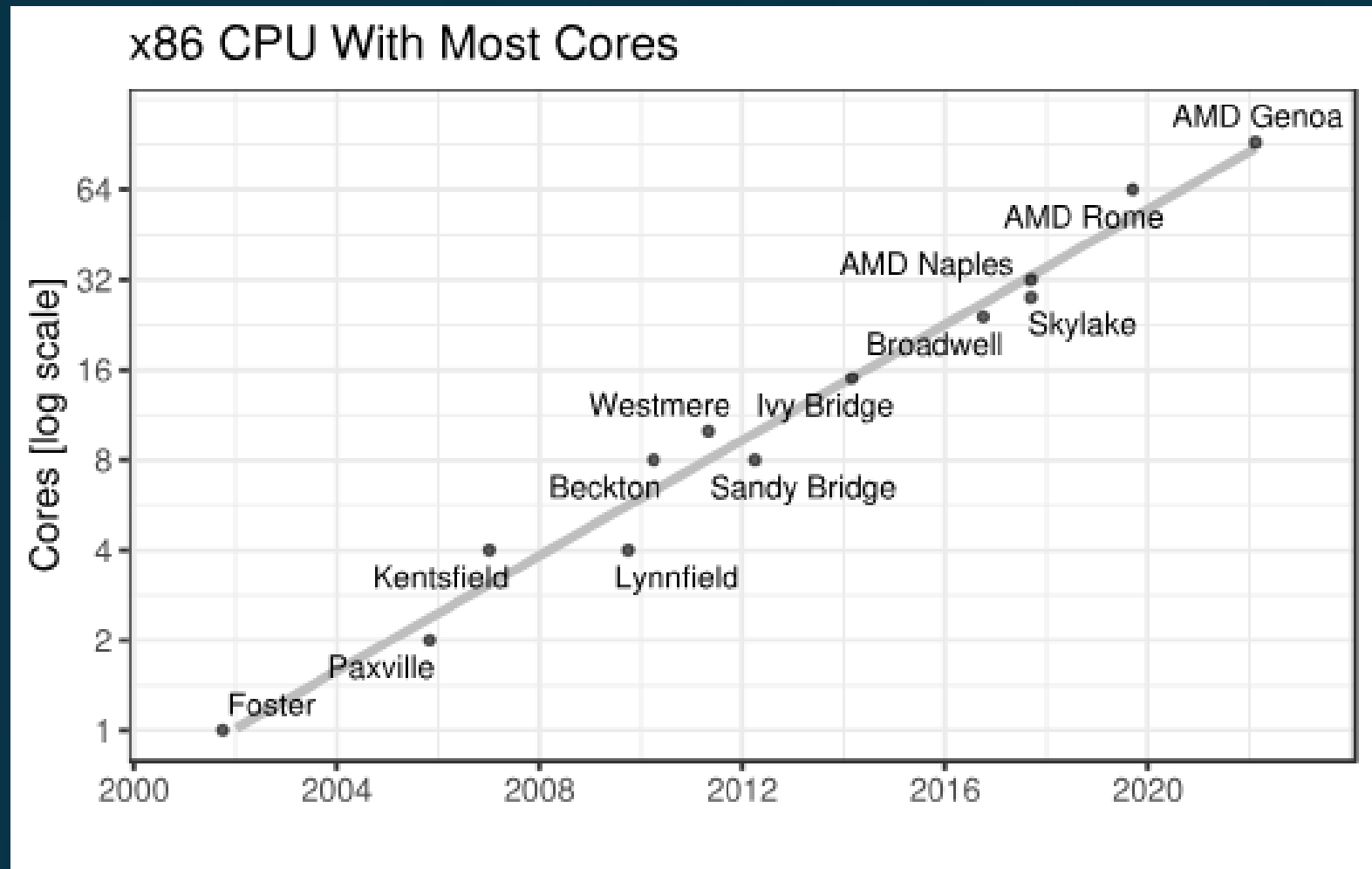
Intel CPU 클럭속도



8088	1979	5
80186	1982	6
80286	1982	4
80286	1985	10
80386	1985	12.5
80486	1989	16
	1990	33
	1991	50
	1992	66
	1994	100
P5	1995	120
	1996	166
	1997	233
	1998	266
	1999	300
P-2	1998	450
P-3	2000	1133
	2001	1400
P-4	2001	2000
	2002	3000
P-4HT	2003	3000
	2004	3400
	2005	3800
P-D	2005	3200
	2006	3600

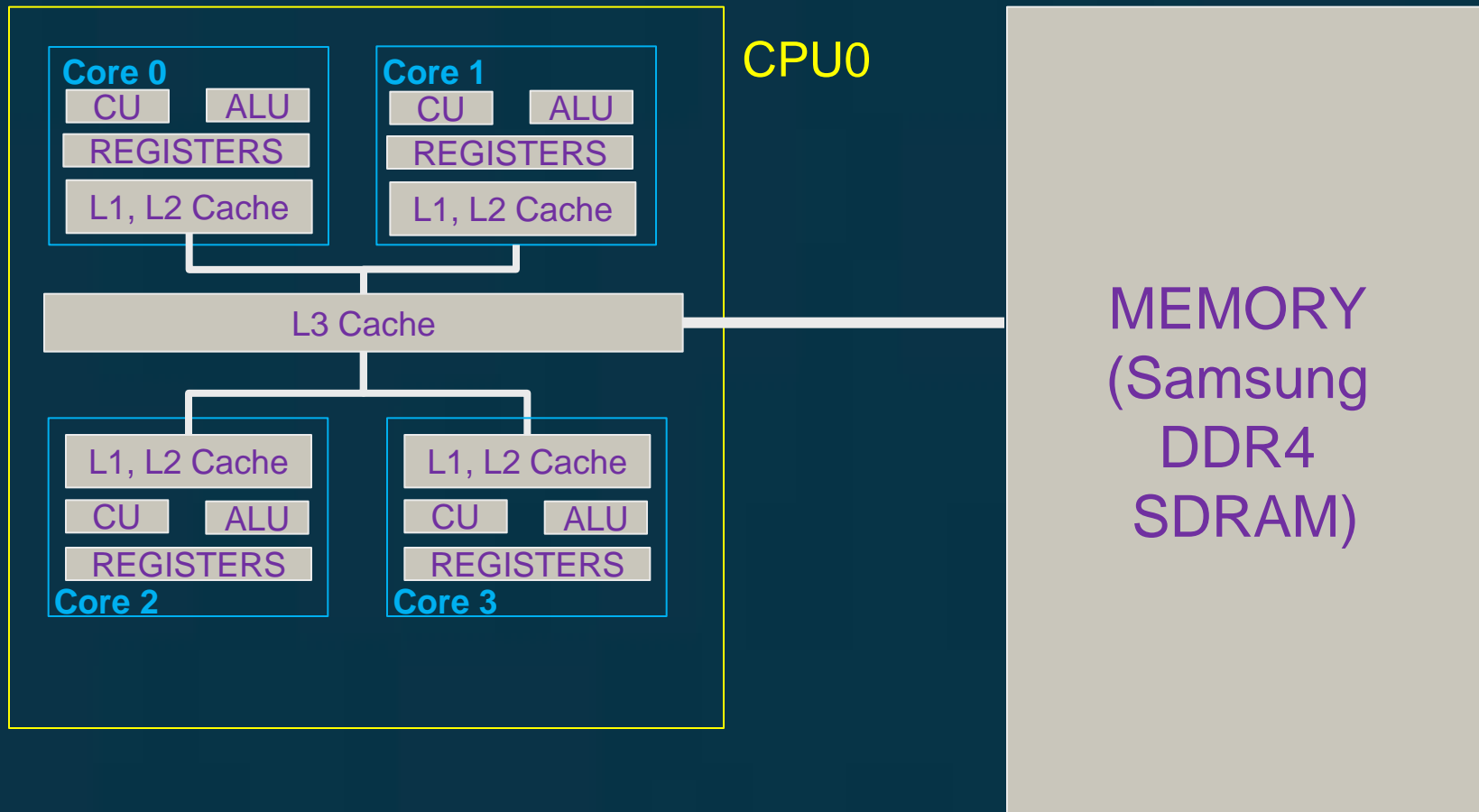


# 멀티코어프로세서



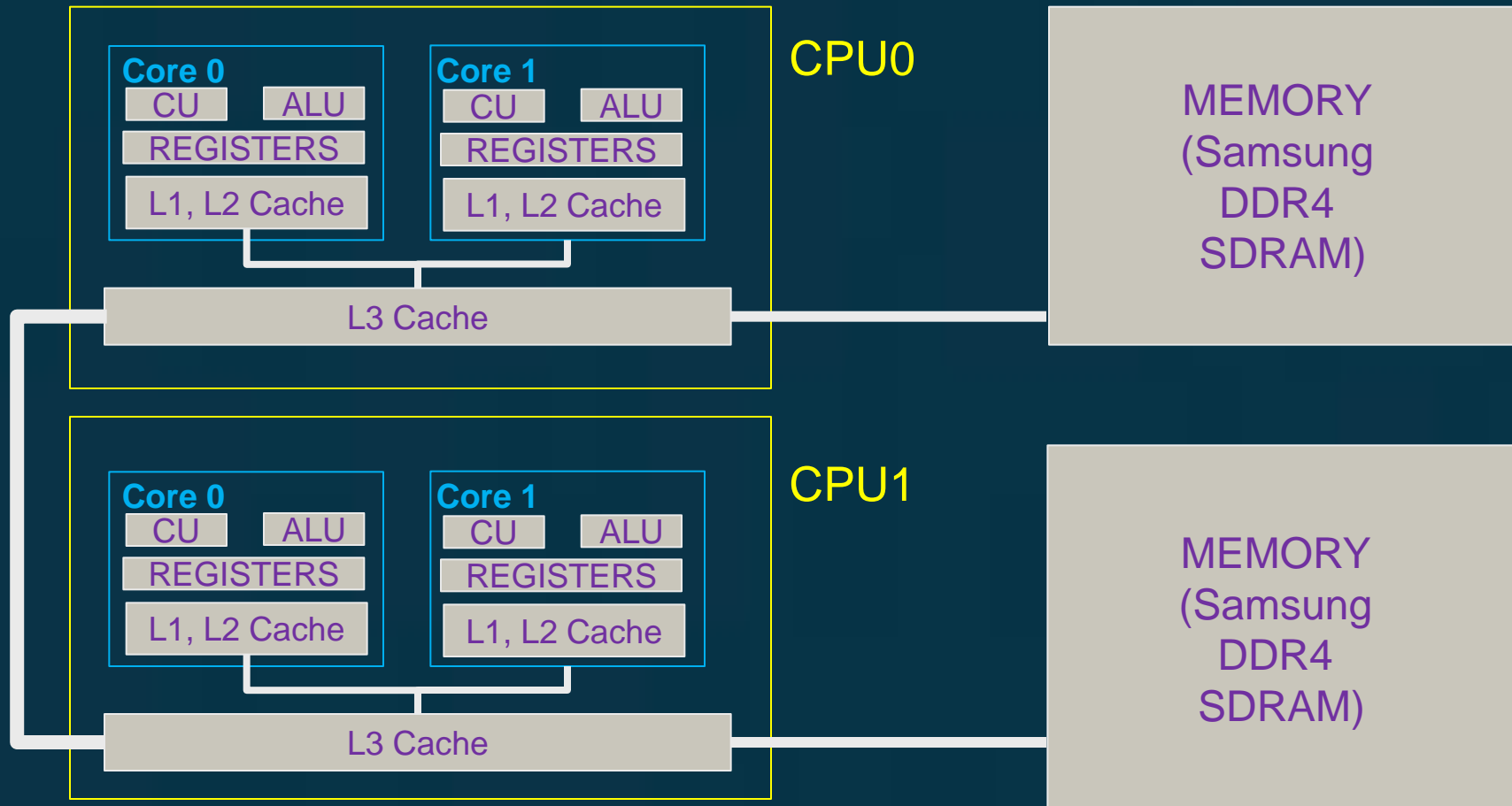
# 멀티코어프로세서

- 멀티코어프로세서 (Quad Core - 1 CPU)



# 멀티코어프로세서

- 멀티 CPU 컴퓨터 (Dual Core - 2 CPU)



# 멀티코어프로세서

- 코어?
  - ALU + Register + CU + Cache
  - 그냥 CPU와 똑같다.
- 멀티코어 CPU
  - 하나의 칩에 여러 개의 프로세서를 같이 포장해 넣은 것
  - 프로그래머의 입장에서는 멀티프로세서와 똑같다.(정확히는 SMP)

# 멀티코어프로세서

- 왜 이렇게 늦게 나왔는가?
  - 프로그래머가 싫어하기 때문
    - 프로그램을 다시 작성하지 않으면 성능향상 **ZERO**
    - 전혀 다른 알고리즘들을 사용해야 한다
    - 디버깅이 더럽게 어렵다
  - 옛날부터 있었다.
    - 과학 기술 계산용 : 기상예측, QCD, 유체역학, N-body problem
      - 수출 금지 품목
      - 무지 비쌌음
    - CRAY X-MP (1982), Pentium 4 Dual (2003)
- 현재는 모든 컴퓨터가 멀티코어

# 멀티코어프로세서

## ● TREND

### — 현재

- 4 core (Quad Core)가 대세
  - 2017년 부터 코어 개수 급속 증가.
- AMD EPYC: 64 core

Intel Xeon Platinum 9200 Family (Cascade Lake AP)						
AnandTech	Cores	Base Freq	Turbo Freq	L3 Cache	TDP	Price
Platinum 9282	56 C / 112 T	2.6 GHz	3.8 GHz	77.0 MB	400 W	head 25k\$ ~ 50k\$
Platinum 9242	48 C / 96 T	2.3 GHz	3.8 GHz	71.5 MB	350 W	shoulders
Platinum 9222	32 C / 64 T	2.3 GHz	3.7 GHz	71.5 MB	250 W	knees
Platinum 9221	32 C / 64 T	2.1				



### 1 AMD 라이젠 스레드리퍼 PRO 5995WX (샤갈 프로)

AMD(소켓sWRX8) / 4세대(Zen3) / 7nm / 64코어 / 128쓰레드 / 기본 클럭: 2.7GHz / 최대 클럭: 4.5GHz / L2 캐시: 32MB / L3 캐시: 256MB / TDP: 280W / PCIe4.0 / 메모리 규격: DDR4 / 3200MHz / 내장그래픽: 미탑재 / 쿨러: 미포함 / 8채널 ECC 메모리 지원


관련기사 컴퓨터가 아니라 CPU만 900만이요? AMD 라이젠 스레드리퍼 프로




등록월 2022.08. | 상품의견 52건 | 브랜드로그 | 관심


정품 8,298,990원 ☐ 122물 ☐

# 멀티코어프로세서

## ● 최신



Alibaba Group 해외 직구 플랫폼


PC-CPU Store >

94.4% 긍정

홈
제품 ▾
피드백



**₩10,194,400**

EPYC 9754 128C/256T CPU 256MB 캐시, 360W TDP 128 코어 256 스레드, 2.25GHz-3.1Ghz 서버 프로세서, 지지대 기가바이트 MZ73-LM1 MBD

번들: CPU

CPU

# 멀티 CPU 컴퓨터

- 멀티코어 이전의 병렬 컴퓨터
  - SMP(Symmetric Multi Processor) Computer
- 옛날
  - 과학기술 계산용
    - Burroughs Corporation introduced the D825 in 1962, a four-processor computer that accessed up to 16 memory modules through a crossbar switch
- 조금 전
  - 온라인 게임 서버 : 리니지 (약 20년 전)
- 현재
  - 고성능을 요구하는 서버 컴퓨터 또는 워크스테이션



# 멀티프로세서 컴퓨터

## ● 게임 콘솔

	PS5	Xbox Series X
Price	\$500 (PS5); \$400 (PS5 Digital Edition)	\$500
Key Exclusives	Spider-Man: Miles Morales, Horizon II: Forbidden West, Gran Turismo Sport	Halo Infinite, Senua's Saga: Hellblade 2, Forza Motorsport 8, State of Decay 3
Backwards Compatibility	Almost all PS4 games, including optimized PS4 Pro titles	All Xbox One games / Select Xbox 360 and original Xbox games
CPU	8-core 3.5 GHz AMD Zen 2	8-core, 3.8 GHz AMD Zen 2
GPU	10.3 teraflop AMD RDNA 2	12.0 teraflop AMD RDNA 2
RAM	16 GB GDDR6	16 GB GDDR6
Storage	825 GB custom SSD	1 TB custom NVMe SSD
Resolution	Up to 8K	Up to 8K
Frame Rate	Up to 120 fps	Up to 120 fps
Optical Disc Drive	4K UHD Blu-ray (Standard PS5 only)	4K UHD Blu-ray

# 멀티코어컴퓨터

- 결론

- 이미 멀티코어(와 CPU)는 현실이다.
- 앞으로 더욱 더 많은 Core를 사용하게 될 것이다.

- 그래서?

- 고성능을 필요로 하는 프로그램은 멀티코어를 지원하지 않으면 살아남지 못한다.
  - 예) 게임, 3D 그래픽, 영상처리, 인공지능

- 지원하지 않으면?

- 경쟁 제품보다 느려진다. dual core 1/2, quad core 1/4,  
.....

# 멀티코어

- 프로그래밍 방법이 바뀌어야 한다.
  - 기존의 프로그램을 Dual Core CPU에서 실행하였다. 속도 향상은 얼마인가?
    - 답) 0%
  - 멀티쓰레드 프로그래밍 을 해야 한다.
  - 좋은 멀티쓰레드 프로그래밍이란?
    - core의 개수에 비례해서 실행속도 증가

# 내용

---

- 강좌 소개
- “멀티쓰레드 프로그래밍”에 대한 소개
- 간단한 멀티쓰레드 프로그래밍 작성법
- Case Study

# 멀티쓰레드 프로그래밍

- 프로그래밍 방법

- C++ 프로그래밍 언어에 멀티쓰레드 라이브러리가 표준으로 존재
- 2011년도에 새로운 C++언어의 표준으로 C++11이 공표되었음 (*ISO/IEC 14882:2011*)
- 표준 C++언어를 지원하는 컴파일러라면 하드웨어나 운영체제에 관계 없이 사용

# 멀티쓰레드 프로그래밍

- 왜 C++11을 써야 하는가 (개인적인 경험)
  - 1985년 “BASIC(또는 FORTRAN)쓰지 왜 PASCAL쓰는가?”
  - 1987년 “PASCAL잘 쓰고 있는데 왜 C를 써야 하지?”
  - 2002년 “우리 C++, STL 써요.” by 송재경
  - 2016년 넷마블 => `std::shared_ptr`, 람다 함수, `std::atomic`,... 이곳 저곳에서 사용.

# 멀티쓰레드 프로그래밍

- 과거의 멀티쓰레드 프로그래밍 방법

- Windows

- WIN32 라이브러리에서 지원되는 API 사용.
    - Windows는 멀티쓰레드에 특화된 OS

- Linux

- pthread API를 사용하면 됨
    - “gcc -pthread test.cpp” 형식으로 사용한다.
    - Pthread는 POSIX thread의 약자

# 멀티프로세서 지원

## ● 운영체제에서의 thread 지원 (1/2)

### — Windows

- 프로세스를 구성하는 원소
- 모든 프로세스는 처음 시작 시 한 개의 thread를 갖고 실행 됨
- 운영체제가 thread를 직접 스케줄링
- 멀티 CPU(또는 Core)라면 여러 개의 쓰레드를 동시에 실행시켜 줌



# 멀티프로세서 지원

## ● 운영체제에서의 thread 지원 (2/2)

### – 리눅스

- 리눅스는 thread라는 개념이 없다.
  - 모든 것은 process
  - Pthread라이브러리가 마치 thread가 존재하는 것처럼 보이게 해준다.
- 리눅스에서 thread를 사용할 수 있다.
  - code, data, 자원을 공유하는 process를 생성할 수 있다.
  - 다른 운영체제의 thread와 다른 것이 없다.

# 멀티 쓰레드 프로그래밍

- [실습] 싱글 쓰레드 프로그램
  - 2를 5000만번 더하는 프로그램

```
#include <iostream>

int sum;

int main()
{
    for (auto i = 0; i < 50000000; ++i) sum = sum + 2;
    std::cout << "Sum = " << sum << endl;
}
```

# 멀티 쓰레드 프로그래밍

- 실습 # 2: 아래의 프로그램을 입력하고 실행하시오.

```
#include <iostream>

int sum;

int main()
{
    for (auto i = 0; i < 50000000; ++i) sum = sum + 2;
    std::cout << "Sum = " << sum << endl;
}
```

# 멀티 쓰레드 프로그래밍

- 멀티 쓰레드 프로그래밍의 시작
  - C++11의 thread

```
#include <thread>

void f();
struct F {
    void operator() ();
};

int main()
{
    std::thread t1{f};    // f() executes in separate thread
    std::thread t2{F()}; // F()() executes in separate thread
}
```

From: <http://www.stroustrup.com/C++11FAQ.html#std-threads>

# 멀티 쓰레드 프로그래밍

- 멀티 쓰레드 프로그래밍의 시작
  - (주의) 모든 쓰레드가 종료할 때 까지 메인 쓰레드가 기다려 주어야 함.

```
#include <thread>

void f();

int main()
{
    std::thread t1{f};    // f() executes in separate thread
    t1.join();            // wait for t1
}
```

From: <http://www.stroustrup.com/C++11FAQ.html#std-threads>

# 멀티 쓰레드 프로그래밍

- 쓰레드 객체 메소드

- join() : thread의 종료까지 기다리는 함수.
- joinable() : thread의 종료를 확인하는 함수
- get\_id() : kernel에서 다루는 thread의 id
- detach() : thread객체에서 thread를 분리한다.
  - thread종료에 상관없이 thread객체를 destruct할 수 있다.
  - thread의 종료를 join과 joinable로 알 수 없다.
- hardware\_concurrency() : 논리(가상) 코어의 개수

# 멀티 쓰레드 프로그래밍

- 자기 자신

- `this_thread` : 주의) `name_space`임

- 예) `cout << "my thread id " << this_thread::get_id();`

- `this_thread`의 API

- `get_id()` : 자기 자신의 `thread_id`
    - `sleep_for()` : 정해진 시간동안 `threa`의 실행을 멈춤 (busy waiting 방지)
    - `sleep_until()` : 정해진 시간까지 `threa`의 실행을 멈춤 (busy wating 방지)
    - `yield()` : 다른 스레드에게 실행 시간을 양보
      - 자기가 할 일의 우선순위가 낮을 경우
      - 작업을 끝내고 다음 작업의 도착을 기다리는 경우 : 루프를 돌면서 작업의 도착을 확인하고 `yield`를 해야 한다.

# 멀티 쓰레드 프로그래밍

## ● yield의 사용법

```
for (;;) {  
    JOB* job = nullptr;  
    do {  
        bool success = job_pool.get_job(&job);  
        while (false == success);  
        job->execute();  
    }  
}
```



```
for (;;) {  
    JOB* job = nullptr;  
    do {  
        bool success = job_pool.get_job(&job);  
        if (true == success) break;  
        this_thread::yield();  
        while (true);  
        job->execute();  
    }  
}
```



# [참조] 멀티 쓰레드 프로그래밍

---

- 멀티 쓰레드 프로그래밍의 시작
  - Windows API 소개
  - LINUX API 소개

# [참조] 멀티 쓰레드 프로그래밍

- 쓰레드 생성

```
HANDLE WINAPI CreateThread(
    __in_opt LPSECURITY_ATTRIBUTES lpThreadAttributes,
    __in SIZE_T dwStackSize,
    __in LPTHREAD_START_ROUTINE lpStartAddress,
    __in_opt LPVOID lpParameter,
    __in DWORD dwCreationFlags,
    __out_opt LPDWORD lpThreadId);
```

- *lpThreadAttributes* : thread의 보안 속성 (NULL로 하면 됨)
- *dwStackSize* : 스택의 크기 (byte) (NULL로 하면 기본값)
- ***lpStartAddress*** : 실행할 함수 이름

```
DWORD WINAPI ThreadProc( __in LPVOID lpParameter );
```

- *dwCreationFlags* : 즉시 실행(NULL) 또는 대기(CREATE\_SUSPENDED)
- *lpThreadId* : thread의 ID를 담을 int 변수의 포인터
- Return값
  - Thread의 handle

# [참조] 멀티 쓰레드 프로그래밍

- 쓰레드 종료 검사

```
DWORD WINAPI WaitForSingleObject(  
    __in HANDLE hHandle,  
    __in DWORD dwMilliseconds );
```

- hHandle : thread의 핸들
- dwMilliseconds : 끝날 때 까지 기다리는 시간
  - INFINITE : 무한정 기다린다.
- Return값
  - WAIT\_OBJECT\_0
  - WAIT\_TIMEOUT
  - WAIT\_FAILED

# [참조] 멀티 쓰레드 프로그래밍

## ● 쓰레드 생성(LINUX)

```
#include <pthread.h>

int pthread_create(pthread_t * thread, pthread_attr_t *
attr, void * (*start_routine)(void *), void * arg);
```

- thread : thread의 id
- attr : 쓰레드 생성시 사용할 옵션 (예: 스택크기)
- start\_routine** : 쓰레드가 수행할 함수
- arg : start\_routine함수에 전해줄 파라미터

# [참조] 멀티 쓰레드 프로그래밍

## ● 쓰레드 종료 대기 (Linux)

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread_id, void **thread_return);
```

—thread\_id : 종료할 쓰레드의 id

—thread\_return : 쓰레드 시작함수의 리턴값

# 멀티 쓰레드 프로그래밍

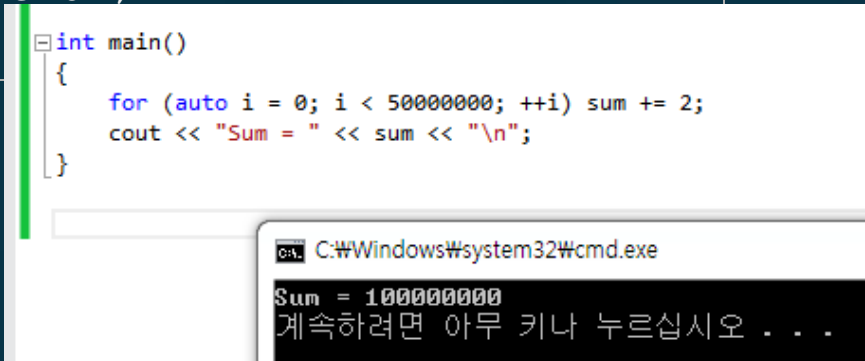
- 기존의 싱글 쓰레드 프로그램
  - 2를 5000만번 더하는 프로그램

```
#include <iostream>

using namespace std;

int sum;

int main()
{
    for (auto i = 0; i < 50000000; ++i) sum = sum + 2;
    cout << "Sum = " << sum << endl;
}
```



```
int main()
{
    for (auto i = 0; i < 50000000; ++i) sum += 2;
    cout << "Sum = " << sum << "\n";
}
```

C:\Windows\system32\cmd.exe

Sum = 100000000

계속하려면 아무 키나 누르십시오 . . .

# 멀티 쓰레드 프로그래밍

- Thread 2개를 만드는 프로그램 작성
  - 전역변수 `sum`
  - 쓰레드가 하는 일은 “`sum = sum + 2;`” 오천만번 수행
    - 하나의 쓰레드는 이천오백만번 수행
  - 쓰레드 종료 후 `sum` 출력

# 멀티 쓰레드 프로그래밍

## ● 실습 # 3 : Thread 2개를 만드는 프로그램 작성

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

using namespace std;
int sum;

void thread_func()
{
    for (auto i = 0; i < 25000000; ++i)
        sum = sum + 2;
}

int main()
{
    thread t1 = thread{ thread_func };
    thread t2 = thread{ thread_func };

    t1.join();
    t2.join();

    cout << "Sum = " << sum << "\n";
}
```



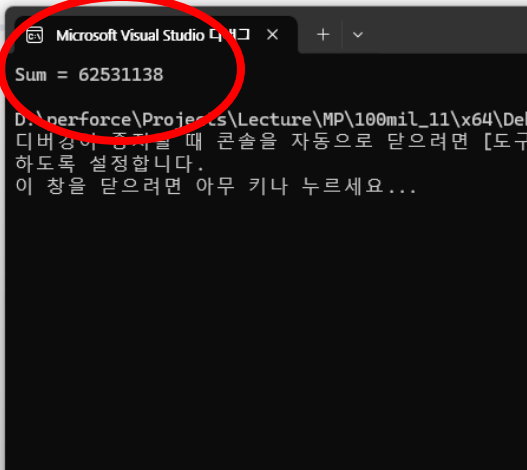
# 멀티 쓰레드 프로그래밍

- 쉬운가?
- 대답은 **NO**

```
void thread_func()
{
    for (auto i = 0; i < 250000000; ++i)
        sum = sum + 2;
}

int main()
{
    thread t1 = thread{ thread_func };
    thread t2 = thread{ thread_func };

    t1.join();
    t2.join();
}
```



The screenshot shows a Visual Studio console window with the output 'Sum = 62531138' circled in red. Below the output, there is a message in Korean: '디버깅이 종료를 할 때 콘솔을 자동으로 닫으려면 [도구] 하도록 설정합니다. 이 창을 닫으려면 아무 키나 누르세요...'.

# 멀티 쓰레드 프로그래밍

- 멀티 쓰레드 프로그래밍에서 중요한 사항
  - 올바른 결과가 나와야 한다.
    - 무한루프에 빠지거나 프로그램이 오류로 종료하면 안 된다.  
=> 당연하지만 힘들다.
  - 멀티 쓰레드로 인한 성능향상이 커야 한다.
    - 성능향상이 적으면 멀티 쓰레드 프로그래밍을 할 이유가 없다.  
=> 여러 가지 요인이 얹혀서 만족할 만한 성능향상을 얻는 것은 어렵다.

# Data Race

- 왜 틀린 결과가 나왔을까?  
— “sum = sum + 2”가 문제이다.

```

00007FF674EB9498 EB 0F 80 FF FF call     __check_for_debugger (07FF674E
    for (int i = 0; i < 1000000; ++i)
00007FF674EB949B C7 45 04 00 00 00 00 mov     dword ptr [rbp+4],0
00007FF674EB94A2 EB 08      jmp     func+2Ch (07FF674EB94ACh)
00007FF674EB94A4 8B 45 04      mov     eax,dword ptr [rbp+4]
00007FF674EB94A7 FF C0      inc     eax
00007FF674EB94A9 89 45 04      mov     dword ptr [rbp+4],eax
00007FF674EB94AC 81 7D 04 40 42 0F 00 cmp     dword ptr [rbp+4],0F4240h
00007FF674EB94B3 7D 11      jge     func+46h (07FF674EB94C6h)
    sum = sum + 2;
00007FF674EB94B5 8B 05 95 8F 00 00 mov     eax,dword ptr [sum (07FF674EC2450h)]
00007FF674EB94BB 83 C0 02      add     eax,2
00007FF674EB94BE 89 05 8C 8F 00 00 mov     dword ptr [sum (07FF674EC2450h)],eax
00007FF674EB94C4 EB DE      jmp     func+24h (07FF674EB94A4h)

```

<DEBUG> => <Windows> => <Disassembly>

# Data Race

- 왜 틀린 결과가 나왔을까?  
— “sum = sum + 2”가 문제다.

쓰레드 1  
CORE 0

쓰레드 2  
CORE 1

MOV EAX, SUM

ADD EAX, 2

MOV SUM, EAX

MOV EAX, SUM

ADD EAX, 2

MOV SUM, EAX

sum = 200

sum = 200

sum = 202

sum = 202

# Data Race

- 원인

- 공유 메모리를 여러 스레드에서 읽고 쓴다.
- 읽고 쓰는 순서에 따라 실행 결과가 달라진다.  
(프로그래머가 예상 못한 결과가 발생.)
- 이것을 **Data Race**라고 한다.

- Data Race의 정의

- 복수개의 스레드가 하나의 메모리에 동시 접근
- 적어도 한 개는 Write



# Data Race

- 해결 방법

- Data Race를 없애면 된다.

- 어떻게?

- Lock과 Unlock을 사용한다. (OS시간에 배움)

- Data Race의 정의

- 복수개의 스레드가 하나의 메모리에 **동시** 접근
      - 적어도 한 개는 Write

- 복수개의 스레드가 **동시**에 접근할 수 없도록 한다.

- 동시에는 하나의 스레드 만 접근할 수 있도록 한다.

# 멀티 쓰레드 프로그래밍

## ● Lock과 Unlock

- C++11 표준에 존재
- Mutex 클래스의 객체 생성 후 lock(), unlock() 메소드 호출

```
#include <mutex>

using namespace std;
mutex mylock;

...

mylock.lock();
// Critical Section
mylock.unlock();
```



# 멀티 쓰레드 프로그래밍

## ●Why

```
mylock.lock();
```

## ●Not

```
lock();
```

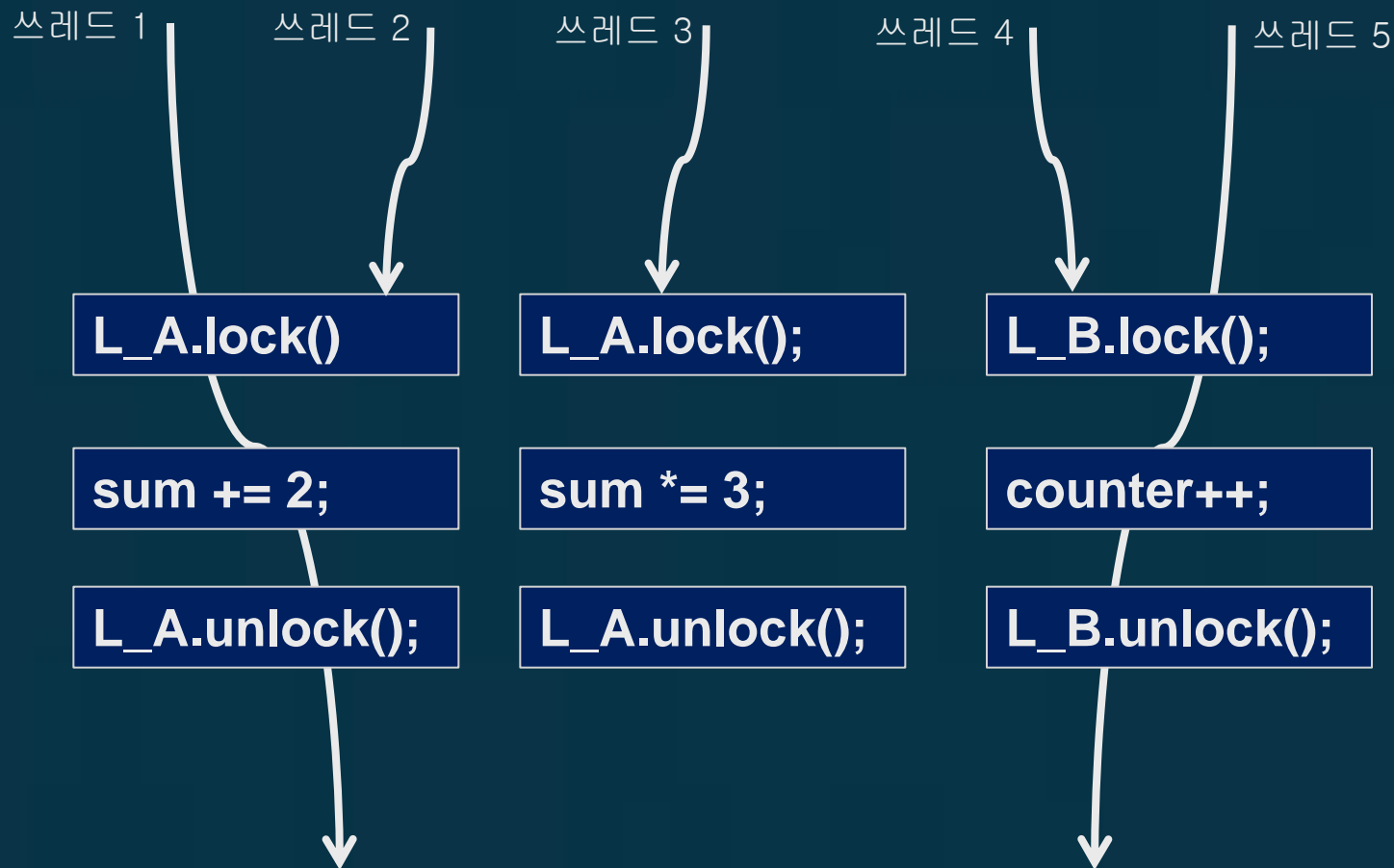
# 멀티 쓰레드 프로그래밍

- Lock과 Unlock : 주의점

- Mutex객체는 전역 변수로.
- 같은 객체 사이에서만 Lock/Unlock이 동작.
  - 다른 mutex객체는 상대방을 모름.
- 서로 동시에 실행돼도 관찰은 critical section이 있다면 다른 mutex객체로 보호하는 것이 성능이 좋음
  - 같은 mutex객체로 보호하면 동시에 실행이 안됨

# 멀티 쓰레드 프로그래밍

- Mutex 객체가 필요한 이유.



# 멀티 쓰레드 프로그래밍

- 실습 #4

- **Mutex** 객체를 사용하여 **DataRace**를 제거하여 올바른 답이 나오도록 실습 #3의 프로그램을 수정하시오.

# 멀티 쓰레드 프로그래밍

## ● 결과

```
using namespace std;
int sum;
mutex mu_a;

void thread_func()
{
    for (auto i = 0; i < 250000000; ++i)
    {
        mu_a.lock();
        sum = sum + 2;
        mu_a.unlock();
    }
}
```

Microsoft Visual Studio

Sum = 1000000000

D:\perforce\Project  
디버깅이 중지될 때  
하도록 설정합니다.  
이 창을 닫으려면 이

# (참고) 멀티 쓰레드 프로그래밍

- WIN32 API에서 Lock과 Unlock

- Critical Section이라고 부름

- `InitializeCriticalSection()`
- `EnterCriticalSection()`
- `LeaveCriticalSection()`

- CRITICAL\_SECTION 변수를 필요로 한다.

- 서로 다른 공유메모리에 대한 접근을 구별하기 위해
- 서로 다른 공유메모리에 대한 접근은 DataRace가 아니다.
- 다른 CRITICAL\_SECTION 변수에 대한 lock은 동시 수행이 가능하다. => **성능향상**

# (참고) 멀티 쓰레드 프로그래밍

## ● CRITICAL\_SECTION 변수 초기화

```
#include <windows.h>

CRITICAL_SECTION  CS_lock;

InitializeCriticalSection(CRITICAL_SECTION *CS_lock);
```

— CS\_lock : 초기화 할 CRITICAL\_SECTION 변수

# (참고) 멀티 쓰레드 프로그래밍

## ● CRITICAL\_SECTION 변수 lock

```
#include <windows.h>

CRITICAL_SECTION  CS_lock;

EnterCriticalSection(CRITICAL_SECTION *CS_lock);
```

- EnterCriticalSection() 이후의 명령어들은 하나의 쓰레드에서만 실행된다.



# (참고) 멀티 쓰레드 프로그래밍

## ● CRITICAL\_SECTION 변수 unlock

```
#include <windows.h>

CRITICAL_SECTION  CS_lock;

LeaveCriticalSection(CRITICAL_SECTION *CS_lock);
```

- Critical Section의 실행이 끝났음을 공지해서 다른 쓰레드가 Critical Section에 들어 올 수 있도록 해준다.

# (참고) 멀티 쓰레드 프로그래밍

## ● Windows win32 API 프로그래밍 예제

```
#include <windows.h>
#include <stdio.h>

int sum = 0;
CRITICAL_SECTION cs;
DWORD WINAPI ThreadFunc(LPVOID lpVoid)
{
    for (int i=1;i<=25000000;i++) {
        EnterCriticalSection(&cs);
        sum = sum + 2;
        LeaveCriticalSection(&cs);
    }
    return 0;
}
int main()
{
    InitializeCriticalSection(&cs);
    HANDLE hThread2 = CreateThread(NULL, 0, ThreadFunc, NULL, 0, NULL);
    HANDLE hThread3 = CreateThread(NULL, 0, ThreadFunc, NULL, 0, NULL);
    WaitForSingleObject(hThread2, INFINITE);
    WaitForSingleObject(hThread3, INFINITE);
    CloseHandle(hThread2);      CloseHandle(hThread3);
    printf("Result is %d\n", sum);
}
```

# (참고) 멀티 쓰레드 프로그래밍

- 리눅스 프로그래밍 예제

```
#include <pthread.h>
#include <stdio.h>

int sum = 0;

pthread_mutex_t mutex;

void *ThreadFunc(void *lpVoid)
{
    for (int i=0;i<5000000000 / 2 ;i++) {
        pthread_mutex_lock(&mutex);
        sum += 2;
        pthread_mutex_unlock(&mutex);
    }
}

int main()
{
    pthread_t thread1, thread2;

    pthread_mutex_init(&mutex,NULL);
    pthread_create(&thread1, NULL, ThreadFunc, (void*) 0);
    pthread_create(&thread2, NULL, ThreadFunc, (void*) 1);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf(" Result is %d\n", sum);
}
```

# 멀티 쓰레드 프로그래밍

---

- 결과가 옳게 나왔다. 만족하는가?
- 큰 문제가 있다.... 느껴지는가?

# 멀티 쓰레드 프로그래밍

## ● 실습 #5 :

- Lock을 사용한 프로그램과 Lock을 사용하지 않은 프로그램의 속도를 비교하시오.
- Thread의 개수를 1,2,4,8개로 변경하면서 측정하시오
  - 쓰레드의 개수가 늘어나면 하나의 쓰레드에서 돌아가는 루프의 회수가 줄어들어야 함. (전체 회수 = 5000만번)

```
#include <chrono>
using namespace std::chrono;

auto t = high_resolution_clock::now();
// 측정하고 싶은 프로그램을 이곳에 위치시킨다.
auto d = high_resolution_clock::now() - t;
cout << duration_cast<milliseconds>(d).count() << " msecs\n";
```

milli second를 측정하는 프로그램

# 멀티 쓰레드 프로그래밍

## ● 실습 힌트

- N 개의 `thread`를 저장할 수 있는 벡터(또는 배열)가 필요.
- `Thread`함수가 수행해야 할 루프의 회수는  $50000000/\text{쓰레드 개수}$
- 2 중 루프 필요
  - 쓰레드 개수 루프
  - 쓰레드 생성 루프
- `Thread`함수에 매개변수 전달.

```
void thread_func(int num_threads) {...}  
  
thread t[0] = thread{ thread_func, 3 };
```

# 멀티 쓰레드 프로그래밍

## ● 실행 시간 비교

	실행시간	결과
1 Threads	..	..
2 Threads	..	..
4 Threads	...	..
8 Threads	...	...

No LOCK

??배의 성능차이

	실행시간	결과
1 Threads	..	..
2 Threads	..	..
4 Threads	..	..
8 Threads	..	..

With LOCK

# 멀티 쓰레드 프로그래밍

## ● 실행 시간 비교 (2021)

	실행시간	결과
1 Threads	76	100000000
2 Threads	141	57675918
4 Threads	166	30286902
8 Threads	150	19423732

No LOCK

100배의 성능차이

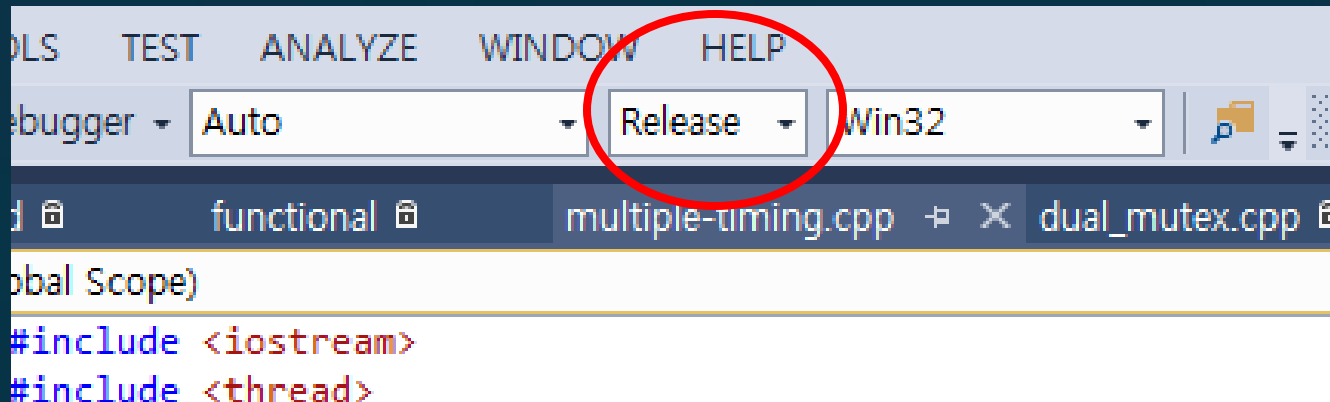
	실행시간	결과
1 Threads	5497	100000000
2 Threads	7157	100000000
4 Threads	7301	100000000
8 Threads	8302	100000000

With LOCK



# 멀티 쓰레드 프로그래밍

- 잠깐!!!
- 성능 측정????
- 디버그 모드에서의 성능 측정은 무의미
  - 멀티쓰레드 프로그래밍의 목적은 오로지 성능
- 이제부터 모든 성능은 릴리즈 모드에서 측정을 하자.



# 멀티 쓰레드 프로그래밍

## ● 실행 시간 비교 (2021) – Release Mode

	실행시간	결과
1 Threads	8	100000000
2 Threads	1	100000000
4 Threads	3	100000000
8 Threads	5	100000000

No LOCK

**100배**의 성능차이

	실행시간	결과
1 Threads	1221	100000000
2 Threads	1215	100000000
4 Threads	1309	100000000
8 Threads	1513	100000000

With LOCK

# 멀티 쓰레드 프로그래밍

## ● 실습 #6 :

- Lock을 사용한 프로그램과 Lock을 사용하지 않은 프로그램의 속도를 비교하시오.
- Thread의 개수를 1,2,4,8개로 변경하면서 측정하시오
  - 쓰레드의 개수가 늘어나면 하나의 쓰레드에서 돌아가는 루프의 회수가 줄어들어야 함. (전체 회수 = 5000만번)
- 릴리즈 모드로 컴파일 하시오
  - G++ 사용시에는 '-Ofast' 옵션을 사용하시오.

# 멀티 쓰레드 프로그래밍

---

- 결과
- 원인
- 해결책 : **VOLATILE**

# 멀티 쓰레드 프로그래밍

## ● 실행 시간 비교 (Release Mode)

	실행시간	결과
1 Threads	..	..
2 Threads	..	..
4 Threads	...	..
8 Threads	...	...

No LOCK

??배의 성능차이

	실행시간	결과
1 Threads	..	..
2 Threads	..	..
4 Threads	..	..
8 Threads	..	..

With LOCK

# 멀티 쓰레드 프로그래밍

- 실행 시간 비교 (2021) – Release Mode + volatile  
[Intel(R) Core(TM) i7-7700 CPU \_3.60GHz]

	실행시간	결과
1 Threads	91	1000000000
2 Threads	61	50436198
4 Threads	49	37940594
8 Threads	52	26257512

No LOCK

10배의 성능차이

	실행시간	결과
1 Threads	1183	1000000000
2 Threads	1203	1000000000
4 Threads	1362	1000000000
8 Threads	1555	1000000000

With LOCK

# 멀티 쓰레드 프로그래밍

- Mutex의 lock() 이라는 물건은
  - 한번에 하나의 쓰레드만 실행 시킴
    - 병렬성 감소, 병렬실행으로 인한 성능향상 감소
  - Lock을 얻지 못하면 Queue에 순서를 저장하고 스핀
    - Lock() 메소드 자체의 오버헤드
  - 심각한 성능 저하

# 멀티 쓰레드 프로그래밍

- 해결 방법은?

- Lock을 쓰지 않으면 된다.
- “Sum += 2”를 하는 동안 다른 thread가 실행되지 못하도록 하면 된다. (lock 없이)

- 어떻게?

- C++11의 **atomic** 메모리 타입을 사용한다.

```
#include <atomic>
std::atomic<int> sum;
```



# 멀티 쓰레드 프로그래밍

## ● Atomic (C++11)

- 전체 쓰레드(Core)에서 Atomic연산은 다른 Atomic연산과 동시에 수행되지 않는다.
- 같은 쓰레드의 Atomic연산의 실행 순서는 프로그래밍 순서를 따르고 절대 바뀌지 않는다.
  - C++11아닌 일반적인 정의 : atomic연산이 아닌 일반적인 연산과도 실행 순서가 바뀌지 않는다. (C++11에서는 지켜지지 않는다.)
- 모든 Atomic연산의 실행 순서는 모든 쓰레드에서 동일하게 관찰된다. (.. 이걸 나중에.. )

# 멀티 쓰레드 프로그래밍

## ● 실습 #6 :

—  $\text{sum} = \text{sum} + 2$ 를 atomic하게 바꾸어 실행하라.

```
#include <atomic>
std::atomic <int> sum;

void ThreadFunc(int num_threads)
{
    for (int i=0; i<50000000 / num_threads; i++)
        sum += 2;
    return 0;
}
```

# 멀티 쓰레드 프로그래밍

- 실행 시간 비교 (2021) – Release Mode + volatile  
[Intel(R) Core(TM) i7-7700 CPU \_3.60GHz]

	실행시간	결과
1 Threads	91	100000000
2 Threads	61	50436198
4 Threads	49	37940594
8 Threads	52	26257512

No LOCK

	실행시간	결과
1 Threads	1183	100000000
2 Threads	1203	100000000
4 Threads	1362	100000000
8 Threads	1555	100000000

With LOCK

	실행시간	결과
1 Thread	262	100000000
2 Thread	550	100000000
4 Thread	799	100000000
8 Thread	833	100000000

Atomic 연산

# 멀티 쓰레드 프로그래밍

- 주의점

- <atomic>의 한계

- <atomic> + <atomic> != <atomic>

- “**sum = sum + 2**”과 “**sum += 2**”는 다르다!

- volatile과 차이는?

- 컴파일러 레벨에서 대응이 끝나는가?

- 성능!

# 멀티 쓰레드 프로그래밍

## ● 실습 #7 :

— 다음 프로그램들의 속도 비교를 하라

- Lock이 없는 처음 프로그램
- Atomic 연산을 적용한 프로그램
- Lock을 사용한 프로그램

# 멀티 쓰레드 프로그래밍

- 결과가 옳게 나왔다. 만족하는가?

	실행시간	결과
1 Thread	..	..
2 Thread	..	..
4 Thread	..	..

No LOCK

	실행시간	결과
1 Thread	..	..
2 Thread	..	..
4 Thread	..	.

With LOCK

	실행시간	결과
1 Thread	..	..
2 Thread	..	..
4 Thread	..	..

Atomic 연산

# 멀티 쓰레드 프로그래밍

- 정답은?

- 처음 부터 Data Race가 적도록 프로그램을 재작성 하는 것이 좋다.
  - Lock이나 Atomic연산 개수를 줄일 수 있다.
- 하지만, Lock이나 Atomic연산을 완전히 없애는 것은 불가능하다.

# 멀티 쓰레드 프로그래밍

- Atomic 연산

- Mutex보다는 좋다.
- 기대하는 만큼의 성능향상이 나오는가?
- HW의 한계, 병렬알고리즘의 필요성



# 멀티 쓰레드 프로그래밍

## ● 실습 #8

- Data Race를 최소화 하도록 프로그램을 변경해 실행해 보자.
- 역시 1, 2, 4, 8 thread에서의 실행시간을 재보자.

# 멀티 쓰레드 프로그래밍

## ● 정답은?

```
void optimal_thread_func(int num_threads)
{
    volatile int local_sum = 0;
    for (auto i = 0; i < 50000000 / num_threads; ++i) local_sum += 2;
    mylock.lock();
    sum += local_sum;
    mylock.unlock();
}
```

```
int main()
{
    vector<thread*> threads;
    for (auto i = 1; i <= MAX_THREADS; ++i)
    {
        sum = 0;
        threads.clear();
        auto start = high_resolution_clock::now();
        for (auto j = 0; j < i; ++j)
            threads.emplace_back(optimal_thread_func, i);
        for (auto tmp : threads) tmp.join();
        auto duration = high_resolution_clock::now() - start;
        cout << i << " Threads" << " " << duration << endl;
    }
}
```

C:\Windows\system32\cmd.exe

```
1 Threads    Sum = 1000000000 Duration = 116 milliseconds
2 Threads    Sum = 1000000000 Duration = 58 milliseconds
4 Threads    Sum = 1000000000 Duration = 30 milliseconds
8 Threads    Sum = 1000000000 Duration = 32 milliseconds
16 Threads   Sum = 1000000000 Duration = 30 milliseconds
32 Threads   Sum = 1000000000 Duration = 28 milliseconds
64 Threads   Sum = 1000000000 Duration = 23 milliseconds
```

계속하려면 아무 키나 누르십시오 . . .

# 멀티 쓰레드 프로그래밍

## ● 만족하는가? (실습 PC)

	실행시간	결과
1 Thread	..	..
2 Thread	..	..
4 Thread	..	..

No LOCK

	실행시간	결과
1 Thread	..	..
2 Thread	..	..
4 Thread	..	..

With LOCK

	실행시간	결과
1 Thread	..	..
2 Thread	..	..
4 Thread	..	..

Atomic 연산

	실행시간	결과
1 Thread	..	..
2 Thread	..	..
4 Thread	..	..

정답

# 멀티 쓰레드 프로그래밍

- 실행 시간 비교 (2021) – Release Mode + volatile  
[Intel(R) Core(TM) i7-7700 CPU \_3.60GHz]

	실행시간	결과
1 Threads	91	100000000
2 Threads	61	50436198
4 Threads	49	37940594
8 Threads	52	26257512

No LOCK

	실행시간	결과
1 Thread	262	100000000
2 Thread	550	100000000
4 Thread	799	100000000
8 Thread	833	100000000

Atomic 연산

	실행시간	결과
1 Threads	1183	100000000
2 Threads	1203	100000000
4 Threads	1362	100000000
8 Threads	1555	100000000

With LOCK

	실행시간	결과
1 Thread	84	100000000
2 Thread	38	100000000
4 Thread	23	100000000
8 Thread	19	100000000

정답

# 멀티 쓰레드 프로그래밍

## ● 만족하는가? (i7-920 – 4core)

	실행시간	결과
1 Thread	116	100000000
2 Thread	62	52661344
4 Thread	49	34748866
8 Thread	43	28327790

No LOCK

	실행시간	결과
1 Thread	3,226	100000000
2 Thread	15,642	100000000
4 Thread	11,245	100000000
8 Thread	14,191	100000000

With LOCK

	실행시간	결과
1 Thread	379	100000000
2 Thread	555	100000000
4 Thread	568	100000000
8 Thread	566	100000000

With Atomic

	실행시간	결과
1 Thread	113	100000000
2 Thread	61	100000000
4 Thread	30	100000000
8 Threads	33	100000000

정답

# 멀티 쓰레드 프로그래밍

- 실행결과 (XEON E5-4620, 8 core X 4 CPU)
  - 10억 만들기
  - Ubuntu 18.04 Linux, gcc 7.5.0

쓰레드	시간	결과
1 개	1424	100000000
2 개	1077	502319024
4 개	757	254006316
8 개	328	137059012
16 개	621	226626432
32 개	519	132260364
64 개	411	107912026

No LOCK

쓰레드	시간	결과
1 개	4299	100000000
2 개	31960	100000000
4 개	14344	100000000
8 개	16684	100000000
16 개	16450	100000000
32 개	16388	100
64 개	15059	100

Atomic

쓰레드	시간	결과
1 개	13762	100000000
2 개	56082	100000000
4 개	78309	100000000
8 개	62968	100000000
16 개	62163	100000000
32 개	-	100000000
64 개	-	100000000

With LOCK

쓰레드	시간	결과
1 개	1425	1000000000
2 개	733	1000000000
4 개	389	1000000000
8 개	205	1000000000
16 개	126	1000000000
32 개	88	1000000000
64 개	77	1000000000

정답

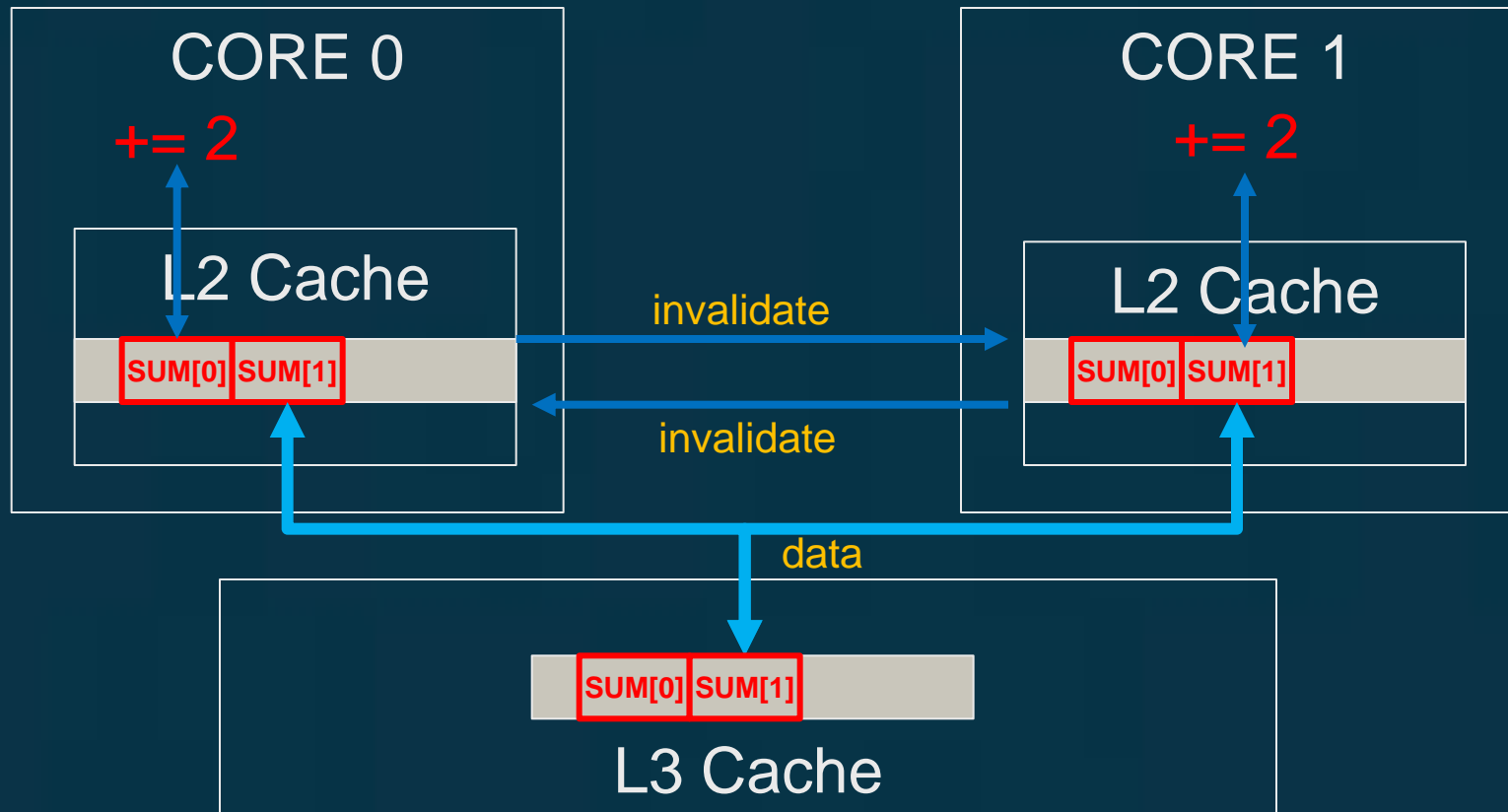
# 다른 정답

- SUM을 local변수로 해야 하는가?
  - SUM[]
  - 스레드 종료 후 합산
- 문제
  - 올바른 값이 나오는가? 성능이 스레드 개수만큼 올라가는가?
  - Cache Thrashing
    - Invalidation PingPong

# 다른 정답

- Cache Thrashing

— 원인 : **False Sharing**





# 다른 정답

## ● Cache Thrashing 해결

– 스레드마다 서로 다른 cache line을 사용하도록 한다.

• `volatile int sum[NUM_THREAD * CACH_LINE_SIZE]`

– C++ `alignas` 키워드 사용

```
struct NUM {  
    alignas(64) volatile int sum;  
};
```

```
NUM num[MAX_THREAD];
```

# 정리

- 병렬 컴퓨팅이란 무엇인가?
- 쓰레드란 무엇인가?
- 왜 멀티쓰레드 프로그래밍을 해야 하는가?
- 멀티쓰레드 프로그래밍은 어떻게 하는가?
  - C++11 표준 라이브러리
- 멀티쓰레드 프로그래밍의 어려움
  - Data Race
  - 성능

# 정리

끝!



# 내용

---

- 강좌 소개
- “멀티쓰레드 프로그래밍”에 대한 소개
- 간단한 멀티쓰레드 프로그래밍 작성법
- Case Study

# Case Study

- 멀티 쓰레드 프로그래밍의 사용처
  - 과학 기술 계산
    - 오래전부터...
  - 멀티 미디어
    - Encoding & Decoding
    - 멀티쓰레드의 구현이 매우 쉬움
      - 독립된 장면들 (key frame 단위)
  - 게임
    - 온라인 게임 서버 1997년 부터
    - 3D 게임엔진 2000년대 중반 부터

# Case Study

- Unreal 4

- 3D 게임엔진

- 3D 그래픽을 구현하는 Library와 Contents제작 툴들의 집합

- [https://www.youtube.com/watch?v=PD5cRnrMqWw&index=29&list=PLZlv\\_N0\\_O1gYfw89JtRX0bTb\\_YbxHOXwz](https://www.youtube.com/watch?v=PD5cRnrMqWw&index=29&list=PLZlv_N0_O1gYfw89JtRX0bTb_YbxHOXwz)

- 선택

- 10억원 정도
    - 무료 사용, 단 백만달러 이상 매출 달성 시 매출의 5%

- 플랫폼

- PC, PS4, XBOX-One, iOS, Android

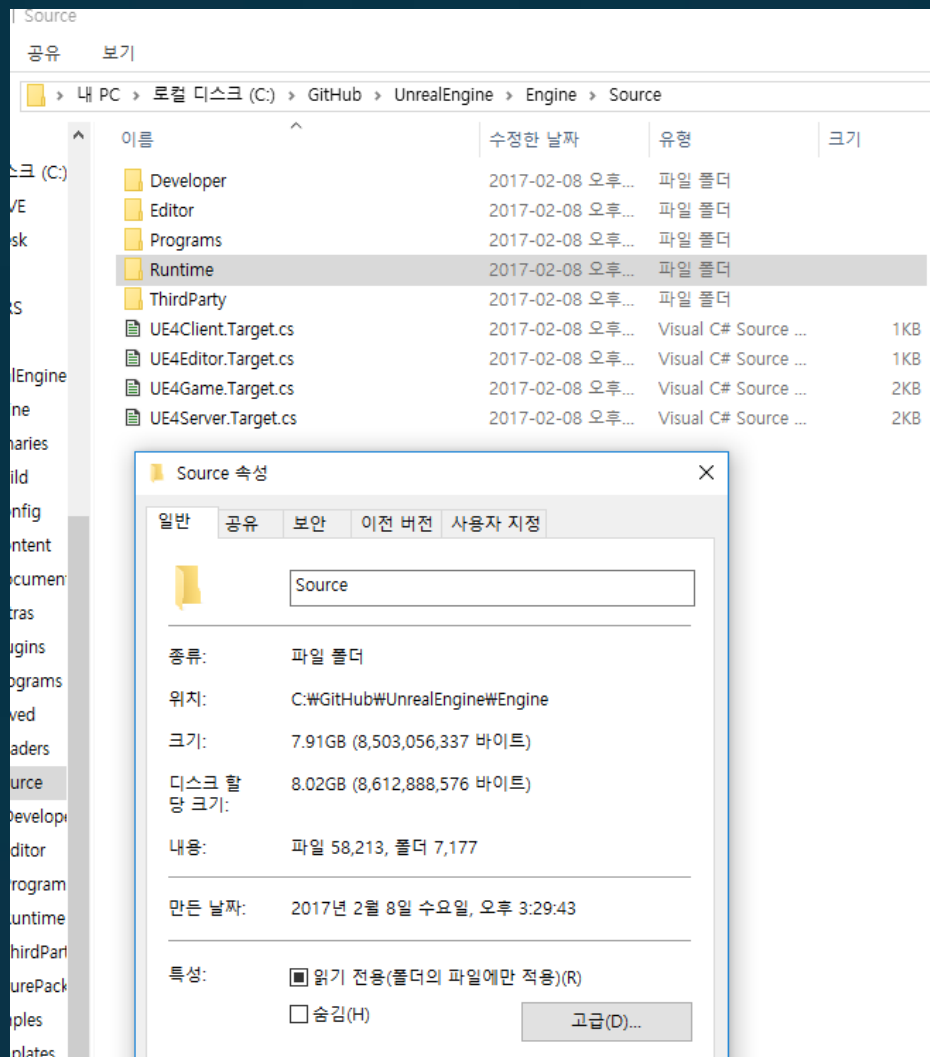
- 2012년 공개

- Heterogeneous 멀티쓰레딩

# Case Study

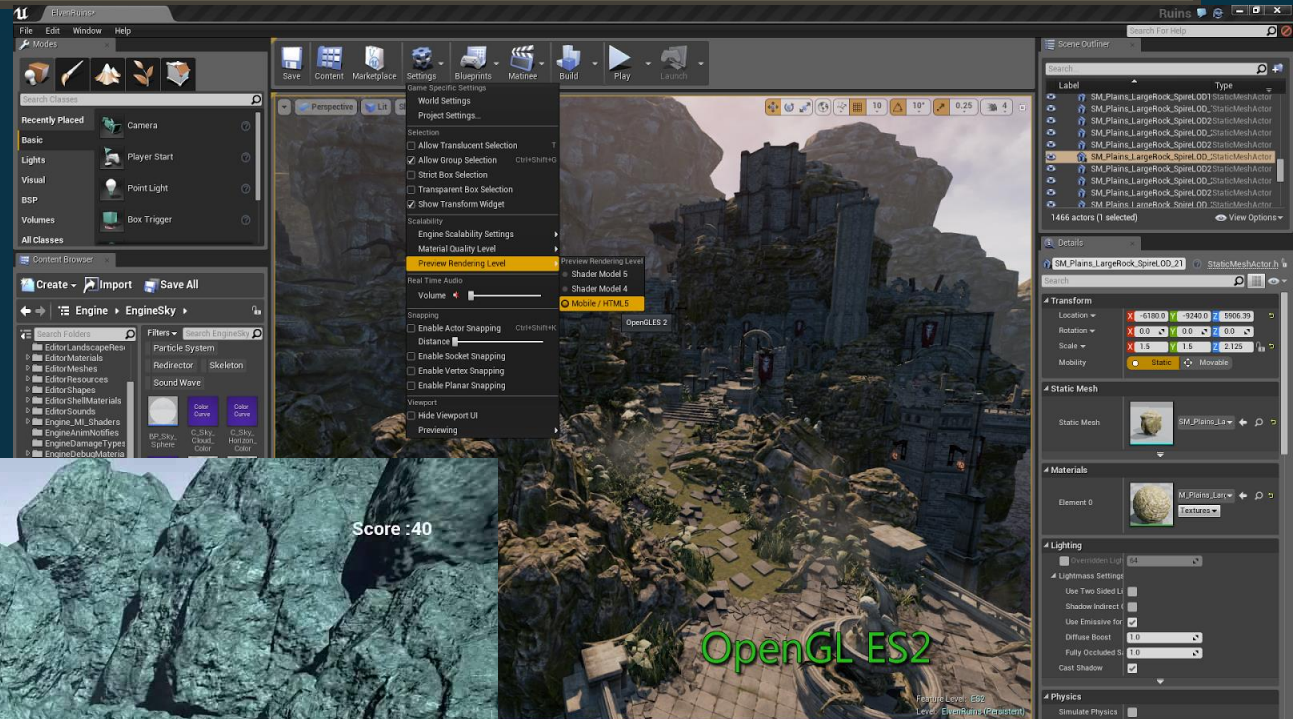
## ● Unreal 4

— 소스코드만 8GB



# Case Study

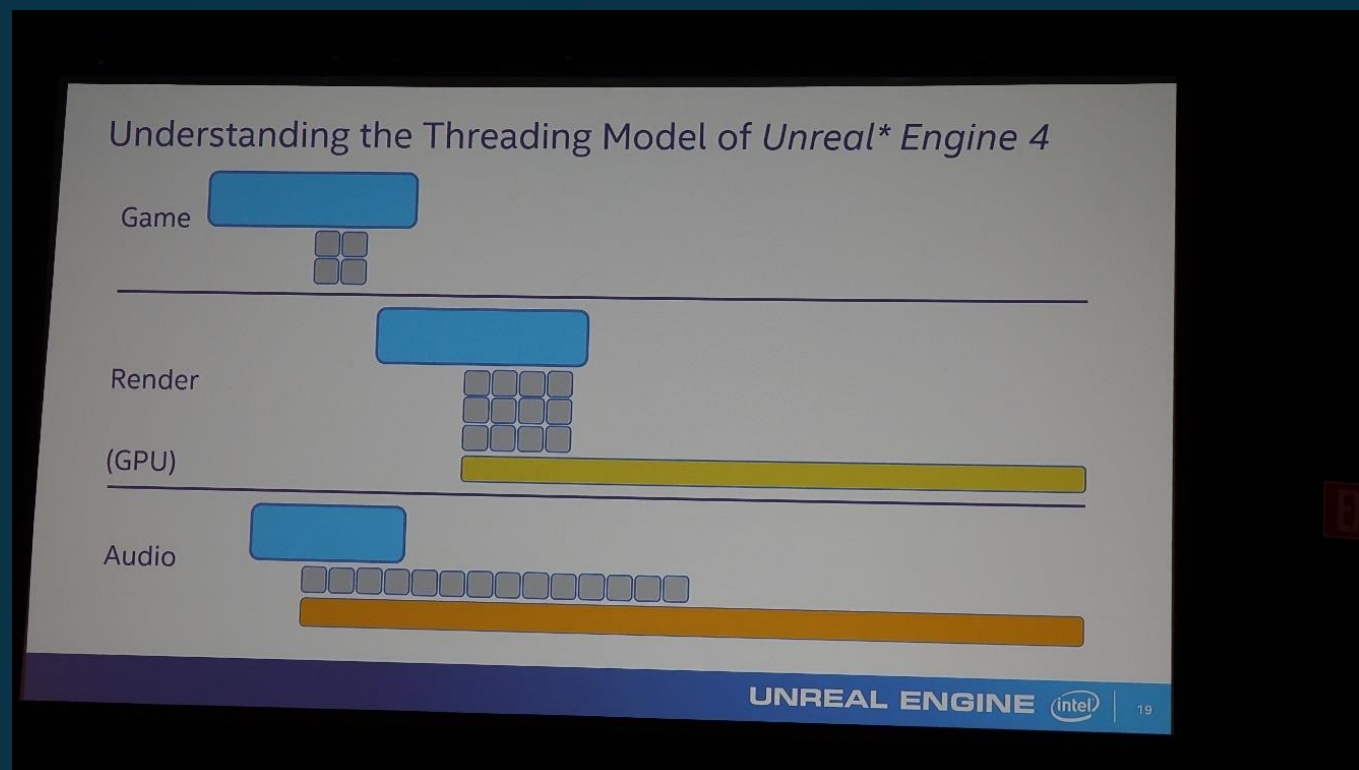
## ● Unreal 4





# Case Study

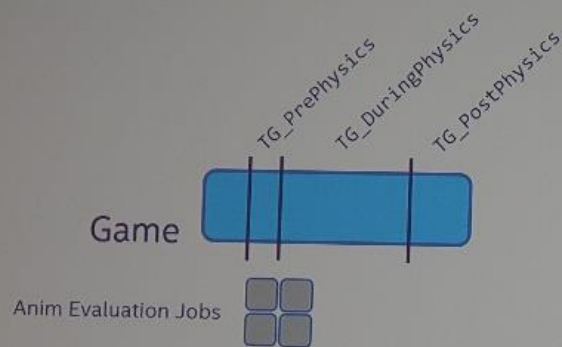
- 3개의 주 쓰레드
- 많은 개수의 Worker Thread 쓰레드
  - 주 쓰레드가 생산한 작업을 병렬 수행.



# Case Study

## ● Game Thread

### Understanding the Threading Model of *Unreal\* Engine 4*: Game Thread (and Friends)



The Game Thread handles updates for gameplay, animation, physics, networking, etc., and most importantly, Actor ticking.

We can control the order in which objects Tick, by using Tick Groups. They don't give you parallelism, but they do allow you to control dependent behavior.

Physics use tasks within the game thread to perform their work

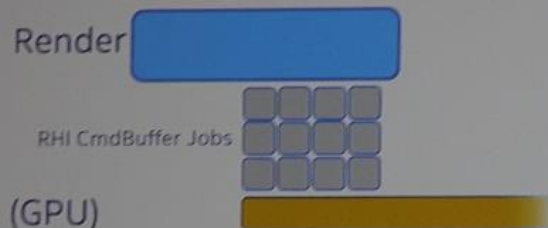
We do let you evaluate animation graphs in parallel, which we use for *Robo Recall\**.

# Case Study

## ● Render Thread

### Understanding the Threading Model of *Unreal\* Engine 4*:

#### Render Thread



The Render Thread handles generating render commands to send to the GPU.

In general, at the beginning of the thread, we do a final update of position and orientation from the HMD, traverse the scene, and then generate command buffers to send to the GPU.

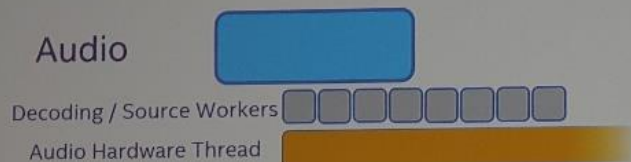
The command buffer generation can be done in parallel to decrease the time it takes to generate commands for the whole scene, and kick off work to the GPU.

# Case Study

## ● Audio Thread

### Understanding the Threading Model of *Unreal\* Engine 4*:

#### Audio Thread



*Robo Recall\** is the first title to ship with the new audio mixing and threading system.

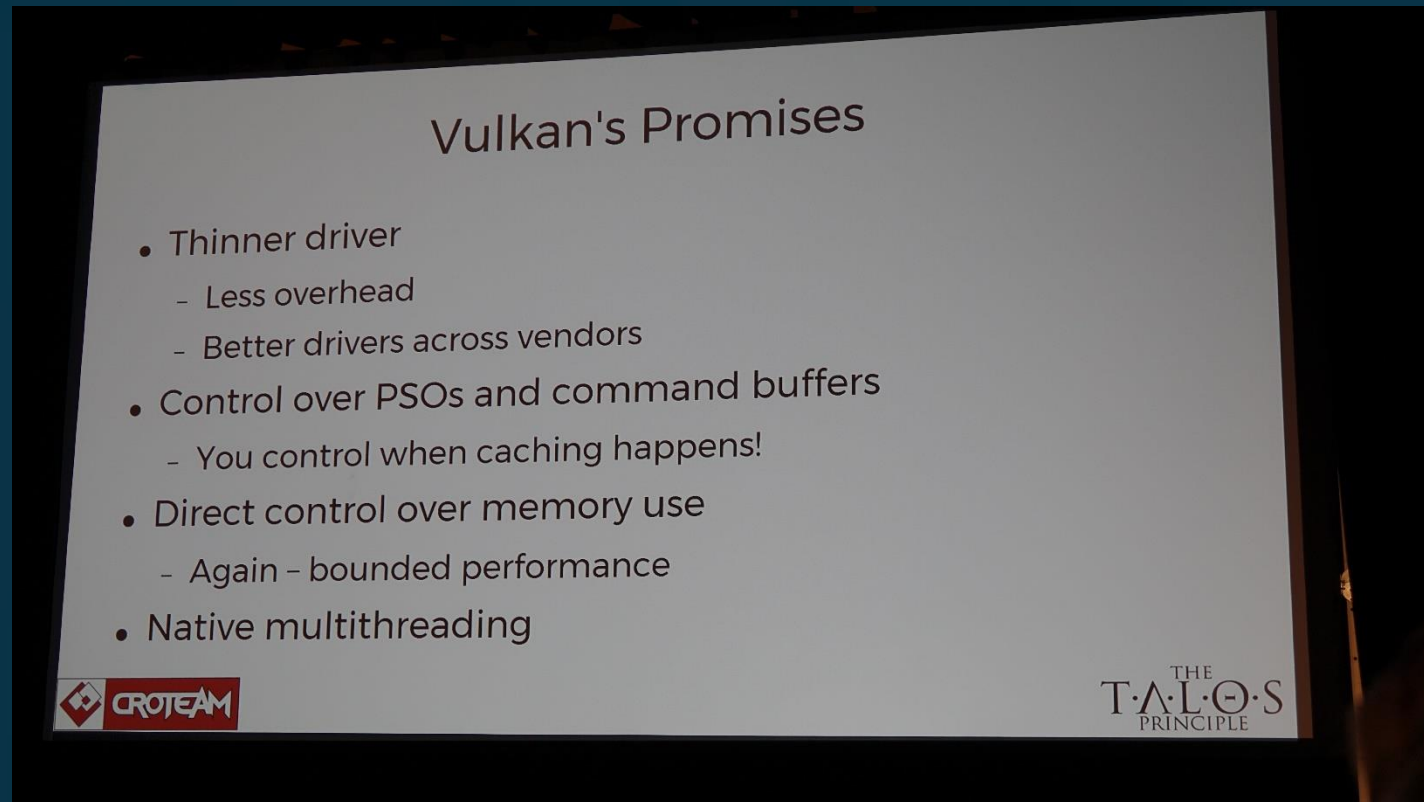
The main audio thread is analogous to the Render Thread, and interfaces the game thread with the lower level mixing threads.

The decoding and source worker tasks decode the audio information, and also do processing like spatialization and HRTF.

The audio hardware thread is a platform dependent thread (e.g. XAudio2 on Windows\*), which consumes the mix.

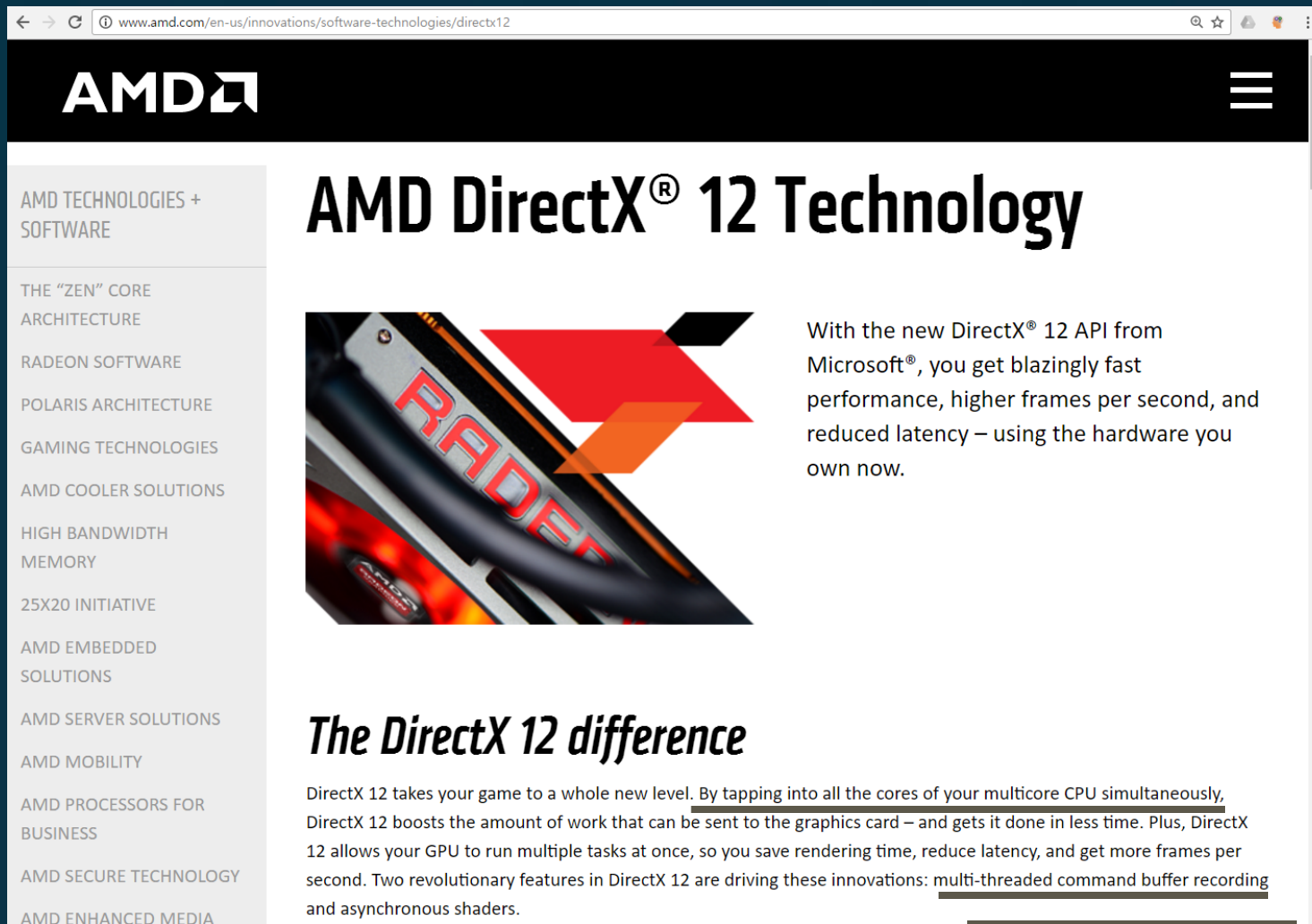
# Case Study

- 3D API의 변천
  - OpenGL -> Vulkan



# Case Study

## ● 3D API의 변천 : D3D11 -> D3D12




The screenshot shows the AMD website's 'DirectX 12 Technology' page. The page features a navigation menu on the left with links to various AMD technologies and solutions. The main content area is titled 'AMD DirectX® 12 Technology' and includes a large image of a Radeon graphics card. To the right of the image, there is a paragraph describing the benefits of DirectX 12. Below this, a section titled 'The DirectX 12 difference' provides a detailed explanation of the technology's capabilities and features.

AMD TECHNOLOGIES + SOFTWARE

- THE "ZEN" CORE ARCHITECTURE
- RADEON SOFTWARE
- POLARIS ARCHITECTURE
- GAMING TECHNOLOGIES
- AMD COOLER SOLUTIONS
- HIGH BANDWIDTH MEMORY
- 25X20 INITIATIVE
- AMD EMBEDDED SOLUTIONS
- AMD SERVER SOLUTIONS
- AMD MOBILITY
- AMD PROCESSORS FOR BUSINESS
- AMD SECURE TECHNOLOGY
- AMD ENHANCED MEDIA

## AMD DirectX® 12 Technology



With the new DirectX® 12 API from Microsoft®, you get blazingly fast performance, higher frames per second, and reduced latency – using the hardware you own now.

### *The DirectX 12 difference*

DirectX 12 takes your game to a whole new level. By tapping into all the cores of your multicore CPU simultaneously, DirectX 12 boosts the amount of work that can be sent to the graphics card – and gets it done in less time. Plus, DirectX 12 allows your GPU to run multiple tasks at once, so you save rendering time, reduce latency, and get more frames per second. Two revolutionary features in DirectX 12 are driving these innovations: multi-threaded command buffer recording and asynchronous shaders.

# Case Study

- 멀티쓰레드 프로그래밍의 종류
  - Heterogeneous 멀티쓰레딩
    - 쓰레드마다 맡은 역할이 다르다.
    - 다른 Code Part를 실행
    - 쓰레드간의 Load Balancing이 힘들다.
    - 병렬성이 제한된다.
  - Homogeneous 멀티쓰레딩
    - Data/Event Driven 프로그래밍
    - 모든 쓰레드는 Symmetric하다.
      - available한 순서대로 다음 작업을 처리한다.
    - 자동적인 load balancing, 제한없는 병렬성
    - 작업 분배 Queue를 비롯한 일반적인 병렬 자료구조 필요.



# Case Study

- MMORPG Game Server

- Massively Multiplayer Online Role Playing Game

- 한 개의 서버 프로세스에 5000명 이상의 사용자가 동시 접속해서 게임 실행

- 게임 서버는 가상 환경을 시뮬레이션하는 네트워크 이벤트 시뮬레이터

- 5000명 플레이어 (소켓)

- 수십만 몬스터 (AI)

- 20km X 20km 월드 (충돌체크, 길찾기)

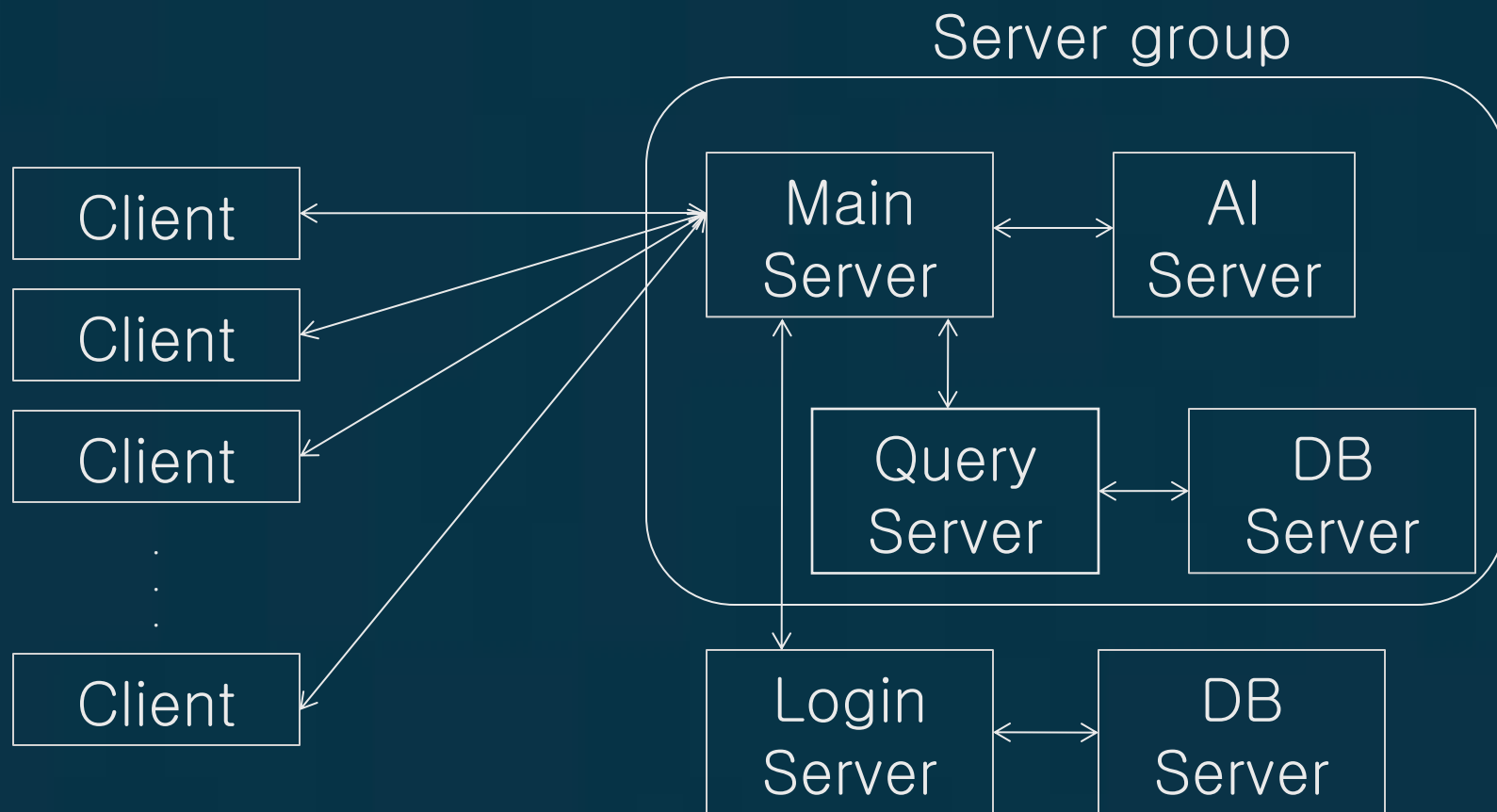


# ● MMORPG



# Case Study

- MMORPG Game Server



# Case Study

## ● MMORPG Game Server의 구현

- Windows에서 제공하는 멀티쓰레드+Network I/O API인 IOCP사용
  - 일반적인 `Select()`함수로는 몇천개의 socket을 관리할 수 없음
  - socket하나당 하나의 쓰레드는 운영체제의 과부하
  - Linux나 iOS는 `epoll` 또는 `kqueue`를 사용
    - 2019년 Linux `io-uring` 추가
- Time Consuming 작업 및 Blocking 작업들의 재배치
  - Throughput과 함께 response time도 중요.

# Case Study

- MMORPG Game Server

- Homogeneous Multithreading

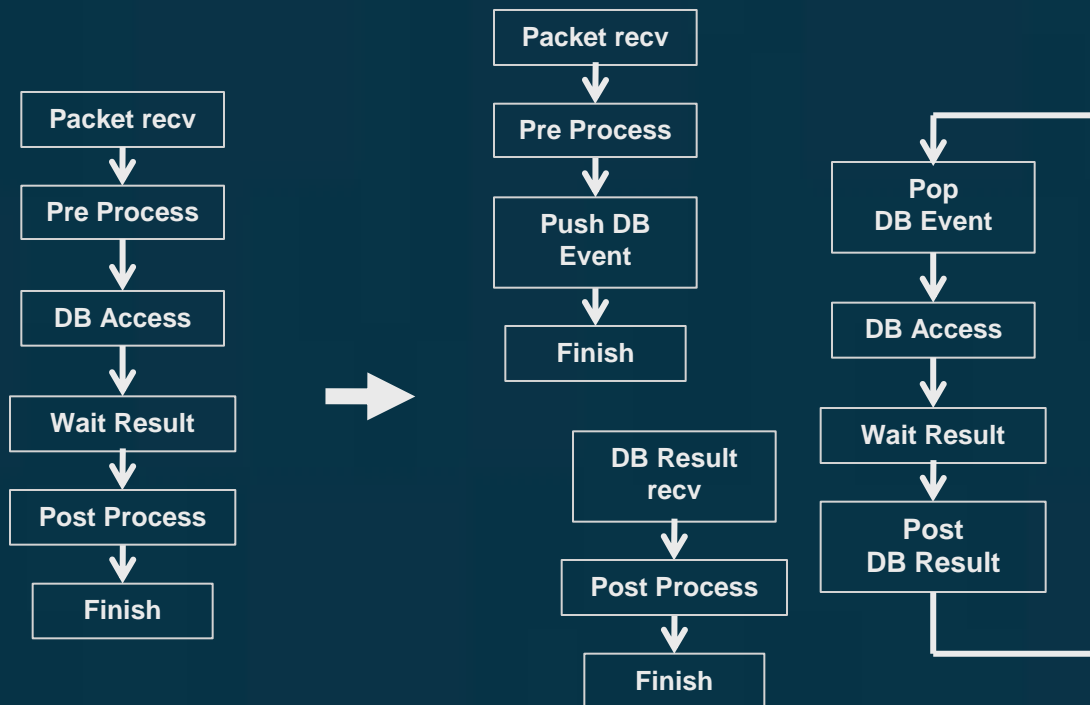
- Worker thread의 pool을 사용
- socket을 통해 packet이 올 때 마다, OS가 thread pool의 thread를 하나 깨워서 packet의 처리를 맡김.
- HW Core 개수 1.5배의 worker threads
- Network 데이터 처리 뿐만 아니라 AI 루틴도 worker thread에서 처리
  - 모든 time consuming작업은 병렬로 처리

# Case Study

## ● MMORPG Game Server

### — 멀티쓰레드 트릭 #1 : DataBase query

- 문제 : database 쿼리는 쓰레드의 blocking을 초래
- 해결 : blocking 전용 쓰레드를 따로 두어서, 작업 전달




# Case Study

## ● MMORPG Game Server

### — 멀티쓰레드 트릭 #2 : NPC AI

- 문제 : NPC가 너무 많다.
- 해결 : Timer Thread를 사용하여, Active한 NPC의 Active한 event만 처리.

```
heart_beat()
{
    if (my_hp < my_max_hp)
        my_hp += HEAL_AMOUNT;
}
```



```
get_damage(int dam)
{
    my_hp -= dam;
    add_timer(my_heal_event, 1000);
}

my_heal_event()
{
    my_hp += HEAL_AMOUNT;
    if (my_hp < my_max_hp)
        add_timer(my_heal_event, 1000);
}
```

```
Event_queue timer_queue

TimerThread()
do {
    sleep(1)
    do {
        event k = peek (timer_queue)
        if (k.starttime > current_time())
            break
        pop (timer_queue)
        post_event_to_workerthread(k)
    } while true;
} while true;
```



# Case Study

## ● MMORPG Game Server

### — Worst Case Tuning

- 평상시의 CPU낭비 OK, 최대 부하일 때 잘 버티는 프로그램이 최고

### — 컨테이너 자료구조를 사용한 객체간 동기화

- Queue, Priority Queue, Set
- Custom 자료구조
  - 1 writer/multi reader Queue
  - Timer queue : multi writer/1 reader Priority Queue
  - View list : Set with copy method

### — Thread & Cache affinity

# 질문?

---